

# A Pedagogical Static Analysis Framework For Teaching Functional Programming

Luis Eduardo Muñoz Guerrero

Facultad de Ingenierías, Universidad Tecnológica de Pereira, Pereira, Risaralda, Colombia

ORCID: <https://orcid.org/0000-0002-9414-6187>

lemunozg@utp.edu.co

---

## Abstract

Static analysis techniques have proven invaluable in identifying potential errors and improving code quality in imperative programming languages. However, their application to functional programming paradigms, particularly in educational contexts, remains underexplored. This paper presents a pedagogical approach to static analysis specifically designed for functional programs, focusing on common patterns, anti-patterns, and learning opportunities that arise in functional programming education. We propose a framework that not only detects errors but also provides educational feedback to help students understand functional programming concepts more deeply. Our approach emphasizes immutability, higher-order functions, recursion patterns, and type system utilization. Through case studies in Haskell and OCaml, we demonstrate how pedagogical static analysis can enhance the learning experience and accelerate student comprehension of functional programming principles.

**Keywords:** Functional Programming, Static Analysis, Programming Education, Pedagogical Tools, Haskell, Ocaml.

## 1. Introduction

Functional programming has gained significant traction in both academia and industry, with languages like Haskell, OCaml, Scala, and F# becoming increasingly popular [1]. Despite this growth, students often struggle with the paradigm shift from imperative to functional thinking [4]. Traditional static analysis tools focus primarily on error detection and performance optimization, but they rarely address the pedagogical needs of learners. The transition to functional programming requires students to adopt new mental models: thinking in terms of immutable data, function composition, and declarative problem-solving rather than step-by-step procedures. Common mistakes include attempting to use mutable state, misunderstanding recursion, and failing to leverage the type system effectively [9].

This paper introduces a pedagogical static analysis framework tailored for functional programming education. Our approach goes beyond traditional error detection to provide contextual, educational feedback that helps students understand not just what is wrong, but why it is wrong and how to think functionally about the problem. Traditional compiler error messages and static analysis tools are designed for experienced developers who already understand the underlying concepts [6]. For students learning functional programming, these messages can be cryptic and unhelpful [7]. For example, a type error message like "Couldn't match expected type 'Int' with actual type '[Int]'" tells an experienced programmer exactly what went wrong, but a novice may not understand the distinction between a value and a list, or why the type system rejected their code [11].

Furthermore, students often write code that compiles and runs but fails to embrace functional programming principles [8], [15]. They might use explicit recursion where a fold would be more appropriate [2], or chain multiple passes over a data structure instead of composing operations. These issues are not errors in the traditional sense, but they represent missed learning opportunities. This paper makes the following contributions: a taxonomy of common functional

programming mistakes and anti-patterns in educational settings; a pedagogical static analysis framework that provides educational feedback tailored to student learning; pattern detection algorithms for identifying opportunities to use higher-order functions, improve recursion schemes, and leverage type system features; case studies demonstrating the effectiveness of pedagogical feedback in Haskell and OCaml courses; and guidelines for implementing pedagogical static analysis tools in functional programming education.

## 2. Background

Functional programming is built on several core principles that distinguish it from imperative programming [1]. First-class functions allow functions to be passed as arguments and returned as values, enabling powerful abstraction mechanisms. Immutability ensures that data structures cannot be modified after creation, eliminating entire classes of bugs related to shared mutable state. Pure functions, which always produce the same output for the same input and have no side effects, make programs easier to reason about and test. Higher-order functions like map, filter, and fold provide generic patterns for processing collections without explicit recursion [2]. Pattern matching enables elegant decomposition of data structures. Lazy evaluation, present in languages like Haskell, allows for infinite data structures and can improve performance by computing only what is needed.

Static analysis examines program code without executing it, identifying potential errors, security vulnerabilities, and code quality issues. Traditional static analysis techniques include type checking, data flow analysis, control flow analysis, and abstract interpretation. In functional languages, type systems are particularly powerful, with features like parametric polymorphism, algebraic data types, and type classes providing strong guarantees about program behavior. However, traditional static analysis focuses on correctness and performance rather than pedagogy. A pedagogical approach must consider the learner's mental model, common misconceptions, and the learning objectives of the course [12].

