Bringing Extract Method Refactoring to the Rust Programming Language: Experiments with REM

Matthew Britton[1], Supervisor: Alex Potanin[2]

Abstract
This report details my current progress on advancing extract method refactoring in the Rust programming language. This project builds on existing work completed by Ilya Sergey, Sewen Thy and others in Adventure of a Lifetime: Extract Method Refactoring for Rust. The first half of the project is focussed on bringing the Rusty Extraction Maestro (REM) tool to a usable state, with the second half focussed on researching evaluating and implementing new extraction methods and techniques.

[1] *School of Engineering, The Australian National University. — matt.britton@anu.edu.au*
[2] *School of Computing, The Australian National University. — alex.potanin@anu.edu.au*

Contents

1. Introduction

Context

This project starts an extension of work done in a previous project by Sewen Thy and Ilya Sergey, in which they theorised and then implemented the beginnings of an automated "Extract Method" refactoring tool for the Rust programming language. Extract Method refactorings are a well known and widely used refactoring approach, although it is also one of the most complicated to implement in practice as it relies on very low-level code changes [2]. Additionally, due to Rust's unique approach to memory safety of ownership and borrowing, and the lifetime system that enforces these rules, it is far more difficult to implement than in other languages. Ilya Sergey et al's contribution focused on the theoretical apects of the problem [1], whilst Sewen Thy's contribution focused on the implementation, [3], with a pure Rust algorithm and a heavy reliance on Intellij IDEA's type inference and static code analysis tools. Their approach was dubbed "Rusty Extraction Maestro" (REM). Figure 1 gives a high-level overview of the REM algorithm.

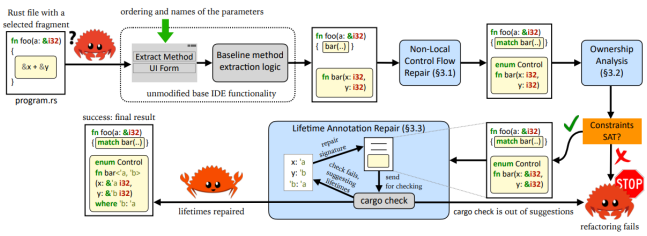


Figure 1. An overview of the REM Algorithm, from [1]

The REM algorithm is a three stage process, beginning after Intellij IDEA has performed an initial code extraction and type inference. If REM fails (i.e. an extraction isn't possible), then the code is returned to its original condition. The stages are:

1. **Controller:** handles any non local control flow. This includes return, break and continue. As rust is an expressions based language [4] these statements can be used in arbitrary combinations, adding substantial complexity. The algorithm is included appendix A1.
2. **Borrower:** infers any ownership annotations. In Rust, variables can be either immutable (imm) or mutable (mut). Additionally, they are either owned by the current scope, or borrowed from another scope.

```
let x = 5; //x is an imm variable
let x_ref = &x; //imm reference to x
let mut y = 5; //z is a mut variable
let ref_y = &mut z; //mut reference to z
//Functions can take references
fn foo(s: &String){println!("{}", s);}
```

The borrower is at its core, a dataflow analysis algorithm that infers the ownership of each variable, and adds the correct information to the function body and scope. The algorithms are included in appendix A2.

3. **Repairer:** focuses on correcting lifetime annotations in Rust functions, particularly those involving references. When a function has borrowed values, lifetimes need to be explicitly managed to ensure that the references remain valid. The repairer first explicitly annotates all lifetimes, using the rust compiler (rustc) to check for errors: `'a` represents the lifetime of `x` [1].

```
fn bar <'a 'b> (x: &'a i32, y: &'b i32)
    -> &'b i32 where 'a: 'b {...}
```

All functions are converted into an intermediate representation (IR), that are the machine equivalent of the above snippet. It then applies the lifetime elision rules [5] to remove any lifetimes the compiler can infer to bring the code back to a human readable state. The algorithm is included in appendix A3

One of the most difficult aspects of any refactoring / optimisation tool is ensuring the correctness of the output. A refactor is "correct" if the orginal behaviour of the program is preserved [6]. Unfortunately, for the REM algorithm, formal verification of the correctness is impossible as it would a model of the formal semantics of (safe) Rust, which doesn't exist [2]. Instead, the toolchain relies on a combination of extensive testing, the rust compiler, and a fail-early approach to ensure that the user is never presented with "incorrect" code.

