Functional paradigms in LISP and its contribution to reliable, maintainable software

Luka Dekanozishvili

***Abstract*—This paper explores the work of John McCarthy on the LISP programming language [1] and focuses on its underlying functional paradigms, its everlasting influence today, and its application in modern programming. The central research question is: In what way do the functional programming paradigms introduced in LISP contribute to the development of more reliable and maintainable software?**

A major challenge in modern-day software development is ensuring correctness and a lack of runtime issues. As programs grow in complexity, stateful monolithic functions with mutable states become harder to properly test. This leads to untested and unreliable programs, which are difficult to validate. LISP introduced key concepts, such as recursion, immutability, and functions as first-class citizens, that offer promising solutions to these issues.

We will explore through logical reasoning how LISP's functional design allows for a more predictable program execution, supports testing philosophies, and how its modularity aids in collaborative development. Although empirical studies on this topic are still emerging, theoretical and practical evidence suggest its usefulness in modern software development.

***Index Terms*—LISP, functional programming paradigms, purity, testability, modularity**

I. Introduction and motivation

The LISP (LISt Processor) language, as described by John McCarthy, operates on singly linked lists, which is a famously recursive data structure. This allows for defining purely recursive functions elegantly through mathematical notation, which is the basis of functional programming (FP). By defining a myriad of primitive functions using a simple syntax, passing functions as arguments becomes a natural part of the language since functions are treated as first-class citizens.

Although McCarthy originally intended LISP to only be a formalism for doing recursive function theory, the practical applications of LISP as a programming language were later realized [2].

Today, Lisp is a family of languages with many dialects. Common Lisp is multi-paradigm, while Clojure focuses on FP.

It is worth noting that most Lisps including John McCarthy's LISP are mutable languages, with exceptions like Clojure, which encourages immutability. Enforcing immutability is in many cases optional, but it comes with benefits, as discussed in the next section.

This paper argues about the benefits of enforcing the aforementioned restrictions, and using functional paradigms to benefit the development process.

II. Main content

A. Functional paradigms

The book "Programming Languages: Principles and Paradigms" states a common misconception among beginner software developers that designing and writing efficient programs is the most important task of the job [3]. In practice, incorrect, unreadable, and hard to maintain software accounts to over fifty percent of the project costs. Tremendous resources could be saved if the focus were shifted to readability, reliability and correctness, but the reality shows that most businesses favor rapid development and software that just works in the present. Programmers who care about software quality often make compromises in order to meet deadlines, as stated by Austin [4].

When software developers are rewarded based on their performance and not the contribution to the software ecosystem, it's no wonder the software becomes unreliable and unmaintainable at later stages, often requiring complete rewrites in different programming languages.

By shifting the underlying paradigms to FP, the focus of the development can be shifted to producing high-quality, maintainable, and reusable software, as discussed in this paper.

Elm is a Lisp focusing on responsive GUI creation that enforces immutability, and referentially transparent and pure functions [5]. If the user makes semantic mistakes in Elm, compile-time errors are presented in all the places, so programming becomes more methodical and step-by-step. Because of what the language restricts, no runtime errors can ever occur because the program will throw compile-time errors instead.

Languages and tools like Elm reward a non-rushed, methodical development process that pays out in the long term because runtime errors can be mitigated and further guarantees about the program execution can be made.

B. The influence on modern programming

Type systems, generics, polymorphism, and type safety all originated from FP since it's easier to reason about in a side-effect-free environment [3]. These are crucial concepts that modern languages embrace - there is a reason why TypeScript

with its type system is generally preferred over JavaScript in web development.

Another well-known example for the benefits of a strict type system is Rust. Since type-mismatch errors are non-existent, logical mistakes have to be fixed before the program is even ran. Because of this, the formal verification of Rust programs is simpler than other imperative languages with looser types, and can be done using tools like the Prusti verifier [6].

Furthermore, Python allows optional types for variables and function signatures. By choosing to add types, the developer gets rewarded with improved suggestions, and is made aware of mismatched-type errors while programming. This results in a better programming experience.

