

Refactoring and Verification in Rust
Expanding the REM toolchain
And providing a novel approach to verification

Matthew Britton

A Research and Development report submitted as part of
ENGN3712: Engineering Research and Development Project

The Australian National University

Research School of Computing, School of Engineering



July 2025

© Copyright by Matthew Britton, 2025

All Rights Reserved

Disclaimer

I hereby declare that the work undertaken in this R&D project has been conducted by me alone, except where indicated in the text. I conducted this work between September 2024 and July 2025, during which period I was an Engineering (R&D) student at the Australian National University. This report, in whole or any part of it, has not been submitted to this or any other university for a degree.

Acknowledgements

TODO write acknowledgements

Abstract

TODO Abstract

Contents

List of Figures

List of Tables

Glossary

Term	Definition
AoL	Adventure of a Lifetime. ?.
GC	Garbage Collector.
REM	Rust Extraction Maestro, the refactoring toolchain implemented in Borrowing Without Sorrowing ?

Table 1: Glossary of Terms

Introduction

Rust is a modern systems programming language that aims to reconcile low-level control with strong safety guarantees. This is in stark contrast to previous norms, where a program was often forced to choose between low-level control (with languages such as C/C++) and memory safety (with garbage collected languages such as Java). However, as prior work like *Adventure of a Lifetime* (AoL) and the subsequent *Borrowing without Sorrowing* plugin have shown, these very features that make Rust robust also complicate automated refactoring - particularly “Extract Method” transformations that are straightforward in garbage collected (GC) languages. Building on these initial efforts, this paper extends the Rusty Extraction Maestro (REM) tool, removing dependencies on outdated IDEs, handling a broader array of Rust constructs (including `async/await` and generics), and introducing a more rigorous verification strategy to ensure that refactorings preserve semantics. By providing a standalone command-line interface and a proof-of-concept Visual Studio Code extension, this project lays the groundwork for seamlessly integrating safe, behavior-preserving “Extract Method” refactoring into mainstream Rust development workflows.

0.1 Motivation and Background

It is an inevitable fact of software development that the engineer will spend a substantial amount of their time modifying and improving existing codebases. Such modifications will range from small bug fixes to complete architecture overhauls. Regardless of how large (or small) the situation is, refactoring - systematic restructuring of the codebase without altering its external behaviour - plays a pivotal role in maintaining code quality and limiting code smell over time. Despite refactoring being commonplace in many systems programming languages like C++ or Java, implementing automated refactoring tools remains challenging, particularly in languages with advanced features and strict safety guarantees, such as Rust.

When it comes to memory management, two schools of thought have dominated for the last half-century. One tradition leans on automated techniques like garbage collection,

while the other relies on manual, explicit management. In this landscape, Rust stands out by enforcing memory safety and concurrency through its rigorous system of ownership, borrowing, and lifetimes. Yet, the very features that make Rust safe also introduce massive hurdles for automated refactoring. A naive, “copy and paste” implementation of extract method refactoring that works in more forgiving languages like Python and Java will hit the proverbial brick wall that is the Rust borrow checker. Whilst the borrow checker and other compilation checks will prevent unintended consequences, the programmer must now deal with a whole host of other issues.

Against this backdrop, *Adventure of a Lifetime* (AoL) ? introduced the conceptual framework for automated “Extract Method” refactoring in Rust. From there, *Borrowing without Sorrowing* ? implemented the concepts outlined in *Adventure of a Lifetime* as an IntelliJ IDEA plugin. This project now builds on those foundations, aiming to extend, improve and explore new frontiers for automated Rust refactoring.

