





# Systematic Review of Functional Programming for Mutable States in the Integration of Imperative Systems

Javier S. Sánchez<sup>1</sup> ; Michael D. Huamancaja<sup>2</sup> ; Jorge L. Ruiz<sup>3</sup> ; Luis C. Rada<sup>4</sup> 

<sup>1,2,3,4</sup> Universidad Tecnológica del Perú, Peru, U19209170@utp.edu.pe<sup>1</sup>, U20226656@utp.edu.pe<sup>2</sup>, C02047@utp.edu.pe<sup>3</sup>, C18380@utp.edu.pe<sup>4</sup>

**Abstract**– This systematic literature review (SLR) evaluates the impact of functional programming (FP) on the detection of mutable states, focusing on the integration with imperative systems and APIs. The PICO methodology and PRISMA protocols were used to elaborate this SLR. In this sense, four questions were posed to guide the search and, through the inclusion and exclusion criteria, this work focused on 49 articles directly related to the topic. The findings show that functional languages such as Haskell and SML offer an open window to memory management, concurrency optimization, and improved resilience in distributed systems. Likewise, tools such as monads and lazy evaluation have a positive impact on strengthening the reliability of critical systems in the face of technical challenges. It is concluded that functional programming and mutable state management are effective, but there is a deficiency in the interaction between imperative and functional systems, especially in large-scale industrial applications. It is suggested that the integration of imperative and functional systems be investigated to improve scalability, modularization and reliability in industrial environments.

**Keywords**-- functional language, functional programming, immutability, imperative systems, mutable states.

## I. INTRODUCTION

In the field of Software Engineering, functional programming (FP) has become a key approach for utilizing third-party APIs. However, conventional data transmission techniques have often proven inefficient, leading to significant delays in processes. These inefficiencies not only impede technological advancements but also hinder progress toward achieving the Sustainable Development Goals (SDGs) for 2025-2030. Despite its potential, FP poses numerous challenges, particularly in addressing mutability between paradigms in diverse contexts, such as optimizing memory management in parallel functional programming and translating programs effectively, as highlighted in prior works [3], [4], [11], [18].

FP is not confined solely to the software domain. It plays a pivotal role in supporting cyber-physical systems and infrastructure management [5]. However, the absence of robust techniques to ensure disentanglement and the inherent complexity of functional languages underscore the urgent need for tools to address these challenges. Prior studies have

highlighted critical aspects requiring attention: the trade-off between FP safety and performance [1], [2], opportunities for enhancing Griset's efficiency through better portability optimizations [3], and ongoing debates about the efficacy of the FIP calculus [4]. Additionally, practical applications and variations in optimization conditions demand further exploration to validate current approaches [17],[21],[30],[41],[43],[45].

In this context, a systematic literature review (SLR) is conducted to evaluate the role of functional programming in improving the detection of mutable states during integration with imperative systems. This study employs a hybrid methodology, combining the PICOC framework to construct precise search equations for scientific databases with the PRISMA guidelines to establish rigorous inclusion and exclusion criteria for relevant information. By addressing these challenges, this SLR aims to provide a comprehensive understanding of FP's potential to enhance mutability detection and integration efficiency.

The remainder of this article is organized as follows: the Methodology section describes the hybrid approach combining PICOC and PRISMA frameworks. The Results section presents the findings from the systematic review, followed by a detailed Discussion that interprets these findings in light of the identified challenges and opportunities. Finally, the Conclusion section summarizes the key contributions and outlines directions for future research.

## II. METHODOLOGY

### A. Methods PICO and PRISMA

To carry out this study, the following research question was proposed: How does functional programming improve the detection of mutable states in the integration with imperative systems? It meets the organizational criteria of the PICO strategy, which allows identifying the components of the question (see Table 1) and associating them with keywords (see Table 2), with a view to performing a structural search of scientific literature, which allowed building the search

equation (see Table 3) that was applied to the SCOPUS database.

#### a.1. PICO Method

Then, the methodology used PRISMA was used to guide the detection process. This procedure included the implementation of inclusion and exclusion criteria, as well as the review of titles, abstracts and full texts of studies. Consequently, the studies that met the established criteria were chosen to be used for the next stage of the study: data collection and examination of results. Thus, review questions of the PICO search strategy were created, considering the proposal of questions that allow the recognition of the key components of the information to be collected [50], which can be validated in Table 1.

TABLE I  
SUMMARY OF PICO

Problem	Literature on functional programming
Intervention	Functional programming languages
Context	Application of functional programming in API migration
Result	Detecting mutable states in imperative systems involving API's

TABLE II  
RESEARCH QUESTIONS

RQ	Research question	Motivation
RQ1	What types of problems have been solved by using functional programming in improving the detection of mutable states in integration with imperative systems?	A number of academic publications from the last 5 years on functional programming are identified
RQ2	What types of functional programming languages are used to improve mutable state detection?	Significant works from the last five years on functional programming languages are identified
RQ3	In what type of space are the types of functional programming languages that are used to improve the detection of mutable states being used?	Significant works from the last five years on the application of functional programming are identified
RQ4	What types of results have been obtained with the application of functional programming in the detection of mutable states in the integration of imperative systems?	Significant works from recent years on functional programming are identified
General search equation (total of 883 research articles)		
( TITLE-ABS-KEY ( "functional programming" OR "mutable states" OR "systems integration" OR "error detection" OR "imperative systems" ) AND TITLE-ABS-KEY ( "Detection strategies" OR "Immutability" OR "State detection" OR "Functional languages" OR "Functional programming" OR "Mutable states" ) AND TITLE-ABS-KEY ( "Functional programming languages" OR "Application space" OR "Detection" OR "Mutable states" OR "Improvement" OR "Immutability" ) AND TITLE-ABS-KEY ( improvement OR maintenance OR effect* OR measurement OR calibration OR routine OR automation OR prediction OR standard* ) )		

For the reading, not only titles and summaries were analyzed, but also the introduction and content of each literature. To do so, the criteria shown in Table 3 were considered.

