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

Australian National University

July 2025

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 **?**
AST	Abstract Syntax Tree.
CFG	Control Flow Graph.
IR	Intermediate Representation.
MIR	Mid-level Intermediate Representation.
LLBC	Low-Level Borrow Calculus
SMT	Satisfiability Modulo Theories.
CMBC	C Bounded Model Checker
LSP	Language Server Protocol.
rustc	The Rust compiler.
IDE	Integrated Development Environment.

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Building on these initial efforts, this paper extends the Rusty Extraction Maestro (REM) tool, removing dependencies on outdated IDEs, handling a broader array of Rust constructs (including async/await and generics), and introducing a more rigorous verification strategy to ensure that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 wihtout 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.

```java
// Before:
public void processNumbers(List<Integer> nums) {
    int sum = 0;
    for (int n : nums) {
        sum += n;
    }
    System.out.println("Sum is: " + sum);
}

// After:
public void processNumbers(List<Integer> nums) {
    int sum = calculateSum(nums);
    System.out.println("Sum is: " + sum);
}

private int calculateSum(List<Integer> nums) {
    int sum = 0;
    for (int n : nums) {
        sum += n;
```

```
    }
    return sum;
}
```

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.

```rust
fn main() {
    let s1 = String::from("Hello");
    let s2 = s1; // Ownership of the string data moves to s2
    // The following line would be invalid if uncommented:
    // println!("{}", s1);
    // ^ value borrowed here after move
    println!("{}", s2); // Prints "Hello"
}
```

   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.

Immutable Borrow

```rust
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);
}
```

Mutable Borrow

```rust
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!"
}
```

Table 1: Immutable and mutable borrowing in Rust

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.

```rust
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:

```rust
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.

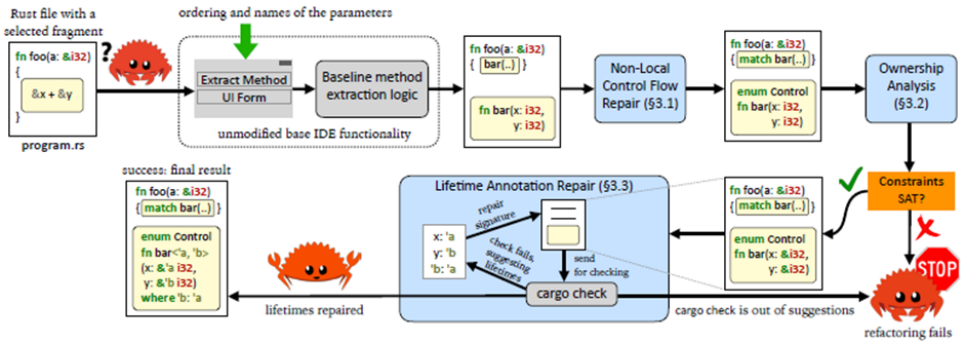

Figure 1: The refactoring process as outlined in *Adventure of a Lifetime*

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 `VSCode` 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 VSCode 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.

Chapter 1

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 wouthout changing its external behavior. It is a key practice for enhancing code maintainability and reducing technicanle debt in software projects **?**. Modern IDES, such as Eclipse and IntelliJ IDEA include a wide array fo automated refactorings, providing developers with quick, semantics- perserviding 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

Welll 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

One of the first efforts to build a Rust refactoring tool was by G. Sam et al. (2017), who created a proof-of-concept refactoring framework utilizing the Rust compiler's internal APIs. The team partnered with Mozilla Research to be among the first to implement the Rust specific refactorings of *Lifetime Elision* and *Lifetime Reification*. This allowed their program to introuce explicit lifetime parameters in instances where the compiler was implicitly inferring them - a refactoring that was brand new to Rust. The challenges encountered illustrated how Rust's stricter scoping and shadowing rules required careful handling of name conflicts during renaming (e.g. avoiding situations where renaming a variable could unintentionally shadow another) **?**. Additionally, the 2017 study concluded that many refactorings are possible with Rust's compiler infrastructure, but ensuring *behavioural preservation* (especially around ownership transfers) requires additional static analyses not needed in langauges without Rust's constraints. Their work on *Lifetime Elision* has since formed part of the REM toolchain, where the rich feedback from the

Rust compiler is leveraged to ensure that the final transformation is both valid Rust and as legible as possible **?**, **?**.