Research in programming education has identified several challenges students face when learning to program [5]. Novices often struggle with abstraction, debugging, and understanding error messages [6], [7]. In functional programming specifically, students must overcome the imperative mindset they may have developed in previous courses [4]. Effective educational tools provide scaffolding that gradually introduces complexity, offer immediate feedback, and explain concepts in terms students can understand [10]. Pedagogical static analysis can serve as an automated tutor, providing guidance at the moment students need it most.

### 3. Taxonomy of Functional Programming Mistakes

Through analysis of student code from multiple functional programming courses, we have identified several categories of common mistakes and anti-patterns [8], [9], [15]. Students with prior imperative programming experience often attempt to translate imperative patterns directly into functional code. This manifests as excessive use of explicit recursion with accumulator parameters mimicking loops, attempts to use mutable references or I/O operations where pure functions would suffice, and sequential thinking rather than declarative composition. For example, a student might write a function to sum a list using explicit recursion with an accumulator, when the idiomatic approach would be to use a fold [2]. While both are correct, the latter better demonstrates understanding of functional abstractions.

Recursion is fundamental to functional programming, but students often struggle with it [9]. Common issues include missing base cases leading to infinite recursion, incorrect recursive calls that do not make progress toward the base case,

and failure to use tail recursion when appropriate for performance. Students may also fail to recognize standard recursion patterns. For instance, structural recursion over lists follows a predictable pattern: handle the empty list base case, then process the head and recurse on the tail. Recognizing this pattern is key to mastering functional programming [16].

Higher-order functions like map, filter, fold, and compose are central to functional programming [1], [2], yet students often write explicit recursion instead. This represents a missed opportunity to learn these powerful abstractions. For example, a student might write a recursive function to square each element of a list, when `map (\x -> x * x)` would be more concise and idiomatic. Detecting these opportunities and suggesting the higher-order alternative helps students build their functional vocabulary [10].

Functional languages typically have rich type systems, but students often fail to leverage them effectively [11]. Common issues include overuse of generic types like lists or tuples instead of defining custom algebraic data types, ignoring type class constraints, and failing to use parametric polymorphism to write generic code. Students may also struggle with type errors, not understanding why the type checker rejected their code [3], [11]. Pedagogical feedback can explain type mismatches in terms of the student's likely intent.

While correctness is the primary concern in education, students should also learn to recognize inefficient patterns [13]. Multiple traversals of the same data structure, using `append (++)` in a loop leading to quadratic complexity, and failure to use lazy evaluation appropriately are common issues. These inefficiencies may not matter for small inputs in homework assignments, but understanding them is important for writing scalable functional code.

**Table 1: Taxonomy of Common Functional Programming Mistakes**

| Category                   | Common Issues                                                                | Impact on Learning                               |
|----------------------------|------------------------------------------------------------------------------|--------------------------------------------------|
| Imperative Thinking        | Excessive explicit recursion, mutable state attempts, sequential composition | Fails to embrace functional paradigm             |
| Recursion Misunderstanding | Missing base cases, non-decreasing calls, no tail recursion                  | Infinite loops, stack overflow, poor performance |
| Underutilizing HOFs        | Manual recursion instead of map/filter/fold                                  | Misses key abstractions, verbose code            |
| Type System Misuse         | Overuse of generic types, ignoring constraints                               | Weak type safety, unclear intent                 |
| Inefficient Patterns       | Multiple traversals, quadratic append operations                             | Poor scalability, performance issues             |

