Towards a Catalog of Refactorings for Elixir

Lucas Francisco da Matta Vegi
Department of Computer Science
Federal University of Minas Gerais (UFMG)
Belo Horizonte, Brazil
lucasvegi@dcc.ufmg.br

Marco Tulio Valente
Department of Computer Science
Federal University of Minas Gerais (UFMG)
Belo Horizonte, Brazil
mtov@dcc.ufmg.br

***Abstract*—Elixir is an emerging functional programming language that is gaining popularity in the industry. However, to the best of our knowledge, no study has yet presented a specialized catalog of refactoring strategies specifically tailored for this language. Therefore, this paper aims to address this research gap by conducting a systematic literature review to explore whether there are existing refactoring strategies, compatible with Elixir, that have been proposed for other functional languages or languages that served as inspiration for the development of Elixir. Our preliminary results indicate that there are 54 refactoring strategies compatible with Elixir code, thus forming a comprehensive catalog of refactoring techniques tailored specifically for this language. To illustrate the application of these refactoring strategies, we have provided code examples that show the transformations resulting from each refactoring. Additionally, these refactorings have been categorized into three distinct groups based on the specific programming features required for the respective code transformations.**

***Index Terms*—refactoring, Elixir, functional programming**

I. Introduction

Elixir is a modern functional programming language renowned for its efficient performance in parallel and distributed computing environments [1]. Conceived in 2012, Elixir draws inspiration from a blend of various programming languages. The language adopts a Ruby-based syntax, which contributes to its user-friendly nature. Similar to Haskell, Elixir employs immutable data structures, rendering it suitable for concurrent programming. Although they possess different syntaxes and features, Elixir integrates with Erlang, enabling interoperability between the two languages as they both run on BEAM, Erlang's virtual machine [2]. Moreover, Elixir incorporates polymorphic and extensible features inherited from Clojure. As a result, the adoption of Elixir is gaining traction, with over 300 companies worldwide using the language, including Discord, Heroku, and PepsiCo.[1]

Just as in any programming language, it is natural to expect Elixir developers to seek design improvements in their codebase through refactoring strategies, which are code transformations that do not alter behavior [3]. The term was first proposed by Opdyke [4] and popularized by Fowler in his well-known catalog containing 72 refactorings for object-oriented code [3].

According to Bordignon and Silva [5], an increasing number of developers are using functional languages in the industry. Conversely, Abid et al. [6] have shown the scarcity of studies conducted on refactoring for these languages. Furthermore, to the best of our knowledge, no study has yet investigated refactorings specifically tailored for Elixir.

To address this research gap, the present paper provides preliminary documentation on refactorings compatible with Elixir that have been proposed for other functional languages or languages that served as inspiration for the creation of Elixir, such as Ruby. The final goal of our research is to propose a specific catalog of refactorings for Elixir. To achieve this goal and considering that refactoring for functional languages has been previously discussed in the literature [7], [8], we opted to conduct a Systematic Literature Review (SLR) to identify such refactoring strategies.

The refactorings that were identified in our SLR study were analyzed and adapted for Elixir when compatible. While not all adaptations are straightforward, we deliberately chose a simple example to illustrate this process. In Erlang, there are several built-in higher-order functions for working with lists (e.g., `lists:map/2`, `lists:foreach/2`, and `lists:filter/2`). In Elixir's `Enum` module there are functions equivalent to those in Erlang. However, sometimes it is recommended to replace calls of such functions by list comprehensions. Thus, it is possible to adapt to Elixir the refactoring Transform to List Comprehension, originally proposed by Sagonas and Avgerinos [9] for Erlang:

```
1  # Before refactoring
2  def square(list) do
3    Enum.map(list, &(&1 * &1))
4  end
5
6  # After refactoring
7  def square(list) do
8    for x <- list, do: x * x
9  end
```

The adaptation for Elixir only involves syntactic changes. In the example, line 3 is equivalent to `lists:map(fun(x) -> x * x end, list)` in Erlang, and the list comprehension in line 8 is equivalent to `[x * x || x <- list]`. Although simple, this refactoring results in code that is more declarative and easier to read [10].