#### a.2. Inclusion and exclusion criteria (PRISMA)

In this order of ideas, 6 inclusion criteria and 5 exclusion criteria were applied, which allowed 883 documents to be related for the development of this review [51]. Therefore, the following inclusion criteria were used:

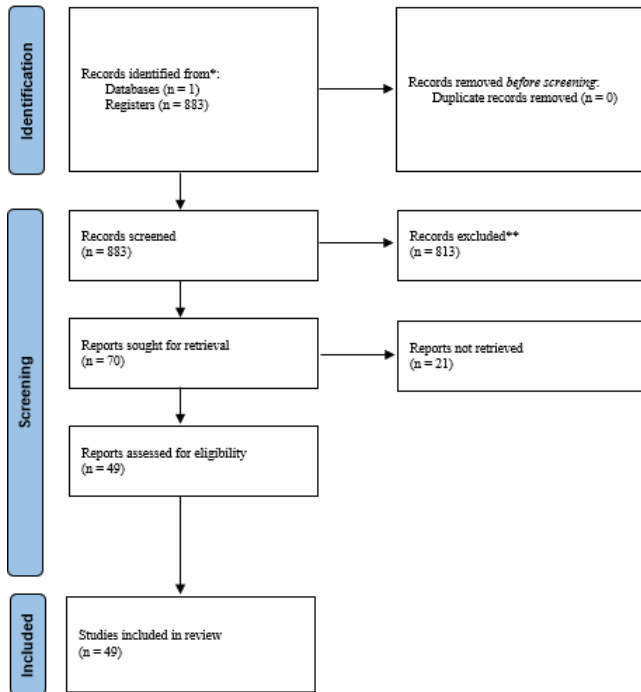
- 1) Research and review articles from 2020 to 2024.
- 2) Research and review articles that are directly related to digital transformation in the optimization of administrative processes.
- 3) Open access research and review articles.
- 4) Research and review articles in English or Spanish.
- 5) For duplicate research and review articles, only one will be taken.

The exclusion criteria used were:

- 1) Research and review articles before 2020.
- 2) Research and review articles that are not directly related to digital transformation in the optimization of administrative processes.
- 3) Non-open access research and review articles.
- 4) Articles research and review in languages other than English or Spanish.
- 5) Articles For duplicate research and review projects, only one will be taken.

Following the established search guidelines, no duplicate articles were found. This search yielded 64 results in SCOPUS. The combination of keywords through Boolean operators resulted in 0 duplicate articles, which were identified by removing them from the data source for further selection. According to these guidelines, 70 possible articles were considered for the SLR, excluding 813 records. In addition, a manual review was carried out with the aim of identifying relevant articles for the systematic review, resulting in 49 results. By reading the abstracts, 65 articles were identified that were used in the results of this SLR. Table 3 presents the PRISMA flow diagram used to select scientific articles that meet the guidelines.

TABLE III  
PRISM DIAGRAM



To carry out the research, the following question was asked: What type of problems have been solved with the use of functional programming in improving the detection of mutable states in the integration with imperative systems? This complies with the organization of the PICO methodology that allows identifying the components of the question (see Table 1) and associating them with keywords (see Table 2), with the intention of carrying out a structured search of previous scientific literature, which allowed constructing the search equation (see Table 2) that was applied in the SCOPUS database.

The PRISMA method was then used to guide the filtering process. This process included the application of inclusion and exclusion criteria, as well as the review of titles, abstracts and full texts of studies. As a result, documents that met the established criteria were selected and served as the basis for the next phase of the study: data extraction and analysis of results.

Taking into account the main question, different points of view were evaluated on: Functional Programming for Mutable States in the Integration of Imperative Systems. Therefore, review questions were developed using the PICO method, taking into account the proposed questions that allow recognizing the key words of the information to be collected, which can be validated in Table 2.

### III. ANALYSIS OF THE RESULTS

#### B. Bibliometric Analysis

According to the SCOPUS database, it is shown that the detection on mutable states in functional programming has been relevant since the end of the 20th century, but its greatest

boom is seen at the beginning of the 21st century, as can be seen in Figure 1.

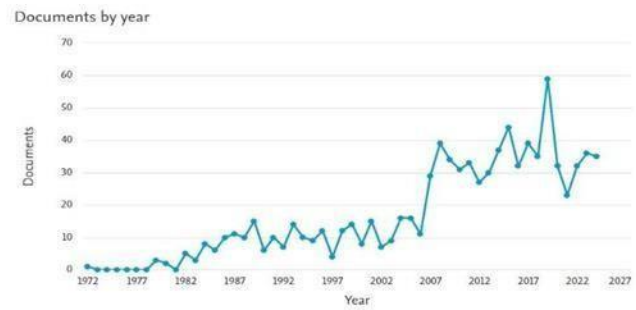


Fig.1 Production of academic literature on Functional Programming for Mutable States in the Integration of Imperative Systems

Figure 2 shows a world map highlighting the countries with the largest academic contributions to the topic “A systematic review of variable state functional programming in imperative systems integration”. The United States (237), the United Kingdom (67), and Australia (26) are the top contributors, indicating the high production of academic literature in these regions. Lines connect different countries and indicate international collaboration on related research, while color depth highlights each country’s emphasis on that particular field.



Fig.2 Countries with the highest contribution of academic literature on Functional Programming for Mutable States in the Integration of Imperative Systems

Figure 3 provides an analysis of the corresponding terms of functional programming, where “functional programming” is the core, and shows its importance in the field. Keywords such as “semantics” and “error detection” are grouped around this core to form specific sub-areas. The concepts of “differentiation” and “higher-order function” are highlighted in the subsets of computation and functional abstraction, respectively, showing the connection between mathematical theory and the practice of software development. Furthermore, the groupings reflect the logical structure and interrelations of

the concepts, which show their versatility and relevance in different contexts of use.