1.5.2 Extract Method (REM)

A major advance in Rust refactoring came with the Rust Extract Maestro (REM), with the theoretical background provided by the work of Costea et al. (2023) in *Adventure of a Lifetime: Extract Method Refactoring for Rust*. Thy et al. (2023) then provided a practical implementation of REM in *Borrowing Without Sorrowing*. REM is a tool that tackled notoriously complex Extract Method refactoring for Rust. REM decomposes the extract-method process into a sequence of transformations, each addressing a specific aspect of Rust's type and borrow rules **?**. The approach begins with a "naïve" function hoisting (moving a block of code into a new function), then applies a series of *automated program repairs* to make the code compile correctly. This process is guided by *oracles* - specialised analyses that resolve issues the compiler would flag. One such oracle is an intra-procedural **ownership analysis** which infers wheter each value moved to the new function should be passed by value, by shared reference (using `&T`), or by mutable reference (using `&mut T`). Appendix 1 (**??**) provides a detailed example of what this looks like in practice.

Another oracle leverages Rust's own compiler (rustc) as a sub-procedure: REM invokes Rust's lifetime checker to identify lifetime inconsistencies, then introduces appropriate lifetime parameters to the extracted function until the borrow checker is satisfied. By iteratively repairing borrow errors in the extracted code, REM ensures the transformation preserves semantics and yields a well-typed program. The REM tool, implemented as an extension to the IntelliJ Rust plugin, was shown to handle complex extractions that involve non-local control flow (e.g. `return` or `break` inside the extracted fragment) and borrowing across function boundaries. In an evaluation on real-world Rust projects, REM could successfully perform extractions that developers had done manually (including cases with nested lifetimes), outperforming existing tools in the scope of code it can handle. Their work demonstrated that seemingly "unsafe" transformations (like cutting a function in half) can be automated in Rust by coupling transformations with verification from the Rust compiler's own checks.

1.5.3 Automated Fixes for Ownership Errors

An alternative angle on Rust refactoring is automatically *fixing* code that violates ownership/borrow rules - essentially a refactoring from "non-compiling" to "compiling" code without changing behavior. **Rust-Lancet** (Yang et al., ICSE 2024) is a tool aimed at *automated ownsership-rule-violation fixing with behaviour preservation*. Given a Rust

source file that fails to compile due to borrow checker errors, Rust-Lancet analyzes the Abstract Syntax Tree (AST) and applies code transformations to eliminate the error while preserving the program's semantics **?**. Under the hood, it uses Rust parsing libraries (`syn` and `quote`) to manipulate the AST and generate patched code. In a similar manner to REM, Rust-Lancet employs an interative repair loop: it first applies a fix (for example, inserting a `clone()` on a value to appease ownership rules), then rechecks the compiler errors and repeats if futher fixes are needed. The tool includes correctness checks to avoid over-fixing; a *behaviour preservation* module validates that the inserted changes do not alter program outputs for typical cases. In an evaluation on over 150 real cases of ownership rule violations (gathered from Rust's test suite and prior research), Rust-Lancet was able to completely fix a large fraction of them with zero false positives (i.e. it never introduced an incorrect fix). It even outperformed suggestions given by the Rust compiler and baseline techniques using large language models, especially in scenarios requiring multiple coordinated edits. While Rust-Lancet is positioned as a bug-fixing tool, it overlaps with refactoring by automatically rewriting code in a semantics-preserving way. Its success highlights how Rust's rigorous rules can be leveraged - the tool knows a patch is correct when the compiler's checks pass and its behavior preservation validation succeeds **?**.

1.5.4 Comparison and Summary

Overall, automated refactoring in Rust is characterized by taking full advantage of the rich and sophisticated output of the Rust compiler. Tools like REM explicitly invoke the borrow checker and type checker as so called oracles **?**, whilst others, like Rust-Lancet, effectively automate a "fix until compile" loop wiht guarantees of behaviour preservation. This is in contrast to many refactoring tools for garbage-collected languages, which operate mostly on the AST or intermediate representation without needing to consider memory lifetimes. The research so far indicates that, with suitable abstractions (e.g. ownership analysis algorithms and lifetime inference strategies), even complex Rust refactorings can be automated soundly. As more of these techniques mature, we expect Rust's refactoring support to begin matching the breadth available for languages like C++ and Java, albeit with different underlying algorithms tailored to Rust's unique semantics.