0.1.1 Do People Actually Refactor?

Despite widespread agreement on the value of continuous, incremental code improvements, one might still wonder: do development teams truly refactor in practice, or is it just an idealized principle? A recent large-scale survey of 1,183 professional developers by the JetBrains team, *One Thousand and One Stories: A Large-Scale Survey of Software Refactoring* ?, sheds light on this question. The results show that refactoring is unquestionably a routine part of modern software engineering:

- **Frequent Refactoring:** Over 77% of surveyed developers reported refactoring code at least once a week, with 40.6% doing so “almost every day” and another 36.9% at least every week.
- **Lengthy Sessions:** Refactoring can be time-intensive; 66.3% of respondents indicated they had spent an hour or more in a single session refactoring their code.
- **Rare “Non-Refactorers”:** Only 2.5% stated that they “never” refactor code. Even within that group, many had at least renamed or relocated code elements—activities the survey authors classify as refactoring.

These findings confirm that refactoring is more than just a best-practice slogan: most teams regularly restructure and clean up their code. At the same time, the significant number of hour-plus refactoring sessions suggests that technical debt can build up to a point where larger-scale efforts become necessary. Understanding how, why, and when developers refactor can thus guide the design of more effective refactoring tools and strategies.

0.1.2 Importance of Refactoring in Software Engineering

In large projects, codebases can quickly become very unwieldy. Refactoring - the process of improving a programs internal structure without altering its external behaviour ? - is a crucial tool that helps engineers keep the technical debt at a manageable level. It ensures that system complexity remains manageable by promoting cleaner abstractions, fewer side effects, and much clearer data flows.

Additionally, whilst refactoring itself doesn't add new features, it often uncovers latent defects and unexpected behaviours or side effects. It can be used to bring clarity to confusing, hundred-plus line methods, and makes testing significantly more straightforward.

Finally, well-factored code is much easier to understand and thus modify. This, in turn, will speed up future iterations, feature implementations, and onboarding of new team members.

0.1.3 What is “Extract Method” Refactoring?

“Extract Method” is a common refactoring technique where a contiguous block of code (often a few lines or a loop body) is extracted into its own function. The original code is replaced with a call to this newly created function. By breaking down the large methods or blocks into smaller, aptly named functions, readers can far more quickly grasp the purpose of each piece of logic.

In Java or other garbage-collected languages, method extraction is straightforward: you collect parameters, create a new function, and replace the original lines with a function call. Rust, however, introduces additional layers of complexity due to ownership and borrowing. The example below (in Java) illustrates how this technique works at a superficial level.

```
1 // Before:
2 public void processNumbers(List<Integer> nums) {
3     int sum = 0;
4     for (int n : nums) {
5         sum += n;
6     }
7     System.out.println("Sum is: " + sum);
8 }
9
10 // After:
11 public void processNumbers(List<Integer> nums) {
12     int sum = calculateSum(nums);
13     System.out.println("Sum is: " + sum);
14 }
15
16 private int calculateSum(List<Integer> nums) {
17     int sum = 0;
18     for (int n : nums) {
19         sum += n;
```

```
20     }  
21     return sum;  
22 }
```

0.1.4 Why Rust Poses Unique Challenges

1. **Ownership:** Each value in Rust has a single owner that governs the value's lifetime. Once the owner goes out of scope, the value is dropped. This design eliminates many types of memory errors (such as use-after-free), but can complicate refactoring: relocating or splitting functionality into separate functions may transfer or divide ownership in unexpected ways.

```
1 fn main() {  
2     let s1 = String::from("Hello");  
3     let s2 = s1; // Ownership of the string data moves to s2  
4     // The following line would be invalid if uncommented:  
5     // println!("{}", s1);  
6     // ~ value borrowed here after move  
7     println!("{}", s2); // Prints "Hello"  
8 }
```

In the above example, `s1` is “moved” into `s2`, making `s1` no longer valid afterward. If a refactoring operation was to pass `s1` into a function that takes ownership, the original owner cannot access it unless ownership is returned. Alternatively, we could end up with code that refuses to compile, forcing the developer to manually debug and fix the refactored code.