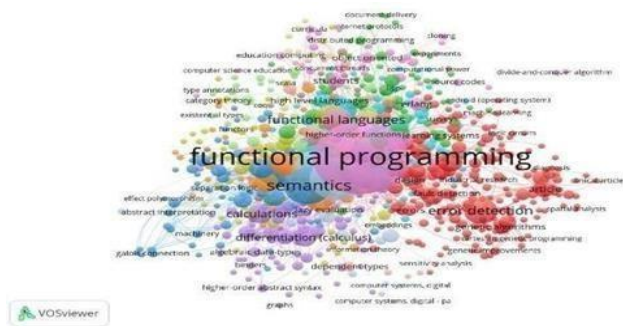


Fig.3 Network Visualization

Figure 4 shows a chart of related terms for functional programming, where “functional programming” appears as the top term, showing its relevance. Concepts such as “semantics,” “error detection,” and “computation” are organized around this core to form specific topic areas. The connection between terms such as “differentiation” and “higher-order function” illustrates the connection between mathematical abstraction and programming practice, and shows the interrelationship that demonstrates their use in different contexts. The color scale represents the evolution of these concepts between 2005 and 2020, showing the progressive development of thinking in this field.

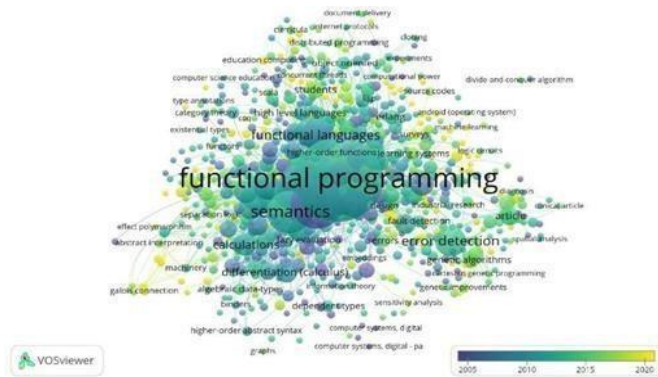


Fig.4 Overlay Visualization

In this order of ideas, the results obtained from the analysis of 49 documents collected from this SLR are explained to answer not only the questions constructed through the PICO methodology: but also to define, through relevant research, the concepts of Functional Programming and mutable states and their importance in Software Engineering, namely:

*b.1. RQ1: What kind of problems have been solved with the use of functional programming in improving the detection of mutable states in integration with systems imperatives?*

The application of functional programming has been shown to address various challenges, particularly in environments requiring resource optimization, data race management, and scalability improvements in concurrent systems. Numerous studies highlight its effectiveness in managing conflicts and enhancing execution in parallel environments through the use of purely functional languages [1], [5], [8], [10], [26], [36], [44], [2], [7], [11], [22], [42], [43], [30], [37].

To further strengthen data integrity, facilitate error detection, and ensure the correctness of critical programs in decentralized systems, type systems, security protocols, and formal verification methods have proven essential. Functional languages, by minimizing risks associated with mutable state, play a pivotal role in achieving these objectives [4], [16], [26], [28], [39], [41], [14], [20], [21], [38], [45]. These approaches not only enhance reliability but also reduce vulnerabilities in software design.

Moreover, category theory and algebraic structures provide foundational tools for modularity and algorithmic optimization. The use of concepts such as higher-order functions and compositionality has significantly advanced the efficiency and scalability of software systems [15], [29], [33], [40]. Functional programming paradigms, particularly lazy evaluation and higher-order types, enable seamless integration and efficient data processing, further broadening their applicability in diverse domains [18], [19], [23], [17].

While these studies showcase the technical advantages of functional programming, there is a pressing need to explore its real-world applications more comprehensively. Practical challenges, such as integration with imperative systems, adoption barriers in industry, and performance trade-offs, remain underexplored. Addressing these gaps could provide deeper insights into how functional programming can complement existing paradigms and address large-scale, real-world problems.

Below, Table 4 illustrates the impact of each research group. Figure 5 presents the distribution of references among five thematic categories, with “Type systems, security, and verification” emerging as the most frequently addressed topic, accounting for 11.29% of the analyzed articles.

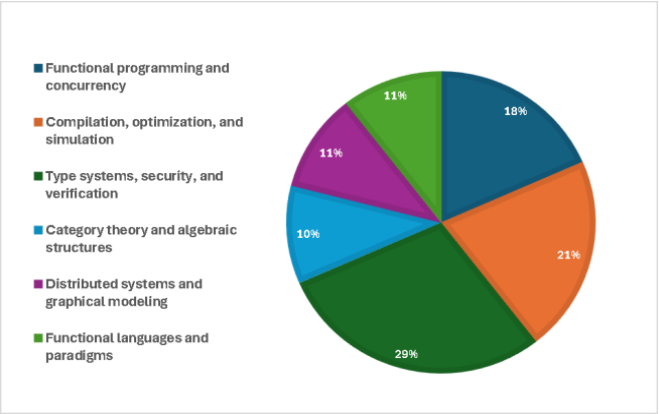


Fig. 5. Solved problems of functional programming languages

TABLE IV  
APPLICATION SPACES OF FUNCTIONAL PROGRAMMING LANGUAGES

Group	Application	References
Functional programming and concurrency	Handling data races, conflict reduction, scalability and optimization in parallel environments, integration with imperative systems.	[1], [5], [8], [10], [26], [36], [44]
Compilation, optimization and simulation	Partial evaluation, symbolic translation, resource optimization and efficient execution in functional and imperative environments, improving computational intensity.	[2], [7], [11], [22], [42], [43], [30], [37]
Type systems, security and verification	Maintaining data integrity, detecting inconsistencies, reducing risks due to mutable state, and correcting critical systems.	[4], [16], [26], [28], [39], [41], [14], [20], [21], [38], [45]
Category theory and algebraic structures	Modularity and expressiveness improvement through algebraic concepts, validation and algorithmic optimization.	[15], [29], [33], [40]
Distributed systems and graphical modeling	Coherence and synchronization in distributed systems, handling complex interactions in graphs.	[34], [35], [6], [40]
Functional languages and paradigms	Innovations in lazy assessment and higher-order types that improve functional integration.	[18], [19], [23], [17]