As part of our research effort, we also provide code examples in Elixir for all refactorings compatible with the language. Besides, as the side conditions that need to be validated to apply a refactoring depend on the characteristics of each

[1]https://elixir-companies.com/en

language [11], our catalog includes tailored documentation of side conditions for Elixir.

Our contributions are twofold: (1) we cataloged 54 refactorings that are compatible with Elixir code, categorizing them into three distinct groups (*traditional refactorings*, *functional refactorings*, and *Erlang-specific refactorings*); and (2) we provided documentation with code examples and tailored side conditions that can support the implementation of automated refactoring tools for Elixir in the future. Indeed, no robust, up-to-date, and widely adopted refactoring tool is available for Elixir, as shown by an exploratory search conducted by us on Hex,[2] Elixir's package manager.

The remainder of this paper is organized as follows. Our research question and methods are presented in Section II. In Section III, we present and discuss our preliminary results. Potential threats to validity are listed in Section IV. In addition, we present related work in Section V. Finally, we conclude the paper and present suggestions for future work in Section VI.

II. Methodology

Due to Elixir being a relatively recent programming language, there is a limited number of studies that explore software engineering and quality attributes of Elixir-based software. Specifically, to the best of our knowledge, no work investigates or catalogs specific refactoring strategies for this language. Therefore, aiming to provide a preliminary catalog of refactoring for Elixir, we initially decided to conduct a Systematic Literature Review (SLR) to answer the following research question:

- **RQ:** What are the refactorings reported in the related literature that apply to Elixir?

Our definition of related literature pertains to papers discussing functional languages or other programming languages that have influenced the development of Elixir, such as Ruby.

We based on guidelines for conducting SLR studies in software engineering [12]–[14] to identify, examine, and evaluate scientific papers related to refactoring in functional programming languages. Figure 1 summarizes the steps we followed in our SLR. Next, we also detail these steps.

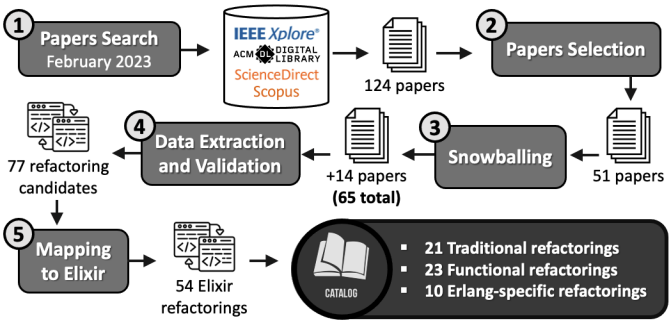

Fig. 1. Overview of our methodology.

1) Papers Search: Our objective was to search for articles on refactoring in well-known functional programming languages or in languages that inspired the creation of Elixir. To build a query string compatible with this objective, we followed five steps proposed by Kitchenham and Charters [14] to find all relevant search terms for a query string:

- We extracted the main terms using our RQ as a basis;
- We added variations and synonyms of the main terms;
- We conducted preliminary searches in order to refine the query string, aiming to obtain comprehensive and accurate results;
- We used "OR" operators to concatenate variations of related terms. Additionally, the "AND" operator was used to combine the main terms;
- We adapted the query string to formats compatible with the limitations of the digital libraries used.