2. **Borrowing:** Instead of copying or moving data, Rust encourages passing references. However, these references can be immutable (`&T`) or mutable (`&mut T`), and only one active mutable reference is allowed at a time. When extracting logic into a separate function, it is important to ensure that references remain valid and continue to adhere to Rust's borrowing rules. The examples below demonstrate how borrowing can be effectively handled. However, if multiple references with overlapping mutable access are introduced by refactoring, the compiler's rules may require parameter signatures or function boundaries to be revised accordingly.
3. **Lifetimes and their Management:** The Rust compiler tracks how long references live to guarantee memory safety. Any function that takes (or returns) references must define or infer lifetime parameters. When we write Rust code normally, such as the example below, the compiler is capable of inferring these lifetimes for the programmer.

```
1 fn echo(message: &str) -> &str { message }
```

When the programmer writes `echo(message: &str) -> &str`, the compiler is quietly inferring a generic lifetime for both the parameter and the return value. In effect, the compiler treats this as if the programmer wrote:

Immutable Borrow	Mutable Borrow
<pre> fn print_message(message: &str) { println!("{}", message); } fn main() { let greeting = ↪ String::from("Hello, ↪ world!"); print_message(&greeting); // ↪ Borrows greeting immutably println!("Still own greeting: ↪ {}", greeting); } </pre>	<pre> fn add_exclamation(message: &mut ↪ String) { message.push('!'); } fn main() { let mut greeting = ↪ String::from("Hello"); add_exclamation(&mut greeting); ↪ // Borrows greeting mutably println!("{}", greeting); ↪ // Prints "Hello!" } </pre>

Table 1: Immutable and mutable borrowing in Rust

```

1 fn echo<'a>(message: &'a str) -> &'a str { message }

```

So, behind the scenes, Rust sees that `message` is a reference that lives at least as long as `'a`, and it infers that the output must share that same `'a` lifetime. The programmer doesn't need to write `<'a>` or annotate the references explicitly, because the language automatically applies its lifetime inference rules whenever possible.

If engineers later extract code into new functions—especially if multiple reference parameters or more complex borrowing patterns are involved—the compiler might no longer be able to figure out the relationships between references, forcing programmers to specify such `'a` parameters explicitly.

4. **Non-local Control Flow** Rust allows `return`, `break`, and `continue` inside code blocks. Extracting a fragment of code containing, for example, an early `return`, into a function changes the flow of control unless handled explicitly.

0.1.5 Adventure of a Lifetime: Automated Refactoring for Rust

A foundational contribution toward automated refactoring for Rust was made by Sergey et al. in *Adventure of a Lifetime* ?. While “Extract Method” refactoring is relatively straightforward in languages like Java, the challenges discussed in the previous section demonstrate why a similar algorithm for Rust is far from trivial. To tackle these unique challenges, *Adventure of a Lifetime* introduced a specialized refactoring framework tailored to Rust's complexities.

Their approach addressed Rust-specific issues using multiple specialized passes. For example, one pass systematically encodes `return`, `break`, and `continue` statements into an auxiliary enum, allowing the caller function to reconstruct the original control flow by matching on variants such as `Ok(...)`, `Return(...)`, etc. Another pass utilizes

constraint-based ownership analysis to precisely determine and assign the correct ownership levels (owned or borrowed references) to variables involved in extracted code fragments.

The systematic handling of these issues ensures correctness with respect to ownership, borrowing, and lifetimes, thereby overcoming the primary hurdles faced when naively transferring techniques from languages like Java directly into Rust. Figure ?? from *Adventure of a Lifetime* outlines the complete refactoring process.