C. Acknowledgment about FP's difficulty

Because learning and mastering FP requires a fundamentally different approach, they are seen as more difficult by the unfamiliar developer. This is why FP languages are often not the first choice for projects. Some people claim that FP or Lisps are "only useful for academics" since they were created for mathematical formalism, and are widely used in the field, but this statement is untrue since functional paradigms and Lisps have been widely adopted in the software development ecosystem, and have proven to be useful.

One example of this would be Emacs Lisp, which is used for configuring Emacs, a family of text editors [7].

Another example is the functional high-level language Erlang, developed by Ericsson, which was developed for telephony switches. It was also used in 3G, LTE networks as well as by Nortel and Deutsche Telekom [8]. Erlang is distributed, fault-tolerant, real-time, and is still widely used to this date.

D. Benefits of modularity

Khanfor et al. highlight how FP encourages program reuse through higher-order functions like map and $fold$, as stated by Wadler [9], [10]. Wadler also mentions that reuse in FP is often "invisible" since it is an inherent part of the language design. They also refer to Hughes et al., according to whom functional developers claim increased productivity due to concise functions, because conventional (imperative) programs consist of "90% assignment statements" [11].

The authors of "Is functional programming better for modularity?" examine the often implied hypothesis that functional programming is inherently superior in supporting modularity than other paradigms such as procedural and object-oriented programming. This conclusion was indirectly drawn from Hughes' work [12], who asserts that modularity is essential to successful programming and that any language aiming to improve productivity should support modularity well [11].

E. Improved testability

Testing in software development, especially in imperative programming, is often neglected since it is difficult to model large, monolithic, stateful programs. Because of the lack of tests, no guarantees about execution can be made every time a part of the program changes.

Even though the same testing methodologies are available to imperative languages, Lisps provide a couple of advantages:

1) Functions in pure Lisps are referentially transparent - given the same input, the output is deterministic (this is the definition of purity). This simplifies writing unit tests and ensures software reliability since unexpected errors and inconsistent return values are mitigated.

Here is an example of an impure and pure function in Python:

```python
# Variable declared globally
interest_rate = 1.1

# Implementation
def apply_interest(user):
    user.money_owed *= interest_rate
    user.save()

# Usage
apply_interest(user)

# Unit test
user = User.objects.create(money_owed=100)
apply_interest(user)
assert_equals(user.money_owed, 110)
```

Listing 1. Impure function in Django, Python's web framework.

```python
# Variable declared globally
interest_rate = 1.1

# Implementation
def calculate_interest(money_owed, interest_rate):
    return money_owed * interest_rate

# Usage
user.money_owed = calculate_interest(
    user.money_owed,
    interest_rate
)

# Unit test
assert_equals(calculate_interest(100, 1.1), 110)
```

Listing 2. Pure function in Python.

In the impure example, the function mutates a global object and relies on a global variable `interest_rate`. Because of this, writing the unit test required setting up a database object, increasing test complexity. If the `interest_rate` changes, the test will fail since it relies on external state, and is impure.

Even though the return values are hard-coded in both unit tests, only the impure implementation's unit test will fail if `interest_rate` is changed since the pure example's test is self-contained. Namely, the pure example explicitly takes the previously declared state as an argument.

Failing unit tests for trivial reasons might cause frustration. Hard to implement unit tests might cause neglect to writing them.

This is however an oversimplified example. It is e.g. necessary to write impure or non-referentially transparent functions when dealing with databases, and the entire program need not be pure. Side effects can be isolated though, and pure functions can still be tested effectively.

2) While "black box" methodologies can be used for checking the inputs and outputs with unit tests, verifying the

models requires consistently simulating the internal state of the functions, which is more difficult with imperative languages. Namely, variables need to be set up in advance and impure internal functions need to be mocked. The functional parts of the program are ignored, which are crucial for the design and implementation, as argued by Howden [13].

If FP principles are followed, verifying models is simple since referentially transparent functions can be written as values, thus modeled more effectively.

This supports test-driven development (TDD), and thus encourages continuous refactoring without fear that the correctness might be violated. For example, "LIFT - The LIsp Framework for Testing" describes a testing suite for Lisps that supports hierarchical and regression tests [14].

