UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FERNANDO HENRIQUE DE ALBUQUERQUE ALVES

On the usage of functional programming concepts in JavaScript programs

Recife
2022

FERNANDO HENRIQUE DE ALBUQUERQUE ALVES

On the usage of functional programming concepts in JavaScript programs

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Engenharia de Software e Linguagens de Programação.

Orientador: Prof. Dr. Fernando José Castor de Lima Filho

Coorientadora: Prof[a]. Dr[a]. Fernanda Madeiral Delfim

Recife

2022

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

A474o Alves, Fernando Henrique de Albuquerque
On the usage of functional programming concepts in JavaScript programs / Fernando Henrique de Albuquerque Alves. – 2022.
61 f.: il., fig., tab.

Orientador: Fernando José Castor de Lima Filho.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2022.

Inclui referências.

1. Engenharia de software. 2. Programação funcional. 3. *Javascript*. I. Lima Filho, Fernando José Castor de (orientador). II. Título.

005.1 CDD (23. ed.) UFPE - CCEN 2022-135

Fernando Henrique de Albuquerque Alves

“On the usage of functional programming concepts in JavaScript programs”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Programação.

Aprovado em: 08/07/2022.

BANCA EXAMINADORA

_____

Prof. Dr. Breno Alexandro Ferreira de Miranda
Centro de Informática / UFPE

_____

Prof. Dr. Rodrigo Bonifacio de Almeida
Departamento de Ciência da Computação / UnB

_____

Prof. Dr. Fernando José Castor de Lima Filho
Centro de Informática / UFPE
(**Orientador**)

ACKNOWLEDGEMENTS

First, I would like to thank my supervisors, Fernando Castor and Fernanda Madeiral, for all their availability and attention to detail spent in supervising this work. I would also like to thank Delano Oliveira for acting directly and diligently. This work would not have had the degree of quality that so many reviewers praise without their joint effort.

I would also like to thank my family, especially my wife, Janaina Lima, who had all the necessary patience and resilience and always believed that I would make it, even in the most challenging and doubtful moments throughout the research period.

Finally, I would like to thank my friends and co-workers who have always been interested in helping me with ideas, constructive criticism about the work, and code reviews. Especially Álvaro Basto, Deyvson Lazaro, João Nunes and Walber Rodrigues.

ABSTRACT

Language constructs inspired by functional programming have made their way into most mainstream programming languages. Many researchers and developers consider that these constructs lead to programs that are more concise, reusable, and easier to understand. Notwithstanding, few studies investigate the prevalence of these structures and the implications of using them in mainstream programming languages. This paper quantifies the prevalence of four concepts typically associated with functional programming in JavaScript: recursion, immutability, lazy evaluation, and functions as values. We divide the latter into two groups, higher-order functions and callbacks & promises. We focus on JavaScript programs due to the availability of some of these concepts in the language since its inception, its inspiration from functional programming languages, and its popularity. We mine 91 GitHub repositories (more than 22 million LOC) written mostly in JavaScript (over 50% of the code), measuring the usage of these concepts from both static and temporal perspectives. We also measure the likelihood of bug-fixing commits removing uses of these concepts (which would hint at bug-proneness) and their association with the presence of code comments (which would hint at code that is hard to understand). We find that these concepts are in widespread use (478,605 occurrences, 1 for every 46.65 lines of code, 43.59% of LOC). In addition, the usage of higher-order functions, immutability, and lazy evaluation-related structures has been growing throughout the years for the analyzed projects, while the usage of recursion and callbacks & promises has decreased. We also find statistical evidence that removing these structures, with the exception of the ones associated to immutability, is less common in bug-fixing commits than in other commits. In addition, their presence is not correlated with comment size. Our findings suggest that functional programming concepts are important for developers using a multi-paradigm language such as JavaScript, and their usage does not make programs harder to understand.

Keywords: functional programming; javascript; mining software repositories.

RESUMO

Constructos de linguagem de programação inspirados pelo paradigma funcional chegaram à maioria das linguagens convencionais. Muitos pesquisadores e desenvolvedores consideram que esses constructos tornam programas mais concisos, reutilizáveis e mais fáceis de entender. No entanto, poucos estudos investigam a prevalência dessas estruturas e as implicações de usá-las em linguagens de programação convencionais. Este trabalho quantifica a prevalência de quatro conceitos tipicamente associados à programação funcional em JavaScript: recursão, imutabilidade, avaliação preguiçosa e funções como valores. Dividimos o último em dois grupos, funções de alta ordem e callbacks & promises. Focamos em programas JavaScript devido à disponibilidade de alguns desses conceitos na linguagem desde seu início, sua inspiração em linguagens de programação funcionais e a popularidade da linguagem. Mineramos 91 repositórios GitHub (mais de 22 milhões de linhas de código) escritos principalmente em JavaScript (mais de 50% do código), medindo o uso desses conceitos de perspectivas estáticas e temporais. Também medimos a probabilidade de commits de correção de bugs removendo usos desses conceitos (o que sugeriria propensão a bugs) e sua associação com a presença de comentários de código mais longos (o que sugeriria um código difícil de entender). Descobrimos que esses conceitos são de uso generalizado (478.605 ocorrências, 1 para cada 46,65 linhas de código, 43,59% de linhas de código). Além disso, o uso de funções de alta ordem, imutabilidade e estruturas relacionadas à avaliação preguiçosa vêm crescendo ao longo dos anos para os projetos analisados, enquanto o uso de recursão e callbacks & promises diminuiu. Também encontramos evidências estatísticas de que a remoção dessas estruturas, com exceção das associadas à imutabilidade, é menos comum em commits de correção de bugs do que em outros commits. Além disso, a presença dessas estruturas não está correlacionada com o tamanho do comentário associado. Nossas descobertas sugerem que os conceitos de programação funcional são importantes para desenvolvedores que usam uma linguagem multiparadigma, como JavaScript, e seu uso não torna os programas mais difíceis de entender.

Palavras-chave: programação funcional; javascript; mineração de repositórios de software.

LIST OF FIGURES

LIST OF TABLES

LISTINGS

CONTENTS

1 INTRODUCTION

The rapid growth of computing power has made it possible to apply computing to complicated tasks and increased the demand for software engineers (WIRTH, 2008). Modern programs have become complex, reaching millions of lines of code written by large program teams over many years. In this context, a language represents an abstraction whose objects and constructs reflect a problem. In a high-level language, for example, a developer deals with numbers, indexed arrays, data types, conditional and repetitive statements instead of bits and bytes, addressed words, jumps, and condition code (ROY et al., 2009).

However, a programming language is not designed in a vacuum but for solving specific problems, and each problem has a paradigm that is best for it. A paradigm is an approach to programming a computer based on a mathematical theory or coherent principles, and each one supports a set of concepts that makes it the best for a particular problem (ROY et al., 2009). In this work, we focus on the functional programming paradigm.

Functional programming is a paradigm where programs are built by defining, applying, and composing functions (SCOTT, 2016). Many researchers (BACKUS, 1978; HUDAK, 1989; HUGHES, 1989) consider that functional programming concepts lead to more concise, reusable, and easier-to-understand programs. Elm, Scheme, Clojure, Erlang, Haskell, and F# are examples of functional programming languages. Multi-paradigm languages, such as JavaScript and Python, also include structures popularized by functional languages, like higher-order functions. JavaScript, for example, takes inspiration from functional languages such as Lisp and Scheme (SATERNOS, 2014). By mixing different paradigms, these languages allow one to solve problems more quickly or efficiently than one could do with a single paradigm.

1.1 PROBLEM STATEMENT

The extent to which developers use functional programming concepts in multi-paradigm programming languages is unknown. Language structures inspired by functional programming, e.g., function literals, have made their way into mainstream programming languages like Java and C++. However, few studies (GALLABA et al., 2015; MAZINANIAN et al., 2017; XU et al., 2020; FIGUEROA et al., 2021) investigate the usage of these structures, and when they do, they evaluate few structures, few projects of multiple areas or are limited only to purely functional languages. In particular, previous works have not investigated whether the use of structures inspired by functional programming is connected to improvements or decreases in code quality. In addition, they do not analyze code adjacent to these structures, such as comments, or how the usage of these structures changes over time. Therefore, in this work, we tackle the problem of understanding how developers use struc-

tures inspired by functional programming in a mainstream multi-paradigm programming language.

Studying how developers use these structures is important for several reasons. Investigating their prevalence can reveal how successful their adoption by the community has been, which can influence the design of future programming languages. Studying the evolution in their usage provides subsidies for the development of new tools to automate tasks such as refactoring, code analysis, and test-case generation. Analyzing their impact on code quality can help technical managers in the decision to adopt or avoid these structures and languages that use them. It can also influence developers and maintainers to conduct refactoring and re-engineering efforts.

1.2 GOALS AND METHODS

The goal of this work is to quantify the prevalence and significance of four concepts typically associated with functional programming in JavaScript: recursion, immutability, lazy evaluation, and functions as values (higher-order functions and callbacks & promises). To do so, we measure the occurrence of these concepts and their structures from static and temporal perspectives. We also measure the likelihood of bug-fixing commits removing their uses, which would hint at bug-proneness, and their association with the presence of code comments, which would indicate code that is hard to understand (HUNT; THOMAS, 1999; BECK, 2004; FOWLER et al., 2019). We focus on JavaScript programs due to the availability of some of these concepts and structures in the language since its inception, its inspiration from functional programming languages (SATERNOS, 2014), and its popularity.

Based on the overall goal, we tackle the following research questions (RQs):

RQ1. How often are functional programming structures used in real software?

RQ2. How has the use of functional programming structures evolved over the years?

RQ3. Are uses of functional programming structures removed more often in bug-fixing commits?

RQ4. Is code that employs functional programming structures associated with longer comments?

Following the guidelines of Easterbrook et al. (2008), these questions are divided into two groups, frequency questions (RQ1 and RQ2) and relationship questions (RQ3 and RQ4). The idea behind frequency questions is to understand the patterns of occurrence of a phenomenon. If no base-rate questions are asked, there is no basis for saying whether a particular situation is typical or unusual. Relationship questions, on the other hand, are

related to the interest in knowing the relationship between two phenomena, specifically whether one is related to the occurrence of the other.

To answer the research questions, we first select repositories from GitHub using established criteria. After that, we give the repositories as input to our tool, psMiner. It starts the analysis process by ordering the repositories by Lines of Code (LOC) and selecting the ones with the most LOC. Then, psMiner begins its main task, which is to mine the repositories for the usage of the functional programming concepts we investigate in this study. It builds an abstract syntax tree from each repository's files and recognizes elements from the source code of these systems as functional programming structures. We double-check the precision of the results produced by psMiner. First, we draw a random sample of code snippets and manually check whether the classification the tool performs is correct. Second, to ensure that every structure is represented since some appear only rarely, we randomly select five instances of each structure we have considered across all the projects and manually check the results. Finally, we create scripts that can process the data captured by psMiner and infer knowledge from it. These results help us answer our research questions.

1.3 CONTRIBUTIONS

The main contribution of this work is the study of the usage of functional programming concepts in JavaScript projects. We found out that 43.59% of the ∼22M analyzed lines of code are related to functional programming structures, and it is possible to find one use of such structures for every 46.65 lines of code. Furthermore, the usage of higher-order functions, immutability, and lazy evaluation-related structures has been growing throughout the years for the analyzed projects, while the usage of recursion and callbacks & promises has decreased. In particular, the usage of immutability-related structures has more than tripled throughout their evolution. Moreover, we find out that the concepts tend to be removed less often in bug-fixing commits than in non-bug-fixing commits, excepting immutability-related structures, and that there is no correlation between comment size and source code including them. These findings highlight the importance of functional programming structures to developers even in an inherently imperative language and suggest that the usage of these structures is less error-prone than not using them and does not make the source code difficult to understand.

As briefly mentioned, we developed psMiner and created some Python scripts to mine and measure the usage of functional programming concepts, respectively. Although they are not frameworks and were not designed with extensibility in mind, it is possible to customize some of their functionalities (we detail how to do that in Section 5.3). We also extracted and organized the data in publicly available data sets (Alves et al., 2022a).

Finally, the paper entitled "On the Bug-proneness of Structures Inspired by Functional Programming in JavaScript Projects" (ALVES et al., 2022b) reports the study presented in this dissertation and is currently under review.

1.4 TEXT ORGANIZATION

The remainder of this document is organized as follows. Chapter 2 describes the foundations of this work, the historical origins of functional programming, examples of languages and concepts commonly related to this paradigm, and an overview of what mining software repositories is and how it can be applied. It also presents the works related to the usage of functional programming structures, the usage of other structures in terms of understandability, and JavaScript data mining. Chapter 3 presents the design of our study, including the research questions, the selection of projects, and how we mined the concepts specifically in JavaScript code and automated this process in psMiner. Chapter 4 presents the results for each research question and discusses the threats to the validity of this study. Chapter 5 presents psMiner, explaining how it works and how it can be used and customized. Finally, Chapter 6 concludes this work with our main findings, their implications, and avenues for future works.

2 FOUNDATIONS

In this chapter, we present the foundations of this work. It is organized into three sections. Section 2.1 is composed of the definition, origin, evolution, and use of concepts that inspired functional programming languages, as well as the definition of multi-paradigm languages and how languages such as JavaScript take advantage of the functional paradigm. We briefly define mining software repositories in Section 2.2 and present where and how it is used in this work. Finally, we describe the related works in Section 2.3.

2.1 FUNCTIONAL PROGRAMMING

Functional programming is a paradigm where programs are built by defining, applying, and composing functions (SCOTT, 2016). Unlike imperative programming, which has roots in the Turing machine, functional programming has roots in the Lambda Calculus (CHURCH, 1936). Many researchers (BACKUS, 1978; HUDAK, 1989; HUGHES, 1989) consider that functional programming concepts are more concise, reusable, and easier to understand. Although academic research was the initial focus of this paradigm, functional programming has been popularized in the industry in the last few years. Elm, Scheme, Clojure, Erlang, and Haskell are examples of functional programming languages.