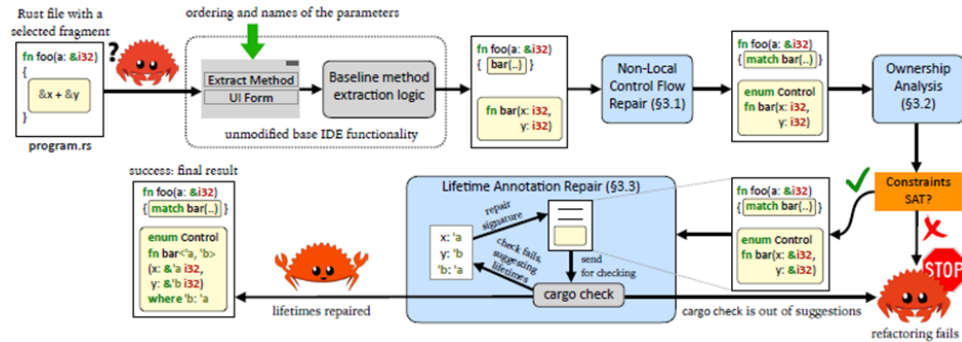


Figure 1: The refactoring process as outlined in *Adventure of a Lifetime*

The authors implemented this approach as *Rusty Extraction Maestro* (REM) on top of IntelliJ IDEA’s Rust plugin. Their empirical evaluation on multiple real-world Rust projects demonstrated that REM handles a wider range of extractions than existing IDE tools, including scenarios involving multiple borrows and nested references. Specifically, REM successfully managed 37 out of 40 tested cases, outperforming both IntelliJ IDEA alone and VSCode’s Rust Analyzer. The study also showed that REM-generated functions extracted quickly (on the order of one to two seconds), making it practical for everyday development. Most of the overhead arose from repeated calls to `cargo check`, although the delay remained acceptable for typical usage scenarios.

However, the evaluation also identified several limitations. REM struggles with code involving complex generics or elaborate trait bounds due to limited type inference capabilities in IntelliJ’s Rust support. Additionally, it lacks support for asynchronous programming constructs (`async/await`) and is unable to correctly extract functions that partially move struct fields. Macros beyond standard-library macros frequently disrupt refactoring attempts. Lastly, REM requires (and also assumes) a compilable project state, as it requires compilation checks to aid with its analysis. It often generates functions with overly verbose type signatures due to explicit lifetime and ownership annotations, despite the application a robust set of lifetime elision rules.

0.2 Problem Statement

0.2.1 Issues with Existing Automated Refactoring Tools

Rust’s unique ownership, borrowing, and lifetime system presents challenges that traditional refactoring tools don’t typically encounter. In languages like Java or Python, transformations such as “Extract Method” can be applied directly with minimal risk of breaking the code. However, Rust’s strict guarantees mean that naïvely lifting code into a new function often results in compilation errors or unintended changes in behavior due to ownership conflicts, borrowing rules, or lifetime mismatches. As a result, programmers often have to manually fix the refactored code—a process that can take significantly longer than rewriting it from scratch. While automated refactoring tools for Rust, such as RustRover and the IntelliJ Rust plugin, handle simple extractions well, they struggle with more complex constructs due to these inherent language constraints.

For instance, IntelliJ IDEA’s Rust plugin and Visual Studio Code’s Rust Analyzer both handle simpler scenarios—like extracting a small code block without complex borrows—but rely heavily on IntelliJ’s type inference or `rustc`. They often provide incomplete coverage for features such as `async/await`, generics with intricate trait bounds, or macros, leading the tools either to produce incorrect code or to refuse to recognize an extraction altogether. Moreover, these tools commonly struggle with non-local control flow, such as an early `return` from deep within a nested block.

By contrast, REM specifically targets these Rust-specific pain points. First, it applies a specialized ownership analysis algorithm that determines exactly how data should be passed—owned versus borrowed—in an extracted method. Second, it reifies non-local control flow operations like `return`, `break`, and `continue` by translating them into a custom enum and then patching the call-site logic. Finally, REM refines the resulting function’s lifetime annotations in a loop, using `cargo check` to repeatedly gather compiler feedback. This structured approach greatly expands the variety of Rust code patterns that can be safely refactored.

