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.

Keywords
Rust, Extract Method Refactoring

[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, however, due to Rust's unique approach to memory safety of 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, [2], 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 [1] gives a high-level overview of the REM algorithm.

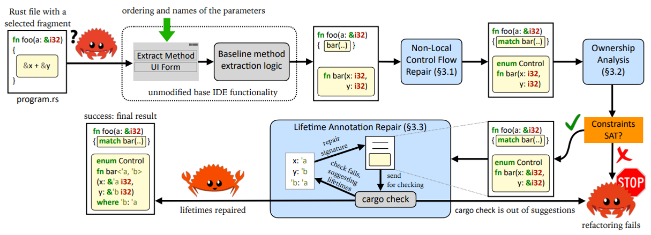

Figure 1. The REM Algorithm

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 [3] 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.

```rust
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
```

```rust
//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].

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

It then applies the lifetime elision rules [4] 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 [5]. 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 exists [1]. Instead, the toolchain relies on extensive testing, the rust compiler, and a fail-early approach to ensure that the user is never presented with “incorrect” code.

2. Work Completed to Date

Literature Review Progress

Research into Advanced Refactoring Methods

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 non-functional.

Another small modification has been changing the way REM handles data. Previously, the three main components of REM (controller, borrower and repairer) worked by each reading and writing to a file, and then passing the file to the next stage of the toolchain. By changing the way data is passed between the (now 5) modules of REM, a roughly 10-20% speedup has been achieved.

[1] See Rust Book - Lifetimes for more information

REM-Extract: Performing inital function extraction in Rust

REM-CLI: A comprehensive command line interface for REM

REM-VSCode: A Visual Studio Code extension for REM

4. Future Work

5. Issues Encountered So Far

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

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

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

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

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
1  $PS \leftarrow []$
2  $R \leftarrow$ collect return statements in $EF$
3  $B,C \leftarrow$ collect top-level break and continue statements in $EF$
4  if $R \cup B \cup C \neq \emptyset$ then
5      $RTY \leftarrow$ BuildReturnType$(R, B, C)$
6      $PS \leftarrow$ UpdateReturnType$(EF, RTY)$ :: $PS$
7      for $l_r \in R$ do $PS \leftarrow (l_r$, return $e \rightsquigarrow$ return Ret($e$)) :: $PS$
8      for $l_b \in B$ do $PS \leftarrow (l_b$, break $\rightsquigarrow$ return Break) :: $PS$
9      for $l_c \in C$ do $PS \leftarrow (l_c$, continue $\rightsquigarrow$ return Continue) :: $PS$
10     $l_E \leftarrow$ find location of the final expression of $EF$
11     $PS \leftarrow (l_E, E \rightsquigarrow$ Ok($E$)) :: $PS$
12     $\overline{CS} \leftarrow$ BuildCasesForReturnType$(RTY)$
13     $l_{\text{caller}} \leftarrow$ location of $E$
14     $PS \leftarrow (l_{\text{caller}}, E \rightsquigarrow$ match $E$ with $\overline{CS}$) :: $PS$
15 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 $PS$
1  $Aliases \leftarrow$ alias analysis on $F$                    /* maps variables to their aliases */
2  $Mut \leftarrow$ CollectMutabilityConstraints($EF$, $Aliases$)
3  $Own \leftarrow$ CollectOwnershipConstraints($EF$, $Aliases$, $F$)
4  $PS \leftarrow [\,]$
5  for $param \in EF.params$ do                    /* derive patches for the signature of EF */
6      $v, \tau, l \leftarrow param.var, param.type, param.loc$
7      if $UNSAT(Mut \cup Own, v)$ then raise RefactorError
8      if $LUB(Mut \cup Own, v) = \langle \text{mut}, \text{ref} \rangle$ then $PS \leftarrow (l, v : \tau \rightsquigarrow v : \&\texttt{mut}\ \tau) :: PS$
9      if $LUB(Mut \cup Own, v) = \langle \text{imm}, \text{ref} \rangle$ then $PS \leftarrow (l, v : \tau \rightsquigarrow v : \&\ \tau) :: PS$
10 for $param \in EF.params$ do                    /* derive the patches for the body of EF */
11     if $LUB(Mut \cup Own, param.var) = \langle \_, \text{ref} \rangle$ then
12         $Exps \leftarrow$ collect from $EF.body$ all the occurrences of $param.var$
13         for $e \in Exps$ do $PS \leftarrow (e.loc, e \rightsquigarrow (*\ e)) :: PS$
14 for $arg \in E.args$ do                         /* derive patches for the call to EF */
15     $v, e, l \leftarrow arg.var, arg.exp, arg.loc$
16     if $LUB(Mut \cup Own, v) = \langle \text{mut}, \text{ref} \rangle$ then $PS \leftarrow (l, e \rightsquigarrow \&\texttt{mut}\ e) :: PS$
17     if $LUB(Mut \cup Own, v) = \langle \text{imm}, \text{ref} \rangle$ then $PS \leftarrow (l, e \rightsquigarrow \&e) :: PS$
```

Algorithm 3: CollectMutabilityConstraints

```
Input  : extracted function $EF$, an alias map $Aliases$
Output : a set $Mut$ of mutability constraints
1  $MV \leftarrow$ collect all the variables in $EF$ that are part of an $lvalue$ expression
2  $MV \leftarrow$ add to $MV$ all the variables in the body of $EF$ that are function call arguments with mutable requirements
3  $MV \leftarrow$ add to $MV$ all the variables in $EF$ that are mutably borrowed
4  $Mut \leftarrow \{\text{imm} <: p.var \mid p \in EF.params \wedge \forall v' \in Aliases(p.var) : v' \notin MV\} \cup$
5           $\{\text{mut} <: p.var \mid p \in EF.params \wedge \exists v' \in Aliases(p.var) : v' \in MV\}$
```

Algorithm 4: CollectOwnershipConstraints

```
Input  : extracted function $EF$, an alias map $Aliases$, original caller function $F$
Output : a set $Ownership$ of ownership constraints
1  $FV \leftarrow$ free variables in $F$ in the code snippet after the call to $EF$
2  $PBV \leftarrow$ collect all vars in $EF.params$ declared as pass-by-value
3  $Borrows \leftarrow PBV \cap \{p.var \mid p \in EF.params \wedge \exists v' \in Aliases(p.var) : v' \in FV\}$
4  $Own \leftarrow$ collect all the vars in $EF$ which are moved into or out of
5  $Ownership \leftarrow \{v <: \text{ref} \mid v \in Borrows\} \cup \{\text{own} <: v \mid v \in Own\}$
```

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