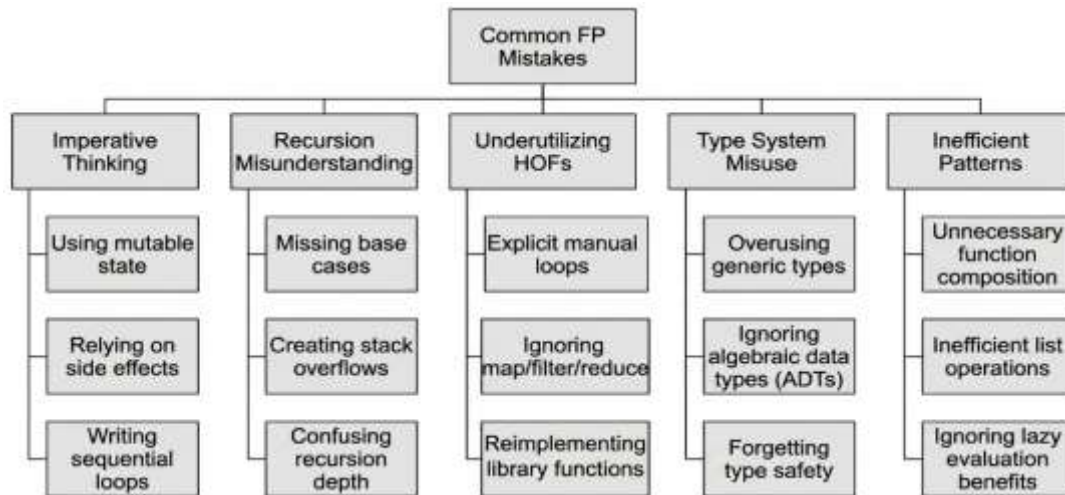


Figure 1: Hierarchical Taxonomy of Common Functional Programming Mistakes in Educational Settings

#### 4. Pedagogical Static Analysis Framework

Our pedagogical static analysis framework consists of three main components: pattern detection, educational feedback generation, and progressive disclosure of complexity. The pattern detection component analyzes the abstract syntax tree (AST) of student code to identify both problematic patterns and opportunities for improvement. We use a combination of syntactic pattern matching and semantic analysis. For example, to detect opportunities to use map, we look for

explicit recursive functions that follow this pattern: a base case returning an empty list, and a recursive case that applies a function to the head and conses it onto the recursive call on the tail [2]. This pattern strongly suggests that map would be more appropriate. Similarly, we detect potential uses of filter, fold, and other higher-order functions. We also identify recursion issues like missing base cases or non-decreasing recursive calls [9].

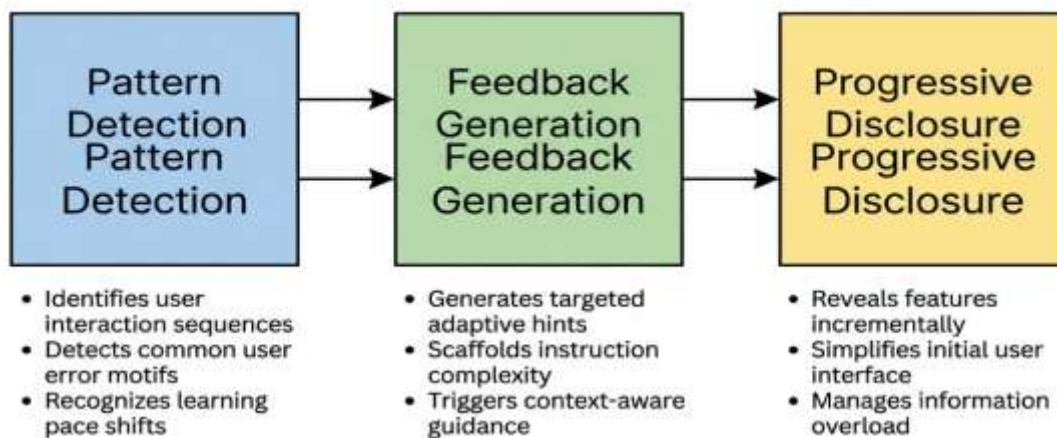


Figure 2: Three-Component Architecture of the Pedagogical Static Analysis Framework