A fundamental characteristic of a functional programming language is that functions can be assigned to variables, passed as arguments, and returned from functions. Also, like mathematical functions, the outputs of a function depend only on its inputs because the functions have no internal state (no side effects) (SCOTT, 2016). It eliminates a significant source of bugs, making the order of execution irrelevant. Variables never change once given a value because functional programs do not have assignment statements. One can evaluate expressions anytime and replace variables by their values and vice versa (referential transparency) (HUGHES, 1989). Furthermore, anonymous functions can be defined.

Functional languages typically include features not generally available in imperative languages, such as lazy evaluation, partial function application, and absence of side effects. A functional language is considered pure when it treats every computation as the evaluation of a mathematical expression, i.e., not allowing state changes and mutable data. Likewise, a function that satisfies this property is called a pure function.

Some widely popular languages, such as Python and Swift, include structures popularized by functional languages since their first versions. In addition, many mainstream imperative languages, such as C#, Java, and C++, have introduced structures and libraries to support a programming style heavily inspired by functional languages. JavaScript, for example, takes inspiration from functional languages like Lisp and Scheme. The Lisp

community itself considers ECMAScript[1], which JavaScript is based on, as a dialect of Lisp (SATERNOS, 2014). By mixing different paradigms, these languages allow one to solve problems more easily or efficiently than one could do with a single paradigm.

2.1.1 Functional programming concepts

This section presents concepts that are considered elements of the functional programming paradigm by multiple authors (WATT, 2004; SCOTT, 2016; KEREKI, 2020). However, they do not represent everything possible in the languages of this paradigm. The concepts presented in this section include those investigated in our study (see Section 3.3) and others that are important to the functional programming paradigm.

Pure functions: Functions are pure when they do not affect any state outside their scope. In other words, they do not cause side effects. An important concept that pure functions follow is that an expression must always evaluate the same result in any context[2]. It means that the output of a function application will always be the same given an input.

Algebraic data types: These are structured types formed by composing other types. Programming languages support various composite values in terms of structuring concepts, such as Cartesian products (tuples, records), mappings (arrays), disjoint unions (algebraic types, discriminated records, objects), and recursive types (lists, trees) (WATT, 2004). In Haskell, for example, it is possible to understand algebraic data types in terms of disjoint unions, as presented in Listing 2.1.

```
data Number = Exact Int | Inexact Float
```

Listing 2.1 – Disjoint Unions in Haskell

In Elm, these custom types are called “union types” and have a very similar syntax to Haskell, as presented in Listing 2.2.

```
type User = Student | Teacher
```

Listing 2.2 – Union Types in Elm

Pattern matching: In the context of programming languages, pattern matching is checking one or more inputs against a pre-defined pattern and seeing if they match. It consists of specifying patterns to which some data should conform, then checking to see if it does, and deconstructing the data according to those patterns (LIPOVACA, 2011). Pattern matching, especially for strings, appears in many programming languages. ML, for example, is known for extending pattern matching to the full range of constructed values - including tuples, lists, records, and variants - and integrating it with static typing and type inference (SCOTT, 2016).

[1] <https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>
[2] <https://elmprogramming.com/pure-functions.html>

Listing 2.3 is an example of a factorial function using pattern matching in Haskell from (LIPOVACA, 2011). It depicts a function recursively, as it is usually defined in mathematics. It starts by expressing that the factorial of 0 is 1. Then it displays that the factorial of any positive integer is that integer multiplied by the factorial of its predecessor (LIPOVACA, 2011).

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Listing 2.3 – Factorial function in Haskell using pattern matching

Recursion: A function is recursive if it calls itself directly or indirectly (HINSEN, 2009). Although this is not specific to functional programming languages, recursion is an important concept because, in the absence of side effects, it provides the only general means of performing repetition (SCOTT, 2016).

Immutability: Immutable data is a standard feature of functional programming, and it is used to prevent changes (mutation) in data structures. In other words, it means the value of an expression depends only on the referencing environment in which it is evaluated and not on the time at which the evaluation occurs. If an expression yields a specific value at one point in time, it is guaranteed to yield the same value at any point in time (referential transparency) (SCOTT, 2016). Immutability, when applied to objects, for example, creates objects that cannot be modified after they have been created (BRADY, 2021).

Functions as values: In functional languages, functions are values, which means that they can be passed as parameters, returned from subroutines, or assigned to variables (SCOTT, 2016). A function that takes another function as an argument or returns a function as its result is called a higher-order function (SCOTT, 2016). This concept is derived from mathematics and can be intuitively considered a function of functions (XU et al., 2020). An example of a higher-order function is `map`, which takes a list l and a function f as its arguments and applies f to each element of l. Listing 2.4 is an example of a higher-order function in Elm. The `map` function takes function *String.length* and list *names* as its arguments and applies the function to each element of the list returning a list of their lengths.

```
names = [ "Fernando", "foo", "bar" ]
lengths = List.map String.length names
```

Listing 2.4 – The map function in Elm

Lambdas: Lambda expressions are a core feature of functional programming. The term Lambda function or Lambda expression is derived from Lambda Calculus[3]. In most of

[3] <https://csmith111.gitbooks.io/functional-reactive-programming-with-elm/content/section2/LambdaFunctions.html>

the functional programming languages, lambda expressions are declared in the form of anonymous functions which can be passed to or returned by other functions similar to Lambda Calculus (MAZINANIAN et al., 2017). In Elm, they start with a "\" followed by a list of parameters, the "`->`", and the function body. Listing 2.5 is an example of lambda expression in Elm. In this work, we do not map lambda functions individually, but we look for thunks that are structured in the form of lambdas and related to lazy evaluation.

```
\n -> n + 1
```

Listing 2.5 – Lambda expression in Elm

Lazy evaluation: This concept is related to the idea that an expression is only evaluated when its value is necessary, e.g., because it is used in an I/O operation. Lazy evaluation is a counterpoint to eager evaluation, where an operator is applied as soon as its operands are known (WATT, 2004). With lazy evaluation, it is possible, for example, to avoid unnecessary calculations and create infinite lists. Languages like Haskell make use of laziness in its core (LIPOVACA, 2011).

Currying: It is the process of transforming a function with multiple arguments into a sequence of functions. Each of them receives one argument from the original function, from left to right, returns a function that receives the other argument, and so on. Upon being called with an argument, each function produces the next one in the sequence, and the last one does the actual calculations (KEREKI, 2020).

In Elm, every function only takes one parameter[4]. So all the functions that accept several parameters are automatically curried by the language. Consider that each space in a function call is a function application in Elm. For example, the function `max` 1 3 is partitioned into two functions, one that receives 1 as an argument and another that receives the other (3) and compares it to 1 to find the maximum value between them. This process is called partial function application. It allows passing not all arguments to a function, applying the given arguments to it, and returning a new one that can be applied to the remaining arguments later. Consider the Listing 2.6. It is possible to define a new function called `addTwo`. It happens because the partial application of a function returns another function.

```
add a b = a + b
addTwo = add 2
```

Listing 2.6 – Partial function application in Elm

[4] <https://learnyouanelm.github.io/pages/06-higher-order-functions.html>

2.1.2 Historical origins of functional programming

In the 1930s, mathematicians like Alan Turing, Alonzo Church, Stephen Kleene, and Emil Post worked in imperative and functional models, developing several formalizations of the notion of an algorithm. Even working largely independently, these formalizations were powerful enough: anything that could be computed in one could be computed in the others (SCOTT, 2016). From all of that work, Turing created the Turing Machine, which computed imperatively, changing values in the cells of its tapes exactly as an imperative program changes the values of variables. In contrast, Church created a model of computation called Lambda Calculus (CHURCH, 1936), based on the notion of parameterized expressions. This model strongly inspired functional programming because computation is done by substituting parameters in expressions, and in a functional program, arguments are passed to functions (SCOTT, 2016).

According to Turner (2012), in 1958, Lisp began life as a project led by John McCarthy at MIT. McCarthy himself wrote the first published account of the language and theory of Lisp in 1960 (MCCARTHY, 1960). Also, in the early 60s, Peter Landin wrote a series of seminal papers on the relationship between programming languages and lambda calculus. In 1966, he described an idealized language family, ISWIM (If you See What I Mean). It was the first appearance of algebraic type definitions used to define structures. Around 1967, PAL, the Pedagogic Algorithmic Language, was created as a vehicle for teaching programming linguistics - it had run-time type checking. Between 1973 and 1975, John Darlington made NPL (New Programming Language) with Rod Burstall. The language was first order, strongly typed, purely functional, and had a call-by-value semantics. It evolved into HOPE (BURSTALL et al., 1980), a higher-order, strongly typed language with explicit types, polymorphic type variables, and purely functional. Also, in Edinburgh, from 1973 to 78, the programming language ML emerged as the meta-language of Edinburgh LCF (GORDON et al., 1979). The language was higher-order, call-by-value, and allowed assignment and mutable data. Standard ML (MILNER et al., 1997), which appeared later, in 1986, is a mix of HOPE and ML but is not pure - it has references and exceptions.

Also, according to Turner (2012), between 1983 and 1986, Miranda emerged as a lazy, purely functional language, polymorphic with type inference, list comprehensions, and optional type specifications. In 1984, an independent group of researchers at Chalmers University implemented Lazy ML. It was a pure, lazy version of ML, used by Lennart Augustsson and Thomas Johnsson as both source and implementation language for their work on compiled graph reduction.

In 1987, a committee of researchers was organized and started work on the Haskell Report at the FPCA'87 Conference. In 1990, they published the first Haskell Report, which describes the motivation for creating the language. In 1992, occurs the creation of GHC (Glasgow Haskell Compiler), an open-source compiler for the language. In 1994,

the Haskell web page[5] was created, the primary information source about the language. Finally, in 2010, another version of Haskell was released, and it stands as the current for most Haskell developers[6].

Miranda strongly influenced Haskell's design, so there are many similarities between the two languages, like purity, higher-order, laziness, static typing, pattern matching, and list comprehensions (HUDAK et al., 2007; TURNER, 2012). In Haskell, functions are also curried but, like many other languages, a function with two arguments may be represented as a function of one argument that returns a function of one argument, so Haskell supports both curried and uncurried definitions (HUDAK et al., 2007).

In 2005, the first release of F# became available. Originally developed at Microsoft Research, Cambridge, it is a strongly typed functional-first, multi-paradigm language. It descends from the ML language and was heavily inspired by OCaml (FANCHER, 2014). It has anonymous functions, immutable variables, lazy evaluation support, higher-order functions, currying, and pattern matching.

In 2007, Rich Hickey released Clojure[7]. It is predominantly a functional programming language and features a rich set of immutable, persistent data structures, functions as first-class objects, and emphasizes recursive iteration instead of side-effect-based looping. Moreover, Clojure is a dialect of Lisp and shares the code-as-data philosophy and an influential macro system.

In 2012, José Valim released the first version of Elixir, a concurrent, functional programming language designed to implement distributed, fault-tolerant systems built on top of Erlang's Virtual Machine. It is structured in functions and modules (groups of functions), supports pattern matching and higher-order functions, and all the data types are immutable[8].

Also, in 2012, Evan Czaplicki designed Elm as his thesis (CZAPLICKI, 2012). Elm includes traditional if-expressions, let-expressions, and case-expressions for pattern matching. It supports higher-order and anonymous functions, partial application of curried functions, immutable values, stateless functions, and static typing with type inference[9]. Table 1 summarizes the important facts about the origin and evolution of functional programming.

Besides those already mentioned, many other languages use concepts typically associated with functional programming. Swift, for example, has higher-order functions, recursion, algebraic data types, and pattern matching, among other features. Java adopted Lambda Expressions (also called closures) since its 8th version, adding support to anony-

[5] <https://www.haskell.org/>
[6] <https://serokell.io/blog/haskell-history>
[7] <https://clojure.org/>
[8] <https://serokell.io/blog/introduction-to-elixir>
[9] <https://elmprogramming.com/>

Table 1 – Timeline of important facts about functional programming.

Period	Fact
1930s	Alan Turing, Alonzo Church, Stephen Kleene, and Emil Post developed several formalizations of the notion of an algorithm.
1958	Lisp began life as a project led by John McCarthy at MIT.
1966	Peter Landin described an idealized language family, ISWIM (If you See What I Mean).
1967	PAL, the Pedagogic Algorithmic Language, was created.
1973—1975	John Darlington made NPL (New Programming Language) with Rod Burstall.
1973—1978	ML emerged as the meta-language of Edinburgh LCF.
1984	An independent group of researchers at Chalmers University implemented Lazy ML.
1985	First release of Miranda.
1986	Standard ML appears as a mix of HOPE and ML.
1987	A committee of researchers was organized and started work on the Haskell Report at the FPCA'87 Conference.
1990	The first Haskell Report, which describes the motivation for creating the language, was published.
1992	GHC (Glasgow Haskell Compiler), an open-source compiler for the language, is created.
2005	First release of F#.
2007	Rich Hickey released Clojure.
2010	The last formal specification of Haskell was made.
2012	In 2012, José Valim released the first version of Elixir.
2012	Evan Czaplicki designed Elm as his thesis.

Source: the author (2022)

mous functions. Python functions are first-class and can be higher-order or anonymous too. Scala includes currying, immutability, and pattern matching.

2.1.3 Multi-paradigm programming languages

A programming language is considered to be multi-paradigm when it includes concepts from multiple paradigms, like the imperative and functional ones. JavaScript and Python are examples of multi-paradigm programming languages.