After these steps, we define the following query string:

```
("Refactoring") AND ("Functional Language" OR "Functional Paradigm" OR "Elixir" OR "Erlang" OR "Haskell" OR "Clojure" OR "Ruby" OR "Lisp")
```

As Elixir is a compilation of features from various other languages such as Erlang, Haskell, Clojure, and Ruby [1], we added these terms concatenated in our query string. Lisp was included due to its historical importance and popularity, as in addition to being the first functional language created [27], it is also widely known and used.[3]

In order to carry out the search, we opted for four reputable digital libraries that index software engineering publications relevant to our research question: IEEE Xplore digital library,[4] ACM digital library,[5] ScienceDirect,[6] and Scopus.[7] The final search was conducted in February 2023, when these terms were searched in the abstracts of the papers retrieved from these digital libraries.

2) Papers Selection: A total of 124 papers were found, which we refer to as P1 to P124. The abstracts of all papers were read by the first author, who is an expert Elixir developer, to select only those related to our research question. In addition, 37 duplicates were eliminated as these papers were returned by more than one digital library. After this step, 51 papers were selected. Our replication package includes the complete compilation of papers, also encompassing those that are duplicated or not-related to our research question.

3) Snowballing: In order to add new studies to our knowledge base, we used a snowballing process based on the recommendations made by Wohlin [15]. The first author analyzed the papers referenced by the 51 papers selected in the previous step and selected those that are relevant for our research question. Later, the abstracts of these papers were read, and 14 of them, which we refer to as P125 to P138, were also selected for further analysis.

[2]https://hex.pm/

[3]https://www.tiobe.com/tiobe-index/

[4]http://ieeexplore.ieee.org

[5]https://dl.acm.org

[6]http://www.sciencedirect.com

[7]https://www.scopus.com

4) Data Extraction and Validation: The 65 papers selected in the previous steps were fully read and analyzed by the first author, aiming to extract refactoring candidate strategies that also apply to Elixir. These refactoring candidates were validated by the second author. Only eight out of 65 papers did not have refactoring candidates extracted. In total, 77 refactoring candidates were extracted by the first author and validated by the second one.

5) Mapping to Elixir: Finally, all 77 refactoring candidates were analyzed by the first author to identify those compatible with Elixir's features and then adapt them to the language's syntax and semantics. Due to language feature incompatibilities, 23 out of 77 refactoring candidates were not selected to compose our catalog of refactoring for Elixir. Some of these refactorings involve, for example, Haskell's static data typing system, which cannot be reproduced in Elixir.

A total of 54 of these candidates had their compatibility with Elixir confirmed, thus code examples that illustrate the transformations were created for the catalog. Besides, we provide tailored documentation of side conditions of each refactoring in Elixir. These refactorings were also categorized into three groups: *Traditional*, which are mainly based on Fowler's catalog [3]; *Functional*, which use programming characteristics of functional languages, such as pattern matching and higher-order functions; and *Erlang-Specific*, which use features unique to the Erlang ecosystem (e.g., OTP, `typespecs`, and `behaviours`).

III. Results

In our systematic literature review, we found 54 refactorings compatible with Elixir, with 21 of them categorized as *Traditional refactorings* (see Table I), 23 as *Functional refactorings* (see Table II), and 10 as *Erlang-specific refactorings* (see Table III). Due to space limitations, we present in these tables only the refactorings that were cited in at least two different papers. For this reason, 12 refactoring strategies found in a single paper are not presented in the tables. However, in our replication package, all 54 refactorings are listed.

According to Murphy-Hill et al. [16] and Golubev et al. [17], Renaming an identifier is the most frequently performed refactoring by developers. As shown in Table I, this fact can also be observed in our study, as Renaming an identifier was not only the most common *Traditional refactoring* but also the most common refactoring among all 54 in our catalog, cited in 35 papers. In Elixir, code identifiers can be `functions`, `modules`, `macros`, `variables`, `map/struct fields`, `modules aliases`, and `modules attributes`. When the name of an identifier does not clearly convey its purpose, it should be renamed to improve readability.