*b.2. RQ2: What types of functional programming languages are used to improve mutable state detection?*

The review of the literature highlights various challenges addressed through functional programming, particularly in optimizing and managing state variables within imperative systems. Functional programming, combined with concurrency, leverages purely functional languages and advanced memory management techniques to mitigate data races and reduce conflicts in simultaneous systems. This approach enhances scalability and minimizes errors in parallel environments [1], [5], [8], [10], [26], [36], [44].

Program compilation and optimization incorporate tools like partial evaluation and symbolic translation, which are critical for efficient and secure execution in environments integrating functional and imperative paradigms [2], [7], [11], [22], [42], [43]. These techniques demonstrate practical benefits in bridging paradigms while addressing real-world performance and reliability concerns.

Type systems, security, and privacy play a pivotal role in maintaining information integrity and identifying inconsistencies within decentralized systems. Functional languages, through robust type systems, reduce risks associated with mutable states and enhance system reliability [4], [16], [26], [28], [39], [41]. Algebraic structures and category theory further contribute by providing conceptual tools that improve modularity, expressiveness, and the integration of functional constructs [15], [29], [33], [40]. These theoretical foundations translate into practical solutions for managing complexity and enhancing software maintainability.

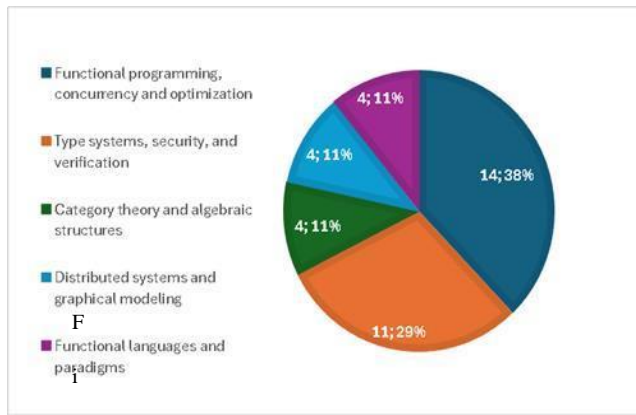
Formal verification procedures are critical for ensuring the correctness of essential programs, especially in systems with mutable states. These methods, combined with distributed computing technologies, tackle the inherent challenges of consistency and synchronization in georeplicated environments [14], [20], [21], [38], [45], [34], [35]. Such approaches are vital for addressing the real-world demands of distributed systems and ensuring their robustness.

In scientific computing and simulation, functional programming simplifies the handling of state variables, enhancing computational efficiency while reducing resource consumption [30], [37]. Graphical modeling and functional frameworks focus on algebraic structures and related mapping techniques to manage complex graph interactions, which is particularly relevant in large-scale data analysis [6], [40].

Functional programming paradigms, with features like lazy evaluation and higher-order types, optimize the integration of functional constructs into existing systems. These innovations enable efficient data processing and improve compatibility with imperative programming approaches [18], [19], [23]. The ongoing research demonstrates how functional programming not only addresses specific technical challenges but also facilitates the integration of mutable state management, bridging the gap between functional and imperative paradigms.

Table 4 provides a detailed breakdown of the impact of each research group by reference count. Figure 6 illustrates the distribution of references across four thematic groups: functional programming, concurrency, and optimization represent the highest proportion (42.86%), emphasizing resource efficiency and scalability. Verification and security in critical systems account for 35.71%, focusing on error detection and distributed systems. Category theory and algebraic structures contribute 14.29%, highlighting modularity and algorithmic optimization. Lastly, functional languages and paradigms comprise 7.14%, underscoring advances in integration through higher-order types.





g

6. Types of functional programming languages used TABLE

TYPES OF PROGRAMMING LANGUAGES USED		
Group	Application	References
Functional programming, concurrency and optimization	Handling data races, scalability, resource optimization and efficient execution.	[1], [5], [8], [10], [26], [36], [44], [2], [7], [11], [22], [42], [43], [30], [37]
Type systems, security and verification	Ensures data integrity, detects inconsistencies and ensures correctness in critical systems.	[4], [16], [26], [28], [39], [41], [14], [20], [21], [38], [45]
Category theory and algebraic structures	Improves modularity and algorithmic optimization.	[15], [29], [33], [40]
Distributed systems and graphical modeling	Synchronization in distributed systems and manipulation of complex graphs.	[34], [35], [6], [40]
Functional languages and paradigms	Improves functional integration through lazy evaluation and higher-order types.	[18], [19], [23], [17]

*b.3. RQ3: In what type of space are the types of functional programming languages that are used to improve the detection of mutable states being used?*

The review of the literature highlights various challenges addressed through functional programming, particularly in optimizing and managing state variables within imperative systems. Functional programming, combined with concurrency, leverages purely functional languages and advanced memory management techniques to mitigate data races and reduce conflicts in simultaneous systems. This approach enhances scalability and minimizes errors in parallel environments [1], [5], [8], [10], [26], [36], [44]. Furthermore, these techniques are increasingly applied in industry for tasks such as resource scheduling and real-time data processing, addressing practical demands for reliability and performance.

Program compilation and optimization incorporate tools like partial evaluation and symbolic translation, which are critical for efficient and secure execution in environments integrating functional and imperative paradigms [2], [7], [11], [22], [42], [43]. These methods are especially valuable in embedded and distributed systems, where performance constraints and error minimization are key. Their practical applications extend to reducing debugging cycles and ensuring code safety in mission-critical environments.