The extent to which developers use functional programming concepts to develop in multi-paradigm programming languages is unknown. Perhaps developers are not even aware they are using these concepts when programming with these languages. Moreover, the use of functional programming concepts in a language that is not purely functional can be confusing. Consider callbacks, for example, which are functions passed as arguments to other functions and executed after them. They induce a non-linear control flow and can be deferred to execute asynchronously, declared anonymously, and may be nested to arbitrary levels, which can be challenging to understand and maintain (GALLABA et al., 2015). Studying how developers use structures inspired by functional programming in a mainstream multi-paradigm programming language can show us how these structures are used in practice and provide subsidies for researchers and tool builders to propose improvements.

In this work, we investigate the use of concepts inspired by the functional programming paradigm in JavaScript, a mainstream, imperative, multi-paradigm language. More specifically, we investigate the prevalence of the usage of functional programming con-

cepts in software projects, their evolution, bug-proneness, and understandability (using association to the presence of code comments as a proxy).

2.2 MINING SOFTWARE REPOSITORIES

Data mining is a set of techniques for uncovering patterns and other valuable information from large data sets[10]. According to Siddiqui e Ahmad (2018), mining software repositories focuses on extracting and analyzing heterogeneous data, like bugs, issues, and source code available in software repositories. Usually, this is done to uncover interesting, helpful, and actionable information about software systems and projects. These data sets can be, for example, code snippets available on sites like Stack Overflow or software repositories from GitHub. Historical and valuable information stored in software repositories provides an excellent opportunity to acquire knowledge and help monitor complex projects and products without interfering with development activities and deadlines. Mining software repositories is so related to software engineering nowadays that several conferences are dedicated to the topic, such as MSR[11].

In this work, we mine 91 open-source repositories in JavaScript and collect data about the usage of 22 functional programming structures related to the concepts of that paradigm.

2.3 RELATED WORKS

We organize related work in terms of three main lines: studies on the usage of functional programming (Section 2.3.1), understandability of structures other than functional programming ones (Section 2.3.2), and mining studies targeting JavaScript projects (Section 2.3.3).

2.3.1 Usage of functional programming structures

We found in the literature studies investigating the usage of specific functional programming structures. For example, Gallaba et al. (2015) investigated the usage of callback in a corpus of 138 JavaScript programs. They found out that, on average, every 10th function definition takes a callback as an argument. Xu et al. (2020) analyzed the use of high-order functions in Scala programs. They collected 8,285 higher-order functions from 35 Scala projects and found out that 6.84% of functions are defined as higher-order functions on average. Figueroa et al. (2021) analyzed the use of monads as a dependency

[10] <https://www.ibm.com/cloud/learn/data-mining>
[11] <https://www.msrconf.org/>

Table 2 – Related works.

Work	Functional programming concepts	Programming language	Investigated factors
Gallaba et al. (2015)	Functions as value (callbacks)	JavaScript	usage frequency
Xu et al. (2020)	Functions as value (high-order functions)	Scala	usage frequency
Figueroa et al. (2021)	Monads	Haskell	usage frequency
Mazinanian et al. (2017)	Lambda Expressions	Java	usage frequency
This work	Recursion, lazy evaluation, functions as value, immutability	JavaScript	usage frequency and evolution, bug-proneness, and association with code comments

Source: the author (2022)

in 85,135 packages in the Haskell language. They found that 32% of the packages depend on the packages that implement monads. Mazinanian et al. (2017) analyzed the usage of lambda expressions in 241 open-source Java projects. The authors found out that the ratio of lambdas introduced per added line of code increased by 54% between 2015 and 2016 and also discovered that developers adopt lambdas for reasons such as making code more concise and avoiding duplication. These are the closest related work to ours. Table 2 summarizes these similarities. We investigated more than one functional programming concept in our work, differently from those studies that focused on one specific concept.

Following along different lines, Lubin e Chasins (2021) studied how programmers write code in several statically-typed functional programming languages, including Haskell, Elm, F#, and others. The authors conducted a grounded theory analysis of 30 programming sessions, combined with 15 semi-structured interviews, and produced a theory of how programmers write code in these languages. They then validated some of the elements of that theory in a controlled experiment and found out, for example, that programmers in these languages tend to code in a cycle of writing a bit of code and running the compiler, even when it is clear that compilation will fail. Furthermore, pattern matching tended to incur a reduced workload compared to combinators.

Kamps et al. (2020) studied structural degradation in Haskell programs by monitoring three static metrics related to size, cohesion, and coupling. The authors leveraged the Gini coefficient to measure structural inequality. They found out that post-release defects correlate significantly with the degree of inequality between the size of the modules in three mature Haskell systems.

2.3.2 Understandability of structures other than functional programming ones

One related work that inspired us, particularly for the research questions, is the work by Gopstein et al. (2018). They used a corpus of fourteen C and C++ projects measuring the prevalence and significance of 15 atoms of confusion, which are small code patterns,

such as conditional operator, that can cause programmers' misunderstandings. Among the research questions of their work are questions that seek to find the frequency of use of atoms, whether the age of projects influences the number of atoms, whether they are removed more frequently in bug-fix commits, and if atoms are commented out more frequently than any other type of structure. All these questions inspired our research questions.

Despite the inspiration, we gather and interpret the data differently than Gopstein et al. (2018). First, how we choose the repositories differs mainly because of the nature of each work. Gopstein et al. (2018) obtained the list of repositories from the US DOD FOSS GRAS (United States Department of Defense Free and Open Source Software Generally Recognized as Safe) and The IDA Open Source Migration Guidelines from the European Commission. In our case, we selected projects with SeArt[12], using the criteria described in Chapter 3. Regarding the research question concerning bug fixes, only one of the fourteen repositories was chosen in their work. They analyzed the repository commit by commit to determine whether it was a bug fix. In our case, we search all repositories as described in Chapter 3. Another example of a different approach is the question related to comments associated with code. Gopstein et al. (2018) sought to understand only the relationship between the existence of comments associated or not with atoms of confusion. In our case, we seek to understand the length of comments rather than simply looking for their presence.

2.3.3 Mining JavaScript projects

We found some studies that mined JavaScript repositories. Hanam et al. (2016) mined 105K commits from 134 server-side JavaScript projects aiming to discover bug patterns. In a study by Campos et al. (2019), they mined code snippets in JavaScript on Stack Overflow to analyze them using ESLinter, a JavaScript linter. Furthermore, they investigated the use of those code snippets in GitHub projects. Saboury et al. (2017a) investigated code smells in 537 releases of five popular JavaScript applications aiming to understand how they impact the fault-proneness of applications. They detected 12 types of code smells (e.g., nested callbacks and variable re-assign) and found out that, on average, files without code smells have hazard rates 65% lower than files with code smells.

Richards et al. (2011) conducted a large-scale study of the use of the `eval` function in JavaScript-based web applications. They recorded the behavior of 337MB of strings given as arguments to 550,358 calls to `eval` exercised in over 10,000 websites and observed that, at the time, between 50% and 82% of the most popular websites used `eval`. They also confirmed, in that context, that `eval` usage is pervasive and not necessarily something problematic.

[12] <https://seart-ghs.si.usi.ch/>

In our work, we mine 91 GitHub repositories (more than 22 million LOC) written mostly in JavaScript (over 50% of the code).

3 STUDY DESIGN

The goal of this study is to quantify the prevalence and significance of recursion, immutability, lazy evaluation, and functions as values in JavaScript programs. To do so, we measure the occurrence of these concepts and their structures from static and temporal perspectives, as well as the likelihood of bug-fixing commits removing their uses and their association with the presence of code comments. In this chapter, we present our research questions (Section 3.1), how we select projects (Section 3.2), how we mine functional programming concepts in JavaScript programs (Section 3.3), and our methods to answer the research questions (Section 3.4).

3.1 RESEARCH QUESTIONS

To understand the use of concepts inspired by functional programming in JavaScript programs, we formulated research questions that would indicate whether the phenomenon in question is a commonplace situation or not (EASTERBROOK et al., 2008). In addition, we are also looking to answer questions relating these structures' use to associated bug-fix rates and comments. Therefore, we focus on the following questions:

RQ1. How often are functional programming structures used in real software?

RQ2. How has the use of functional programming structures evolved over the years?

RQ3. Are uses of functional programming structures removed more often in bug-fixing commits?

RQ4. Is code that employs functional programming structures associated to longer comments?

RQ1 aims to gauge the pervasiveness of functional programming structures in real-world JavaScript software. RQ2 provides a temporal perspective, measuring the evolution in using these structures in the studied projects. For RQ3, we are interested in assessing bug-proneness. Although it is difficult to identify the causes, we can use the code that changes when bugs are fixed as a proxy (SLIWERSKI et al., 2005). Therefore, the rationale for RQ3 is that if bug-fixing commits are more likely to remove instances of functional programming structures than non-bug-fixing commits, this may indicate that these structures are bug-prone. Finally, the rationale behind RQ4 is that, since developers write comments to help them better understand the functioning and the intent of code snippets, we expect code with longer comments to be more troublesome to understand than code with shorter comments, as reported in previous work (BUSE; WEIMER, 2008; STEIDL et al., 2013; AMAN et al., 2015). With this question, we seek to understand whether there is an association between comment length and functional programming structures. We provide more details about these questions in Section 3.4.

3.2 PROJECT SELECTION

We first select the repositories to be analyzed to answer our research questions. We aim to select a representative sample of mature repositories written mainly in JavaScript. We also want our sample to follow good GitHub data-mining practices. More specifically, as mentioned by Hinsen (2009), the selected projects should be active, i.e., with at least one commit in the last six months, at least 1,000 commits, and 1,000 issues overall, not personal or archived projects, and created before six months ago. We use GitHub because of its popularity among developers, documented ways of accessing its content through APIs, and the possibility of accessing open-source software (OSS) from diverse domains.

Unfortunately, GitHub does not offer ways of directly obtaining a list of repositories using the earlier criteria. It provides channels to search for repositories through its search API[1] and using the advanced search[2] but not without having to develop a software system to group the returned data. So, after testing some methods (KRISHNA et al., 2018) and tools (e.g., Reaper (MUNAIAH et al., 2017)), we decided to use a tool called SeArt [3] (DABIC et al., 2021), because it allows us to get a list of repositories with the mentioned criteria.

We use the following settings to search repositories using SeArt: `Language` (JavaScript), `Number of commits` (1,000, at minimum), `Number of contributors` (2, at minimum, to avoid personal repositories), `Number of issues` (1,000, at minimum), `Created at` (before 2021-01-01, six months before collection date), `Last commit at` (after 2021-01-01, six months after collection date) and `Exclude forks` (yes). This search returns 357 repositories. We then remove the archived ones, not found (git cloning returned a `not found` error), and the ones whose GitHub topics were related to documentation or guides. After this filtering step, we keep 338 repositories.

With these results, we run a tool called `cloc`[4] to obtain the Count of Lines of Code (CLOC) of each repository because we use this information in some research questions and also to prioritize repository processing. We ignore some folders (`node_modules`, `coverage`, `build`, `bin`, `stories`, `dist` and `3rdParty`) trying to avoid code that is usually related to the build process, third-party libraries, or auto-generated code. Despite searching only for JavaScript projects, we consider code written in TypeScript. According to its website[5], TypeScript is just *"JavaScript with syntax for types"*. In other words, we accept all extensions related to JS and TS as options for `cloc`. Executing `cloc` resulted in a count of 35,396,336 lines of code. Finally, we order the repository list by CLOC and select the 100 projects with the most LOC. Since we could not parse nine of them, our final list has 91 projects amounting to 22,326,070 LOC. Table 3 summarizes some statistics of the

[1] <https://docs.github.com/en/rest/reference/search>
[2] <https://github.com/search/advanced>
[3] <https://seart-ghs.si.usi.ch/>
[4] <https://github.com/kentcdodds/cloc>
[5] <https://www.typescriptlang.org/>

Table 3 – Selected projects.

	Min	25%	50%	75%	Max
LOC	101,952	116,077	173,106	284,617	1,857,932
# Stargazers	17	380	2,982	17,840.5	171,203
# Contributors	13	76	171	346	1,504
# Commits	1,765	5,924	9,768	17,849	77,667
# Issues	1,007	1,878	2,879	6,939	21,538
# Pull requests	1,002	1,775.5	2,971	5,089	36,763

Source: the author (2022)

selected projects. It shows that our sample comprises mature repositories with extensive histories in the number of commits and diverse in several aspects such as popularity (i.e., number of stars) and number of contributors.

3.3 MINING USAGES OF FUNCTIONAL PROGRAMMING CONCEPTS IN JAVASCRIPT PROJECTS

When mining for the usage of functional programming concepts, we search for specific blocks of code that can represent those concepts described in Section 2.1.1. To process them, we developed a tool, named psMiner (see Chapter 5), that can build an AST (Abstract Syntax Tree) from the files of each repository and recognize elements from the source code of these systems as functional programming structures. Our tool was developed by extending the TypeScript Compiler API because it allows us to infer types in a way that other parsers we have tested (e.g., Esprima[6]) were not able to, adding more reliability to the results.

We also verify the precision of the results produced by our tool in two ways. First, we draw a random sample of 384 code snippets and manually check whether the classification the tool performs is correct. To have a confidence level of 95% that the real value is within ±5% of the measured value, 384 or more samples are needed. To reduce the impact of disproportionately large projects, we randomly select a project and then randomly pick one code snippet including a potential functional programming structure from that project. We repeat this procedure 384 times. In this step, we did not find any misclassifications. Second, to ensure that every structure is represented, since some appear only rarely, e.g., the `flatMap` function, we randomly select five instances of each structure we have considered, across all the projects, totaling 105 code snippets. Also, we did not find any misclassification in this step. Table 4 presents the complete list of structures.

In the remainder of this section, we explain the JavaScript structures we selected to represent the functional programming concepts described in Section 2.1.1.