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

Jung et al. (2018) introduced **RustBelt**, a foundational framework for reasoning about Rust's type system and memory safety. In their landmark paper, *RustBelt: Securing the Foundations of the Rust Programming Language*, they proved the soundness of Rust's core type system. By embedding a Rust-like language (λRust) in the Iris framework (a higher-order concurrent separation logic), RustBelt verified that well-typed safe Rust code cannot exhibit undefined behaviour **?**. This was the first independent verification of Rust's claims about memory and thread safety. Additionally, it proivdfed a foundation on which other verification effors could build: one can assume that safe code is memory-safe, allowing them to focus verification effort either on higher-level functional correctness or the remaining pitfalls of unsafe code.

1.6.2 Verification via Functional Translation (CHARON and AENEAS)

One prominent approach to verifying Rust programs is to reduce the problem to verifying *pure functional code*. **AENEAS** (Ho & Potzenko, ICFP 2022) is a toolchain that translates a substantial subset of Rust into a purely functional language, suitable for input to existing proof assistants or model checkers **?**. The key insight of Aeneas is that for many Rust programs (those not using interior mutability or unrestricted unsafe code), explicit memory reasoning can be eliminated. Aeneas introduces an intermediate representation called *Low-Level Borrow Calculus* (LLBC), inspired by Rust's mid-level IR (MIR), which makes the ownership and borrowing structure of a program explicit. Crucially, LLBC's semantics is *ownership-centric*: instead of a heap, it uses an abstraction of *loans* and *borrows* to track aliasing, meaning individual values carry information about what they borrow.

Ho and Protzenko went on to define a set of pure, value-based semantics for LLBC - one with no pointer addresses - therby "capturing the essence of the borrow mechanism" in a mathematical form **?**. They then translate LLBC into a pure lambda calculus[1], effectively converting Rust code into a functional program with the same behavior. The result is that one can verify properties of the Rust program by verifying the translated functional program using a theorem prover of choice. For example, one might translate Rust to Coq or Fstar and prove postconditions about the output, with Aeneas ensuring

[1]See this excellent introduction to the Lambda Calculus for more information: https://plato.stanford.edu/entries/lambda-calculus/

those properties carry back to the original Rust code.

To tackle tricky issues like termination of borrows across function boundaries, AENEAS introduced a notion of *backwards functions* to conservitavely approximate the "borrow graph" of a program and ensure the translated code remains termination-friendly. The approach was shown to significantly reduce the burden on proof engineers, since they no longer needed to reason about low-level memory safety - Rust's type system has already done the heavy lifting for them.

From there, Ho et al., (ICFP 2024) presented *Sound Borrow-Checking for Rust via Symbolic Semantics* **?**, with the goal of addressing two open questions from AENEAS's methodology.First, they established a link between Aeneas's LLBC and a traditional heap-based semantics, by proving that LLBC's admittedly unusual modelling is faithful to a standard memory model, ensuring that verifiying programs in LLBC is not "proving the wrong thing". Second, they introduced a set of formal *symbolic semantics* for Rust that act as a sound abstraction of concrete execution, and prove that these symbolic semantics are capable of correctly approximating Rust's behavior. In essence, they show that if a program is accepted by their symbolic interpreter (which enforces borrow rules), then there exists a corresponding concrete execution in a real heap that is memory-safe. This result formally validates the idea that the borrow checker (or a tool mimicking it) prevents all illicit behaviors.

A very important outcome of this work is a proof that the **symbolic interpreter can server as a verified borrow-checker** for LLBC programs. This provides increased confidence that tools like Aeneas (which rely on symbolic execution as an intermediate) are built on solid theoretical ground, and it paves the way for future verified compilers or static analyzers for Rust that incorporate a proven-correct borrow checker. The broader significance is that Rust's complex lifetime rules are now accompanied by a machine-checked proof of their soundness, something rare for industrial language features.

1.6.3 Static Analysers and Model Checkers

Static Analysis

Beyond full formal verification, several tools bring lightweight verification or bug-finding to Rust. **PRUSTI** (Astrauskas et al., OOPSLA 2019) is a static *deductive verifyer* that uses Rust's types to simplify program annotation and verification, **?**. Prusti translates Rust code into the Viper verification language, using Rust's lifetime and ownership information to implicitly encode permissions for verification **?**. This means a developer can write specifications (pre/post-conditions, invariants) for Rust functions, and Prusti will prove them, assuming no undefined behavior in safe code. The advantage of leveraging Rust's type system is evident - e.g., Prusti knows that two mutable references cannot

alias, which corresponds to a simple permission separation in the verification condition. Over the past few years, Prusti has grown to handle significant subsets of Rust, and it even provides a Visual Studio Code extension for interactive use **?**.