Testing is particularly important since writing tests alongside the main program accelerates the development, as functions are debugged while they are being implemented. Buchan et al. report that empirical studies demonstrate increased confidence in the product and, although subjective, a higher quality codebase through TDD, despite skepticism from developers in the beginning [15].

While TDD principles can also be applied to imperative languages, and the same tooling is available, it is often neglected if the program becomes too large to test efficiently.

F. Examples

Run-length encoding (RLE) is a technique used in lossless data compression to reduce file size by utilizing repetition. In RLE, consecutive occurrences of the same data are stored as a single instance of said data along with a count of the consecutive occurrences. For example, $rle(\text{"}aaaBxxxxkk\text{"}) = \text{"}3a1B4x2k\text{"}$.

Below are examples of RLE's implementations using different paradigms.

```python
def rle(string):
    output = ''
    count = 1

    for i in range(1, len(string)):
        if string[i] == string[i-1]:
            count += 1
        else:
            output += str(count) + string[i-1]
            count = 1

    output += str(count) + string[-1]

    return output
```

Listing 3. RLE in Python using imperative paradigms [16].

```haskell
rle = map (head &&& length) . group
```

Listing 4. RLE in Haskell using functional paradigms.

The first listing contains many indices and array access operations. Because of this, it is prone to off-by-one errors. The second listing utilizes higher-order and high-level functions, and is much easier to grasp once you know what each function does.

Furthermore, if the functional implementation has bugs, it will be because of a major logical flaw and not because of a few untested inputs. This is a good thing since we want to know as soon as possible if the program is not correct. The absence of indices and numbers intuitively supports this case.

Whereas in the imperative implementation it is possible a single incorrect index will produce unwanted results only for specific inputs. Guaranteeing this will not happen requires formal step-by-step verification.

Though it is worth noting for this example that the return value of the two listings differ, as the second returns a list of tuples, and not a string.

An important distinction has to be made here: paradigms are not limited to languages. The following shows a functionally equivalent program in Python using functional paradigms:

```python
def rle(s):
    return list(map(lambda g:
        (g[0], len(list(g[1]))), groupby(s)))
```

Listing 5. RLE in Python using functional paradigms.

III. Discussion

Functional programming is not the most common paradigm for early-stage startups. Their most common requirements are rapid development, and release of new features as well as ease of hiring engineers. Since functional programming is not the most popular paradigm, most job-seeking software developers focus on what is in demand, i.e,. imperative programming, thus it is harder to find many good FP engineers fast.

However, a lot of late-stage startups opt in for a complete rewrite if technical dept piles up as a result of rushing, and adding in new features or making changes causes breakage, instability, or takes too much time. This is where we think FP shines. As discussed in the previous section, FP offers many advantages that safeguard the program execution, and allows for safer refactoring. In a lot of cases, this is what the late-stage startups are looking for since downtime or crashes in production directly translate to loss of revenue.

Scala is a type-safe functional language that runs on the Java Virtual Machine (JVM), and focuses on scalability. Its key features include multi-paradigm abilities, so allowing to program imperatively when necessary, and thus enabling teams to slowly implement a paradigm shift. It is often a desirable language for rewrites, as was the case for Twitter. Eriksen reports about the motives behind choosing Scala, as well as the team's experience [17]. Namely, Scala's functional systems design, avoiding mutable state, how using a "research language" turned out in practice. According to Eriksen, the team had no prior experience with Scala or the JVM, but this did not turn out to be a challenge.

OCaml is a statically typed functional language that catches bugs at compile-time. Jane Street Capital, a trading company, has adopted OCaml as its primary development language. In "Caml trading - experiences with functional programming on Wall Street" Minsky and Weeks discuss how OCaml helps them produce readable, correct, efficient, and adaptable programs rapidly, and how it gives them an advantage over competitors using imperative languages [18].

IV. Acknowledgments

We did not discuss object-oriented paradigms (OOP) in this paper due to time constraints, and since the research question is scoped around Lisps and FP. However many observations made in this paper may also apply to OOP in contrast with FP, and future work could explore this further.