[6] <https://esprima.org/>

3.3.1 Recursion

We looked for function declarations whose names are used as call expressions once or more inside their bodies. We consider only direct recursion, i.e, we do not account for cases where a function f calls a function g which calls f. Previous work has shown that indirectly recursive calls are uncommon (CARTER et al., 2018). Furthermore, this analysis would considerably increase the processing time since it would require the identification of cycles in the program call graph.

3.3.2 Immutability

When parsing for immutability, we look for structures representing the idea of shallowly copying a data structure (object or array) or preventing it from being changed. We do not, for example, look for deep clones or libraries related to immutability (e.g., Ramda, Underscore.js). Therefore, we consider four scenarios for immutability.

The first case we consider is the use of the `Object.freeze` method. It prevents an object (and its prototype) from being changed. When analyzing the repositories, we used the TypeScript type checker to infer when a call expression has a left-hand side expression with an object constructor, whose name is `freeze`, to reduce the probability of false positives.

Next, we consider the use of spread syntax for immutability because it is used to (shallowly) copy or destructure arrays and objects without modifying them. To process these structures, we search for array literal expressions that represent spread elements and spread assignments. Listing 3.1 shows an example of an object (`person`) being shallowly copied into another object (`anotherPerson`). This new object has its `age` property changed to 51, but that does not affect the first object.

```
const person = { age: 50 };
const anotherPerson = { ...person, age: 51 };

console.log(person.age); // 50
console.log(anotherPerson.age); // 51
```

Listing 3.1 – Example of spread syntax.

In addition to the spread syntax for shallow copies, we also look at two other structures with the same purpose, `Object.assign` (with an empty object passed in the first parameter) and `Array.slice` (with no arguments taken). In the first situation, we look for a call expression with precisely two arguments where the first one is an empty object and the second one is an object. We also take uses of `Array.slice` into account because, when no arguments are taken, the `slice` function returns a copy of an array in its entirety, in a

similar way to those previously mentioned. Parsing this structure requires only searching for call expressions from arrays whose names are `slice` with zero arguments.

3.3.3 Lazy evaluation

Although JavaScript does not support lazy evaluation inherently, it includes mechanisms that can delay the evaluation of an expression (or execution of a statement) until it is necessary. This work examines two such mechanisms, named generator functions and thunks.

Generator functions are not directly inspired by functional programming (KEREKI, 2020). Notwithstanding, we consider them a way of achieving lazy evaluation because instead of immediately processing an expression when invoked, these functions return a particular type of iterator called generator. This iterator only has its value consumed when the generator's `next` method is called, executing the function until it finds the `yield` keyword. With generators, it is possible, for example, to create infinite lists, a typical example of the uses of lazy evaluation (HUGHES, 1989). An asterisk token can identify generator functions or methods in their declarations.

In Listing 3.2, there is a generator function that returns numeric values every time its `next` method is called. It is important to note that, thanks to lazy evaluation, it is possible to use an infinite data structure (`Infinity`) without running out of memory.

```
function* range() {
    let count = 0;
    for (let i = 0; i < Infinity; i++) {
        count++;
        yield i;
    }
    return count;
}
const iterator = range();
console.log(iterator.next().value); // 0
console.log(iterator.next().value); // 1
```

Listing 3.2 – Example of a generator function.

A thunk is a nullary function literal, i.e., an arrow function that has no parameters. Thunks encapsulate computations that are only executed when they are actually invoked. Consequently, they can also be used to delay evaluation (KEREKI, 2020). In Listing 3.3, the expression "2 + 2" is only evaluated when the thunk `four` is called.

```
const four = () => 2 + 2;
```

Listing 3.3 – Example of a thunk.

3.3.4 Functions as values

Due to the emphasis of this work on JavaScript, we divide functions as values into two groups: **higher-order functions (HOFs)** and **callbacks & promises**.

When parsing for HOFs, we look for two specific scenarios. The first scenario occurs when a function takes another function as an argument and uses it to traverse a list applying it to each component of the list (WATT, 2004). To identify this scenario, we look for function names that refer to native functions from `Array.prototype` (`every`, `filter`, `find`, `findIndex`, `flat`, `flatMap`, `forEach`, `map`, `reduce`, `reduceRight` and `some`), and receive functions as arguments to traverse a list. We search for type-inferred arguments that are arrow functions, function expressions, or type-inferred functions. We ignore cases where TypeScript is unable to infer the type of the argument.

The second scenario consists of non-native functions (created by a developer) returning another function as their result. We disregard non-native function declarations that take functions as parameters due to a limitation of the Typescript compiler API that does not infer functions types in that manner. This does not include callbacks, which we address later in this section. In addition, there are many ways to call functions in JavaScript but we decided to only parse `property access` expressions, i.e., of the form `o.f()`. We are also not considering `Array.prototype` overriding. We discuss these limitations in more detail in Section 4.6.

Callbacks are functions passed as an argument to another (parent) function and they are typically used in asynchronous calls such as timeouts and `XMLHttpRequests` (`XHRs`). Callbacks are executed after the parent function has completed its execution (FARD; MESBAH, 2013), that is, "as a handler to be called in response to some future event." (SCOTT, 2016). In this work, we only look for functions that use callbacks, that is, functions whose parameter names are called once or more inside their bodies. More specifically, we consider scenarios where declared functions are passed as arguments or where the argument is an anonymous function.

A promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. We consider it an excellent example of the native use of functions as values because when promises are created, it is possible to pass at least two arguments (two functions) that will be called when a promise succeeds (`resolve`) or fails (`rejects`). Promises became popular in JavaScript as an approach to discipline the use of callbacks. To parse them, we look only for declarations of promises, so we type-check `new` expressions creating objects of type `Promise`. Listing 3.4 presents a promise that fulfills with the value 1 when it resolves, printing it in the console.

```
new Promise((resolve) => {
  resolve(1)
```

```
}).then(console.log) // 1
```

Listing 3.4 – Example of a promise.

3.4 METHODS

To answer **RQ1**, we first identify all uses of functional programming concepts from the main revision of each repository. To do this, we visit every AST node searching for specific node kinds that can include uses of these concepts, as explained in the previous section. This process is done in conjunction with the data extraction of RQ2, except that, for RQ1, we only consider the revision marked with GitHub's main revision SHA.

To get data to answer **RQ2**, we go through each commit for each repository, analyzing one snapshot (due to computational cost) per month when available. This process starts on the main revision date and ends with the first available commits made in the repositories. For each set of functional programming concepts we identify, we save this data to use during the analysis step.

To address **RQ3**, we need to obtain commits and classify them as bug-fixing or non-bug-fixing commits. To do so, we rely on labels provided by developers to categorize issues and pull requests on GitHub repositories. We use the REST and GraphQL GitHub APIs to fetch issues and pull requests based on queried labels related to bugs and then obtain the commits that closed these issues and pull requests for every analyzed project.

First, we define a list of terms that are related to bugs, which is composed of `bug`, `error`, `defect`, `failure`, `fault`, and `exception`. These terms are used in a query to fetch labels related to bugs in a given repository. In this matching process, we ignore labels that contain one bug-related term together with the term `unconfirmed` or `not`.

After that, we start the process for each repository included in our study. We first get all labels from a given repository and classify them as bug-related labels or not according to the terms described above. Then, we fetch issues and pull requests from the repository because when we were developing our extraction tool, there was no way of directly downloading commits based on the labels of the associated issues and pull requests. Thus, for each bug-related label in the repository, we download up to 1,000 closed issues or pull requests since the last repository commit date using a GraphQL query. We chose not to recursively download all issues and pull requests because it would take much longer to get all the data in each repository, which would also add considerable processing time to the data analysis. We only consider the issues that were closed by a commit or by a pull request through the GraphQL GitHub closing event (`CLOSED_EVENT`) so that we can associate them with their fixing commits. Finally, from the selected issues and pull requests, we fetch the last associated commit and consider them bug-fixing commits. To download non-bug-fixing commits, we follow the same steps, but the query to fetch the issues and pull requests is made so as not to bring data that contains the bug-related

labels. We consider that the commits found in these issues and pull requests are not from bug fixes.

Our tool analyzes the repository versions from the bug-fixing and non-bug-fixing commits to identify and count how many occurrences of functional programming structures were removed. Note that this analysis is performed on the complete snapshots of the identified commits and their parents, not the `git` difference between them. We consider the entire snapshot because we leverage the available information to improve the precision of detecting some functional programming structures—loading only the modified parts of the code limit TypeScript's ability to infer types.

To answer **RQ4**, we visit every AST node on each repository's main git revision, looking for comments. To parse them, we use the TypeScript Compiler API to collect information about the positioning of the comments, based on ranges from the full text of each source file and node positioning (`full start` for leading comments and `end` for trailing comments), their types (`leading` or `trailing`), whether they are adjacent to functional programming structures or not, and whether they have JSDoc tags. In addition, we also remove repeated comments, as the same comment can appear in more than one AST node and persist in CSV files.

4 STUDY RESULTS

In this chapter, we present the results of the study. Each one of the four research questions (RQs) is addressed in its respective section.

4.1 HOW OFTEN ARE FUNCTIONAL PROGRAMMING STRUCTURES USED IN REAL SOFTWARE? (RQ1)

In our corpus of 91 open-source JavaScript projects, we identified 478,605 occurrences of functional programming structures. Considering that the analyzed projects have 22,326,070 lines of code, that means that there is one use of functional programming concepts for every 46.65 lines of code, on average. Table 4 shows a breakdown of these occurrences in terms of these structures (column "Occurrences (#)") and the percentage of the overall LOC of the analyzed projects related to functional programming (column "% LOC").

If we examine the number of lines of code of the functional programming structures, 43.59% of all of them, in the projects, are related to functional programming. It means that almost one out of every two lines in these projects is related to functional programming. This number may sound inflated, but it makes sense when we consider that, for example, for a higher-order function, we include all the lines of the function in this count, as we show in Listing 4.1. The rationale for this conservative approach is to account for the code affected by functional programming in its entirety. For example, for the code snippet in Listing 3.4, we would count three lines of code. It may lead to some lines of code being counted more than once for different structures, e.g., a callback may use higher-order functions and invoke `Object.freeze`. This is the reason why, if we add up all the percentages in the column "% LOC" of Table 4, the result will be greater than the aforementioned 43.59%. Different approaches could have been employed, but then identifying which parts of the code pertain to functional programming concepts and which ones do not become fuzzy. For example, in an anonymous function used as a callback, should we ignore its body when counting the lines of code? The answer is not clear. We mitigate this problem by presenting both the number of LOC and the number of occurrences of each structure.

In total, the most pervasive concept is lazy evaluation, with 299,520 occurrences. The least pervasive one is recursion, with only 7,879 occurrences. When we consider the structures related to the concepts, the most and least frequent ones are, respectively, `thunks`, with 298,797 instances, and the higher-order function `reduceRight`, with only 25 occurrences. Furthermore, we did not find occurrences of `flat` functions. When we consider the number of lines of code, callbacks & promises comprise 4,168,527 LOC, i.e.,

Table 4 – Prevalence of functional programming structures.

Concept	Structure	% LOC	Occurrences (478,605)
Higher-order functions	every	0.0133%	0.1272% (609)
	filter	0.0753%	1.0971% (5,251)
	find	0.0181%	0.4461% (2,135)
	findIndex	0.0021%	0.0501% (240)
	flat	0%	0% (0)
	flatMap	0.0010%	0.0079% (38)
	forEach	0.6475%	2.9024% (13,891)
	map	0.3179%	2.1454% (10,268)
	reduce	0.0879%	0.4643% (2,222)
	reduceRight	0.0005%	0.0052% (25)
	some	0.0160%	0.2513% (1,203)
	Non-native	8.8892%	7.4418% (35,617)
	Total	10.0688%	14.9388% (71,499)
Immutability	Array.slice	0.0034%	0.1529% (732)
	Object.assign	0.0067%	0.2171% (1,039)
	Object.freeze	0.0526%	0.3661% (1,752)
	Spread Assignment	0.6263%	5.4366% (26,020)
	Spread Element	0.0916%	1.0403% (4,979)
	Total	0.77%	7.213% (34,522)
Callbacks & Promises	Callback	17.8525%	9.2330% (44,190)
	Promise	0.8767%	4.3867% (20,995)
	Total	18.67%	13.6197% (65,185)
Lazy evaluation	Generator	0.0435%	0.1510% (723)
	Thunk	15.3588%	62.4308% (298,797)
	Total	15.4023%	62.5818% (299,520)
Recursion		1.2233%	1.6462% (7,879)

Source: the author (2022)

18.67% of all the LOC in the analyzed projects. Immutability, the functional programming concept with the least lines of code, comprises only 172,462 LOC, i.e., 0.77% of all the LOC.