Aims

This project aims to improve and extend on the REM tool by addressing key limitations and researching (and subsequently implementing) new approaches to refactoring.

1. **Fixing REM's Issues:** Remove the dependency on (*i*) `rustc` and (*ii*) Intellij IDEA to make REM a standalone tool. Additionally, reduce file I/O overhead (and other unnecessary overhead) by optimising the tools internal data handling.
2. **Platform-Agnostic Code Extraction:** Build a complete, end-to-endsolution for method extraction using *Rust Analyzer* (RA) for the code extraction, wrapped in a single, user-friendly CLI tool written entirely in Rust.
3. **VSCode Extension Proof of Concept:** Develop a VSCode extension to showcase how REM can be easily integrated into editors, with potential for extending to other environments.
4. **Exploring New Refactorings:** The big ticket item for this research project where the bulk of my time will be spent, investigating extract method refactoring for the far more advanced areas of *Generics* and *Asynchronous Code*, with the potential to expand into, *Macros, Closures, and Const Functions*, time permitting. *Generics* and *Asynchronous Code* present special challenges due to (a) the complex lifetimes and (b) the lack of information available at compile time. The starting point for this is *Generics*, as Adventure of A Lifetime [1] touches on the topic.
5. **Final Aim: Merging REM with RA:** An ultimate, ambitious goal for this project is to merge the REM toolchain with RA, Rust's official language server protocol (LSP) implementation [3]. This would require proving substantially more of the correctness of REM (although as previously noted, a full correctness proof is impossible), allowing its functionality to be available across any Rust-supported editor. If that is not feasible, then a well featured VSCode extension is a suitable alternative.

2. Theory Work Completed to Date

Literature Review Progress

A literature review is in the process of being written, and is around 60-75% complete. A draft will be completed by the Christmas break, and it will be finished by the end of the summer break (although it will still be subject to review and future revisions). It (broadly) covers the following topics, with topics marked with * potentially being added:

- An Overview of Extract Method Refactoring.
- Rust's Memory Management Model (Ownership and Borrowing), and how its' type system both helps and hinders refactoring tools.
- * Evolving Standards and Best Practices in the Rust Ecosystem.
- Existing Refactoring Tools and Their Limitations.
- Previous Work on the REM Tool by Ilya Sergey et al and Sewen Thy.
- Advanced Refactoring Techniques in Other Languages.
- Testing and Validation Strategies for Refactoring Tools
- * Comparative Analysis with Other Programming Languages.
- * Integration with Developer Tools.

Research into Advanced Refactoring Methods

Recent research on advanced refactoring methods, primarily focused on Java, highlights the importance of enhancing tools to handle the complexities of asynchronous programming and generics. Lin et al. developed ASYNCDROID [7], a tool that refactors improperly used Android async constructs, transforming them into more efficient forms like

[1] See Rust Book - Lifetimes for more information

[2] Formal correctness proofs for refactorings are very rare, even for languages with fully formalised semantics [1]

[3] https://rust-analyzer.github.io/

`IntentService`. This approach addresses memory leaks and lost results, by automating these complex refactorings.
Similarly, Zhang et al. proposed ReFuture [8], a tool for automating refactoring in Java projects using CompletableFuture, improving asynchronous code efficiency through static analysis techniques like visitor pattern and alias analysis. This tool demonstrated high effectiveness in large codebases, emphasizing the need for advanced analysis to manage dependencies and optimize task scheduling.
Additionally, Marticorena et al. explored the impact of generics on method extraction refactorings in Java, noting the necessity for refactoring tools to evolve as programming languages incorporate more sophisticated type systems [9].
These studies, while centered on Java, underscore the broader need for developing refactoring methods for modern language features, particularly in environments with strict memory and type constraints.