In a similar vein, *RefinedRust: A Type Sysem for High-Assurance Verification of Rust Programs* (Gäher et al., 2024) introduced a refinement type system for Rust that can verify both safe and certain unsafe code by embedding checking into the type system **?**. RefinedRust's types allow expressing rich safety and correctness properties, and a prototype tool translates annotated Rust into a Coq model for verification. This line of work brings Rust closer to languages like Dafny or Liquid Haskell in terms of having a path to verification integrated with the language's types.

Model Checking

The most notable tool for *model checking* is **Kani**, a bounded model checker for Rust developed by AWS. Kani translates Rust programs (including those with `unsafe` code) into verification conditions that are checked by a SAT/SMT solver (built on the C Bounded Model Checker (CBMC) backend) **?**. What sets Kani apart is that it operates on Rust's MIR (Mid-level IR), leveraging information about traits and generics directly rather than reducing to LLVM bitcode. By working at the MIR level, Kani retains high-level semantics like trait object behavior, which allowed the team to verify complex properties involving dynamic trait objects (Rust's form of dynamic dispatch). VanHattum et al. (ICSE 2023) describe how Kani handles dynamic trait dispatch soundly, making it the first model checker to support the full range of Rust trait features **?**. In experiments, they found that using Rust-specific knowledge (MIR and trait info) improved verification performance by 5× - 15× compared to an equivalent analysis at the LLVM level. Kani is being used to verify critical systems components (for example, AWS has applied it to verify parts of their cryptographic libraries and virtualization code), focusing on memory safety and user-provided assertions in performance-critical Rust code. Other dynamic analyzers like MIRI (an interpreter that detects undefined behavior in Rust by executing code in a defined environment) complement these verification tools by checking one execution at a time but with a semantic model that catches unsafe violations; however, MIRI is more a testing aide than a formal verifier.

1.6.4 Summary

We have shown that the Rust verification ecosystem in incredibly rich and includes: foundational proofs of the language's safety (RustBelt, symbolic semantics), translation-based verifiers (Aeneas, Prusti, RefinedRust) that build on the guarantees of Rust's types, and model checkers (Kani) that leverage Rust's own compiler internals for efficient exhaustive exploration. Each of these harnesses Rust's key feature—its rich type system—as an

asset to simplify or optimize verification. This contrasts with verification for languages like C/C++, where the lack of a guaranteed memory safety baseline means much effort is spent just to mitigate wild pointers or data races. In Rust's case, the baseline guarantees free the verifier to focus on higher-level properties or the remaining unsafe corners of the language. The research surveyed here shows a trend of *integrating verification with the language's design*: from using the borrow checker in a proof assistant, to encoding Rust lifetimes as logical predicates, to running model checkers on MIR. This tight integration yields verifiers that are both sound (no false negatives on safety) and relatively precise, making formal assurance of Rust software increasingly attainable.

1.7 Integration with IDEs and Language Servers

The ultimate goal for refactoring and verification tools is seamless integration into developers' workflows. For Rust, integration has meant working closely with the compiler or the emerging Language Server Protocol (LSP). However, `rustc` was not originally desinged as a reusable library for IDE features, which led to the creation of separate projects to support code intelligence and analysis. The first generation Rust Language Server (RLS) attempted to use the compiler internals to power IDE features, but it struggled with performance and completeness as Rust evolved. Today, the community relies on **rust-analyzer**, a from-scratch implementation of a Rust language server that emphasizes speed and modularity **?**. Rust-analyzer constructs its own parse tree and does its own type inference, enabling on-the-fly responses to editor queries without running the full compiler on every change. This architecture is conducive to implementing refactorings as lightweight code transformations called "assists." In fact, rust-analyzer already supports basic refactorings like *Rename* and *Extract Variable*, and *Extract Method* as editor assists. These operate on rust-analyzer's internal syntax tree and use its knowledge of ownership and types to ensure the edits are valid. For example, the Extract Function assist initially had limitations with Rust's generics and lifetimes - extracting code from a generic function would produce errors because the assist didn't propagate required type parameters or lifetime parameters to the new function. However, even with substantial development by the community, certain assists still fall significantly short of the capabilities of tools developed specifically for purpose, such as REM.