Table 4 also presents the proportions of all the occurrences of functional programming concepts represented by each structure. Furthermore, the sum of the occurrences of thunks and callbacks accounts for 71.66% of all the occurrences of functional programming structures in the analyzed projects.

```
function () {
    var canSetImmediate = typeof window !== 'undefined'
    && window.setImmediate;
    var canPost = typeof window !== 'undefined'
    && window.postMessage && window.addEventListener
    ;
```

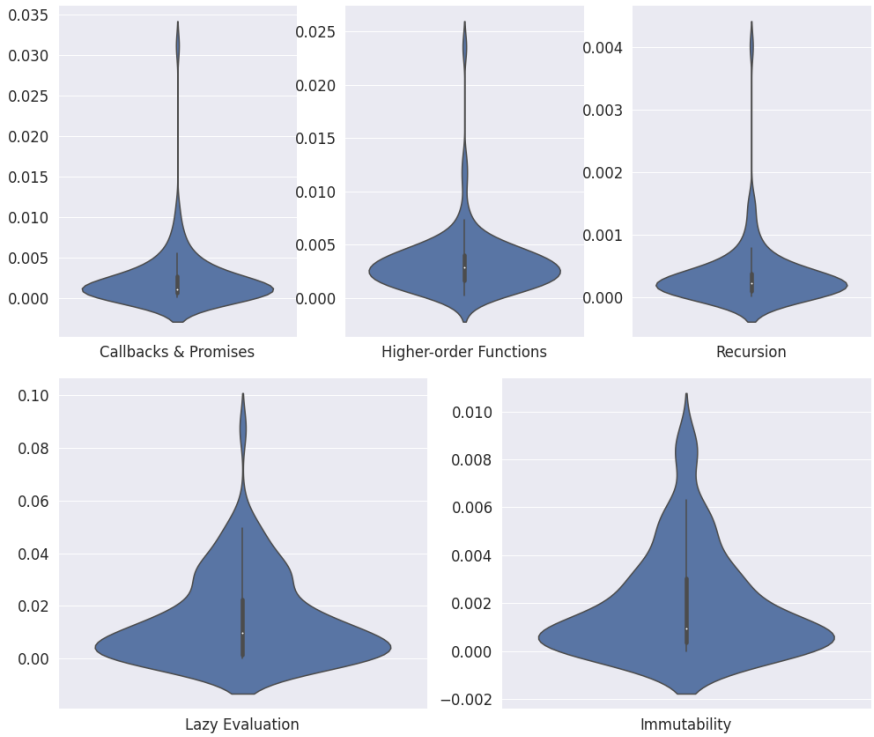
```
    if (canSetImmediate) {
        return function (f) { return window.setImmediate(f) };
    }

    if (canPost) {
        var queue = [];
        window.addEventListener('message', function (ev) {
            var source = ev.source;
            if ((source === window || source === null) && ev.data === '
                process-tick') {
                ev.stopPropagation();
                if (queue.length > 0) {
                    var fn = queue.shift();
                    fn();
                }
            }
        }, true);

        return function nextTick(fn) {
            queue.push(fn);
            window.postMessage('process-tick', '*');
        };
    }

    return function nextTick(fn) {
        setTimeout(fn, 0);
    };
}
```

Listing 4.1 – Example of a higher-order function mined in a repository (ajaxorg/ace).

Figure 1 shows the distributions of the usages of functional programming concepts in the repositories. The LOC of the projects normalizes the distributions. Note that the scale is different for each violin plot. The usage of callbacks & promises, higher-order functions, and recursion is overall consistent for the projects, which is observed by the density of projects in the median of the distributions. However, this consistency does not exist with immutability-related structures and the lazy-evaluation-related ones.

Key takeaways for RQ1. We found out that functional programming concepts are used very often in the analyzed projects, on average, once for every 46.65 lines of code. In addition, the code related to these structures comprises 43.59% of all the LOC in these projects. The most popular of these structures are `thunks`. They represent 62.43% of all functional programming structures and occur six times more often than the second most popular one, callbacks.

Figure 1 – Distribution of use of functional programming concepts in repositories



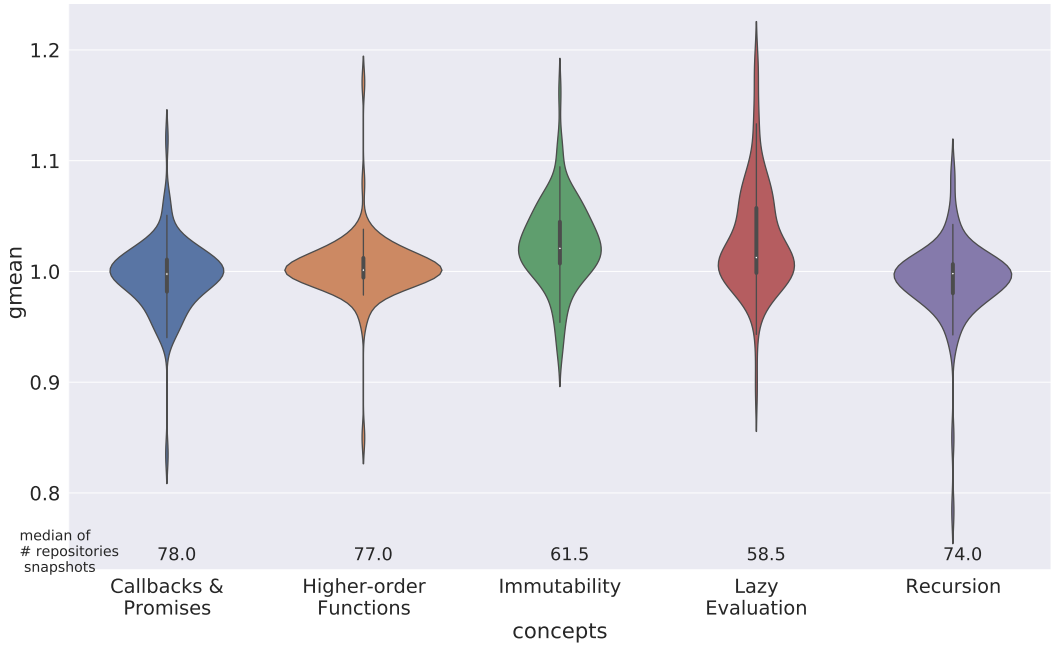
Source: the author (2022)

4.2 HOW HAS THE USE OF FUNCTIONAL PROGRAMMING STRUCTURES EVOLVED OVER THE YEARS? (RQ2)

We also investigate how the use of functional programming concepts changes throughout the evolution of the 91 repositories. We adopt the following approach. First, for each monthly snapshot (see Section 3.4) of each project (totaling 5,757 monthly snapshots), we count the number of lines of the functional programming concepts, normalized by the overall number of LOC of the project. Second, we calculate the percentage change for each pair of consecutive monthly snapshots. If any of the two snapshots has a zero value, we discard it and use the next one with a non-zero value. Finally, we compute the geometric mean of all the percentage changes for each repository and structure. The geometric mean is useful to summarize changes in percentages over time. For example, if the geometric mean of the percentage changes for one of the functional programming concepts, for one project, is exactly 1.0, this means that the use of that concept did not change throughout the evolution of that project. In case it is 1.01, this means that, on average, the use of that concept grew 1% per subsequent snapshot. Since the values are normalized by the number of LOC, this is a real growth in the use of functional programming.

The violin plot in Figure 2 presents the result of the processing. Each violin shows the distribution of the geometric means of the analyzed projects for each functional programming concept. The white dot in each violin indicates the median geometric mean.

Figure 2 – Distribution of evolution of functional programming concepts in repositories



Source: the author (2022)

Table 5 – Order statistics for the geometric means summarizing the evolution in the use of functional programming structures in the analyzed projects. A value of 1.0 means no change.

Concept	Min	25%	50%	75%	Max
Callbacks & promises	0.834839	0.981676	0.997667	1.010751	1.119621
Higher-order functions	0.849989	0.994403	1.001208	1.012228	1.170697
Immutability	0.926663	1.007126	1.020825	1.044866	1.161784
Lazy evaluation	0.895239	0.998629	1.012532	1.057329	1.185875
Recursion	0.783807	0.980235	0.998015	1.006601	1.089953

Source: the author (2022)

The numbers at the bottom of the plot show the median number of snapshots based on the geometric means calculated. The plot shows that all the medians sit close to 1.0. Table 5 presents the order statistics for the data in the plot. It is possible to see that 50% of the projects exhibited a monthly growth of 2.0825% in the use of immutability structures. Since we are considering 61.5 snapshots (as shown at the bottom of the plot), this represents an overall growth of 255% ($1.0208258^{61.5}$) in the use of immutability-related structures, on average. The usage of two other functional programming concepts also grew in the analyzed projects: lazy evaluation (107.21%) and higher-order functions (9.74%). For the two remaining concepts, there was a reduction. Overall usage of recursion fell by 13.67% and callbacks & promises by 16.65%.

Key takeaways for RQ2. The use of functional programming structures has been growing throughout the years for the analyzed projects. However, this growth is uneven and inconsistent for all functional programming concepts. Usage of immutability-related structures has grown by more than 255% whereas usage of structures for lazy evaluation has doubled. On the other hand, the usage of recursion and callbacks & promises has decreased by 13.67% and 16.65%, respectively.

4.3 ARE USES OF FUNCTIONAL PROGRAMMING STRUCTURES REMOVED MORE OFTEN IN BUG-FIXING COMMITS? (RQ3)

In our corpus of 91 open-source JavaScript projects, in this research question, we use 151,489 commits, where 42,929 are classified as bug-fixing commits and 117,558 as non-bug-fixing commits. To analyze the error-proneness of functional programming concepts, we start from the null hypothesis that there is no relationship between bug-fixing commits and the removal of functional programming structures. More specifically, we formulate five null hypotheses, one for each functional programming concept. We perform the chi-square test considering two dimensions: bug-fixing vs. non-bug-fixing commits and with vs. without removal of functional programming structures. Since we test five hypotheses, we apply the `Bonferroni` adjustment to the alpha. Thus, we use an alpha of 0.01.

Table 6 presents the obtained p-values for the chi-square test. In all comparisons, the null hypotheses are rejected, excepting immutability. In other words, there is a relationship between the removal of functional programming structures and bug-fixing commits but not for the latter. For all the tests that rejected the null hypothesis, the p-values are orders of magnitude lower than 0.01.

We then calculate the odds ratio to quantify the odds of functional programming structures removed in a bug-fix commit. Table 6 shows the results for the functional programming concepts. For example, the odds ratio for recursion is 0.622. It indicates

Table 6 – The p-values and odds ratios for the relationship between bug-fixing commits and the removal of functional programming structures. For all the cases, excepting immutability, the removal of functional programming structures is **less** likely to occur in bug fixing commits than in non-bug fixing commits.

Concept	Chi-square test (p-value)	Odds ratio
Recursion	4.580×10^{-93}	0.622
Lazy evaluation	2.591×10^{-7}	0.958
Higher-order functions	5.049×10^{-14}	0.931
Callbacks & Promises	1.196×10^{-52}	0.826
Immutability	0.015276	

Source: the author (2022)

that functional programming structures are 37.77% **less** likely to be removed in bug-fixing commits than in non-bug-fixing commits. This same phenomenon can be observed for all the cases that rejected the null hypothesis, although less intense. Structures related to lazy evaluation, higher-order functions, and callbacks & promises are 4.16%, 6.89% and 17.38% less likely to be removed in bug-fixing commits, respectively. These results suggest that using functional programming structures in JavaScript programs is less bug-prone than not using them, except for immutability.

Key takeaways for RQ3. Functional programming structures tend to be removed less often in bug-fixing commits than in non-bug-fixing commits. It can be observed for all the functional programming concepts, excepting immutability. The difference is more prominent for recursion and callbacks & promises than for other functional programming concepts. They are 37.77% and 17.38% less likely to be removed in bug-fixing commits, respectively.

4.4 IS CODE THAT EMPLOYS FUNCTIONAL PROGRAMMING STRUCTURES ASSOCIATED TO LONGER COMMENTS? (RQ4)

Code comments aim to make code easier for developers by explaining how it works or the rationale behind its leading design and implementation decisions. Some authors (BECK, 2004; HUNT; THOMAS, 1999; FOWLER et al., 2019) argue that comments indicate problems with their associated code, especially if they appear within methods. Fowler et al. (2019) states the following about code comments:

> *It is surprising how often you look at thickly commented code and notice that the comments are there because the code is bad.*

We investigate whether comments associated with code that leverages functional programming concepts tend to be longer than comments for code that does not include these structures. We expect longer comments associated with code that is harder to understand (AMAN et al., 2015; STEIDL et al., 2013; BUSE; WEIMER, 2008). Our sample has 1,644,133 comments, 17,772 of them are adjacent to the functional programming structures and 1,626,361 are not. Of these comments, 131,342 are trailing and 1,512,791 are leading. 311,365 are using JSDoc tags. The most commented concept is higher-order functions with 13,842 comments. The one with the least is immutability.

We use the point-biserial correlation coefficient to check the correlation between comment size (a continuous variable) and the presence of functional programming structures (a dichotomous variable) in the associated code snippet. We do not take JSDoc tag comments into account as they have a specific structure used to generate documentation for coarse-grained entities, e.g., entire methods or classes. Table 7 shows the results of the

Table 7 – Correlations between usage of functional programming structures and code comment size. The p-values only indicate statistical significance for functional programming in general. For all the cases, correlation was negligible.

Concept	Point-biserial coefficient	p-value
Recursion	-12.064×10^{-5}	0.889
Lazy evaluation	-13.906×10^{-5}	0.872
Higher-order functions	-50.456×10^{-5}	0.560
Callbacks & Promises	-23.197×10^{-5}	0.789
Immutability	-6.026×10^{-5}	0.944
All	-57.107×10^{-5}	0.510

Source: the author (2022)

point-biserial correlation for each of the functional programming concepts. For all the concepts the correlations are negligible, and the p-values suggest that it is not possible to infer a relationship between comment size and the use of functional programming concepts. Considering the combination of all structures, without distinguishing between the concepts, we obtain a p-value of 0.510 and a correlation of -57.107×10^{-5}. This correlation is negligible, and the p-value indicates no statistical significance. This result suggests that code including functional programming concepts is not more challenging to understand than code that does not.

Key takeaways for RQ4. We found no correlation between comment size and code with functional programming concepts. When examining the different functional programming concepts separately it is not possible to ascertain whether there is a relationship or not.

4.5 DISCUSSION

The JavaScript language includes support for higher-order functions and enables every function to be treated as a value since its first version in 1995. More recent versions of the ECMAScript specification, e.g., ES 6, in 2015, extend this support with structures such as arrow functions and the `const` declaration. This study shows that these structures have widespread adoption in the analyzed projects. At the same time, besides the use of callbacks (GALLABA et al., 2015; GALLABA et al., 2017; SABOURY et al., 2017b), we are not aware of any other study in the literature that studies how ideas from functional programming influence software development in JavaScript. Researchers have an opportunity to fill in this knowledge gap by investigating in-depth topics such as how developers use immutability structures, how to support these developers in building (mostly) purely functional programs in JavaScript, and how to refactor existing systems to leverage these structures.

