

YaleNUSCollege

Borrowing without Sorrowing

Implementing Extract Method Refactoring for Rust

Sewen Thy

**Capstone Final Report for BSc (Honours) in
Mathematical, Computational and Statistical Sciences**

Supervised by: Prof. Ilya Sergey

AY 2022/2023

Yale-NUS College Capstone Project

DECLARATION & CONSENT

1. I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.
2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property ([Yale-NUS HR 039](#)).

ACCESS LEVEL

3. I agree, in consultation with my supervisor(s), that the Thesis be given the access level specified below: [check one only]

Unrestricted access

Make the Thesis immediately available for worldwide access.

Access restricted to Yale-NUS College for a limited period


Make the Thesis immediately available for Yale-NUS College access only from _____ (mm/yyyy) to _____ (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):
_____.

After this period, the Thesis will be made available for worldwide access.

Other restrictions: (please specify if any part of your thesis should be restricted)

Sewen Thy, Elm College

Name & Residential College of Student



Signature of Student

01/04/2023
Date

Ilya Sergey 

Name & Signature of Supervisor

01/04/2023
Date

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Ilya Sergey, for his invaluable guidance and unwavering support throughout my thesis work. His endless encouragement, insightful feedback, and willingness to push me beyond my limits have been instrumental in shaping my research and helping me achieve my goals.

I am also indebted to the members of the VERSE lab, especially Andreea Costea and Kiran Gopinathan, for their invaluable assistance in organizing formalizations, algorithms, and guiding implementations. Without their help, this thesis would not have been possible.

I would like to extend a special thanks to Juwon, who has been a constant source of support and encouragement throughout this journey. Her patience, understanding, and unwavering faith in my abilities have been indispensable to my success.

Finally, I would like to thank my friends and suitemates for their endless encouragement and for sharing in both the joy and the stress of this experience. From late-night study sessions to wine nights, your support has been a crucial part of my Yale-NUS experience.

Thank you all for helping to make this journey such a meaningful and rewarding one.

YALE-NUS COLLEGE

Abstract

B.Sc (Hons)

Borrowing without Sorrowing Implementing Extract-Method Refactoring for Rust

by **Sewen THY**

Rust is a programming language frequently favored for system and concurrency applications due to its safety features and precise memory control but it lacks proper support for extract method refactoring which allows for better code reuse and maintainability.

This thesis outlines a non-trivial implementation of an extract method refactoring for Rust within the IntelliJ's plugin. I saw common patterns in developer's manual extract method refactoring from open-source projects and implemented my strategy to support extracting functions that: (1) contains non-local control flows, (2) requires safe and minimal borrowing, and (3) requires lifetime annotations. I designed the solutions using a combination of static syntactical treatment, constraint analysis, and a novel use of program repair for refactoring to determine the lifetime bounds. I then evaluate my implementation using real-world open-source projects and comparing them to the state-of-the-art implementations.

Keywords: Rust, refactoring, extract method, lifetime, IntelliJ.

Contents

Acknowledgements	ii
Abstract	iii
1 Introduction	1
1.1 Problem Statement	1
1.2 Contributions	1
2 Background	2
2.1 Rust	2
2.1.1 Rust’s Memory Model	2
2.1.2 Important Constructs in Rust	3
2.2 Clean Code	4
2.2.1 Extract Method Refactoring	4
2.3 Early Rust Refactoring	5
3 Motivations	7
3.1 Methodology	7
3.2 Real-world Examples	8
3.2.1 Zola	8
3.2.2 Rust	9
3.2.3 Gitoxide	11

3.3	Categories of Extraction Patterns	12
3.3.1	Ownership and Mutability	12
3.3.2	Non-elidable lifetimes	13
4	The Extract Method Algorithm	16
4.1	Non-local Control Flows	16
4.2	Least Permissive Borrowing	17
4.3	Lifetime Repairs	20
4.4	Implementation	22
4.4.1	Failure modes	23
5	Evaluation	26
5.1	Effectiveness	26
5.2	Efficiency	32
5.3	Discussions	34
5.3.1	Type Inferences	34
5.3.2	Cargo Check Trade Off	37
5.3.3	Technical Detail	37
6	Conclusion	39
	Bibliography	40
A	Example Code Refactoring	42
A.1	Zola	42
A.2	Rust	42
A.3	Gitoxide	43

Chapter 1

Introduction

1.1 Problem Statement

Rust is a programming language frequently favored for system and concurrency applications due to its safety features and precise memory control. However, conventional refactoring method is not as effective for extracting complex methods in Rust which hinders code reuse making it hard to achieve good maintainability.