As shown in Table II, the most extracted refactoring from the *Functional refactorings* category in our study was Generalise a function definition. In total, 22 papers cited this refactoring in the context of Erlang or Haskell. It proposes using higher-order functions to eliminate duplicated code among functions, as shown in the following example:

TABLE I
Traditional refactorings

Refactoring	Description	#
Rename an identifier	Renames identifiers (e.g., `functions`, `modules`, `macros`, etc.) to clearly convey their purpose	35
Extract function	Creates a new function by extracting code snippets from existing functions	26
Folding against a function definition	Replaces duplicated code with a call to an existing equivalent function	21
Moving a definition	Moves a definition (e.g., `function` or `struct`) from one module to another	16
Add or remove a parameter	Used when it is necessary additional information from the callers of a function or the opposite situation	14
Introduce or remove a duplicate definition	Temporarily duplicates an identifier to test a change on it without losing the original	11
Inline function	Replaces all calls to a function with its body	10
Grouping parameters in tuple	Groups some of a function's parameters into a `tuple` (a.k.a. Introduce Parameter Object)	10
Reorder parameter	Improves the readability of a function's interface by reordering its parameter list	10
Remove dead code	Removes definitions that are not being used	9
Remove import attributes	Removes `import` directives by replacing all calls to imported functions with fully-qualified calls	8
Merge expressions	Breaks large expressions into smaller parts and assign them to local variables	6
Simplifying nested conditional statements	Eliminates unnecessary nested conditional statements	5
Introduce import	Replaces fully-qualified function calls with calls that use only the name of the imported functions	4
Splitting a large module	Splits a `module` into several new ones, moving to each new `module` the attributes and functions with related purposes	4
Introduce overloading	Creates variations of a function using the same name	2
Temporary variable elimination	Eliminates variables that only store results to return or intermediate values	2

#: The number of papers citing the refactoring.

```
1  # Before refactoring
2  def foo(list) do
3    list_comprehension = for x <- list, do: x * x
4    Enum.map(list_comprehension, &(&1 * 3))
5  end
6
7  def bar(list) do
8    list_comprehension = for x <- list, do: x + x
9    Enum.filter(list_comprehension, &(rem(&1, 4) == 0))
10 end
11
12 # After refactoring
13 def gen(list, gen_op, trans_op, trans_args) do
14   list_comprehension = for x <- list, do: gen_op.(x,x)
15   trans_op.(list_comprehension, trans_args)
16 end
17
18 def foo(list) do
19   gen(list, &Kernel.*/2, &Enum.map/2, &(&1 * 3))
20 end
21
22 def bar(list) do
23   gen(list, &Kernel.+/2, &Enum.filter/2, &(rem(&1,4) == 0))
24 end
```

Before Generalise a function definition, we have two similar functions. The `foo/1` takes a list and transforms it into two steps. First, it squares each of its elements (line 3)

and then multiplies each element by three (line 4), returning a new list. Similarly, `bar/1` receives a list, doubles the value of each element (line 8), and then returns a new list containing only the elements divisible by four (line 9).

Although `foo/1` and `bar/1` transform lists in different ways, they have duplicated structures. This refactoring generalizes these functions by introducing a new function `gen/4` (line 13). Thus, the bodies of `foo/1` and `bar/1` are replaced with calls to `gen/4` (lines 19 and 23). Note that `gen/4` is a higher-order function as its last three parameters are also functions.