Not only are functional programming structures in widespread use, but their use has also been growing even when we normalize that growth based on the number of lines of code in each project. As pointed out in Section 4.2, on average, the frequency of occurrence of structures related to lazy evaluation, mainly thunks, has grown by more than 100% throughout the life of these projects. Although there was a reduction in recursion and callbacks & promises, these reductions were comparatively small. In the former case, the reduction may be explained by the growth in the use of higher-order functions, since many of these functions (`map`, `filter`, `reduce`) perform operations that are typically implemented recursively. In the latter case, part of the reduction can be explained by the use of alternative structures, such as `async-await`, introduced in ECMAScript 8 (2017), which provide a more disciplined way of using promises. Similarly, the growth in the use of thunks can be partially explained by the publication of the ECMAScript 6 specification, which introduced these structures. Notwithstanding, it does not explain why 62% of all the occurrences of functional programming structures in projects pertain to this case. Investigating these changes in tendencies in more depth is left for future work.

An important point raised during the execution of this research was whether we should consider `const` declarations or not. A `const` declaration can be seen as connected to immutability because it declares a block-scoped, read-only variable. So, the value of a `const` variable will not be reassigned or redeclared within the same scope because its reference is immutable. Despite this, the Mozilla Developer Network Web Docs[1] consider that `const` is not an immutability structure because such as variable may be assigned an object or array, which is mutable. If we hypothetically consider this element to be related to the functional programming concept of immutability in our analyses, it would significantly impact the results. For example, it becomes the most pervasive structure, with 900,858 instances. About one in every 25 lines of code in the analyzed projects would consist of a `const` declaration. Also, almost two out of every three occurrences of functional programming structures would be uses of `const` declarations. The sum of the occurrences of `const` declarations, thunks, and callbacks would account for 90% of all the occurrences of functional programming structures. Furthermore, the usage of immutability structures would have grown, on average, by approximately 400% throughout the evolution of the analyzed projects.

Since the early days of functional programming, it has been touted to as a way to write code that is clearer and easier to understand. In his Turing Award Lecture, Backus (1978) remarked about a functional program that *"its structure is helpful in understanding it without mentally executing it"*. A decade later, Hughes (1989) argued that *"[functional programming] allows improved modularization."*. According to him, mechanisms such as lazy evaluation and higher-order functions make it possible to write simpler programs by decomposing them into small, easy-to-write and read pieces. Hudak (1989) emphasized

[1] <https://developer.mozilla.org/>

the importance of immutability, arguing that *"although the notion of referential transparency may seem like a simple idea, the clean equational reasoning that it allows is very powerful, not only for reasoning formally about programs but also informally in writing and debugging programs."* These authors and their ideas have been very influential in the Programming Languages and Software Engineering communities. Even though they focus on statically-typed, purely functional languages, this paper shows that there is potential benefit in leveraging these ideas, even if only partially, in the context of a dynamically-typed, imperative programming language. Our study provides evidence that the use of structures inspired by functional programming does not make code harder to understand while being potentially less bug-prone. Considering that functional programming structures have seen little empirical evaluation, this is an important step that can motivate the community to conduct more studies.

Finally, we tried to understand if there is any correlation between comment length and the usage of functional programming-inspired structures in the neighboring code, but this is not the only aspect to be considered. The presence of Self-Admitted Technical Debt (SATD) comments has been analyzed by some works (BAVOTA; RUSSO, 2016; MALDONADO et al., 2017). Analyzing whether there is any relationship between SATD comments and the use of functional programming is another dimension for future work. In addition, an in-depth analysis can also be performed on the removal of these comments.

4.6 THREATS TO VALIDITY

Construct validity. Some structures cannot be directly related to their concept. For example, we considered *spread* as a structure that promotes immutability because developers use it to create copies of objects and arrays when they do not want to change the original ones. However, someone can use it to create copies for another intention that is not immutability.

Some ways of mining structures were not considered because they were not representative enough. For example, there are many ways to call functions in JavaScript, but, after sampling four GitHub popular projects (React, Angular, ESLint, and Hoodie), we discovered that only 0.07% of function call expressions were not `property access expressions`, so we decided to ignore other access expressions, e.g., *element access expression*, when parsing. We also searched for `Prototype` overriding of higher-order functions, e.g., redefinitions of function `map`, and found no cases in our random sample. It means that it is possible to identify native higher-order JavaScript functions by their name and target object type.

Furthermore, as in most implementations of source code analysis, our parser has some limitations. For example, we did not find any case of `flat` functions. An in-depth analysis would show the reasons for it. Moreover, in a literal array where each element is in one line,

and there is a comment in each line, the parser only returns the comment in the last line of the literal array. Also, due to the nature of the structures we mine and the high dynamism of the JavaScript language, the precision of the mining process may be negatively affected. We gauge the precision of the tool we have built by manually checking the results produced by it, as reported in Section 3.3. We did not find any misclassifications.

Internal validity. To mine the usage of functional programming structures, we analyzed the entire projects' versions to find additions and removals instead of the actual changes between pairs of versions. Analyzing the changes would increase the engineering effort for implementing the parser and make it impossible to mine some structures. However, by analyzing the entire versions of the projects, we do not have the mapping of the actual structures from one version to another, which is a threat to our study. For instance, if a given existing structure in a version of the project was deleted in a subsequent one, but in that subsequent version, a different structure, but of the same kind, was added, our parser will count it as the same structure.

Moreover, for RQ3, we hypothesize that if functional programming structures are deleted in bug-fixing commits, their usage might be bug-prone. This hypothesis assumes that all changes in a bug-fixing commit are about the bug fix. However, research projects have shown that bug-fixing commits contain other changes that are not related to bug fixes (HERBOLD et al., 2021).

Finally, for RQ4, we compared the size of comments in source code related to functional programming structures with the size of comments in other source code. Our analysis is performed at the AST level, and a comment might belong to more than one AST node since comments are at the line level. It would bias our results because the same comment would be counted for several nodes. To solve that, each comment found in multiple AST nodes was considered only for the first node. However, the existing threat to our study's validity is that a comment might belong to AST nodes related to functional programming structures and other code, and we cannot know for sure what the comment is about. In such a case, we keep the comment for one AST node of each type of code (related and non-related to functional programming structures).

External validity. Even though we searched for projects from various domains, it is still possible that our sample is not representative enough among the many repositories available on GitHub. Furthermore, it is impossible to relate our findings with enterprise software development, mainly because we analyzed only open-source software. Moreover, the functional programming structures we choose may not be representative when generalizing functional languages. As we decided to focus on some scenarios, we know that we do not evaluate several other functional programming structures, so we can not say how representative these other structures would be.

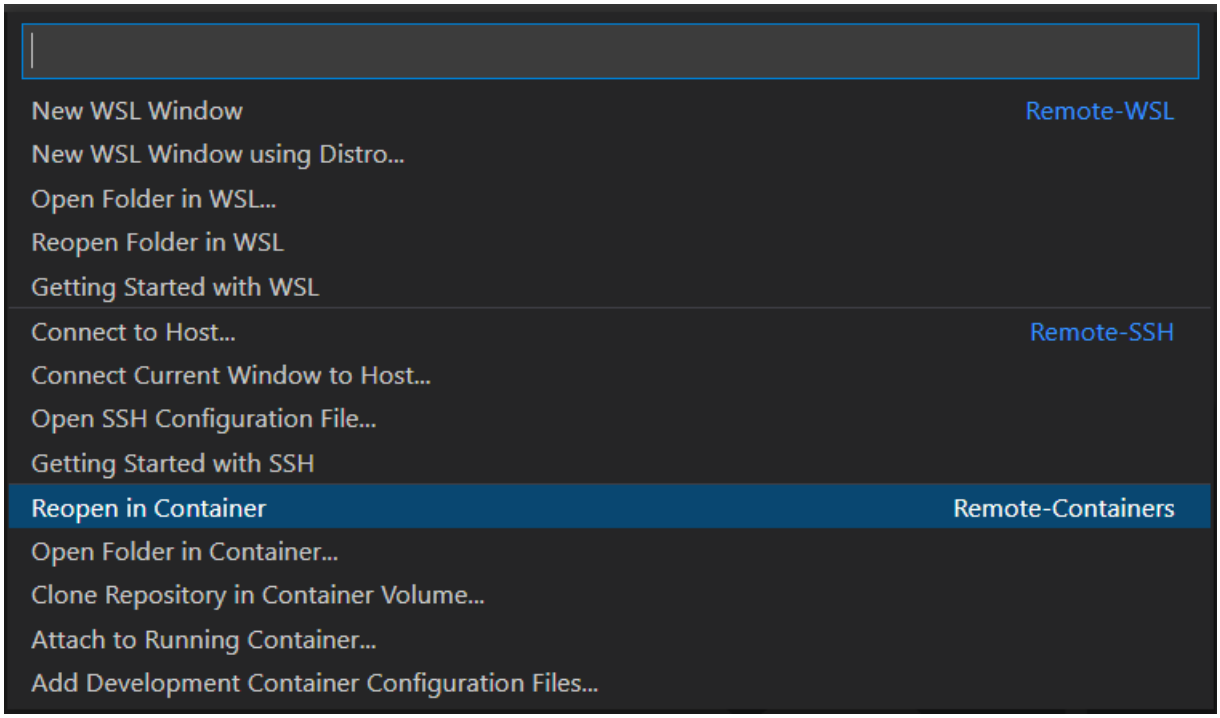
5 PSMINER

This chapter presents psMiner, which is the **p**rogramming **s**tructures' **Miner** developed in this work to mine software repositories. It explains how psMiner can be installed and configured (Section 5.1), how it is implemented (Section 5.2), and how it can be customized to be used in further studies (Section 5.3).

psMiner is publicly available at
<https://doi.org/10.5281/zenodo.6425005> (Alves et al., 2022a).

5.1 INSTALLING & CONFIGURING

Figure 3 – Reopen in Container option listed in Visual Studio Code menu.



Source: the author (2022)

Some applications must be installed and configured to run psMiner. Docker and Visual Studio Code are examples of this. In addition to them, the Remote Containers extension[1] should also be installed to deal with running containers in Visual Studio Code. After that, it is only necessary to open the project in Visual Studio Code and select the option "Reopen in Container" from this extension. It will open the project in a Docker container with the project's dependencies installed. In cases where it is not possible to

[1] <https://code.visualstudio.com/docs/remote/containers>

use Docker, psMiner can be installed using a local instance of Node.js[2]. In both cases, the central command to run the mining process is `npm start`.

Besides that, it is mandatory to configure a `.env` file to run this project. This file accepts five variables:

TOKEN: In this variable, it is required to place a Personal Access Token (PAT) from GitHub[3] with permissions `read:gpg_key`, `repo` and `user` with all options selected. This token is used to make requests to GitHub APIs.

NODE_OPTIONS: This is where the environment variables are passed to Node. By default, we recommend passing the `max-old-space-size` setting to at least 8192 (8GB of memory). This option increases the memory space available to psMiner while it is running.

START: This variable is the starting point index for psMiner to create a range of repositories to be analyzed. The minimum value is zero, and the maximum value is the number of available repositories in `results.csv`, the output file of the SeArt (Section 3.2) tool, minus one.

END: This variable is the final index in the list of repositories to psMiner analyze. The value must always be greater than or equal to START. The maximum value is the number of repositories available in the `results.csv` file.

STEP: This variable indicates how many repositories should be downloaded and analyzed in parallel while running psMiner. The database back-end that the tool employs, SQLite, does not support parallelism, though. Therefore, when using it, the value of this variable should always be 1.

5.2 DEVELOPMENT

This section describes each step of the mining tool. It is where we seek to justify every decision made during the development of the tool. When possible, we go into more specific details regarding commands executed or other tools that have been added.

5.2.1 Input data pre-processing

Before starting the data extraction, psMiner must preprocess the SeArt file (`results.csv`). In our study, preprocessing consisted of obtaining the number of lines of codes (LOC) in

[2] <https://nodejs.org/>
[3] <https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token>

each repository. This data is extracted by running `cloc`[4]. It is an NPM package, created by Kent C. Dodds, based on another tool of the same name[5], which counts the number of files, the number of blank lines, comment lines, and code lines of the source files in a set of repositories. For each analyzed project, `cloc` measures the number of LOC for each language employed in its source files and export their counts to CSV files. For our study, we consider only the sum of all the lines of code instead of counting languages separately. With this information in hand, psMiner exports a new file (`cloc.csv`) concatenating the information available in `results.csv` adding a new column, named `cloc`, to store the generated information. This file is the starting point for all further analyses.

5.2.2 Data extraction from repositories

After completing the process described in the previous section, psMiner extracts data from the repositories. It loads the list of selected repositories and sorts them by size (column `cloc`) to analyze the largest repositories first. It does this because these repositories take longer to load and have the potential to have more of the functional programming concepts we are looking for. If it is not possible to analyze all selected repositories, these are the ones with the highest priority. It then validates through GitHub's REST API if any selected repository was archived. As the repository environment on GitHub is very dynamic, this possibility exists. After that, psMiner downloads all the chosen repositories to be analyzed.

Downloading a repository includes verifying whether it already exists in the destination folder, is up to date, and the default revision branch is still available (if not, psMiner fetches from GitHub which one is the new default using the REST API). It also includes checking if the branch `master` was renamed to `main`[6] in cases where the `master` is the default branch in `cloc.csv` file. psMiner uses `simple-git`[7] to run git commands when manipulating repositories.

We created a GraphQL query that returns all labels from a repository and groups them by name. psMiner transforms all the labels into lowercase letters and removes duplicates before storing them. psMiner then filters the list of labels searching for the ones related to bug classification, stores them in CSV files, and then classifies commits.