1.2 Contributions

This thesis aims to:

- contribute a non-trivial implementation of an extract method refactoring for Rust within the IntelliJ's plugin.
- categorizes common patterns seen in manual extract method refactoring.
- support extracting functions with non-local control flows and maintaining the same semantics.
- support extracting functions that borrows values from its caller.
- support extracting functions that requires named lifetime annotations by contributing a novel implementation for refactoring using program repair.

Chapter 2

Background

2.1 Rust

2.1.1 Rust's Memory Model

To provide its strong safety guarantees, Rust enforces a unique memory model. While other languages like Python and Java relies on garbage collectors which has performance implications, C and C-like languages allows developer freedom over precise memory which is more error-prone, Rust has an ownership model which provides higher-level control and also cleans up memory without a garbage collector (Matsakis and Klock II, 2014, Klabnik and Nichols, 2019).

In Rust's **memory model**, values have its owner and references to the values are called "borrows". Value can be "moved" to another owner as well. There can only be one owner of a value at any one time and when the owner is not live, the value is dropped. This allows Rust to stop memory leaks. For borrowing values, Rust allows either multiple immutable borrows and no mutable borrows or only one mutable borrow. With this Rust ensures that there is only ever one-writer-multiple-reader scenarios which is a thread-safe guarantee (Klabnik and Nichols, 2019, Matsakis and Klock II, 2014).

The Rust compiler, `rustc`, contains the borrow checker which analyzes the lifetime of the borrows and making sure that each borrow is sound. For example, it is unsound to borrow a value and have that reference live longer than the owner, so the borrow checker would flag this as a compiler error (Klabnik and Nichols, 2019, Matsakis and Klock II, 2014).

When a value is passed to a function, it will either be moved, i.e. a change of ownership, or it is borrowed by the function (Klabnik and Nichols, 2019, Matsakis and Klock II, 2014). This affects semantics of the program if the value is later used after it is passed into the function. This is the important part to take into consideration when extracting Rust's method as the extracted method could violate Rust's memory management rules. For example, this can happen by having the extracted function taking ownership of a value that is needed in the original function later or by borrowing the same values as mutable and keeping them alive at the same time or even by borrowing multiple references that have different lifetime and not specifying its bounds making the borrow checker rejects the extracted method.

2.1.2 Important Constructs in Rust

Structs are used to structure data and can have either values or references as its field. If it has reference fields or a struct with a lifetime slot as a field, then it need to have a lifetime annotation for said reference or slot (Klabnik and Nichols, 2019).

Traits are a set of interfaces that a struct can implement. Trait methods can have lifetime annotations to ensure whatever implements it meets the borrow checker guarantee (Klabnik and Nichols, 2019).

Hence, references, structs, and trait have lifetime slots.

Lifetimes are non-lexical and is calculated by the borrow checker to ensure that each reference (or borrow) is legal i.e. it cannot have unsoundness or undefined behaviour by violating the terms of its borrow (Matsakis, 2016). Scope refers to the lifetime of a value, corresponding to the span of time before that value gets freed (or, put another way, before the destructor for the value runs)(Matsakis, 2016). This describes how long a value is valid. Lifetime refers to the lifetime of a reference, corresponding to the span of time in which that reference is used (Matsakis, 2016).

2.2 Clean Code

For any application, keeping clean code is paramount for achieving maintainability and an important part of clean code is good code reusability (Martin, 2008). Ensuring the same functionalities are factored out into modules or functions reduces code duplication. This allows future maintainer to only ever modify one module or function. Therefore, refactoring tools need to make such refactoring as easy as possible to encourage clean code practices like extract method.