Type systems, security, and privacy play a pivotal role in maintaining information integrity and identifying inconsistencies within decentralized systems. Functional languages, through robust type systems, reduce risks associated with mutable states and enhance system reliability [4], [16], [26], [28], [39], [41]. Algebraic structures and category theory further contribute by providing conceptual tools that improve modularity, expressiveness, and the integration of functional constructs [15], [29], [33], [40]. These theoretical foundations translate into practical solutions for managing complexity and enhancing software maintainability, with notable applications in large-scale data management and financial systems.

Formal verification procedures are critical for ensuring the correctness of essential programs, especially in systems with mutable states. These methods, combined with distributed computing technologies, tackle the inherent challenges of consistency and synchronization in georeplicated environments [14], [20], [21], [38], [45], [34], [35]. Such approaches are vital for addressing the real-world demands of distributed systems and ensuring their robustness. For instance, verification frameworks are now being employed in cloud services to guarantee data consistency and system resilience.

In scientific computing and simulation, functional programming simplifies the handling of state variables, enhancing computational efficiency while reducing resource consumption [30], [37]. Applications include optimizing simulations in physics and engineering, where managing large datasets and ensuring precision are critical. Graphical modeling and functional frameworks focus on algebraic structures and related mapping techniques to manage complex graph interactions, which is particularly relevant in large-scale data analysis [6], [40].

Functional programming paradigms, with features like lazy evaluation and higher-order types, optimize the integration of functional constructs into existing systems. These innovations enable efficient data processing and improve compatibility with imperative programming approaches [18], [19], [23]. For instance, the adoption of lazy evaluation in big data frameworks has demonstrated significant improvements in processing speed and resource utilization. The ongoing research demonstrates how functional programming not only addresses specific technical challenges but also facilitates the integration of mutable state management, bridging the gap between functional and imperative paradigms.

Table 4 provides a detailed breakdown of the impact of each research group by reference count. Figure 6 illustrates the distribution of references across four thematic groups: functional programming, concurrency, and optimization represent the highest proportion (42.86%), emphasizing resource efficiency and scalability. Verification and security in critical systems account for 35.71%, focusing on error detection and distributed systems. Category theory and algebraic structures contribute 14.29%, highlighting

modularity and algorithmic optimization. Lastly, functional languages and paradigms comprise 7.14%, underscoring advances in integration through higher-order types.

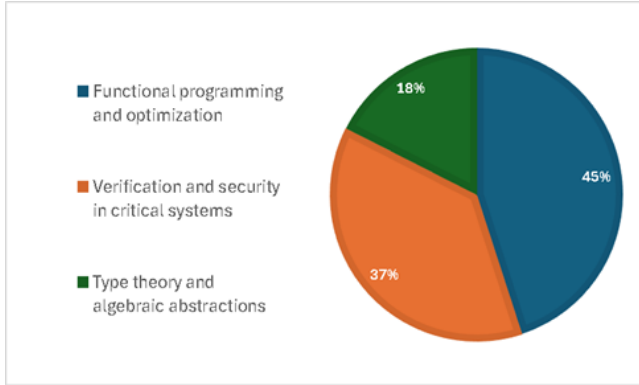


Fig 7. Types of spaces used in functional programming

TABLE VI  
TYPES OF SPACE USED IN THE PF FOR IMPROVED DETECTION OF  
MUTABLE STATES

Group	Application	References
Functional programming and optimization	Compiler optimization, data structure management, teaching recursion and robustness in distributed systems.	[1], [2], [3], [4], [9], [10], [11], [12], [13], [26], [27], [31], [32], [33], [35], [36], [40], [42]
Verification and security in critical systems	Correction, security and error detection in critical, distributed and multicore systems using verification tools.	[5], [6], [8], [14], [24], [34], [38], [20], [25], [23], [46], [47], [39], [43], [44]
Type theory and algebraic abstractions	Memory management, concurrent abstractions and theorem validation using algebraic structures.	[16], [17], [37], [40], [33], [34], [44]

*b.4. RQ4. What types of results have been obtained with the application of functional programming in the detection of mutable states in the integration of imperative systems?*

Previous academic literature has highlighted several challenges addressed through functional programming, particularly related to the optimization and management of state variables in imperative systems. Functional programming leverages advanced techniques such as defunctionalization and partial specialization to enhance memory management and performance, enabling seamless integration in environments prone to adverse effects [1], [2], [11], [22], [29], [35], [37], [43]. These approaches have been effectively applied in real-world scenarios, including optimizing cloud infrastructure and improving the resilience of embedded systems.

Mutable state management is another crucial area where functional languages excel. By employing monad analysis and gradual transformations, functional programming ensures the transparent and safe handling of mutable states, strengthening the resilience of decentralized systems [4], [7], [9], [12], [16],

[17], [18], [19]. These techniques are instrumental in industries such as blockchain and distributed ledger technologies, where reliability and fault tolerance are paramount.

Formal verification and reliability mechanisms play a critical role in ensuring the rectifiability of critical systems. Tools like Cogent and automated testing frameworks improve verifiability in complex contexts, facilitating early error detection and robust system design [20], [28], [31], [32], [38]. In practice, these tools are widely adopted in safety-critical domains like aerospace and medical device software, where errors can have severe consequences.

Parallelism and concurrency are addressed through innovative approaches such as list homomorphism and software transactional memory (STM) models. These techniques provide scalability and conflict resolution in parallel systems, enabling more efficient utilization of multicore architectures [5], [33], [36], [37], [40], [44]. Advances in language design, including asynchronous session types and hybrid languages, have significantly increased the expressiveness and safety of high-level programming, facilitating their adoption in industries such as telecommunications and financial technology [30], [40], [41], [45], [46].