As there was no way of directly downloading commits with specific labels from GitHub at the time, psMiner took another approach to obtain this data. Using a GraphQL query, it downloads closed issues (or pull requests) up to the limit imposed by GitHub (1,000) for each bug label in each repository since the last repository commit date (obtained in the SeArt data file). It only considers the issues closed by a commit or a pull request

[4] <https://github.com/kentcdodds/cloc>
[5] <https://github.com/AlDanial/cloc>
[6] <https://github.com/github/renaming>
[7] <https://www.npmjs.com/package/simple-git>

through the `CLOSED_EVENT` event. In other words, psMiner considers that a commit or a pull request that closed last the issue is the one that fixed the bug (after all, an issue can be closed or reopened several times). psMiner classifies them as bug fixes and stores them in a SQLite database. In this case, it uses Sequelize[8], a TypeScript ORM, to manage data. This way, psMiner checks if there is already a commit with the same URL while storing it, avoiding duplicated data.

When an issue was closed because of a pull request (instead of a commit), psMiner gets the resource information (in this case, a pull request) through a GraphQL query (`fetchPullRequest`). We do this because if psMiner were to pull the information all at once while downloading issues, it could have millions of nested results, thus maxing out the limit[9] of a GraphQL query in GitHub. This fetched information only considers up to the limit imposed by GitHub (100) of the most recent commits that remain associated with the pull request plus the commit that was responsible for merging it (`mergeCommit`). psMiner considers that these commits are from bug fixes and then follows the same steps for each pull request returned from this search.

When downloading the list of non-bug-fix commits, psMiner repeats the steps, slightly changing the query to fetch them. It uses a property called `-label`, populated with all those labels used in the step of capturing bug labels. This property excludes issues or pull requests from the search with these labels. We consider that the commits found in these issues and pull requests are not from bug fixes and save them in the SQLite database. It excludes commits that are not part of the repository but are related to it (for example, when related to external pull requests). These commits require us to process their original repositories, which may be unavailable considering our resources. psMiner does the same to listed commits that are not part of the repository anymore, e.g., because of a removed branch or fork[10].

Next, psMiner identifies differences by comparing the complete snapshots of the project before and after a commit because the TypeScript parser cannot make type inferences accurately without loading the entire projects. Moreover, psMiner checks whether the commit exists locally through the command `git cat-file -s` and, if not available, it performs a `git fetch origin` passing the commit's SHA as a parameter. In this manner, a temporary branch called `FETCH_HEAD` is created to represent the state of the repository locally at that revision. After that, psMiner checks out to this new branch and runs the commit size verification process (to ensure that the commit is downloaded). With the commit locally available, it is necessary to load the list of changed files using the command `git diff-tree` with some parameters that ignore file removals and file renaming without changes. In the first case, it is not possible to analyze the file further (because it is not

[8] <https://sequelize.org/>
[9] <https://docs.github.com/en/github-ae@latest/graphql/overview/resource-limitations#rate-limit>
[10] <https://docs.github.com/pt/github/committing-changes-to-your-project/troubleshooting-commits/commit-exists-on-github-but-not-in-my-local-clone>

available in the revision), and in the second one, there are no code changes other than the filename. From this list of files, it analyzes only those that fit the same criteria used to filter files during the CLOC counter step, and for each file found, psMiner runs a `git log -n 2` command to get the SHA from the previous commit that also changed that file. psMiner uses this list of differences for each repository to start parsing when the commits were made. It parses for every difference found, obtaining functional programming structures for the list of files of each difference. It keeps track of these differences in the SQLite database. A comparison is made between the FPS lists of the involved revisions, looking for different FPS quantities from one revision to another. psMiner then compares if there are fewer structures in that revision representing the base commits. We consider that a commit removed functional programming structures when this happens. psMiner tracks this information in CSV files marking whether it came from a bug-fix or not, based on the classification made in the previous steps. Otherwise, it considers that the number of functional programming structures grew or did not change, storing only SHA information and whether it came from a bug-fix.

5.2.3 Parsing JavaScript projects

During parsing, psMiner identifies the functional programming concepts and structures listed in Chapter 3. Moreover, as already said, our parser uses the TypeScript compiler API because it provides a series of methods that help us traverse the Abstract Syntax Tree (AST) by identifying certain types and syntaxes available in the analyzed code.

Algorithm 1 represents the parsing process. It is done by a function that receives three parameters: (i) the list of files of the project that psMiner will analyze, (ii) a boolean to check if, during the parsing process, it is necessary to analyze the comments, and (iii) if we want to filter the parsing to only some files. We pass the list of files and an object of options as parameters to TypeScript's `createProgram` method (Algorithm 1, line 4). The object of options has two properties: `allowJs` (passed `true` to consider JavaScript files too) and `removeComments` (passed `false` to avoid discarding comments during the parser). It instantiates a `Program` object and gives access to methods that manipulate source files. `Program` is the type in TypeScript's API of the object that stores all the parsed information, e.g., the available files from a repository.

After creating the `Program` object, psMiner loads the type checker through the `getTypeChecker` method available in it (Algorithm 1, line 6). This checker is fundamental for locating some structures and was one of the great motivators for using the TypeScript compiler API instead of other parsers. Next, it checks if the third parameter (the list of files we want to load) has been passed (Algorithm 1, line 7), so it has to filter the source files that will be loaded in the program to those passed as parameters (Algorithm 1, line

Algorithm 1 Parsing of Functional Programming Structures.

Input: $fileNames$ – the list of files of the project to be analyzed
Input: $parseComments$ – a boolean to check if it is necessary analyzing the comments
Input: $includeOnly$ – filter to some files
Output: $records$ with parsed structures
Output: $comments$ with parsed comments
1: **let** $records \leftarrow []$
2: **let** $sourceFiles \leftarrow []$
3: **let** $comments \leftarrow []$
4: **const** $options \leftarrow \{allowJs : true, removeComments : false\}$
5: **const** $program \leftarrow createProgram(fileNames, options)$
6: **const** $typeChecker \leftarrow program.getTypeChecker()$
7: **if** $includeOnly.length > 0$ **then**
8: $\quad sourceFiles \leftarrow program.getSourceFiles().filter(() \rightarrow includeOnly)$
9: **else**
10: $\quad sourceFiles \leftarrow program.getSourceFiles()$
11: **end if**
12: **for const** $sourceFile\ of\ sourceFiles$ **do**
13: $\quad forEachChild(sourceFile, visit)$
14: **end for**
15: **return** $\{records, comments\}$

8). If not, it loads the list of source files (Algorithm 1, line 10). It then visits each AST node from these source files looking for structures from that moment on using the `visit` method (Algorithm 1, line 13). Algorithm 2 represents that `visit` method. Inside this method, there are three steps. The first step parses the comments when the parameter for this is `true` (Algorithm 2, line 1). The second is where psMiner mines structures (Algorithm 2, line 4-23), and the third is where the `visit` method is called recursively for each child node (Algorithm 2, line 24).

When the mining process enters the first step, psMiner loads a source file, looks for leading or trailing comment ranges (positioning intervals occupied by the code), and returns an array with the identified comments. This array of objects contains the line numbers where the block of comments starts and ends, whether there is any `JSDoc` tag inside it, whether it is a leading comment, in which file this comment was, and its length (considering the positioning).

In the second step, psMiner has a switch statement (Algorithm 2, line 4) with some cases as entry points to find structures. The TypeScript compiler differentiates between kind and type, and psMiner considers the node syntax kind in these cases. The former is related to what is possible to find statically, and the latter to what can be found by inferring types.

For the syntax kind `CallExpression` (Algorithm 2, line 5), it is able to find immutability structures and higher-order functions. The search for immutability, for example, makes

Algorithm 2 Function visit.

```
Input: node – AST node
    if parseComments then
 2:     comments ← commentsParser.parse(node)
    end if
 4: switch node.kind do
        case SyntaxKind.CallExpression
 6:         searchForStructures()                    ▷ immutability and HOFs
            break
 8:     case SyntaxKind.ArrowFunction
        case SyntaxKind.FunctionDeclaration
10:     case SyntaxKind.MethodDeclaration
        case SyntaxKind.FunctionExpression
12:         searchForStructures()    ▷ HOFs, lazy evaluation, recursion and callbacks
            break
14:     case SyntaxKind.ArrayLiteralExpression
        case SyntaxKind.ObjectLiteralExpression
16:         searchForStructures()                    ▷ immutability
            break
18:     case SyntaxKind.NewExpression
            searchForStructures()                    ▷ promises
20:         break
        case SyntaxKind.VariableStatement
22:         searchForStructures()                    ▷ const statements
            break
24: forEachChild(sourceFile, visit)
```

use of the type checker to infer whether the expression inside a node is an object or an array when looking for `Object.freeze`, `Object.assign`, or `Array.slice`.

When looking for non-native higher-order functions, lazy evaluation, recursion, or callbacks (not taking into account `async` methods in any of them), psMiner considers the syntax types `ArrowFunction`, `FunctionDeclaration`, `MethodDeclaration`, and `FunctionExpression` (Algorithm 2, line 8-11). For spread search cases, it takes into account `ArrayLiteralExpression` and `ObjectLiteralExpression` kinds (Algorithm 2, line 14-15). In both, the process is similar: calling a search method of each type of structure and returning the records found.

Next, psMiner has a case for the kind `NewExpression` (Algorithm 2, line 18). We also pass the type checker as a parameter, but it is used to get the `Symbol` of the node. If the name of this `Symbol` is "`Promise`", then psMiner considers that it found an instantiation of a promise and returns its record.

To identify `const` statements, it uses `VariableStatement` nodes (Algorithm 2, line 21). psMiner temporarily creates a file from the node text and checks if the node flag is the `Const` enum in the first statement. If so, we assume that psMiner has found a `const`

Table 8 – Syntax node kinds.

Syntax node kind	Concepts
CallExpression	Immutability, functions as values (higher-order functions)
ArrowFunction, FunctionDeclaration, MethodDeclaration, FunctionExpression	Lazy evaluation, recursion, functions as values (higher-order functions and callbacks)
ArrayLiteralExpression, ObjectLiteralExpression	Immutability
NewExpression	Functions as values (promises)
VariableStatement	Immutability

Source: the author (2022)

statement. In Table 8, we summarize each used syntax node kind relating them to their respective concepts.

Finally, in the third step, the parser calls the `visit` method recursively until there are no more child nodes.

5.2.4 Exporting

After parsing the structures, psMiner exports what data the previous step had returned. We created two functions for this: `exportRecords` and `exportComments`. The first one receives as parameters the authored date and `SHA` from revision commit, the list of found records, the name of the repository, and a flag to determine whether it is necessary to remove the exporting file if it already exists. The tool exports this data to a CSV file. The second function takes a list of records, a list of comments, and the repository's name where the tool found the comments. In this function, the tool filters all comments inside the list of records to compare them with the comments list. This way, psMiner can identify which comments are related to functional programming structures or not. psMiner removes duplicate comments from the filtered list before this comparison. After the comparison, the data is persisted in a CSV file, recording a list of comments with additional information such as the name of the repository they belong to, whether they are from a functional programming structure or not, and which functional programming concept they are and which structure.

5.2.5 Sampling

Seeking to validate the parsing step, we created two functions to generate functional programming structure samples. The first function randomly chooses one of the available repositories, loads the list of structures from its main revision, randomly chooses one of these structures, and repeats this process until it finds 384 samples. This approach finds structures randomly but may miss a structure that has rare occurrences. Therefore, we chose to create a function with another approach.

The second function starts by loading all found structures into their respective main revisions in all repositories. psMiner then randomly chooses up to five structures of each type and stores them in a list. Then, it groups the list by repository and sends the list of files where the structures are to the parser. Finally, psMiner exports the data of each structure in a CSV file so that they can also be manually evaluated.

5.3 TOOL CUSTOMIZATION

Although psMiner is not a framework and was not designed with extensibility in mind, it is possible to customize some functionality of it. Some examples are presented as follows.

New types of preprocessing. A new file can be generated from SeArt for pre-processing. For this, it is only necessary that the name of the exported file is `results.csv`, that it is in the `data` folder, and that the command `npm run cloc` has been run to generate the `cloc.csv` file.

Other JavaScript-based programming languages. It is also possible to change the types of languages taken into account during the analysis. For that, in the `getCLOC` function in `src/utils.ts`, the option `--include-lang` must change for the new languages to be analyzed (and that cloc supports).

Topics to be removed. There is a set of words used to remove repositories that are documentation or guides. These words can also be changed in the `filterReposByTopic` function inside `src/extractor/repos.ts`.

GitHub labels. The terms used when searching for labels can also be modified. Inside the `exportBugLabels` function in `src/extractor/labels.ts` are the rules and terms used to filter labels.

Number of samples. Another option available is to generate a different number of samples of the concepts and programming structures sought. In this case, it is only necessary to change the value of the constant `NUM_OF_SAMPLES` inside the `generateCommitSample` function in `src/sampler/index.ts`.

Other functional programming concepts. Finally, extending psMiner with new functional programming structures is possible. It is necessary to create a file in `src/parser/structures` with a search function and all the logic to parse inside it. After that, the new structure needs to be added to the `visit` method in both `src/parser/program.ts` and `src/sampler/index.ts`.

6 CONCLUSION

In this work, we tackle the problem of understanding how developers use structures inspired by functional programming in a mainstream multi-paradigm programming language. We analyzed 91 projects amounting to more than 22 million lines of code, measuring the prevalence and significance of these concepts (recursion, immutability, lazy evaluation, and functions as values) from static and temporal perspectives. We also measured the likelihood of bug-fixing commits removing uses of these concepts and their association with the presence of code comments. In this chapter, we present our main findings (Section 6.1), implications (Section 6.2), and avenues for future works (Section 6.3).