3. Coding Progress

Modifications to REM

The most important modification to the original toolchain has been decoupling it from the rust compiler. Previously, REM was heavily tied to rustc, the rust compiler. It relied compiler specific representations of files and their syntax trees. Because of this dependency, REM required a very specific buildchain, and could not be used outside of this build context. Additionally, when I took over the project, changes to external libraries meant that the toolchain was completely nonfunctional. Some work is still required to bring the toolchain off of nightly rust onto stable rust.
Additionally, REM's data handling has been improved. The original approach involved the three main components (controller, borrower, and repairer) passing data via intermediary files. By restructuring data flow among the now five modules, a 5-10% speedup was achieved.

REM-Extract: Performing inital function extraction

The REM-Extract tool is a drop in replacement for the initial code extraction that REM used to rely on IntelliJ IDEA to perform. It is a pure Rust solution, relying on RA to perform the code extraction, in preparation for the REM toolchain to be integrated into RA at a later date. At the highest level of abstraction, the extractor performs two tasks:

1. Extracting the function from the source code (i.e. directly copying the users selected code into a new function).

2. Inferring the types of the variables in the function, and adding them to the function signature, as well as annotating the caller with the correct variables. Type inference is the "compile-time process of reconstructing missing type information in a program" [10]. Because of Rust's very strict type system, this is a far more complex task than in a dynamically typed languages like Python or Ruby.

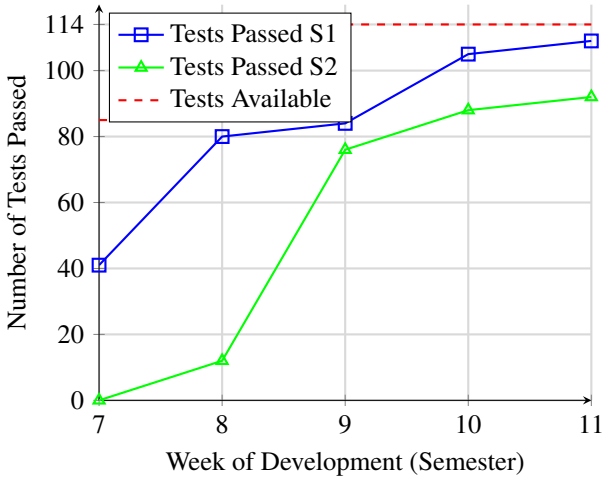
Extracting the function becomes an exponentially more complex task as the more complex language features of Rust are used. In the below example, the extraction of *extracted_fn* is relatively trivial, but demonstrates the basic principles of the extraction tool.

```rust
// Pre-Extraction
fn foo() -> u32 {
    let n = 2; let m = 1;
    n + m
}
// After Extraction
fn new_foo() -> u32 {
    let n = 2;
    extracted_fn(n)
}
fn extracted_fn(n: u32) -> u32 {
    let m = 1;
    n + m
}
```

Appenix A4 contains a more complex example of a function that the toolchain is working towards supporting.

The graph below depicts the development progress of the extraction tool over the last 5 weeks. The tool will continue to be developed as the capabilities of the toolchain are expanded. Most of the test failures at this point are for the reasons outlined in section 5.1.

REM-Extract Development Progress



REM-CLI: A comprehensive command line interface for REM

The REM-CLI provides a unified interface for developers to interact with the 5 separate modules of REM. It streamlines the process of performing method extraction by offering an intuitive command-line interface that handles everything from code analysis to extraction. By consolidating the toolchain into a single, easy-to-use CLI, REM removes the need for users to manually manage multiple stages of the extraction process. It is also needed to itegrate the functionality into a code editor.

REM-VSCode: A Visual Studio Code extension for REM

The REM-VSCode extension is a proof of concept that demonstrates how the REM tool can be easily integrated into any code editor. It provides a user-friendly interface within the far more popular Visual Studio Code editor. Because the entire backend of REM is now written in Rust, the extension is able to be independent of a specific platform, and thanks the the previous work on decoupling the tool from rustc, it no longer requires a very specific environment / buildchain to run. It is available on the VSCode Marketplace, but isn't finalised yet.

4. Timeline for Future Work

The following is a rough timeline for the remaining work on the project. It is subject to change depending on how much work Solar Car takes up over the summer period, and the features we decide to implement beyond *generics* and *async*.