2.2.1 Extract Method Refactoring

Extract method refactoring is characterized by the following features:

1. one or more block(s) of (duplicated) code is replaced with a function call.
2. a new function is created containing the block of (duplicated) code.

For example, consider the following changes:

```
1 fn foo() -> int {
```

```
2   let x = 1;
3 -  println!("x: {}d", x);
4 -  x
5 +  bar(x)
6 }
7 +fn bar(x: i32) -> int {
8 +  println!("x: {}d", x);
9 +  x
10 +}
```

It is considered an extract method refactoring as (1) lines 3-4 are replaced with line 5, a function call to `bar` and the new function `bar` is added on lines 7-10.

2.3 Early Rust Refactoring

Some early works in refactoring was done by [Sam, Cameron et al.](#) which implemented some related renaming and lifetime elisions ([Sam, Cameron et al., 2017](#)). Our tool also implements lifetime elisions as part of our readability optimization. They also mentioned how macro poses challenge which we did encounter here in our analysis of borrowing. Although as we shall conclude later, it would be mostly solved, at least in our use case once we can query the types of variables using `rustc`.

[Schäfer, Ekman et al.](#) (along with ([Schäfer, Verbaere et al., 2009](#), [Schäfer and de Moor, 2010](#))) has been very influential in informing works on refactoring by giving sound framework while being in Java. While not in Rust, these works proposed formalized methods to have semantics preserving structural changes for Java which provides a good basis for our future work in formalizing our patches.

[Schäfer, Ekman et al.](#) has also influenced a master thesis on automated refactoring for Rust by [Ringdal](#) although they did not attempt to do automated lifetime annotations and their evaluations seem to suggest they alter program semantics ([Ringdal, 2020](#)). Furthermore, their refactoring takes on average around 2 seconds (but can take up to 8 seconds).

Chapter 3

Motivations

To show the prominence of these extract method refactoring done manually in the real-world, we conducted a search through commit messages on open-source projects on [GitHub](#) and display the results below.

Note that it is quite hard to find these examples because most extract method refactorings are not contained within its own commit and is mostly packed with other refactors into one commit and only very detail oriented developers would have put that they did indeed extract a method in this refactoring commit.

Throughout this thesis the state of the art refactoring tools we use is IntelliJ Rust plugin (0.4.186.5143-223) and Visual Studio Code's Rust Analyzer plugin (v0.3.1451) with latest versions as of February 2023.

3.1 Methodology

First, we gathered the example projects and big open-source projects using Rust from [this list](#) of popular and the Rust language as well.

Then within each project, we run search through its `git` commit log using the following command to filter for ones related to extract method refactoring:

```
1 $ git log --grep="extract\|\_move\|refactor" -i --all \
```

```
2 --full-history
```

Once we got the shorter commit list, we manually verify whether each change in the code is actually an extract method refactoring using the characterization in [2.2.1](#).

3.2 Real-world Examples

From the list of examples, we will highlight a few commits in the open source projects and see the difference between how a developer manually extracts the function and what IntelliJ's Rust plugin outputs.

3.2.1 Zola

[Zola](#) is a lightweight web framework in Rust that is actively maintained. We found a [commit](#) that extracts a piece of code with non-local control flow (see more in [appendix A.1](#)). We simplified the pattern below where you can see how the code is simply returning a `String` if everything is okay and, otherwise, terminates the function `markdown_to_html` early on line 8:

```
1 pub fn markdown_to_html (...) -> Result<Rendered>
2 { let fixed_link = if ... { return ...; } else { ... }; }
```

Extraction by author

Within this commit, the author simply use a `Result<T, E>` crate which allows for differentiating the success and failure results and then propagate the control-flow back to the main `markdown_to_html` function:

```
1 pub fn fix_link(_: &str, _: &RenderContext) -> Result<String>
2 { let result = if ... {return Err(...)} else {...}; Ok(result) }
```

```
3 pub fn markdown_to_html (...) -> Result<Rendered> {
4     let fixed_link =
5     match fix_link(...) { Ok(r) => r, Err(...) => {return ...} };
6 }
```