Integration into the popular IntelliJ IDEA (via the Rust plugin) has also seen some development. The REM tool was originally built atop the IntelliJ plugin, indicating that industrial IDEs can serve as a proving ground for research prototypes **?**. In principle, once validated, these prototypes can be upstreamed, bringing advanced refactorings to everyday developers. A challenge here is maintaining the tools: Rust's syntax and compiler behavior evolve, so refactoring algorithms must be kept in sync. One way this is being

addressed is by using Rust's stable analysis APIs where possible. For instance, tools like Rust-Lancet chose to use the community-maintained `syn` crate for parsing so that they aren't tied to intermal compiler data structures that change nightly **?**. Similarly, projects to support large-scale refactoring use cases (such as Facebook's code migration tools or the Rust-to-Rust transformations needed in the Rust 2018 edition uplift) relied on either compiler suggestions or external scripting. The Rust compiler team has introduced machine-applicable suggestions in compiler error messages, which tools like **rustfix** can apply automatically. This essentially allows the compiler to assist in simple refactors (e.g. renaming a deprecated syntax, adding a missing lifetime specifier) by emitting a structured suggestion. Whist not as general as a true refactoring engine, this has helped with batch changes across large codebases.

1.7.1 Compiler Internals for Ownership and Borrowing

It's worth noting how Rust's compiler architecture influences refactoring and verification tools. Rustc itself goes through a series of analyses: it parses into an AST (with desugaring), performs name resolution and type checking (including lifetime inference), and then produces MIR where borrow checking and other analyses occur. Because of this, tools integrated at various stages have different views of the program. For example, a refactoring tool working on an AST (like rust-analyzer's assists or Rust-Lancet) must re-run portions of the compiler (or mimic them) to ensure that after transformation the code still passes borrow check. Meanwhile, a verification tool like Kani that operates on MIR has the advantage of piggybacking on the actual borrow-checker's results - it can trust that MIR is already free of borrow rule violations and focus on exploring execution paths **?**. As research progresses, we may see more refactoring tools move to operate on MIR or use the borrow checker in the loop, because MIR provides a simpler, desugared view of the code with explicit borrow regions (e.g. the Polonius project, a next-generation borrow checker, could potentially expose an API for tools to query borrow relationships). For now, practical refactoring uses higher-level representations (AST plus perhaps some compiler query for type info) because it's easier to map edits back to source code.

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 ispired 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 sover 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 refacotring 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 ands a tool-driven migration. The incdeasing adoption of language servers means refactoring capablilities 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 Conclusion

The landscape of Rust refactoring and verification has rapidly evolved, marrying the Rust compiler's strengths with novel algorithms from the research community. Automated refactoring tools for Rust have progressed from simple renaming utilities to sophisticated systems that integrate program repair and static analysis (REM and Rust-Lancet) to handle ownership and lifetime challenges **?**, **?**. Verification tools leverage Rust's unique guarantees to simplify proofs – whether by translating to pure functional models (Aeneas) **?**, by encding lifetime reasoning into logical frameworks (RustBelt, RefinedRust) **?**, **?**, or by exploring executions with model checkers (Kani) that understand Rust-specific features **?**. A unifying theme is that Rust's ownership semantics, once seen purely as a hurdle, have become an enabler for both refactoring and verification: they localize the reasoning about memory, allowing tools to either confidently transform code or assert properties with the guarantee that certain classes of bugs are already ruled out.

In conclusion, Rust's combination of strong static checks and low-level performance has fostered a unique blend of research. The advances in refactoring mean that developers can expect increasing assistance in restructuring Rust code without fear of breaking it. Meanwhile, the maturation of formal methods for Rust suggests that full verification of critical Rust code is becoming practical, supported by tools that integrate with Rust's development ecosystem. By structuring the literature along clear axes (refactoring vs. verification, formal foundations vs. practical tools, compiler phases involved, safe vs. unsafe scope), future reviews can further improve clarity and coverage. The works cited here collectively show that Rust's initially steep learning curve for both humans and tools can be mitigated by clever automation and rigorous semantics – turning Rust's famed safety guarantees from a burden into a powerful ally for program analysis and transformation.

Chapter 2

Proposed Approach

2.1 High Level Strategy

2.2 Detailed Methodology

2.3 Verification Strategy

Chapter 3

Implementation

3.1 System Architecture

3.2 Data Structures

3.3 Integration with External Tools

3.4 User Workflow and Experience

Chapter 4

Evaluation and Experimental Results

4.1 Experimental Setup

4.2 Evaluation Criteria

4.3 Experimental Scenarios

4.4 Results and Analysis

Chapter 5

Conclusions and Future Work

5.1 Summary of Contributions

5.2 Limitations

5.3 Potential Improvements

5.4 Broader Impact

Appendices

6.1 Appendix 1: Passing by Value and Reference

This appendix provides three short, self-containted examples of "naïve" function hoisting in Rust. In each example, we will define a function that is getting refactored, and we will demonstrate the case where the arguments to the function are passed by value and the case where they are passed by reference. To achieve this, we are imagining an automated refactoring tool that initially takes an inline block of code and moves it into a new function. The compilation errors (if any) guide a subsequent ownership analysis, which deduces how each variable should be passed: by value (`T`), by shared reference (`&T`), or by mutable reference (`&mut T`).