Month	Tasks
October	Continue with detailed research into refactoring. Write the mid-semester report.
November	Finish draft of the lit review. Continue in-depth research on generics and async for REM.
December	Work with Alex to develop algorithm for Generics. Begin initial coding/experimentation on REM updates. Break over Christmas during university shutdown.
January	Revise lit review. Develop algorithm for Async. Continue implementing key features. Write tests for initial changes, and verify.
February	Finalize lit review. Scrape github / open-source projects for examples similar to [3]. Meet with Ilya to verify algorithms.
March	Work through bulk of coding before university work starts to ramp up. Integrate features and gather data. At this stage, the projects implementation should be mostly finished. Start on final report writing.
April	Complete core implementation and testing. Write the bulk of the report. Decide on VSCode implementation or RA pull request, and complete.
May	Finalize coding and testing. Complete a bulk data analysis to verify the work. Complete and proofread the final report. Submit final report to Ilya for feedback.
June	Create the seminar presentation. Submit final report and present the seminar.

5. Issues Encountered So Far

Non-Deterministic Nature of the Extraction Algorithm

The extraction algorithm used by RA is non-deterministic, meaning that for the same input code, the output of the extraction process can vary. This results in different versions of semantically equivalent but syntactically distinct code. In other words, while the extracted code maintains the same functionality, its structure and readability can change between runs. This poses a significant challenge for testing the tool, as each run may yield different outputs for the same input, making consistent verification difficult.

For example in Figure 2, a test of function extraction, RA may choose to pass variables by reference to the extracted function, dereferencing them inside the function body. In another instance, it might pass the same variables by ownership, altering the code's appearance but not its behavior. While both versions of the code will likely compile to identical machine code, the readability and clarity of the resulting code can vary.

```
Test 99 | PASSED | FAILED | PASSED: return_to_parent in 1.82s
============================================================
Differences or compilation errors found for test 'return_to_parent':
Differences found between expected and output:
  fn foo() -> i64 {
      let n = 1;
-     let k = match fun_name(n) {
+     let k = match fun_name(&n) {
          Ok(value) => value,
          Err(value) => return value,
      };
      (n + k) as i64
  }

- fn fun_name(n: i32) -> Result<i32, i64> {
-     let m = n + 1;
+ fn fun_name(n: &i32) -> Result<i32, i64> {
+     let m = *n + 1;
      return Err(1);
      let k = 2;
      Ok(k)
  }

  fn main() {

  }

============================================================
```

Figure 2. An example of the non-deterministic nature of the extraction algorithm. Red is expected output, green is the returned code .

Performance Bottlenecks in Large Codebases

The extraction tool's performance is a notable limitation in its current form. Since it is designed to be modular and not tied to any specific codebase, it must build a representation of the entire project to perform code extraction. While this approach works well for small, isolated test cases, it becomes inefficient when applied to larger, project-scale codebases. In such cases, the tool faces significant performance bottlenecks. To address this, the tool will need to either (a) operate on a targeted subset of the codebase, focusing only on the relevant files and functions, or (b) efficiently cache and reuse results from RA's existing analysis to avoid redundant computations. Both approaches present technical challenges that require careful consideration.

Acknowledgments

I would like to thank my supervisor, Alex Potanin, for his guidance and support. Additionally, I would like to thank Sasha Pak for listening in on all of our meetings and providing valuable feedback. Finally, I would like to thank Sewen Thy, the original developer of REM, for his help in understanding the project and his assistance in getting me started.

Source Code

The source code for this project can be found at the following repositories:

- REM-cli
- REM-extract — (Crates.io)
- REM-vscode
- REM-utils (Crates.io)
- REM-controller — (Crates.io)
- REM-borrower — (Crates.io)
- REM-repairer — (Crates.io)
- REM-constraint — (Crates.io)

References

[1] Sewen Thy, Andreea Costea, Kiran Gopinathan, and Ilya Sergey. Adventure of a lifetime: Extract method refactoring for rust. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

[2] Eman Alomar, Mohamed Wiem Mkaouer, and Ali Ouni. Behind the intent of extract method refactoring a systematic literature review. *IEEE Transactions on Software Engineering*, PP, 12 2023.

[3] Sewen Thy. Borrowing without sorrowing: Implementing extract method refactoring for rust. 2023.

[4] Aaron Matsakis, Nicholas Turon. The rust rfc book: Statements and expressions. 2020.