TABLE II
FUNCTIONAL REFACTORINGS

Refactoring	Description	#
Generalise a function definition	When different functions have equivalent expressions, these expressions can be generalized into a new higher-order function	22
Turning anonymous into local functions	Transforms duplicated anonymous function (`lambda`) into a new named function	9
From tuple to struct	Transforms `tuples` into `structs`, thus providing a more abstract interface for the data and improving code readability	7
Introduce Enum.map/2	Replaces a list expression in which each element is generated by calling the same function with a call to `Enum.map/2`	6
Merging multiple definitions	Functions that have common code are merged into a new function that performs the processing done by the original ones	6
Splitting a definition	This refactoring is the inverse of MERGING MULTIPLE DEFINITIONS	5
Nested list functions to comprehension	Transforms nested calls of `Enum.map/2` and `Enum.filter/2` into a list comprehension (a.k.a. deforestation)	4
Struct field access elimination	Replaces direct access to fields of a `struct` with the use of temporary variables and pattern matching	4
Widen or narrow definition scope	Modifies the scope of functions defined in a nested way	4
Eliminate single branch	Eliminates control statements that have only one possible flow	3
Inline macro substitution	Replaces instances of an unnecessary `macro` with its body	3
Introduce pattern matching over a parameter	Uses pattern matching and multi-clause functions to replace conditional statements (i.e., `if`, `unless`, `cond`, and `case`)	3
Transform to list comprehension	Transforms calls to `Enum.map/2` and `Enum.filter/2` into list comprehensions	3
Equality guard to pattern matching	Replaces a variable extracted from a `struct field`, that is only used in a `guard` equality comparison, with pattern matching	2
Function clauses to/from case clauses	Transforms a multi-clause into a single-clause function, mapping function clauses into clauses of a `case` statement	2
List comprehension simplifications	Inverse of TRANSFORM TO LIST COMPREHENSION and NESTED LIST FUNCTIONS TO COMPREHENSION	2
Static structure reuse	Eliminates recreations of identical `tuples` or `lists` by assigning them to variables	2
Struct guard to matching	Transforms a guard that unnecessarily use `is_struct/1` or `is_struct/2` call, into explicit pattern matching	2
Transform a body-recursive function to a tail-recursive	Converts a body-recursive function into a tail-recursive one to improve performance	2
Transforming list appends and subtracts	Transforms calls to the `Enum.concat/2` and `Enum.reject/2` into uses of the `Kernel.++/2` and `Kernel.-/2` operators	2

#: The number of papers citing the refactoring.

Since the scope of the *Erlang-specific refactorings* is narrower, this category had a smaller number of extractions compared to the others. As shown in Table III, the most extracted refactorings from this category were ADD TYPE DECLARATIONS AND CONTRACTS and INTRODUCE/REMOVE CONCURRENCY, both found in four papers. These refactorings respectively make use of the features `typespecs` and `GenServer` from the Erlang ecosystem.

TABLE III
ERLANG-SPECIFIC REFACTORINGS

Refactoring	Description	#
Add type declarations and contracts	Uses `typespecs` to create Elixir's custom types, thereby naming recurring data	4
Introduce/remove concurrency	Uses concurrent processes (e.g., `GenServer`) to achieve a more optimal mapping between Elixir's processes and parallel activities	4
From defensive to non-defensive programming style	Transforms defensive-style error handling code into Elixir's `supervised processes` (a.k.a. "let it crash style")	3
From meta to normal function application	Replaces calls to `apply/3` with calls to functions that have modules, names, and parameter lists defined at compile time	3
Generate function specification	Uses `typespecs` to specify the types of parameters and of the return value of a function	2

#: The number of papers citing the refactoring.

Similar to Fowler's catalog [3], our catalog of refactorings for Elixir can aid developers in improving existing code design in this language, offsetting the degradation experienced by software over time. Future research can explore real Elixir systems to expand this catalog and quantify the prevalence of these refactorings. Additionally, investigations into removing code smells [25], [26] in Elixir systems can be guided by our catalog of refactorings. Finally, automated tools for executing our refactoring strategies can be proposed.

> **RQ answer:** In total, 54 refactorings reported in the literature for other functional languages and/or languages that served as inspiration for the creation of Elixir, are compatible with this language. More details about each of these refactorings, including code examples and tailored side conditions, are available at **https://github.com/lucasvegi/Elixir-Refactorings**

IV. THREATS TO VALIDITY

Since we chose to retrieve papers only in digital libraries that support searching for query string terms exclusively in the abstracts of publications, SpringerLink[8] was not utilized by us. Therefore, there is a risk that important works may not have been retrieved in our review. To mitigate this threat to external validity, we incorporated a snowballing step into our methods, enabling the retrieval of relevant papers published in a broader range of venues.

