ScalaF: Functional Refactoring Suggestions for Scala

Shiv Kiran Bagathi
IIT Bombay, India
shivk121617@gmail.com

Shrikha Mahanty
IIT Mandi, India
helloshrikha@gmail.com

Dasari Gnana Heemmanshuu
IIT Bombay, India
gnanaheemmanshuu@gmail.com

Manas Thakur
IIT Bombay, India
manas@cse.iitb.ac.in

Abstract

Scala supports both object-oriented and functional programming styles. Yet many developers, especially those from imperative backgrounds, write Scala code in an OO style, missing benefits such as immutability, conciseness, and effortless parallelization.

We present ScalaF, a VSCode plugin with a Scala backend that helps bridge this gap. ScalaF detects non-idiomatic patterns and suggests semantics-preserving refactorings, transforming loops and conditionals into higher-order functions and pattern matches. It scales to files up to 5k LoC in under 3 seconds, making it suitable for real-time IDE usage.

By reducing technical debt and surfacing functional best practices in the developer's workflow, ScalaF lowers the barrier to writing idiomatic functional Scala, and serves as both a productivity aid and an educational resource.

1 Introduction

Refactoring is the process of restructuring code to improve its design, readability, and maintainability without altering its external behavior. It plays a crucial role in long-term software quality, especially in large-scale systems where code evolves frequently. In Scala – a hybrid OO-functional language – refactoring can help shift codebases from imperative patterns to idiomatic functional styles that emphasize immutability, declarative logic, conciseness, and easy parallelization.

Many Scala developers, especially those new to the language or transitioning from object-oriented paradigms, tend to write verbose or imperative code that underutilizes the expressive constructs Scala offers. Idiomatic Scala leverages first-class functions, pattern matching, higher-order functions, and immutability to write elegant and efficient code. This project targets the transformation of such non-idiomatic code by building a practical and extensible tool that detects and suggests refactorings aligned with these idioms.

ScalaF integrates directly into the VSCode development environment and uses a Scala-based backend for static program analysis. It identifies patterns such as loop constructs and conditional expressions, analyzes them via syntax trees, and proposes functional replacements—e.g., replacing loops with `map`, `filter`, or `fold`, and nested `if-else` chains with `match` expressions. A particular benefit of these constructs, apart from conciseness, is that they tend to have easily parallelizable versions available commonly as libraries.

Unlike existing tools such as IDE-based linters or command-line frameworks [4], our tool emphasizes live feedback in the editor and offers an extensible framework for adding more transformations. The backend uses Scalameta [5] and quasiquotes to parse, analyze, and reconstruct Scala syntax trees, enabling accurate and idiomatic rewrites. Designed as a modular and language-aware system, it provides practical value to developers while laying the groundwork for future enhancements.

2 System Architecture and Features

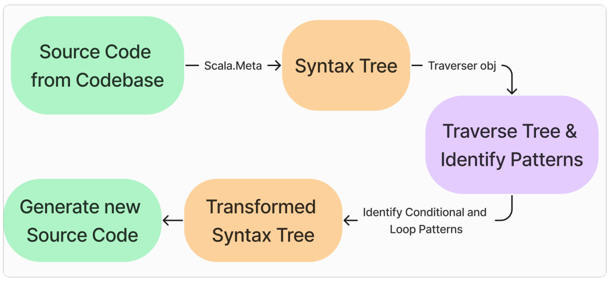
Source Code from Codebase
Scala.Meta
Syntax Tree
Traverser obj
Traverse Tree & Identify Patterns
Identify Conditional and Loop Patterns
Transformed Syntax Tree
Generate new Source Code

Figure 1. ScalaF: Refactoring workflow.

ScalaF is composed of two components:

- **Static Analysis Backend**: Written in Scala, this backend performs tree traversal and pattern matching over source code using Scalameta [5]. It identifies constructs such as loops (`for`, `while`) and conditionals (`if-else`, `else-if`) and refactors them into functional equivalents like `map`, `filter`, and pattern matching. The backend source code is available as a GitHub repository [1].
- **VSCode Plugin**: Built with JavaScript and the VSCode Extension API, it communicates with the backend, parses results, and visually annotates the source file. The plugin handles whole-file refactoring, provides hover-based tips, and enables selective application through a dedicated sidebar interface. The plugin source code is available as a GitHub repository [2].

The complete refactoring workflow of ScalaF, as shown in Figure 1, is summarized below:

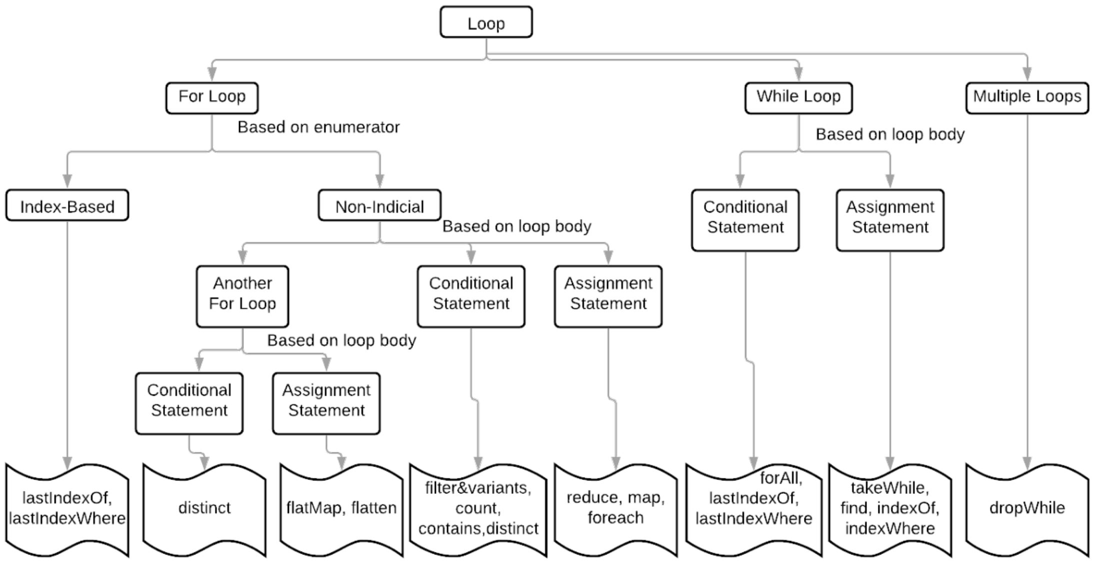

Figure 2. Decision tree for loop refactoring.

1. Parsing Scala source code into an abstract syntax tree (AST) using Scalameta.
2. Traversing the AST to detect imperative patterns such as loops and conditionals.
3. Reconstructing transformed trees using quasiquotes to generate functional alternatives.
4. Displaying suggestions in the editor via hovers, sidebars, and code actions.

2.1 Loop Refactorings

ScalaF categorizes loops based on structure (e.g., index-based, condition-based, nested) and maps them to corresponding Scala collection methods, often leveraging parallel collections for performance improvements. Figure 2 gives illustrates the decision logic we use for loop refactorings. Examples of these refactorings are given in Appendix A.

2.2 Conditional Refactorings

ScalaF also transforms nested `if-else` and `if-else-if` ladders into idiomatic `match` expressions or pattern-matched functional constructs. Parallelism is included when map/filter transformations are applied over collections. Examples of these refactorings are given in Appendix A.

2.3 Parallelization-Aware Refactorings

In addition to transforming control-flow structures into functional equivalents, ScalaF also detects opportunities for parallel execution using Scala's parallel collections [3]. These refactorings improve performance for CPU-bound tasks, especially when element-wise operations are independent and order-insensitive.

Scala supports parallel execution via collections under the `scala.collection.parallel` package, such as `ParVector`, `ParArray`, and `ParSet`. Any standard collection can be parallelized using the `.par` conversion. For example:

```
val nums = (1 to 1_000_000).toList
val sum = nums.par.reduce(_ + _)
```

ScalaF statically detects such cases and recommends `.par` transformation when applicable. These refactorings are especially beneficial in multi-threaded or compute-intensive applications, enabling better CPU utilization with minimal code changes. Because parallel collections use the fork/join framework underneath, they abstract away thread management and promote safe, functional concurrency.

3 VSCode Plugin Implementation

The frontend plugin is developed using TypeScript and the VSCode Extension API. It provides real-time integration with the Scala backend, automatically detecting and suggesting refactorings as developers write or modify code.

In particular, the plugin invokes a local backend JAR via Node's `child_process.spawn`, feeding the active file content through stdin. The backend returns a structured JSON containing a list of suggestions, each with original code location, transformed output, and corresponding Scala method. The plugin then registers a `HoverProvider` that surfaces suggestions when the user hovers over refactorable code. It also registers a `CodeActionProvider` which enables users to apply refactorings interactively. Finally, we created a dedicated `SidebarViewProvider` to list all refactorable entries in the file; developers can preview, copy, or apply suggestions directly from this sidebar.

We have also implemented performance optimizations including memoization for repeated analysis, and bundling the backend as a fat JAR using `sbt-assembly`, to reduce startup latency and to avoid repeated compilation.

We summarize the features of the VSCode Plugin below:

- *Live developer feedback.* Continuous background analysis integrates with the VSCode UI to highlight actionable refactorings without interrupting development.
- *Idiomatic Scala suggestions.* The tool identifies common imperative constructs and recommends concise, functional alternatives such as `map`, `filter`, and `match`.
- *Modular and extensible design.* Both frontend and backend are modular. New analysis rules or refactoring templates can be added without affecting existing features.
- *Interactive sidebar interface.* Developers can browse, preview, and apply refactorings from a custom sidebar panel, improving accessibility.
- *Open source and portable.* The plugin and backend are open-source, platform-independent, and can be extended to support other editors or languages with minimal effort.

3.1 Developer Experience

While this paper focuses on the architecture and design of ScalaF, the accompanying talk will illustrate the developer experience. Specifically, we will show how the plugin highlights refactorable code, surfaces hover-based suggestions, provides sidebar navigation, and supports preview-before-apply workflows with concrete examples.