6.1 MAIN FINDINGS

Our investigation has revealed that the projects employ functional programming structures intensively: they occur, on average, once for every 46.65 lines of code, and more than 44% of all the lines of code in these projects are related in some way to these structures. Furthermore, there is some evidence that their adoption is growing intensively. Immutability- and lazy evaluation-related structures have exhibited growths of 255% and 107% along with the evolution of the analyzed projects. We also found that functional programming structures, except the immutability-related ones, tend to be removed less often in bug-fixing commits than in non-bug-fixing ones. For callbacks & promises and recursion, occurrences of these concepts are 17.38% and 37.77% less likely to be removed in bug-fixing commits, respectively. Finally, we did not find a correlation between comment size and the use of functional structures.

6.2 IMPLICATIONS

Our findings highlight that it is possible to say that developers who work with primarily written JavaScript open-source projects can benefit from using the structures we have investigated, especially by knowing that they are less bug-prone. However, as we have shown, the structures related to immutability suggest the opposite, so in this case, caution is necessary mainly because their use has grown. We also demonstrated that mixing different paradigms is something that developers internalize in such a way that functional programming concepts are used very often in the analyzed projects.

It is also important to note that researchers now have a parameterized tool to download and handle structures related to JavaScript-based languages from repositories on GitHub. Although it was not designed with customization, the tool is easily expandable and adaptable. In addition, we managed to reduce the lack of knowledge in the area and

also demonstrated that there is room for research related to functional programming in multi-paradigm languages such as JavaScript.

6.3 FUTURE WORKS

There are some ideas for future works that we believe are relevant and that would positively add to the work.

One of these ideas would be the addition of other functional programming concepts such as pattern matching, pure functions, or indirect recursion. Also, it is essential to understand why immutability concepts had different results for bug-fixing commits and why thunks are so prevalent. Furthermore, we would like to analyze if there are cultural differences between code written in TypeScript, JavaScript Web, and JavaScript for Node.js applications. It would undoubtedly make our method more robust and mature.

We believe it would be beneficial to increase the number of analyzed projects. With more time and computational resources, compelling results may be found from the repositories not analyzed by this project.

Specifically, regarding the analysis of the removal of functional programming structures in bug-fix commits, we believe that it is worth checking further whether a structure was removed in the comparison between commits. It would imply comparing not only the number of structures from each commit but also checking if the structure was moved to another file, for example.

Analyzing whether there is any relationship between SATD comments and the use of functional programming is another dimension for future work.

Still, concerning the analysis made in the comments, it would be interesting to conduct a study with developers to assess the length of comments associated with functional programming structures. We believe this would bring more confidence to our results.

Finally, we also would like to modularize the psMiner better, making it more customizable and easy to adapt for use in other studies. It would include making the code available for external collaboration in some repository on GitHub, for example.

REFERENCES

ALVES, F.; OLIVEIRA, D.; MADEIRAL, F.; CASTOR, F. *Functional JavaScript data and tools*. Zenodo, 2022. Disponível em: <https://doi.org/10.5281/zenodo.6425005>.

ALVES, F.; OLIVEIRA, D.; MADEIRAL, F.; CASTOR, F. *On the Bug-proneness of Structures Inspired by Functional Programming in JavaScript Projects*. arXiv, 2022. Disponível em: <https://arxiv.org/abs/2206.08849>.

AMAN, H.; AMASAKI, S.; SASAKI, T.; KAWAHARA, M. Lines of comments as a noteworthy metric for analyzing fault-proneness in methods. *IEICE Trans. Inf. Syst.*, v. 98-D, n. 12, p. 2218–2228, 2015.

BACKUS, J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, Association for Computing Machinery, New York, NY, USA, v. 21, n. 8, p. 613–641, aug 1978. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/359576.359579>.

BAVOTA, G.; RUSSO, B. A large-scale empirical study on self-admitted technical debt. In: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2016. p. 315–326.

BECK, K. *Extreme Programming Explained: Embrace Change*. 2nd. ed. [S.l.]: Addison-Wesley, 2004.

BRADY, S. *Immutability and Design Patterns in Ruby*. 2021.

BURSTALL, R. M.; MACQUEEN, D. B.; SANNELLA, D. T. Hope: An experimental applicative language. In: *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*. New York, NY, USA: Association for Computing Machinery, 1980. (LFP '80), p. 136–143. ISBN 9781450373968. Disponível em: <https://doi.org/10.1145/800087.802799>.

BUSE, R. P.; WEIMER, W. R. A metric for software readability. In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2008. (ISSTA '08), p. 121–130. ISBN 9781605580500. Disponível em: <https://doi.org/10.1145/1390630.1390647>.

CAMPOS, U. F.; SMETHURST, G.; MORAES, J. P.; BONIFáCIO, R.; PINTO, G. Mining rule violations in javascript code snippets. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2019. p. 195–199.

CARTER, J. K.; ALNAELI, S. M.; VAZ, W. S. Empirically examining the quality of source code in engineering software systems. In: *2018 IEEE International Conference on Electro/Information Technology, EIT 2018, Rochester, MI, USA, May 3-5, 2018*. [S.l.]: IEEE, 2018. p. 641–644.

CHURCH, A. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, John Hopkins University Press, v. 58, n. 2, p. 345–363, 1936.

CZAPLICKI, E. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University*, v. 30, 2012.

DABIC, O.; AGHAJANI, E.; BAVOTA, G. Sampling Projects in GitHub for MSR Studies. In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2021. p. 560–564.

EASTERBROOK, S.; SINGER, J.; STOREY, M.-A.; DAMIAN, D. Selecting empirical methods for software engineering research. In: *Guide to advanced empirical software engineering*. [S.l.]: Springer, 2008. p. 285–311.

FANCHER, D. *The Book of F#: Breaking Free with Managed Functional Programming*. [S.l.]: No Starch Press, 2014.

FARD, A. M.; MESBAH, A. Jsnose: Detecting javascript code smells. In: IEEE. *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. [S.l.], 2013. p. 116–125.

FIGUEROA, I.; LEGER, P.; FUKUDA, H. Which monads haskell developers use: An exploratory study. *Science of Computer Programming*, v. 201, p. 102523, 2021. ISSN 0167-6423. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167642320301313>.

FOWLER, M.; BECKER, P.; OPDYKE, W.; BRANT, J.; ROBERTS, D.; BECK, K. *Refactoring: Improving the Design of Existing Code*. 2nd. ed. [S.l.]: Pearson Education, 2019.

GALLABA, K.; HANAM, Q.; MESBAH, A.; BESCHASTNIKH, I. Refactoring asynchrony in javascript. In: *Proceedings of the IEEE 33rd International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2017. p. 353–363.

GALLABA, K.; MESBAH, A.; BESCHASTNIKH, I. Don't Call Us, We'll Call You: Characterizing Callbacks in JavaScript. In: *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.: s.n.], 2015. p. 1–10.

GOPSTEIN, D.; ZHOU, H. H.; FRANKL, P.; CAPPOS, J. Prevalence of confusing code in software projects: atoms of confusion in the wild. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. [S.l.: s.n.], 2018. p. 281–291.

GORDON, M. J.; MILNER, A. J.; WADSWORTH, C. P. *Edinburgh LCF: a mechanised logic of computation*. [S.l.]: Springer, 1979.

HANAM, Q.; BRITO, F. S. d. M.; MESBAH, A. Discovering bug patterns in javascript. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2016. (FSE 2016), p. 144–156. ISBN 9781450342186. Disponível em: <https://doi.org/10.1145/2950290.2950308>.

HERBOLD, S.; TRAUTSCH, A.; LEDEL, B.; AGHAMOHAMMADI, A.; GHALEB, T. A.; CHAHAL, K. K.; BOSSENMAIER, T.; NAGARIA, B.; MAKEDONSKI, P.; AHMADABADI, M. N.; SZABADOS, K.; SPIEKER, H.; MADEJA, M.; HOY, N.; LENARDUZZI, V.; WANG, S.; RODRíGUEZ-PéREZ, G.; COLOMO-PALACIOS, R.; VERDECCHIA, R.; SINGH, P.; QIN, Y.; CHAKROBORTI, D.; DAVIS, W.; WALUNJ, V.; WU, H.; MARCILIO, D.; ALAM, O.; ALDAEEJ, A.; AMIT, I.; TURHAN, B.; EISMANN, S.; WICKERT, A.-K.; MALAVOLTA, I.; SULIR, M.; FARD, F.; HENLEY, A. Z.; KOURTZANIDIS, S.; TUZUN, E.; TREUDE, C.; SHAMASBI, S. M.;

PASHCHENKO, I.; WYRICH, M.; DAVIS, J.; SEREBRENIK, A.; ALBRECHT, E.; AKTAS, E. U.; STRüBER, D.; ERBEL, J. A fine-grained data set and analysis of tangling in bug fixing commits. In: *Accepted in Empirical Software Engineering*. [S.l.: s.n.], 2021.

HINSEN, K. The promises of functional programming. *Computing in Science & Engineering*, IEEE, v. 11, n. 4, p. 86–90, 2009.

HUDAK, P. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, Association for Computing Machinery, New York, NY, USA, v. 21, n. 3, p. 359–411, sep 1989. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/72551.72554>.

HUDAK, P.; HUGHES, J.; JONES, S. P.; WADLER, P. A history of haskell: Being lazy with class. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2007. (HOPL III), p. 12–1–12–55. ISBN 9781595937667. Disponível em: <https://doi.org/10.1145/1238844.1238856>.

HUGHES, J. Why Functional Programming Matters. *The Computer Journal*, v. 32, n. 2, p. 98–107, 1989.

HUNT, A.; THOMAS, D. *The Pragmatic Programmer*. [S.l.]: Addison-Wesley, 1999.

KAMPS, S.; HEEREN, B.; JEURING, J. Assessing the quality of evolving haskell systems by measuring structural inequality. In: *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*. [S.l.]: ACM, 2020. p. 67–79.

KEREKI, F. *Mastering JavaScript Functional Programming - Second Edition*. [S.l.]: Packt, 2020.

KRISHNA, R.; AGRAWAL, A.; RAHMAN, A.; SOBRAN, A.; MENZIES, T. What is the connection between issues, bugs, and enhancements? In: IEEE. *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. [S.l.], 2018. p. 306–315.

LIPOVACA, M. *Learn you a haskell for great good!: a beginner's guide*. [S.l.]: no starch press, 2011.

LUBIN, J.; CHASINS, S. E. How statically-typed functional programmers write code. *Proc. ACM Program. Lang.*, v. 5, n. OOPSLA, p. 1–30, 2021.

MALDONADO, E. D. S.; ABDALKAREEM, R.; SHIHAB, E.; SEREBRENIK, A. An empirical study on the removal of self-admitted technical debt. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2017. p. 238–248.

MAZINANIAN, D.; KETKAR, A.; TSANTALIS, N.; DIG, D. Understanding the use of lambda expressions in java. *Proc. ACM Program. Lang.*, ACM, New York, NY, USA, v. 1, n. OOPSLA, oct 2017.

MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 3, n. 4, p. 184–195, apr 1960. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/367177.367199>.

MILNER, R.; TOFTE, M.; MACQUEEN, D. *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997. ISBN 0262631814.

MUNAIAH, N.; KROH, S.; CABREY, C.; NAGAPPAN, M. Curating github for engineered software projects. *Empirical Software Engineering*, Springer, v. 22, n. 6, p. 3219–3253, 2017.

RICHARDS, G.; HAMMER, C.; BURG, B.; VITEK, J. The eval that men do - A large-scale study of the use of eval in javascript applications. In: *Proceedings of the 25th European Conference on Object-Oriented Programming*. Lancaster, UK: Springer, 2011. (Lecture Notes in Computer Science, v. 6813), p. 52–78.

ROY, P. V. et al. Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*, IRCAM/Delatour France, v. 104, p. 616–621, 2009.

SABOURY, A.; MUSAVI, P.; KHOMH, F.; ANTONIOL, G. An empirical study of code smells in javascript projects. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.: s.n.], 2017. p. 294–305.

SABOURY, A.; MUSAVI, P.; KHOMH, F.; ANTONIOL, G. An empirical study of code smells in javascript projects. In: *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.: s.n.], 2017. p. 294–305.

SATERNOS, C. *Client-Server Web Apps with JavaScript and Java: Rich, Scalable, and RESTful*. [S.l.]: " O'Reilly Media, Inc.", 2014.

SCOTT, M. L. *Programming Language Pragmatics*. 4. ed. Amsterdam: Morgan Kaufmann, 2016. ISBN 978-0-12-410409-9.

SIDDIQUI, T.; AHMAD, A. Data mining tools and techniques for mining software repositories: A systematic review. In: _____. [S.l.: s.n.], 2018. p. 717–726. ISBN 978-981-10-6619-1.

SLIWERSKI, J.; ZIMMERMANN, T.; ZELLER, A. When do changes induce fixes? In: *Proceedings of the 2005 International Workshop on Mining Software Repositories*. [S.l.: s.n.], 2005. p. 1–5.

STEIDL, D.; HUMMEL, B.; JUERGENS, E. Quality analysis of source code comments. In: *2013 21st International Conference on Program Comprehension (ICPC)*. [S.l.: s.n.], 2013. p. 83–92.

TURNER, D. A. Some history of functional programming languages. In: SPRINGER. *International Symposium on Trends in Functional Programming*. [S.l.], 2012. p. 1–20.

WATT, D. A. *Programming language design concepts*. [S.l.]: John Wiley & Sons, 2004.

WIRTH, N. A brief history of software engineering. *IEEE Annals of the History of Computing*, IEEE, v. 30, n. 3, p. 32–39, 2008.

XU, Y.; WU, F.; JIA, X.; LI, L.; XUAN, J. Mining the use of higher-order functions. *Empirical Software Engineering*, Springer, v. 25, n. 6, p. 4547–4584, 2020.