The threat to internal validity relates to factors that could introduce biases into our results. To mitigate this, all steps of

[8]https://link.springer.com

paper selection, data extraction, and mapping to Elixir were independently validated by the second author, thus minimizing individual perception biases.

Finally, the main threat to construct validity concerns the format of the query string. In a systematic literature review, there is a risk that important results may be overlooked due to the omission of specific combinations of terms in the query string. To mitigate this threat, we followed five steps proposed by Kitchenham and Charters [14] to define relevant search terms for the query string.

V. Related Work

To the best of our knowledge, this is the first work that catalogs refactorings for Elixir. However, there are other studies that present refactorings and tools for functional languages.

Li et al. [18]–[20] present an automatic refactoring tool for Erlang called Wrangler. To illustrate the use of this tool, some refactoring strategies for this language are presented. Similarly, Sagonas and Avgerinos [9] and Lövei et al. [21] respectively propose the tools Tidier and RefactorErl, accompanied by refactoring strategies for Erlang. In addition to Erlang, Haskell is also the subject of studies on refactoring. Brown et al. [22] briefly describe a number of refactorings implemented in the Haskell refactorer, HaRe. These refactoring strategies for Haskell and Erlang were compared by Li and Thompson [11], showing that each language has its own constraints and challenges, which therefore justifies the proposition of language-specific refactoring catalogs.

Other SLR studies have examined various areas related to refactoring [6], [23], [24], but none of them specifically focused on functional languages like our work.

VI. Conclusion and Future Work

This paper reported a systematic literature review to extract, adapt, and document 54 refactorings that are compatible with Elixir, thereby proposing a first catalog specifically tailored for this language.

As future work, we intend to expand our catalog through a grey literature review and by mining software repositories on GitHub. Additionally, **we plan to identify Elixir-specific refactorings** by analyzing the catalog of code smells specific to Elixir also proposed by us [25], [26]. This analysis can help us to discover refactoring strategies for eliminating these code smells that are not yet covered in our present study. Next, we also plan to conduct surveys and interviews with Elixir developers to validate our catalog of refactorings. Finally, we plan to dedicate efforts towards the development of a tool that can automatically refactor Elixir code.

Replication Package. We provide the complete dataset used in this paper at https://doi.org/10.5281/zenodo.7999830.

Acknowledgment. This research is supported by a grant from Finbits: http://www.finbits.com.br.