Integration with imperative systems highlights the adaptability of functional languages. For example, Haskell has effectively merged with platforms like Excel, demonstrating its potential to improve user productivity and manage adverse effects in diverse environments [13], [23], [24], [25], [27], [39]. This integration bridges the gap between academic research and practical applications, showcasing the versatility of functional programming in addressing industry-specific challenges.

Functional programming thus not only solves specific problems related to dynamic state detection but also facilitates a strong integration with imperative systems. This dual capability delivers innovative solutions in terms of efficiency, safety, and reliability, making it a powerful paradigm for addressing real-world challenges.

The accompanying pie chart illustrates the distribution of references across key areas of focus. Functional programming and optimization represent 53.85% of the references, emphasizing advancements in compiler optimization and improvements in distributed systems. Verification and security in critical systems account for 38.46%, underscoring their importance in ensuring system accuracy and safety. Lastly, type theory and algebraic abstractions contribute 7.69%, highlighting their role in memory management and theorem validation. In summary, the data underscores the critical contributions of functional programming to optimization and security in critical systems.

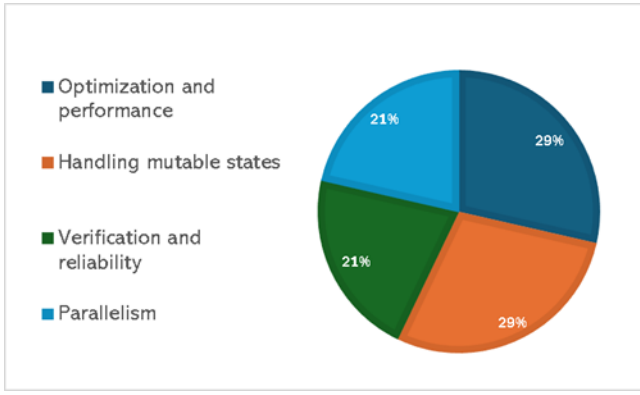


Fig. 8. Results obtained from functional programming

TABLE VII  
TYPES OF SPACE USED IN THE PF FOR IMPROVED DETECTION OF  
MUTABLE STATES

Group	Application	References
Optimization and performance	Improves memory management and performance on side-effected systems using defunctionalization and partial specialization.	[1], [2], [11], [22], [29], [35], [37], [43]
Handling mutable states	Safe and efficient management of mutable states through monad analysis and gradual transformations.	[4], [7], [9], [12], [16], [17], [18], [19]
Verification and reliability	Tools like Cogent and automated testing streamline the verification of critical systems.	[20], [28], [31], [32], [38], [44], [47]
Parallelism	Strategies such as list homomorphism and STM improve parallelism and scalability in systems.	[5], [33], [36], [37], [40], [44]

#### IV. DISCUSSION

From the articles shown in this SLR, it is possible to observe from reading them that a clear vision is shown of how applying functional programming can obtain an improvement in the detection and management of mutable states. Thanks to this, integral solutions such as efficient memory management, concurrency optimization and constant improvement in distributed systems can be obtained. Similarly, functional programming shows, from mechanisms such as lazy evaluation, monad analysis and the use of higher-order types, how it can exponentially improve error detection in mutable states [1], [5], [8], [26], [36]. However, contrary to what has been expressed, there are endless cases of success in the management of mutable states in concurrent and distributed programming environments, there is evidence that in these scenarios the direct interaction between imperative and functional systems in large-scale industrial applications is not focused.

Based on the above considerations, functional languages such as Haskell, SML, among others, play a fundamental role

in improving the detection of mutable states using lazy evaluation, higher-order types, and incorporation into strict type systems [18], [19], [23].

From this, a solution is found to problems found in memory management, synchronization in distributed environments, and concurrency optimization. Likewise, in type systems and verification tools, the addition of these languages presents a fundamental part to guarantee the integrity and security of the data, a key concern in systems with mutable states [4], [16], [28], [41].

To illustrate these considerations, the results show in which environments these languages can be applied, emphasizing academic contexts, distributed systems, and concurrent environments. Additionally, it is observed that functional programming is usually used in the optimization of algorithms, the verification of critical programs, and the improvement of security in distributed and embedded systems [5], [6], [26], [34]. It is worth highlighting that academic and research environments are the ones that have the most access to this type of technologies, implying that the adoption of these techniques is not so common in industrial environments [14], [24], [34].

#### V. CONCLUSIONS

Functional programming and mutable state management are effective, thanks to features such as management, and improved resilience in distributed systems. On the other hand, the studies reflect a deficiency in the direct interaction between laziness, higher-order types, and monad analysis. In addition, various technical problems have been solved, including concurrency optimization, memory imperative and functional systems, especially in large-scale industrial applications.

The application of functional techniques is more common in academic and research environments than in industry. Functional languages such as Haskell and SML offer a fundamental advantage for algorithm optimization and verification of critical systems, standing out in the areas of security and synchronization in distributed systems. However, to increase progress in industrial environments, extensive research is suggested in the integration of imperative and functional systems, in order to improve scalability, modularization, and reliability in commercial and large-scale environment.

#### VI. ACKNOWLEDGEMENTS

We would like to express our sincere gratitude to the Technological University of Peru for their unwavering support and resources throughout this research. Their commitment to fostering academic excellence has been invaluable.