However, REM has several limitations. It heavily depends on the IntelliJ IDEA plugin for Rust, which is outdated and no longer supported. Additionally, it relies on numerous undocumented and rapidly changing `rustc` internals, leading to frequent build issues as third-party packages update and become incompatible. Another major drawback is that REM can only refactor code that compiles, as it relies on `cargo check`, which still requires successful compilation. Lastly, it passes multiple temporary files around on disk, making it incompatible with tools like `rust-analyzer`, which operate entirely in memory through the Language Server Protocol.

0.2.2 Need for Verification

An automated refactoring toolchain is only as good as its ability to preserve the program’s behavior. Rust refactoring poses particular challenges here due to its strict ownership and lifetime rules. A good refactoring must ensure that the transformed program maintains the exact observable behavior as before. Although comprehensive test coverage is the primary practical means of ensuring the preservation of functionality, tests are often slow and limited—they identify only *that* a problem exists, not precisely *where* it was introduced.

Verifying semantic preservation after transformations is notoriously challenging, particularly in Rust. Rust’s unique language features complicate semantic preservation proofs or automated verification techniques. Moreover, Rust code produced by REM may include constructs—such as enums for reified control flow or explicit lifetime annotations—that can appear unfamiliar or unnecessarily complex to human readers. While this complexity does not imply semantic changes, confirming correctness remains essential. Therefore, employing rigorous semantic validation (automated, partial, or manual) alongside refactoring is necessary to ensure transformations remain safe, accurate, and reliable.

0.3 Project Aims and Objectives

This project seeks to improve and extend the capabilities of the **Rusty Extraction Maestro** (REM) tool while making the refactoring process more accessible and verifiable. Specifically, the aims can be grouped into three key areas: enhancing REM’s standalone functionality, establishing a solid approach to verifying refactorings, and developing a proof-of-concept **VSCo** extension. A long-term ambition is to explore potential integration with **Rust Analyzer** if future architectural changes permit.

0.3.1 Improve and Extend REM

A principal goal is to remove REM’s existing dependencies on both **rustc** and IntelliJ IDEA, allowing the tool to function independently. By offering a standalone solution, REM would be simpler to incorporate into various workflows, free of the overhead or version constraints tied to external IDEs and compilers. This will be achieved through a single comprehensive CLI that links all stages of REM together.

To ensure broader adoption, another focus is to provide **VSCo** integration in the form of a user-friendly plugin. This plugin should install easily, require minimal setup, and offer automated **Extract Method** support for Rust directly in Visual Studio Code. In parallel, the project aims to expand REM’s capabilities to handle advanced Rust features, such as **async/await** constructs, macros, closures, **const** functions, and more intricate

generics. Addressing these challenging language constructs will make **REM** suitable for a greater variety of real-world Rust codebases.

0.3.2 Verification of **REM** / Refactorings

Automated refactorings are only meaningful if they preserve the program’s original semantics. While typical “copy-and-paste” style transformations rarely alter program logic, **REM**’s deeper rewrites—which may include lifetime adjustments or reified control-flow enums—require stronger assurances that the resulting code truly behaves as intended. To achieve this, the project intends to develop a theoretical framework (or algorithm) for proving equivalence between the original and refactored code.

An emerging strategy involves functional translation—moving refactored Rust code into a simpler, more mathematically tractable language subset or an IR for proof purposes. Recent development to toolchains like **CHARON?** and **AENEAS?** can assist by allowing for a complete translation of a Rust program into a set of functional, proof oriented languages like **F*** and **Coq**. Though complete verification of all Rust code remains infeasible due to the language’s evolving semantics, partial or scenario-specific proofs can still bolster confidence in **REM**’s correctness. Demonstrating a handful of end-to-end examples—from extraction in Rust to verified equivalence in a proof assistant—will illustrate the viability of this approach. Moreover, the verification pipeline has the potential to be integrated into the main body of **REM**, providing the programmer with a guarantee that their code’s functionality hasn’t been affected or alternatively, the verification tool can be leveraged to execute a comprehensive series of case studies that further validate **REM**’s functionality.