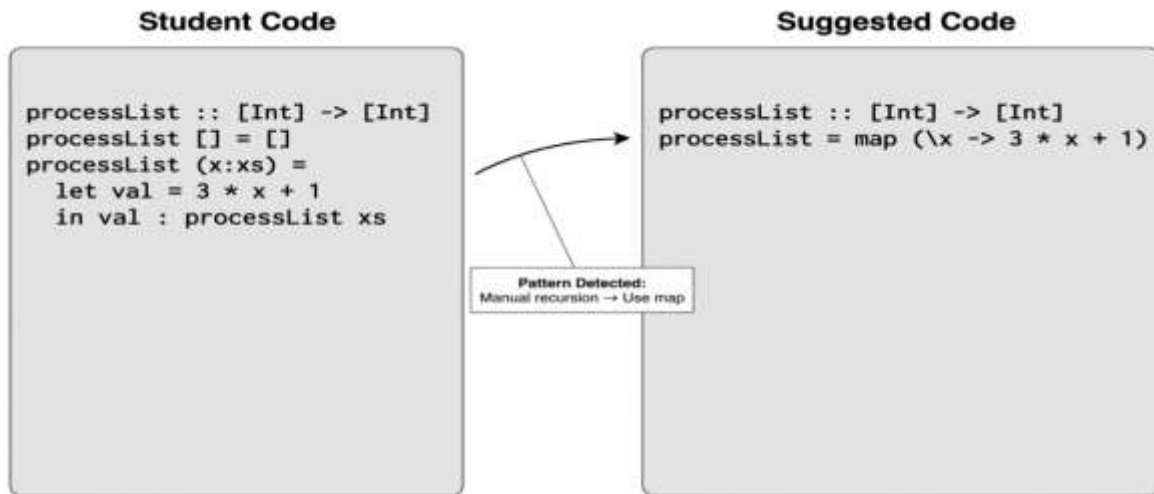
Once patterns are detected, the system generates educational feedback tailored to the student's level and the learning objectives [12]. Feedback includes several components: identification of what pattern or issue was detected; explanation of why this matters and what concept it relates to; a concrete example showing the suggested improvement; rationale for why the suggested approach is better (more idiomatic, more efficient, more maintainable); and links to relevant documentation or course materials. The tone is encouraging and educational rather than critical [6], [7]. Instead of "Error: inefficient code," we might say "This works correctly! You could make it more concise using the map

function, which is designed for exactly this pattern of transforming each element of a list."

Not all feedback is appropriate at all times. Early in a course, students are still learning basic syntax and recursion. Suggesting higher-order functions at this stage might be overwhelming [5]. Our framework supports progressive disclosure, where the level of feedback adapts to the student's progress [10]. Instructors can configure which analyses are active at different points in the course. For example, in the first few weeks, the tool might focus on recursion correctness and basic type errors. Later, it might suggest higher-order functions and point out efficiency issues.

**Table 3: Pedagogical Static Analysis Framework Components**

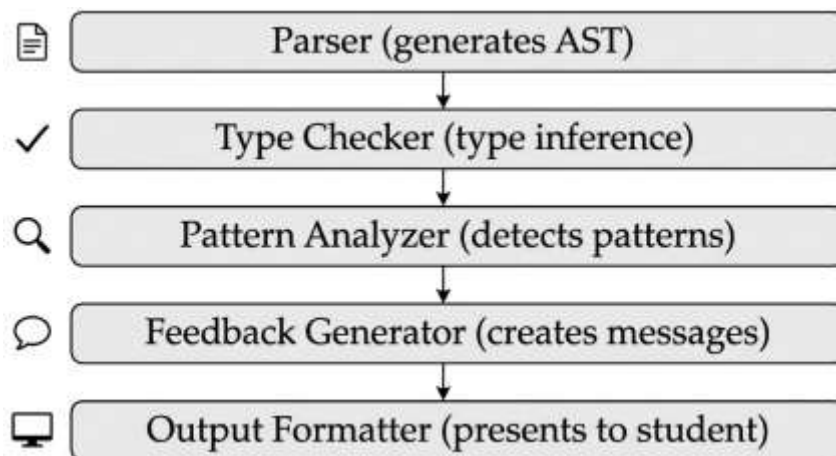
| Component              | Function                                                      | Techniques Used                                     |
|------------------------|---------------------------------------------------------------|-----------------------------------------------------|
| Pattern Detection      | Identifies problematic patterns and improvement opportunities | AST analysis, syntactic matching, semantic analysis |
| Feedback Generation    | Creates educational messages tailored to student level        | Template instantiation, context-aware messaging     |
| Progressive Disclosure | Adapts feedback complexity to student progress                | Configurable analysis rules, staged introduction    |