[5] Aaron Matsakis, Nicholas Turon. The rust rfc book: Lifetime elision. 2020.

[6] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. *SIGPLAN Not.*, 38(5):220–231, May 2003.

[7] Yu Lin, Semih Okur, and Danny Dig. Study and refactoring of android asynchronous programming (t). pages 224–235, 2015.

[8] Yang Zhang, Zhaoyang Xie, Yanxia Yue, and Lin Qi. Automatic refactoring approach for asynchronous mechanisms with completablefuture. *Applied Sciences*, 14(19), 2024.

[9] Raul Marticorena, Carlos López, Yania Crespo, and F. Javier Pérez. Refactoring generics in java: A case study on extract method. pages 212–221, 2010.

[10] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.

Appendices

A1. Non Local Controlflow Repair [1]

Algorithm 1: FixNonLocalControl

Input : an extracted function EF, an introduced function call expression E (*i.e.*, $EF(\ldots)$) in the caller
Output: a list of patches PS to apply to the refactored file

$PS \leftarrow []$
$R \leftarrow$ collect `return` statements in EF
$B, C \leftarrow$ collect top-level `break` and `continue` statements in EF
if $R \cup B \cup C \neq \emptyset$ **then**
  $RTY \leftarrow$ BuildReturnType(R, B, C)
  $PS \leftarrow$ UpdateReturnType$(EF, RTY) :: PS$
  for $l_r \in R$ **do** $PS \leftarrow (l_r,$ `return` $e \rightsquigarrow$ `return Ret(`e`)`$) :: PS$
  for $l_b \in B$ **do** $PS \leftarrow (l_b,$ `break` $\rightsquigarrow$ `return Break`$) :: PS$
  for $l_c \in C$ **do** $PS \leftarrow (l_c,$ `continue` $\rightsquigarrow$ `return Continue`$) :: PS$
  $l_E \leftarrow$ find location of the final expression of EF
  $PS \leftarrow (l_E, E \rightsquigarrow$ `Ok(`E`)`$) :: PS$
  $\overline{CS} \leftarrow$ BuildCasesForReturnType(RTY)
  $l_{\text{caller}} \leftarrow$ location of E
  $PS \leftarrow (l_{\text{caller}}, E \rightsquigarrow$ `match` E `with` $\overline{CS}) :: PS$
return PS

A2. Ownership and Borrowing Repair [1]

Algorithm 2: FixOwnershipAndBorrowing

Input : the extracted function *EF*, the expression *E* of the call to *EF*, original function *F*
Output: a set of patches *PS*

Aliases ← alias analysis on *F* /* maps variables to their aliases */
Mut ← CollectMutabilityConstraints(*EF*, *Aliases*)
Own ← CollectOwnershipConstraints(*EF*, *Aliases*, *F*)
PS ← [ ]
for *param* ∈ *EF.params* **do** /* derive patches for the signature of EF */
  v, *τ*, *l* ← *param.var*, *param.type*, *param.loc*
  if *UNSAT*(*Mut* ∪ *Own*, *v*) **then** raise RefactorError
  if *LUB*(*Mut* ∪ *Own*, *v*) = ⟨mut, ref⟩ **then** *PS* ← (*l*, *v* : *τ* ⇝ *v* : &**mut** *τ*) :: *PS*
  if *LUB*(*Mut* ∪ *Own*, *v*) = ⟨imm, ref⟩ **then** *PS* ← (*l*, *v* : *τ* ⇝ *v* : & *τ*) :: *PS*
for *param* ∈ *EF.params* **do** /* derive the patches for the body of EF */
  if *LUB*(*Mut* ∪ *Own*, *param.var*) = ⟨_, ref⟩ **then**
    Exps ← collect from *EF.body* all the occurrences of *param.var*
    for *e* ∈ *Exps* **do** *PS* ← (*e.loc*, *e* ⇝ (∗ *e*)) :: *PS*
for *arg* ∈ *E.args* **do** /* derive patches for the call to EF */
  v, *e*, *l* ← *arg.var*, *arg.exp*, *arg.loc*
  if *LUB*(*Mut* ∪ *Own*, *v*) = ⟨mut, ref⟩ **then** *PS* ← (*l*, *e* ⇝ &**mut** *e*) :: *PS*
  if *LUB*(*Mut* ∪ *Own*, *v*) = ⟨imm, ref⟩ **then** *PS* ← (*l*, *e* ⇝ &*e*) :: *PS*