#### VII. REFERENCES

- [1] Arora, J., Muller, SK, & Acar, UA (2024). Disentanglement with Futures, State, and Interaction. Proceedings of the ACM on



- Programming Languages, 8(POPL), 1569–1599. <https://doi.org/10.1145/3632895>
- [2] Arora, J., Westrick, S., & Acar, U.A. (2023). Efficient Parallel Functional Programming with Effects. *Proceedings of the ACM on Programming Languages*, 7(PLDI), 1558–1583. <https://doi.org/10.1145/3591284>
  - [3] Berklind, K.J., & Fehr, E. (1982). A consistent extension of the lambda-calculus as a base for functional programming languages. *Information and Control*, 55(1–3), 89–101. [https://doi.org/10.1016/S0019-9958\(82\)90458-2](https://doi.org/10.1016/S0019-9958(82)90458-2)
  - [4] Boyland, J.T. (2010). Semantics of fractional permissions with nesting. *ACM Transactions on Programming Languages and Systems*, 32(6), 1–33. <https://doi.org/10.1145/1749608.1749611>
  - [5] Brachthäuser, J.I., Schuster, P., & Ostermann, K. (2018). Effect handlers for the masses. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 1–27. <https://doi.org/10.1145/3276481>
  - [6] Brandon, W., Driscoll, B., Dai, F., Berkow, W., & Milano, M. (2023). Better Defunctionalization through Lambda Set Specialization. *Proceedings of the ACM on Programming Languages*, 7(PLDI), 977–1000. <https://doi.org/10.1145/3591260>
  - [7] Chlipala, A. (2008). Parametric higher-order abstract syntax for mechanized semantics. *ACM SIGPLAN Notices*, 43(9), 143–156.
  - [8] CRARY, K., KLIGER, A., & PFENNING, F. (2005). A monadic analysis of information flow security with mutable state. *Journal of Functional Programming*, 15(2), 249–291. <https://doi.org/10.1017/S0956796804005441>
  - [9] Danvy, O., Malmkjær, K., & Palsberg, J. (1996). Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 18(6), 730–751. <https://doi.org/10.1145/236114.236119>
  - [10] DARAIS, D., & HORN, D. van. (2019). Constructive Galois Connections. *Journal of Functional Programming*, 29, e11. <https://doi.org/10.1017/S0956796819000066>
  - [11] Deng, Z., Fu, X., & Wang, H. (2018). An IMU-Aided Body-Shadowing Error Compensation Method for Indoor Bluetooth Positioning. *Sensors*, 18(1), 304. <https://doi.org/10.3390/s18010304>
  - [12] Dennis-Jones, E., & Rydeheard, D.E. (1993). Categorical ML — Category-theoretic modular programming. *Formal Aspects of Computing*, 5(4), 337–366. <https://doi.org/10.1007/BF01212406>
  - [13] Fowler, S., Lindley, S., Morris, J.G., & Decova, S. (2019). Exceptional asynchronous session types: session types without tiers. *Proceedings of the ACM on Programming Languages*, 3(POPL), 1–29. <https://doi.org/10.1145/3290341>
  - [14] Hall, C. v., Hammond, K., Peyton Jones, S.L., & Wadler, P.L. (1996). Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2), 109–138. <https://doi.org/10.1145/227699.227700>
  - [15] Hatcliff, J. (1998). Foundations for partial evaluation of functional programs with computational effects. *ACM Computing Surveys*, 30(3es), 13. <https://doi.org/10.1145/289121.289134>
  - [16] Henderson, P.B., & Romero, F.J. (1989). Teaching recursion as a problem-solving tool using standard ML. *ACM SIGCSE Bulletin*, 21(1), 27–31. <https://doi.org/10.1145/65294.71190>
  - [17] Inoue, K., Seki, H., & Yagi, H. (1988). Analysis of functional programs to detect run-time garbage cells. *ACM Transactions on Programming Languages and Systems*, 10(4), 555–578. <https://doi.org/10.1145/48022.48025>
  - [18] Jung, R., Jourdan, J.-H., Krebbers, R., & Dreyer, D. (2021). Safe systems programming in Rust. *Communications of the ACM*, 64(4), 144–152. <https://doi.org/10.1145/3418295>
  - [19] Kaki, G., Earanky, K., Sivaramakrishnan, K., & Jagannathan, S. (2018). Safe replication through bounded concurrency verification. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 1–27. <https://doi.org/10.1145/3276534>
  - [20] Kameyama, Y., Kiselyov, O., & Shan, C. (2015). Combinators for impure yet hygienic code generation. *Science of Computer Programming*, 112, 120–144. <https://doi.org/10.1016/j.scico.2015.08.007>
  - [21] Kanabar, H., Vivien, S., Abrahamsson, O., Myreen, M.O., Norrish, M., Pohjola, J. Å., & Zanetti, R. (2023). PureCake: A Verified Compiler for a Lazy Functional Language. *Proceedings of the ACM on Programming Languages*, 7(PLDI), 952–976. <https://doi.org/10.1145/3591259>
  - [22] Kellison, A.E., & Hsu, J. (2024). Numerical Fuzz: A Type System for Rounding Error Analysis. *Proceedings of the ACM on Programming Languages*, 8(PLDI), 1954–1978. <https://doi.org/10.1145/3656456>
  - [23] Khajenejad, M., & Martinez, S. (2023). Guaranteed Privacy of Distributed Nonconvex Optimization via Mixed-Monotone Functional Perturbations. *IEEE Control Systems Letters*, 7, 1081–1086. <https://doi.org/10.1109/LCSYS.2022.3231223>
  - [24] Lämmel, R., Thompson, S., & Kaiser, M. (2013). Programming errors in traversal programs over structured data. *Science of Computer Programming*, 78(10), 1770–1808. <https://doi.org/10.1016/j.scico.2011.11.006>
  - [25] Leucker, M., Noll, T., Stevens, P., & Weber, M. (2005). Functional programming languages for verification tools: a comparison of Standard ML and Haskell. *International Journal on Software Tools for Technology Transfer*, 7(2), 184–194. <https://doi.org/10.1007/s10009-004-0184-3>
  - [26] Liu, Y.A., & Teitelbaum, T. (1995). Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1), 1–39. [https://doi.org/10.1016/0167-6423\(94\)00031-9](https://doi.org/10.1016/0167-6423(94)00031-9)
  - [27] Lorenzen, A., Leijen, D., & Swierstra, W. (2023). FP2: Fully in-Place Functional Programming. *Proceedings of the ACM on Programming Languages*, 7(ICFP), 275–304. <https://doi.org/10.1145/3607840>
  - [28] Lu, S., & Bodík, R. (2023). Griset: Symbolic Compilation as a Functional Programming Library. *Proceedings of the ACM on Programming Languages*, 7(POPL), 455–487. <https://doi.org/10.1145/3571209>
  - [29] Mason, I. A. (1988). Verification of programs that destructively manipulate data. *Science of Computer Programming*, 10(2), 177–210. [https://doi.org/10.1016/0167-6423\(88\)90026-3](https://doi.org/10.1016/0167-6423(88)90026-3)
  - [30] McDermott, D., & Mycroft, A. (2024). Galois connecting call-by-value and call-by-name. *Logical Methods in Computer Science*, Volume 20,
  - [31] Issue 1(1), 13:1–13:43. [https://doi.org/10.46298/lmcs-20\(1:13\)2024](https://doi.org/10.46298/lmcs-20(1:13)2024)
  - [32] Mokhov, A. (2022). United Monoids. *The Art, Science, and Engineering of Programming*, 6(3). <https://doi.org/10.22152/programming-journal.org/2022/6/12>
  - [33] Niehren, J., Schwinghammer, J., & Smolka, G. (2006). A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3), 338–356. <https://doi.org/10.1016/j.tcs.2006.08.016>
  - [34] O’CONNOR, L., CHEN, Z., RIZKALLAH, C., JACKSON, V., AMANI, S., KLEIN, G., MURRAY, T., SEWELL, T., & KELLER, G. (2021). Cogent: uniqueness types and certifying compilation. *Journal of Functional Programming*, 31, e25. <https://doi.org/10.1017/S095679682100023X>
  - [35] Ramsey, N. (2022). Beyond Reloop: recursive translation of unstructured control flow to structured control flow (functional pearl). *Proceedings of the ACM on Programming Languages*, 6(ICFP), 1–22. <https://doi.org/10.1145/3547621>
  - [36] Rocha, RCO, Góes, LFW, & Pereira, FMQ (2019). Automatic parallelization of recursive functions with rewriting rules. *Computer Science Programming*, 173, 128–152. <https://doi.org/10.1016/j.scico.2018.01.004>
  - [37] SCHMIDT-SCHAUS, M., SABEL, D., & SCHÜTZ, M. (2008). Safety of Nöcker’s strictness analysis. *Journal of Functional Programming*, 18(04), 503–