Extraction by IntelliJ's Rust plugin

When we use the IntelliJ's Rust plugin, the extraction did not account for this non-local control flow. It simply copies the code block and using its own syntactic traversal and type inference to populate the signature for its input and output types:

```
1 pub fn fix_link(_: &RenderContext, mut _: &mut Option<Error>,
2     _: &Cow<str>) -> String {
3     let fixed_link =
4     if ... {return Event::Html(Borrowed(""));} else {...};
5     fixed_link }
6 pub fn markdown_to_html (...) -> Result<Rendered>
7 { let fixed_link = fix_link(...); }
```

This did not compile because the non-local return is treated as a return value from the extracted `fix_link` function rather than the caller `markdown_to_html` function.

3.2.2 Rust

The **Rust** language itself is actively maintained and contained many refactoring. We found a **commit** that extracts a piece of code with named lifetime parameter (see more in appendix [A.2](#)). Below is the signature of the main `try_extract_error_from_fulfill_cx` function whose body is being extracted.

```

1 fn _<'tcx>(mut _: Box<dyn TraitEngine<'tcx> + 'tcx>,
2   _: &InferCtxt<'_, 'tcx>, _: ty::Region<'tcx>,
3   _: Option<ty::Region<'tcx>>) -> Option<DiagnosticBuilder<'tcx>>

```

Extraction by author

The author of the commit recognizes the complex uses of named lifetimes, as well as the fact that there are 2 lifetimes for the `&InferCtxt<'_, 'tcx>` input so these lifetime cannot be elided safely. Therefore, in the extracted method `try_extract_error_from_region_constraints`, they copy over the same kind of named lifetimes (as well as some unrelated new refactoring) and then replaces the main `try_extract_error_from_fulfill_cx` function with a function call and appropriate parameters:

```

1 fn _<'tcx>(_: &InferCtxt<'_, 'tcx>, _: ty::Region<'tcx>,
2   _: Option<ty::Region<'tcx>>, _: &RegionConstraintData<'tcx>,
3   ...) -> Option<DiagnosticBuilder<'tcx>>

```

Extraction by IntelliJ's Rust plugin

Since there are extra unrelated refactoring, we are not assessing whether the refactoring done by the plugin will result in the exact same code but that it handles the necessary annotations of named lifetimes that cannot be elided. However, the plugin did not do any annotations.

```

1 fn _(_: &InferCtxt, _: Region, _: Option<Region>) ->
2   Option<DiagnosticBuilder<ErrorGuaranteed>>

```

Even though in the original function there are named lifetimes that could be simply copied over, this plugin's implementation did not account for it and so this extracted code did not compile. Furthermore, while the type

inference in IntelliJ Rust plugin knows that the struct `Region` requires a lifetime annotation, it did not attempt to do any annotation there either.

3.2.3 Gitoxide

Gitoxide is project that implements git in Rust to take advantage of Rust's safety. We found a **commit** that extracts a function that utilizes struct with lifetimes in them (this particular example, we will also examine for evaluation in section 5.1) (see more in appendix A.3).

Extraction by author

The author of the commit recognizes that the structs' lifetimes needed some bounding together. Therefore, in the extracted method `extract_include_path`, the author annotated the two lifetimes as the same:

```
1 fn _<'a>(_: &mut File<'a>, _: &mut Vec<values::Path<'a>>,  
2   id: SectionId)
```

Extraction by IntelliJ's Rust plugin

Again the plugin did not do the annotations, and it borrows the `id` variable rather than simply taking ownership of it when there is no further use of `id`.

```
1 fn _(_: &mut File, _: &mut Vec<Path>, id: &SectionId)
```

Furthermore, it did not qualify the `Path` struct properly and this also caused a compiler error in addition to the missing lifetime annotations.

3.3 Categories of Extraction Patterns

I categorized the common patterns of extract method refactoring below that I intends address with this thesis with toy-size examples to demonstrate each category. Since non-local control flow was demonstrated very nicely by [subsection 3.2.1](#), we will skip that toy example.