References

[1] S. Juric, Elixir in action, 2nd ed., Manning, 2019.
[2] D. Thomas, Programming Elixir |> 1.6: functional |> concurrent |> pragmatic |> fun, 1st ed., Pragmatic Bookshelf, 2018.
[3] M. Fowler and K. Beck, Refactoring: improving the design of existing code, 1st ed., Addison-Wesley, 1999.
[4] W. F. Opdyke, "Refactoring object-oriented frameworks," PhD thesis. University of Illinois at Urbana-Champaign, USA, 1992.
[5] M. D. Bordignon and R. A. Silva, "Mutation operators for concurrent programs in Elixir," in IEEE Latin-American Test Symposium (LATS), pp. 1-6, 2020.
[6] C. Abid, V. Alizadeh, M. Kessentini, T. N. Ferreira, and D. Dig, "30 years of software refactoring research: a systematic literature review," ArXiv, vol. abs/2007.02194, pp. 1-23, 2020.
[7] H. Partsch and R. Steinbruggen, "Program transformation systems," ACM Computing Surveys, vol. 15, no. 3, pp. 199-236, 1983.
[8] P. Hudak, "Conception, evolution, and application of functional programming languages," ACM Computing Survey, vol. 21, no. 3, pp. 359-411, 1989.
[9] K. Sagonas and T. Avgerinos, "Automatic refactoring of Erlang programs," in 11th ACM SIGPLAN conference on principles and practice of declarative programming (PPDP), pp. 13-24, 2009.
[10] T. Avgerinos and K. Sagonas, "Cleaning up Erlang code is a dirty job but somebody's gotta do it," in 8th ACM SIGPLAN workshop on Erlang, pp. 1-10, 2009.
[11] H. Li and S. Thompson, "Comparative study of refactoring Haskell and Erlang programs," in 6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), pp. 197-206, 2006.
[12] B. Barn, S. Barat, and T. Clark, "Conducting systematic literature reviews and systematic mapping studies," in 10th Innovations in Software Engineering Conference (ISEC), pp. 212-213, 2017.
[13] D. Budgen and P. Brereton, "Performing systematic literature reviews in software engineering," in 28th International Conference on Software Engineering (ICSE), pp. 1051-1052, 2006.
[14] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," 2007.
[15] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in 18th International Conference on Evaluation and Assessment in Software Engineering (EASE), pp. 1-10, 2014.
[16] E. R. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," IEEE Trans. Soft. Eng., vol. 38, no. 1, pp. 5-18, 2012.
[17] Y. Golubev, Z. Kurbatova, E. A. AlOmar, T. Bryksin, and M. W. Mkaouer, "One thousand and one stories: a large-scale survey of software refactoring," in European Soft. Eng. Conf. and Symposium on the Foundations of Soft. Eng. (ESEC/FSE), pp. 1303-1313, 2021.
[18] H. Li, S. Thompson, L. Lövei, Z. Horváth, T. Kozsik, A. Víg, and T. Nagy, "Refactoring erlang programs," in 12th International Erlang/OTP User Conference (EUC), pp. 1-10, 2006.
[19] H. Li, S. Thompson, G. Orosz, and M. Tóth, "Refactoring with wrangler, updated: data and process refactorings, and integration with eclipse," in 7th ACM SIGPLAN workshop on Erlang, pp. 61-72, 2008.
[20] H. Li and S. Thompson, "A domain-specific language for scripting refactorings in Erlang," in Fundamental Approaches to Software Engineering (FASE), J. Lara and A. Zisman (eds.), Lecture Notes in Computer Science, vol. 7212, pp. 501-515, 2012.
[21] L. Lövei, C. Hoch, H. Köllö, T. Nagy, A. N. Víg, D. Horpácsi, R. Kitlei, and R. Király, "Refactoring module structure," in 7th ACM SIGPLAN workshop on Erlang, pp. 83-89, 2008.
[22] C. Brown, H. Li, and S. Thompson, "An expression processor: a case study in refactoring Haskell programs," in Trends in Functional Programming (TFP), R. Page, Z. Horváth, and V. Zsók (eds.), Lecture Notes in Computer Science, vol. 6546, pp. 31-49, 2011.
[23] J. Al Dallal and A. Abdin, "Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: a systematic literature review," IEEE Trans. on Soft. Eng., vol. 44, no. 1, pp. 44-69, 2018.
[24] S. Singh and S. Kaur, "A systematic literature review: refactoring for disclosing code smells in object-oriented software," Ain Shams Engineering Journal, vol. 9, no. 4, pp. 2129-2151, 2018.
[25] L. F. M. Vegi and M. T. Valente, "Understanding code smells in Elixir functional language," Empirical Software Engineering, vol. 28, no. 4, pp. 1-32, 2023.
[26] L. F. M. Vegi and M. T. Valente, "Code smells in Elixir: early results from a grey literature review," in 30th IEEE/ACM International Conference on Program Comprehension (ICPC), pp. 580-584, 2022.
[27] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part I," Commun. ACM, vol. 3, no. 4, pp. 184-195, 1960.