**Figure 4: Example of Pattern Detection and Feedback Generation for Map Opportunity**

### 5. Implementation

We have implemented prototype pedagogical static analysis tools for Haskell and OCaml. Both implementations follow the same general architecture but are adapted to the specific features and idioms of each language. The tool consists of several stages. First, the parser generates an AST from the source code. The type checker performs type inference and checking, providing type information for later stages. The

pattern analyzer traverses the AST, applying pattern detection rules. The feedback generator produces educational messages based on detected patterns [12]. Finally, the output formatter presents feedback to the student in a readable format. The system is designed to be extensible, allowing instructors to add custom pattern detection rules and feedback templates specific to their course [14].

**Figure 3: Five-Stage Processing Pipeline of the Static Analysis Tool**

Pattern detection rules are specified declaratively, making it easy to add new patterns. Each rule consists of a pattern to match in the AST, a condition to check (optional), and a feedback template to instantiate when the pattern is found. For example, the rule for detecting map opportunities matches a recursive function with a specific structure, checks that it is not already using map, and generates feedback suggesting the use of map with a concrete example based on the student's code [10].

The tool can be integrated into the student's development environment in several ways [13], [14]. It can run as a command-line tool that students invoke manually, as a plugin for editors like VS Code or Emacs, or as part of an automated grading system that provides feedback on submitted assignments. Integration with the development environment allows for immediate feedback as students write code, similar to how modern IDEs provide real-time error checking and suggestions.

## 6. Case Studies

We evaluated our pedagogical static analysis framework in two functional programming courses: an undergraduate Haskell course and a graduate OCaml course. In both cases, students used the tool throughout the semester and provided feedback on its usefulness. The undergraduate Haskell course had 45 students, most with prior experience in Java or Python but no functional programming background. The tool was introduced in the third week, after students had learned basic syntax and recursion. Students reported that the tool helped them understand when to use higher-order functions. One student commented, "I kept writing recursive functions until

the tool showed me I was just reinventing map. After seeing that a few times, I started thinking about whether map or filter would work before writing recursion." Analysis of student code showed that by mid-semester, students were using higher-order functions more frequently and writing more idiomatic Haskell. The tool was particularly effective at helping students understand fold, which many found initially confusing.

The graduate OCaml course had 20 students with diverse backgrounds. Some had functional programming experience, while others were learning it for the first time. The tool was configured to provide more advanced feedback, including suggestions about using modules, functors, and advanced type system features. Students appreciated the feedback on efficiency issues. One student noted, "I didn't realize that using @ in a loop was so inefficient. The tool explained why and showed me how to use List.rev\_append instead. That kind of practical advice is really valuable." The tool also helped students understand OCaml's type system better, particularly with respect to polymorphic variants and GADTs. By explaining type errors in educational terms, the tool reduced frustration and helped students develop better mental models. In both courses, students who used the tool regularly showed improved code quality and better understanding of functional programming concepts. Surveys indicated that students found the tool helpful, with an average rating of 4.2 out of 5 for usefulness. Instructors reported that students asked fewer basic questions about when to use certain functions, freeing up office hours for deeper conceptual discussions. The tool effectively served as a first line of feedback, catching common issues before students needed to ask for help.