0.3.3 Build a Proof-of-Concept Extension for VSCode

A strong emphasis is on producing a proof-of-concept **VSCode** extension that showcases **REM** in action within one of the most commonly used Rust development environments. This extension should seamlessly integrate into a developer’s normal workflow, from code editing to debugging, without burdensome setup. By offering an easy-to-install plugin, the refactoring workflow becomes more accessible, and potential refinements can be validated in real-world coding sessions.

A core demonstration will involve extracting method fragments in a variety of Rust files and showing how the extension generates well-typed, behavior-preserving code. The key detail here is that **REM** relies on its new, standalone core, instead of a platform or program specific implementation.

0.3.4 Potential for the Long Term Goal of Merging REM with Rust Analyzer

Finally, looking beyond the immediate development schedule, an intriguing possibility is merging REM's functionality with **Rust Analyzer**, the official language server for Rust. **Rust Analyzer** already converts source code into an intermediate representation to power syntax highlighting, code completion, and basic refactorings. If REM can leverage that IR, it might inject advanced lifetime analysis and partial verification steps directly into the analyzer pipeline.

However, **Rust Analyzer** is deliberately kept lightweight and incremental, while REM aspires to be a comprehensive solution that tackles complex rewriting tasks. Hence, if integration proves overly burdensome, a compromise could be to develop a stripped-down subset of REM for merging. This long-term exploration depends on how **Rust Analyzer** evolves and whether the Rust community adopts more sophisticated refactoring workflows within a single unified environment. The alternative solution of a standalone **VSCode** extension designed to be used alongside **Rust Analyzer** is also seen as acceptable in this case.

Literature Review

1.1 Introduction

Rust is a modern systems programming language that enforces memory safety through a strict ownership and borrowing discipline. This discipline, enforced by Rust’s borrow checker, ensures that well-typed Rust programs are free from data races and dangling pointers without requiring a GC ?. However, Rust’s unique type system — centered on ownership and lifetimes — poses new challenges for code transformation tools and formal reasoning. This literature review surveys academic work on automated refactoring techniques for Rust, approaches to verification of Rust programs (including formal methods and model checking), and efforts to formalize Rust’s semantics. We compare Rust’s refactoring and verification tooling to those of other languages, and we discuss integration with development environments. The goal is to highlight the state of the art in making Rust programs easier to evolve and prove correct, while pinpointing structural refinements for a clearer organization of this body of work.

1.2 Overview of Refactoring

As previously mentioned, refactoring is the process of restructuring existing code to improve its internal structure without changing its external behavior. It is a key practice for enhancing code maintainability and reducing technical debt in software projects ?. Modern IDEs, such as Eclipse and IntelliJ IDEA include a wide array of automated refactorings, providing developers with quick, semantics-preserving code transformations ?. In theory, these automated refactorings offer a safe way to restructure code (backed by well-defined preconditions to preserve behaviour ?) and should be widely used in practice.

However, empirical studies show a disparity between tool support and actual usage. Developers often remain reluctant to use automated refactoring features, preferring manual code changes ?. For example, even with “Rename” refactoring available at the click of a button, many programmers still rely on find-and-replace or other manual methods ?.

The survey also noted that many developers expressed a desire to better understand their IDE’s refactoring capabilities. This suggests that while the tools exist, there is a gap in awareness and useability of the tooling available. Additionally, as evidenced by the lack of adoption of even “Rename” refactorings, challenges still remain in getting developers to trust automated tools to perform complex manipulations.