4 Evaluation

We evaluated ScalaF on synthetic Scala programs ranging from 100 to 5000 lines of code, executed on a Core i7 machine with 8GB RAM. The focus was on measuring refactoring throughput and responsiveness.

Accuracy of Transformations. ScalaF prioritizes precision by surfacing only semantics-preserving refactorings. Recall may be lower since the tool is conservative in hybrid codebases, ensuring correctness even if some opportunities are missed.

Performance Benchmark. Refactoring loops into functional constructs such as `par.reduce` enables parallel execution, which can improve runtime on multi-core machines with minimal code changes.

Table 1 reports average times (in seconds) for tree construction and loop/conditional refactorings. The tool scales well, completing all transformations under 3 seconds for 5k LoC.

Most time is spent in backend syntax tree generation. We mitigate this via backend packaging and parallelized refactorings using `.par`, enabling fast, scalable suggestions for functional code improvements.

Table 1. Refactoring Time vs Code Size

LoC	Tree	Cond.	Loop
100	0.61	0.14	0.40
1k	1.66	0.66	1.05
5k	2.73	1.18	2.10

5 Limitations

ScalaF applies only semantics-preserving transformations and avoids trivial rewrites. In hybrid codebases mixing mutable and immutable constructs, it is conservative, sometimes leaving constructs unrefactored to ensure correctness. Purity and side-effect analysis are not yet integrated.

6 Conclusion and Future Work

In this proposal, we described a practical and extensible system called ScalaF for assisting Scala developers in writing idiomatic, functional code through automated refactoring suggestions. By bridging static code analysis with interactive editor integration, ScalaF encourages better coding practices without disrupting developer workflow. ScalaF's modular design, real-time feedback, and open-source availability position it as both a productivity aid and an educational resource. As we expand ScalaF's capabilities, our goal remains to make functional programming in Scala more intuitive, accessible, and part of the everyday development experience.

In future, we plan to expand ScalaF's capabilities in both depth and usability. On the refactoring front, upcoming enhancements include support for advanced functional patterns such as currying, partial application, and function composition. We also aim to integrate automatic, version-aware rewrites that adapt to language-level changes. On the user experience side, future iterations of the plugin might support inline previews using VSCode CodeLens, custom rule support, and IDE portability using a language server protocol (LSP) implementation.

A Appendix: Refactoring Examples

This appendix presents representative refactorings detected and suggested by ScalaF across a wide range of control-flow constructs. The examples are grouped into conditional refactorings (below) and loop refactorings (Table 2), showing original imperative code and its functional counterpart obtained using ScalaF.

Original:

```
for (x <- xs) {
  if (x == 1) result += f(x+1)
  else if (x == 2) result += f(x+2)
  else result += f(x+3)
}
```

Refactored:

Table 2. Examples of loop refactorings suggested by ScalaF.

Original Code Snippet	**Refactored Output**	**Used Scala Methods**
`for (x <- xs) sum += x`	`xs.par.reduce((x, y) => x + y)`	`reduce + par`
`for (x <- xs) println(x)`	`xs.par.foreach(println)`	`foreach + par`
`for (x <- xs) list += g(x)`	`xs.par.map(g)`	`map + par`
`for (x <- xs) if (h(x)) list += g(x)`	`xs.par.filter(h).map(g)`	`filter + map + par`
`while (h(xs(i))) { list += xs(i); i += 1 }`	`xs.par.takeWhile(h)`	`takeWhile + par`
`while (i < xs.length && xs(i) != x) i += 1`	`xs.par.indexWhere(_ == x)`	`indexWhere + par`
`for (x <- xs; i <- x) ans += i`	`xs.par.flatten`	`flatten + par`
`for (x <- xs) if (!l.contains(x)) l += x`	`xs.par.distinct`	`distinct + par`

```
result = xs.par.map {
  case 1 => f(x+1)
  case 2 => f(x+2)
  case _ => f(x+3)
}
```

Original:

```
if (password == enteredPassword)
  println("Authenticated")
else
  println("Invalid")
```

Refactored:

```
(password == enteredPassword) match {
  case true => println("Authenticated")
  case false => println("Invalid")
}
```

References

[1] CompL. 2025. *ScalaF BackEnd: Part of the ScalaF Refactoring Framework.* https://github.com/CompL-Research/ScalaF-Backend

[2] CompL. 2025. *ScalaF VSCode Plugin: Part of the ScalaF Refactoring Framework.* https://github.com/CompL-Research/ScalaF-VSCode

[3] Scala Documentation. 2025. Parallel Collections in Scala. https://docs.scala-lang.org/overviews/parallel-collections/overview.html. Accessed: 2025-07-19.

[4] Namrata Malkani and Manas Thakur. 2021. *Refactoring Scala Programs to Promote Functional Design Patterns.* https://www.cse.iitb.ac.in/~manas/docs/posters/ecoop21-a.pdf

[5] Scalameta. 2025. *Scalameta: Library to Read, Analyze, Transform and Generate Scala Programs.* https://scalameta.org