**Table 2: Summary of Case Study Results**

| Course              | Students | Tool Usage Period | Usefulness Rating | Key Outcomes                                            |
|---------------------|----------|-------------------|-------------------|---------------------------------------------------------|
| Haskell (Undergrad) | 45       | Week 3-16         | 4.3/5.0           | Increased HOF usage, better fold understanding          |
| OCaml (Graduate)    | 20       | Week 1-16         | 4.1/5.0           | Improved efficiency awareness, better type system usage |
| Combined Average    | 65       | 13-16 weeks       | 4.2/5.0           | Reduced basic questions, improved code quality          |

## 7. Discussion

The case studies demonstrate that pedagogical static analysis can be an effective educational tool. By providing immediate, contextual feedback, the tool helps students learn functional programming concepts more quickly and deeply. However, the quality of feedback is crucial. Generic or overly technical messages are not helpful to novices. The feedback must be tailored to the student's level and explain concepts in accessible terms. This requires careful design and testing of feedback messages.

Our approach has several limitations. First, pattern detection is based on syntactic and simple semantic analysis, which may miss some opportunities or produce false positives. More sophisticated analysis using abstract interpretation or symbolic execution could improve accuracy but would be more complex to implement. Second, the tool cannot

understand the student's intent. If a student deliberately writes explicit recursion for practice, the tool may still suggest using a higher-order function. Allowing students to suppress certain suggestions would address this issue. Third, the tool is currently limited to Haskell and OCaml. Extending it to other functional languages like Scala, F#, or Clojure would require adapting the pattern detection rules to the idioms and features of those languages.

Several directions for future work are promising. First, incorporating machine learning to personalize feedback based on individual student progress could make the tool more effective. By tracking which suggestions students find helpful and which they ignore, the system could adapt its feedback strategy. Second, extending the tool to detect higher-level design issues, such as poor module structure or inappropriate use of side effects, would provide more comprehensive

feedback. This would require more sophisticated program analysis techniques. Third, integrating the tool with automated grading systems could provide students with immediate feedback on assignments. This would reduce the time between submission and feedback, which is known to improve learning outcomes. Finally, conducting longitudinal studies to measure the long-term impact of pedagogical static analysis on student learning would provide stronger evidence of its effectiveness.

## 8. Related Work

Several tools provide static analysis for functional languages, but few focus on pedagogy. HLint for Haskell suggests improvements to code style and idiom, but its feedback is aimed at experienced developers rather than students. OCaml's compiler provides excellent type error messages, but they can be cryptic for novices [11]. In the broader programming education space, tools like Dr. Racket provide pedagogical support for learning Scheme, including teaching languages with restricted features and educational error messages [5]. Helium [3] and Ask-Elle [10] represent significant efforts in creating pedagogical tools for Haskell, with Helium focusing on improved error messages and Ask-Elle providing adaptive tutoring. Our work extends this approach to statically-typed functional languages and focuses specifically on helping students learn functional programming idioms. Research on compiler error messages has shown that novice-friendly messages can significantly improve learning outcomes [6], [7], [8]. Our work builds on this by providing not just better error messages, but proactive suggestions for improvement even when code is correct. Recent work on automated feedback generation [12] and static analysis in programming courses [13], [14] demonstrates the growing interest in this area.

## 9. Conclusion

Pedagogical static analysis represents a promising approach to improving functional programming education. By providing immediate, contextual, educational feedback, such tools can help students overcome the challenges of learning a new programming paradigm. Our framework demonstrates that it is possible to detect common functional programming mistakes and anti-patterns and provide feedback that helps students learn. Case studies in Haskell and OCaml courses show that students find the tool helpful and that it improves code quality and understanding.

As functional programming continues to grow in importance, tools that support learning will become increasingly valuable. Pedagogical static analysis is one piece of the puzzle, complementing traditional instruction and helping students develop the skills they need to think functionally. The key insight is that static analysis tools for education must be designed with pedagogy in mind from the start. Simply adapting tools built for professional developers is not sufficient. By focusing on the learner's needs, mental models, and common misconceptions, we can create tools that truly enhance the educational experience.