6.1.1 Example 1: Passing by Value

Original Code (inline)

```
fn main() {
    let x = 42;
    let y = x * 2;
    println!("y is {}", y);
}
```

Naïve Hoisted Code (before fixes)

Suppose an automated refactoring tool decides to "hoist" the multiplication logic into a new function:

```
fn hoisted_block(x: i32) {
    let y = x * 2;
    println!("y is {}", y);
}

fn main() {
    let x = 42;
    hoisted_block(x);
}
```

In this example, no errors occur because after calling `hoisted_block(x)`, we do not need `x` again in `main`, so moving (passing ownership) is safe. The automated repair analysis detects that `x` is no longer needed in `main` and decides it can be passed by value.

6.1.2 Example 2: Passing by Shared Reference

Original Code (inline)

```
fn main() {
    let text = String::from("Hello world!");
    let length = text.len();

    // Some inline block that only reads from `text`:
    println!("First word is: {}", get_first_word(&text));
    println!("Total length is: {}", length);
}

fn get_first_word(s: &str) -> &str {
    s.split_whitespace().next().unwrap_or("")
}
```

Naïve Hoisted Code (before fixes)

Let's say the refactoring tool decides to hoist the usage of `get_first_word` (or a bit more logic around it) into a new helper function:

```
fn main() {
    let text = String::from("Hello world!");
    let length = text.len();

    // Some inline block that only reads from `text`:
    println!("First word is: {}", get_first_word(&text));
    println!("Total length is: {}", length);
}

fn get_first_word(s: &str) -> &str {
    s.split_whitespace().next().unwrap_or("")
}
```

This version won't compile as expected if the function `print_first_word_block` takes `text` by value. Once `text` is moved into `print_first_word_block`, we can no longer use `text` afterward in `main`. The compiler will complain that `text` is moved, thus invalidating the line `println!("Total length is: ", length);` if we needed `text` for anything else.

Automated Repair Yields

```
fn print_first_word_block(text: &String) {
    // Only reads from `text`, so it only needs a shared reference.
    println!("First word is: {}", get_first_word(text));
}

fn main() {
```

```
    let text = String::from("Hello world!");
    let length = text.len();

    print_first_word_block(&text); // Pass a shared reference
    println!("Total length is: {}", length);
}
```

What's going on?

- **Read-only usage**: The function `print_first_word_block` just needs to read the string (in order to print the first word).
- **Shared reference**: Because the string must remain valid afterward in `main`, the ownership analysis decides that passing by shared reference (`&String`) is correct.

6.1.3 Example 3: Passing by Mutable Reference

Original Code (inline)

```
fn main() {
    let mut values = vec![1, 2, 3];

    // Some inline block that mutates `values`:
    values.push(4);
    println!("Values are now: {:?}", values);
}
```

Naïve Hoisted Code (before fixes)

Hoisting the push operation into a new function could look like this:

```
fn push_block(values: Vec<i32>) {
    let mut v = values;
    v.push(4);
    println!("Values are now: {:?}", v);
}

fn main() {
    let mut values = vec![1, 2, 3];
    push_block(values);
    println!("Final values: {:?}", values); // error: `values` was moved
}
```

Here, we have a compilation error similar to Example 2: once `values` is passed by value, ownership is transferred, and we cannot use `values` again in `main`. However, in this case, we truly do need to mutate `values`, and we want `main` to see the updated vector.

Automated Repair Yields

```
fn push_block(values: &mut Vec<i32>) {
    values.push(4);
    println!("Values are now: {:?}", values);
}

```

```
6  fn main() {
7      let mut values = vec![1, 2, 3];
8      push_block(&mut values);
9      println!("Final values: {:?}", values); // Now values is still in scope and updated
10 }
```

What's going on?

- **Mutation required**: The code needs to modify the original vector.
- **Mutable reference**: By passing `&mut Vec<i32>`, the function can mutate `values` in place, and `main` retains ownership of `values` to use afterward.