Despite a lot of research being FP-centric, finding solid empirical evidence in industrial and enterprise contexts proved to be challenging. Due to a limited availability of published enterprise data, we relied on logical reasoning where empirical validation was unavailable.

While our presented research is FP-centric, our focus is not to dismiss imperative programming and its practicality. FP is not a universal solution, and both paradigms can have different purposes as well as be used simultaneously like in a codebase in Scala. During the evaluation and discussion we tried to maintain an objective view.

That being said, we believe FP is underrepresented outside of research, and its adoption could provide tangible benefits in a general-purpose programming environment.

V. Related work

"The Mechanical Evaluation of Expressions" describes how forms of expression from programming languages can be modeled using Church's λ-notation, a formalism also referenced by McCarthy [19], [20]. Many of Landin's ideas originate from the works of McCarthy, Church, Curry and the authors of ALGOL 60. However, his paper offers a more gradual introduction to expressing computer algorithms through a similar formal notation.

"Functional Program Testing" presents and describes a testing concept in detail that has added benefits over the widely known black-box unit testing and structural testing because of its implementation [13]. In a collection of scientific programs, this method was able to more reliably identify implementation mistakes. It's noteworthy that this method requires a deep understanding of the program which is not necessary for structural testing. Though, as the author states, these methods are complementary and do not rule each other out.

The chapter Functional Programming Paradigm from "Programming Languages: Principles and Paradigms" introduces FP and how it works by explaining its core concepts [3]. The main focus includes the pure paradigm, "real" languages in practice, as well as Lambda calculus. This is a good entry point into the theory of FP because many parallels are drawn to other paradigms, and comparisons with examples are plentiful. Most of LISP's distinguishing characteristics are covered here.

References

[1] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[2] John McCarthy. *History of LISP*, page 173–185. Association for Computing Machinery, New York, NY, USA, 1978.

[3] Maurizio Gabbrielli and Simone Martini. *Functional Programming Paradigm*, pages 335–368. Springer International Publishing, Cham, 2023.

[4] Robert D. Austin. The effects of time pressure on quality in software development: An agency model. *Information Systems Research*, 12(2):195–207, 2001.

[5] Evan Czaplicki. Elm : Concurrent frp for functional guis. 2012.

[6] Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. The prusti project: Formal verification for rust. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods*, pages 88–108, Cham, 2022. Springer International Publishing.

[7] Bill Lewis, Richard M Stallman, and Dan LaLiberte. *GNU Emacs Lisp reference manual*. Free Software Foundation, 1998.

[8] Wikipedia contributors. Erlang (programming language) — Wikipedia, the free encyclopedia, 2025. [Online; accessed 25-June-2025].

[9] Abdullah Khanfor and Ye Yang. An overview of practical impacts of functional programming. In *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, pages 50–54, 2017.

[10] P. Wadler. How to solve the reuse problem? functional programming. In *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*, pages 371–372, 1998.

[11] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 01 1989.

[12] Ismael Figueroa and Romain Robbes. Is functional programming better for modularity? In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU 2015, page 49–52, New York, NY, USA, 2015. Association for Computing Machinery.

[13] W.E. Howden. Functional program testing. *IEEE Transactions on Software Engineering*, SE-6(2):162–169, 1980.

[14] Gary King. Lift—the lisp framework for testing. Technical report, Technical report, University of Massachusetts, 2001.

[15] Jim Buchan, Ling Li, and Stephen G. MacDonell. Causal factors, benefits and challenges of test-driven development: Practitioner perceptions. In *2011 18th Asia-Pacific Software Engineering Conference*, pages 405–413, 2011.

[16] GeeksForGeeks. Run length encoding in python. https://www.geeksforgeeks.org/python/run-length-encoding-python/, 2024.

[17] Marius Eriksen. Scaling scala at twitter. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, New York, NY, USA, 2010. Association for Computing Machinery.

[18] YARON MINSKY and STEPHEN WEEKS. Caml trading – experiences with functional programming on wall street. *Journal of Functional Programming*, 18(4):553–564, 2008.

[19] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 01 1964.

[20] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, NJ, USA, 1985.