Research into automated refactoring aims to bridge this gap by making tools more reliable and intelligent. For instance, Tip (2007) demonstrated that refactorings can be modeled with type constraints to systematically explore alternative valid program structures ?. Such approaches use static analyses to ensure that transformations maintain type correctness and behavioral equivalence, giving developers confidence in the automation. Nevertheless, the human factor (usability, understandability of tool behavior) remains critical.

1.3 Rust’s Ownership Model and Formal Semantics

Rust’s ownership model introduces compile-time enforced rules for aliasing and memory lifetime. Each value in Rust has a single owning scope, and references (borrows) must obey strict rules that prevent concurrent mutation and use-after-free ?,?. While this model provides strong safety guarantees, it complicates formal semantics and tool development. Early efforts to rigorously define Rust’s semantics culminated in *RustBelt*, which provided the first machine-checked safety proof for a realistic subset of Rust’s type system ?. *RustBelt* demonstrated that safe Rust code (even when using unsafe internals of standard libraries) is memory and thread safe, by modeling Rust in the Coq proof assistant ?. This foundational work established confidence in Rust’s core design and set the stage for verifying more complex properties. Subsequent research has extended Rust’s semantic foundation to cover aliasing in unsafe code (e.g. Stacked Borrows and Tree Borrows models for pointer aliasing) ?, and to connect Rust’s high-level rules with low-level memory models. These formal models provide a basis on which verification tools and refactoring tools can reason about Rust code behavior. In summary, Rust’s enforced aliasing discipline, while posing challenges, has inspired a rich line of work in formal semantics that underpins safe refactoring and verification.

1.4 Automated Refactoring in Rust vs Other Languages

Well established languages like java and C# benefit from decades of IDE support for refactorings such as *Rename*, *Extract Method* and *Extract Module / Variable*. Rust, being newer and with far more complex compile-time rules, has lagged behind in automated refactoring support ?. The ubiquitous *Extract Method* is “widely used in all major IDEs” for other languages, but implementing it for Rust is surprisingly non-trivial due to Rust’s ownership and lifetime constraints ?. In contrast to a language like Java, where extracting a function involves mostly syntactic rearrangement, Rust’s refactoring tools must also infer where to borrow or clone data and how to introduce lifetime parameters to satisfy the borrow checker ?, ?. Early comparisons noted that Rust’s lack of reflection and unstable compiler APIs made it harder to build refactoring tools as robust as those for Java / C#. Even in the two years since REM was released, compiler APIs have changed so much that it proved impossible to compile the original project without significant rewrites. On the positive side, Rust’s compiler errors can guide manual refactorign by pinpointing violations of borrowing rules, giving programmers immediate feedback; this meansd that if a refactoring compiles in Rust, it is likely behaviour-preserving ?. Nonetheless, the consensus in both academia and industry is that first-class refactoring tools are needed to manage large Rust codebases with the same ease developers expect in other environments ?, ?.

1.5 Automated Refactoring Techniques for Rust

1.5.1 Renaming and Simple Refactorings

1.5.2 Extract Method (REM)

1.5.3 Automated Fixes for Ownership Errors

1.5.4 Comparison and Summary

1.6 Verification and Formal Methods for Rust

Rust’s appeal for building reliable software has spurred interest in applying formal verification techniques to Rust programs. Traditional deductive verification and model checking must be adapted to account for Rust’s ownership, lifetimes, and possibly unsafe code. In this section, we review key verification tools and techniques, and the formal semantics advances enabling them.