Algorithm 3: CollectMutabilityConstraints

Input : extracted function *EF*, an alias map *Aliases*
Output: a set *Mut* of mutability constraints

MV ← collect all the variables in *EF* that are part of an *lvalue* expression
MV ← add to *MV* all the variables in the body of *EF* that are function call arguments with mutable requirements
MV ← add to *MV* all the variables in *EF* that are mutably borrowed
Mut ← {imm <: *p.var* | *p* ∈ *EF.params* ∧ ∀*v*′ ∈ *Aliases*(*p.var*) : *v*′ ∉ *MV*} ∪
  {mut <: *p.var* | *p* ∈ *EF.params* ∧ ∃*v*′ ∈ *Aliases*(*p.var*) : *v*′ ∈ *MV*}

Algorithm 4: CollectOwnershipConstraints

Input : extracted function *EF*, an alias map *Aliases*, original caller function *F*
Output: a set *Ownership* of ownership constraints

FV ← free variables in *F* in the code snippet after the call to *EF*
PBV ← collect all vars in *EF.params* declared as pass-by-value
Borrows ← *PBV* ∩ {*p.var* | *p* ∈ *EF.params* ∧ ∃*v*′ ∈ *Aliases*(*p.var*) : *v*′ ∈ *FV*}
Own ← collect all the vars in *EF* which are moved into or out of
Ownership ← {*v* <: ref | *v* ∈ *Borrows*} ∪ {own <: *v* | *v* ∈ *Own*}

A3. Lifetime Repair [1]

Algorithm 5: FixLifetimes

Input : a cargo manifest file *CARGO_MANIFEST* for the whole project, extracted function *EF*
Output: patched extracted function *EF′*

1 *EF′* ← clone *EF*
2 *EF′* ← update *EF′* by annotating each borrow in *EF′.params* and *EF′.ret* with a fresh lifetime where none exists
3 *EF′* ← update *EF′* by adding the freshly introduced lifetimes to the list of lifetime parameters in *EF′.sig*
4 **Loop**
5 *err* ← (cargo check *CARGO_MANIFEST*).errors
6 **if** *err* = ∅ **then** break /* refactoring is completed */
7 *suggestions* ← collect lifetime bounds suggestions from *err*
8 **if** *suggestions* = ∅ **then** raise RefactorError /* refactoring failed */
9 *EF′* ← apply *suggestions* to *EF′*
// readability optimisations:
10 *EF′* ← collapse the cycles in the `where` clause of *EF′.sig*
11 *EF′* ← apply elision rules

A4. Example Extraction Goal

```rust
/// The function the user wants to refactor.
/// The sleep calls are simulating long, complex operations.
/// The example is contrived, but the refactoring is applicable to real-world code.
async fn compute_sum_async<T>(items: &[T]) -> T
where
    T: std::ops::Add<Output = T> + Copy + Send,
{
    // First Extraction Block
    let sum = {
        let sleep = task::sleep(std::time::Duration::from_millis(10)).await;
        items[0]
    };

    for &item in &items[1..] {
        // Second Extraction Block
        sum = {
            task::sleep(std::time::Duration::from_millis(10)).await;
            sum + item
        };
    }
    sum
}

/// The output of the refactoring.
/// Note how the generic type T is still appropriately constrained.
/// And that the async blocks have been extracted into separate functions, with
/// the appropriate constraints.
async fn compute_sum_async<T>(items: &[T]) -> T
where
    T: std::ops::Add<Output = T> + Copy + Send,
{
    let mut sum = initialize_sum(items).await;

    for &item in &items[1..] {
        sum = add_items_async(sum, item).await;
    }
    sum
}

async fn initialize_sum<T>(items: &[T]) -> T
where
    T: Copy,
{
    task::sleep(std::time::Duration::from_millis(10)).await;
    items[0]
}

async fn add_items_async<T>(a: T, b: T) -> T
where
    T: std::ops::Add<Output = T> + Copy,
{
    task::sleep(std::time::Duration::from_millis(10)).await;
    a + b
}
```