551.<https://doi.org/10.1017/S0956796807006624>

- [38] Schöpp, U. (2014). On the Relation of Interaction Semantics to Continuations and Defunctionalization. *Logical Methods in Computer Science*, Volume 10, Issue 4(4), 1–41. [https://doi.org/10.2168/LMCS-10\(4:10\)2014](https://doi.org/10.2168/LMCS-10(4:10)2014)
- [39] Schröder, L., & Mossakowski, T. (2009). HasCasl: Integrated higher-order specification and program development. *Theoretical Computer Science*, 410(12–13), 1217–1260. <https://doi.org/10.1016/j.tcs.2008.11.020>
- [40] Seger, C.-JH, Jones, R.B., O’Leary, J.W., Melham, T., Aagaard, M.D., Barrett, C., & Syme, D. (2005). An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9), 1381–1405. <https://doi.org/10.1109/TCAD.2005.850814>
- [41] Selsam, D., Hudon, S., & de Moura, L. (2020). Sealing pointer-based optimizations behind pure functions. *Proceedings of the ACM on Programming Languages*, 4(ICFP), 1–20. <https://doi.org/10.1145/3408997>
- [42] Sieczkowski, F., Pyzik, M., & Biernacki, D. (2023). A General Fine-Grained Reduction Theory for Effect Handlers. *Proceedings of the ACM on Programming Languages*, 7(ICFP), 511–540. <https://doi.org/10.1145/3607848>
- [43] Thaler, J., & Siebers, P.-O. (2019). A tale of lock-free agents: towards Software Transactional Memory in parallel Agent-Based Simulation. *Complex Adaptive Systems Modeling*, 7(1), 5. <https://doi.org/10.1186/s40294-019-0067-9>
- [44] THOMPSON, S., & LI, H. (2013). Refactoring tools for functional languages. *Journal of Functional Programming*, 23(3), 293–350. <https://doi.org/10.1017/S0956796813000117>
- [45] Voigtländer, J., & Johann, P. (2007). Selective strictness and parametricity in structural operational semantics, inequationally. *Theoretical Computer Science*, 388(1–3), 290–318. <https://doi.org/10.1016/j.tcs.2007.09.014>
- [46] WAKELING, D. (2007). Spreadsheet functional programming. *Journal of Functional Programming*, 17(1), 131–143. <https://doi.org/10.1017/S0956796806006186>
- [47] Westrick, S., Arora, J., & Acar, U.A. (2022). Entanglement detection with near-zero cost. *Proceedings of the ACM on Programming Languages*, 6(ICFP), 679–710. <https://doi.org/10.1145/3547646>
- [48] White, D.R., Arcuri, A., & Clark, J.A. (2011). Evolutionary Improvement of Programs. *IEEE Transactions on Evolutionary Computation*, 15(4), 515–538. <https://doi.org/10.1109/TEV>
- [49] B.A. Kitchenham and Charters, S. “Guidelines for performing systematic literature reviews in software engineering” version 2.3
- [50] Page, M. J.; McKenzie, J. E.; Bossuyt, P. M.; Boutron, I., Hoffmann, T. C.; Mulrow, C. D.; Moher, D. (2021). The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. *The BMJ*, 372 doi:10.1136/bmj.n71 [https://www.dropbox.com/s/e35alub972x1jkv/PRISMA\\_2020\\_statement\\_definitivo-Espa%C3%B1ol%20%28completo%29.pdf?dl=0](https://www.dropbox.com/s/e35alub972x1jkv/PRISMA_2020_statement_definitivo-Espa%C3%B1ol%20%28completo%29.pdf?dl=0)