1.6.1 RustBelt and Type-System Soundness

1.6.2 Verification via Functional Translation (CHARON and AENEAS)

1.6.3 Static Analysers and Model Checkers

1.6.4 Summary

1.7 Integration with IDEs and Language Servers

1.7.1 Compiler Internals for Ownership and Borrowing

1.7.2 Industry Perspective

1.8 Related Work Beyond Rust: Refactoring in Other Languages and Paradigms

The challenges and advances in refactoring outlined throughout this review are not unique to Rust. Many other languages have inspired research and into automated refactoring and its many pitfalls. The easiest case study is Java, being one of the earliest targets of refactoring tools, and it has a rich literature. Beyond the generics case study mentioned earlier (?), researchers have looked at refactoring concurrent Java code, refactoring to use new language features, and large-scale restructuring of legacy systems. Zhang et al. (2024) recently proposed an automated refactoring approach for Java’s asynchronous programming using the `CompletableFuture` API. Their tool, **ReFuture**, integrates static analysis (visitor patterns, alias analysis, etc.) to identify where an older async construct (like using raw threads or Futures) can be transformed into a modern `CompletableFuture` chain ?. Impressively, **ReFuture** was evaluated on 9 large Java projects, including *Hadoop* and *ActiveMQ*, and managed to refactor 639 out of 813 eligible code segments automatically, without introducing errors ?. This demonstrates that automated refactoring can manage to tackle performance or paradigm-migration tasks, not just cosmetic and maintenance changes.

In the mobile app domain, Lin et al. (2015) addressed refactoring for Android’s asynchronous constructs. Android apps often mis-use the `AsyncTask` class, leading to memory leaks or lost UI updates. Lin and his team developed **ASYNCDROID**, a tool to refactor improper `AsyncTask` usage into a safer structures like `IntentService` ?. Their formative study found that about 45% of the `AsyncTask` occurrences in real apps could be automatically refactored by **ASYNCDROID**’s rules, and an additional 10% could be handled with minor manual tweaks, resulting in more than half of problematic cases being fixed

largely automatically. The tool was implemented as an Eclipse plugin and offered typical refactoring tool conveniences including a preview of the changes and a rollback option. Importantly, when the authors submitted some of these refactoring patches to open-source projects, the maintainers accepted them, validating that the transformations were indeed considered improvements with no behavioral regressions ?. This work illustrates how automated refactoring can be applied in a specialized context (mobile asynchronous UI code) by encoding domain-specific knowledge as transformation rules and preconditions.

Other notable refactoring research outside of Rust includes work on functional languages and dynamically-typed languages. In Haskell, for example, the HaRe tool (Li et al., 2005) brought refactoring support to a lazy, purely functional language, requiring careful handling of Haskell’s scoping, type classes, and purity concerns ?. In dynamic languages like JavaScript and Python, refactoring is complicated by the lack of static types; researchers have explored heuristic and runtime-analysis-based approaches to perform refactorings safely (e.g., refactoring Python’s modules or JavaScript’s callbacks to promises). While we will refrain from delving into specific studies for those here, it is worth noting that each language tends to spawn its own refactoring research to address unique features.

Finally, it is important that we note that refactoring is very closely intertwined with other software engineering tasks like code smell detection, program repair and modernisation. Refactoring tools are sometimes used as building blocks for automated program repair, such as applying behavior-preserving transformations to enable a later bug-fixing change, with one such case even mentioned in *Adventure of a Lifetime* ?. Tools like Facebook’s JScodeshift or Python’s Bowler allow developers to script custom refactorings (essentially semi-automated transformations), which blurs the line between a “human-driven” refactoring and a tool-driven migration. The increasing adoption of language servers means refactoring capabilities can be made available in a wide range of editors, not just heavyweight IDEs ?. This has been a positive development for languages like Rust and Go, where a language server can provide refactoring assists consistently across many development environments.

1.9 Conclusions and Insights

Proposed Approach

2.1 High Level Strategy

2.2 Detailed Methodology

2.3 Verification Strategy

Implementation

3.1 System Architecture

3.2 Data Structures

3.3 Integration with External Tools

3.4 User Workflow and Experience

Evaluation and Experimental Results

- 4.1 Experimental Setup
- 4.2 Evaluation Criteria
- 4.3 Experimental Scenarios
- 4.4 Results and Analysis

Conclusions and Future Work

5.1 Summary of Contributions

5.2 Limitations

5.3 Potential Improvements

5.4 Broader Impact