3.3.1 Ownership and Mutability

When a value is passed to a function, it will either be “moved”, i.e. a change of ownership, or a it is borrowed by the function. This affects semantics of the program if the value is later used after it is passed into the function.

```
1 pub fn original_foo() {
2   let mut x = String::new();
3   println!("x={}", x);
4   x.push('x');
5   println!("x={}", x);
6 }
```

In cases where the value is not modified say we are extracting line [3](#) (blue), only an immutable references is needed. Otherwise, say we are extracting line [4](#) (orange), we will need an mutable reference as we are modifying the value of x.

```
1 pub fn new_foo() {
2   let mut x = String::new();
3   extract_immutable(&x);
4   extract_mutable(&mut x);
5   println!("x={}", x);
6 }
```

```
7 fn extract_immutable(x: &String) {println!("x={}", *x);}
8 fn extract_mutable(x: &mut String) {(*x).push('x')};
```

In either case, since `x` is used in line 5, it needs to be alive there too so the value cannot be moved into the extracted function. Furthermore, we need to dereference `x` with `*` when we use it as it is passed in as a reference.

3.3.2 Non-elidable lifetimes

There are specific scenarios whereby simply borrowing the values is not enough to get the extraction to compile because while the function was inlined it has other constraints that are tied implicitly to it. One of such constraint is the lifetimes of the references. While the borrow checker can figure out the lifetime through its analysis when the extracted block is inlined, when the extracted block is in a separate function, those analysis can yield unsatisfying results unless we make the correct implicit constraints explicit i.e. by annotating the right lifetimes with the correct bounds to the extracted function signature. Below are some scenarios that requires those explicit lifetime annotations.

Different in/out lifetime: when extracting function that requires values to be borrowed by the function but those borrows might have different lifetimes, we need to make those lifetimes explicit.

```
1 const W: i32 = 5; // 'static
2 pub fn original_foo () { // scope a
3     let x = 1;
4     let x_ref = &x;
5     let mut z : &i32;
6     { // scope b
```

```

7     let y = 2; z = &y;
8     z = if *z < *x_ref { &y } else { &W };
9     println!("{}", *z);
10  }
11 }

```

Semantically, `z` is assigned either a reference of `y` or `W` but since it is only being used within scope `b` where both of these values are live, `z` can be assigned and used safely.

We want to extract the condition definition of `z` and we need to ensure semantically that the reference of `y` that we pass will live as long as the output reference. Hence, one valid extraction is `bar`.

```

1 ... z = bar(x_ref, z, &y); ...
2 fn bar<'a>(x_ref: &i32, z: &i32, y: &'a i32) -> &'a i32
3 { if *z < *x_ref { y } else { &W } }

```

This means that the reference `y` will live exactly as long as output reference returned by `bar`. Note that since there are more than one reference in the input to `bar`, the borrow checker concludes that it is unclear how long the output reference should live so you need to provide that context—however, when it was inlined, the borrow checker knows that it will only live until the end of scope `b`.

Lifetime bounds: there are cases where annotating the named lifetimes are not enough, the relationship between those named lifetimes must be specified with bounds. This example deals with lifetimes that requires strict bounds to be extracted correctly.

```

1 pub fn foo(){
2     let x = 1;
3     { let p : &mut i32 = &mut 0; *p = &x; println!("{}", **p); }

```

```
4 }
```

We see that `p` is a mutable reference so any assignment to `p` needs to live at least as long as `p` (otherwise, you have `p` borrowing a value that is no longer live—which is a safety issue known as a dangling pointer) so the bounds here for `&x` requires `'b: 'a` which in Rust lifetime bounds syntax means lifetime `'b` needs to live at least as long as lifetime `'a`. Hence, one valid extraction is `bar`.

```
1 ... bar(p, &x); ...  
2 fn bar<'a, 'b: 'a>(p: &mut &'a i32, x: &'b i32) { *p = &x; }
```

Chapter 4

The Extract Method Algorithm

4.1 Non-local Control Flows

We need to patch the program when the caller have control flows that is within the extracted function body (e.g. the caller has a `return`, or a loop control `continue` that is now within the extracted function body). To propagate the control flow back up to the caller from the extracted function, we need to patch the extracted function return type and patch the call site to perform the same branching using the return of the extracted function.