## References

- [1] J. Hughes, "Why Functional Programming Matters," *The Computer Journal*, vol. 32, no. 2, pp. 98–107, Apr. 1989. DOI: 10.1093/comjnl/32.2.98
- [2] G. Hutton, "A Tutorial on the Universality and Expressiveness of Fold," *Journal of Functional Programming*, vol. 9, no. 4, pp. 355–372, Jul. 1999. DOI: 10.1017/S0956796899003500
- [3] B. Heeren, D. Leijen, and A. van IJzendoorn, "Helium, for Learning Haskell," in *Proc. 2003 ACM SIGPLAN Workshop on Haskell (Haskell '03)*, Uppsala, Sweden, Aug. 2003, pp. 62–71. DOI: 10.1145/871895.871902
- [4] M. M. T. Chakravarty and G. Keller, "The Risks and Benefits of Teaching Purely Functional Programming in First Year," *Journal of Functional Programming*, vol. 14, no. 1, pp. 113–123, 2004. DOI: 10.1017/S0956796803004805
- [5] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, *How to Design Programs: An Introduction to Programming and Computing*, 2nd ed. Cambridge, MA: MIT Press, 2018. URL: <https://htdp.org>
- [6] G. Marceau, K. Fisler, and S. Krishnamurthi, "Mind Your Language: On Novices' Interactions with Error Messages," in *Proc. 10th SIGPLAN Symp. New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2011)*, Portland, OR, Oct. 2011, pp. 3–18. DOI: 10.1145/2048237.2048241
- [7] G. Marceau, K. Fisler, and S. Krishnamurthi, "Measuring the Effectiveness of Error Messages Designed for Novice Programmers," in *Proc. 42nd ACM Technical Symp. Computer Science Education (SIGCSE '11)*, Dallas, TX, Mar. 2011, pp. 499–504. DOI: 10.1145/1953163.1953308
- [8] A. Altmir and N. C. C. Brown, "37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data," in *Proc. 46th ACM Technical Symp. Computer Science Education (SIGCSE '15)*, Kansas City, MO, Feb. 2015, pp. 522–527. DOI: 10.1145/2676723.2677258
- [9] V. Tirronen, S. Uusi-Mäkelä, and V. Isomöttönen, "Understanding Beginners' Mistakes with Haskell," *Journal of Functional Programming*, vol. 25, Article e11, 2015. DOI: 10.1017/S0956796815000179
- [10] A. Gerdes, B. Heeren, J. Jeuring, and L. T. van Binsbergen, "Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback," *International Journal of Artificial Intelligence in Education*, vol. 27, no. 1, pp. 65–100, Mar. 2017. DOI: 10.1007/s40593-015-0080-x
- [11] B. Wu, J. P. Campora III, and S. Chen, "Learning User Friendly Type-Error Messages," *Proceedings of the ACM on Programming Languages*, vol. 1, OOPSLA, Article 106, Oct. 2017. DOI: 10.1145/3133930
- [12] H. Keuning, J. Jeuring, and B. Heeren, "A Systematic Literature Review of Automated Feedback Generation for Programming Exercises," *ACM Transactions on Computing Education*, vol. 19, no. 1, Article 3, Sep. 2018. DOI: 10.1145/3231711
- [13] D. Liu and A. Petersen, "Static Analyses in Python Programming Courses," in *Proc. 50th ACM Technical Symp. Computer Science Education (SIGCSE '19)*, Minneapolis, MN, Feb. 2019, pp. 666–671. DOI: 10.1145/3287324.3287503
- [14] R. Luukkainen, J. Kasurinen, U. Nikula, and V. Lenarduzzi, "ASPA: A Static Analyser to Support

Learning and Continuous Feedback on Programming Courses. An Empirical Validation," in Proc. ACM/IEEE 44th Int. Conf. Software Engineering: Software Engineering Education and Training (ICSE-SEET '22), Pittsburgh, PA, May 2022, pp. 29–39. DOI: 10.1145/3510456.3514149

- [15] N. C. C. Brown and A. Altdmri, "Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs," ACM Transactions on Computing Education, vol. 17, no. 2, Article 7, Jun. 2017. DOI: 10.1145/2994154
- [16] P. B. Vasconcelos and R. Marques, "Teaching Introductory Functional Programming Using Haskellite," in Trends in Functional Programming in Education (TFPIE 2024), EPTCS 424, pp. 49–66, 2024. arXiv:2508.03640