Algorithm 1 simply collects the facts about `return` statements, and any loop control i.e. `break`, and `continue` statements that control the loop outside of the extracted function. We then patch the return type based on the controls we need to propagate using an enumerator/variant type. Then we pattern match on the generated variants and generate the patch for the call site.

Algorithm 1: FIXNONLOCALCONTROL

```

1 Input: call expression  $E_{\text{caller}}$  (in function  $F'$ ), extracted function  $EF$ , original
   function  $F$ 
2 Output: a list of patches  $PS$ 
   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  $PS \leftarrow (l_r, \text{return } e \rightsquigarrow \text{return Ret}(e))$  ::  $PS$ ;
   8:   for  $l_b \in B$  do  $PS \leftarrow (l_b, \text{break} \rightsquigarrow \text{return Break})$  ::  $PS$ ;
   9:   for  $l_c \in C$  do  $PS \leftarrow (l_c, \text{continue} \rightsquigarrow \text{return Continue})$  ::  $PS$ ;
  10:   $l_E \leftarrow$  find location of final expression of  $EF$ 
  11:   $PS \leftarrow (l_E, E \rightsquigarrow \text{Ok}(E))$  ::  $PS$ 
  12:   $\overline{CS} \leftarrow$  BUILDCASESFORRETURNTYPE( $RTY$ )
  13:   $l_{\text{caller}} \leftarrow$  location of  $E_{\text{caller}}$ 
  14:   $PS \leftarrow (l_{\text{caller}}, E_{\text{caller}} \rightsquigarrow \text{match } E_{\text{caller}} \text{ with } \overline{CS})$  ::  $PS$ 
  15: end if
  16: return  $PS$ 

```

4.2 Least Permissive Borrowing

For the extracted function to be correct and have the least permissions, the caller must give enough permissions (and have that permission to give) so the extracted function can perform the necessary operations, and the caller must still be able to perform its operations (e.g. the extracted function shouldn't own the value that the caller still need).

We define permission as a pair of mutability and ownership, $\langle m, o \rangle$, and a *less-than* relation in [Figure 4.1a](#). With this pair we can define a partial-order between the permissions in [Figure 4.1b](#) which is intuitive regarding ownership and mutability i.e. ownership is more permissive than borrows and mutability is more permissive than immutability. This can be mapped to equivalent Rust types in [Figure 4.1c](#).

Within all the constraints we find for a parameter, we can determine its weakest possible permission by:

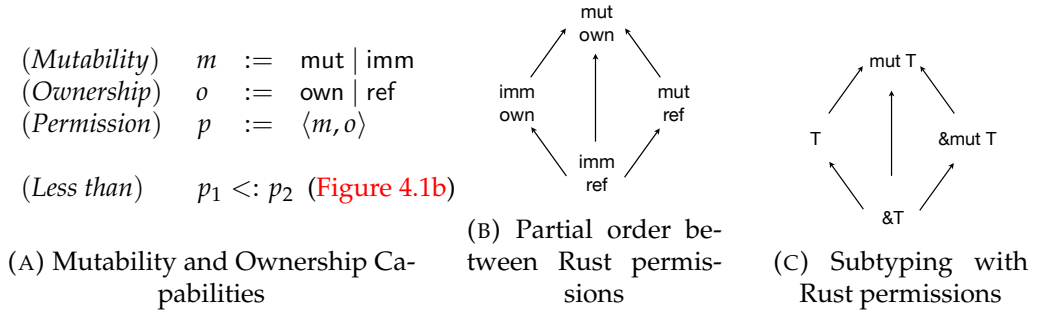


FIGURE 4.1: A combination of mutability and ownership permission maps to a Rust type constructor

$$\text{LUB}(\mathcal{C}, v) \stackrel{\text{def}}{=} \text{LUB}(\{\langle m, o \rangle \mid \langle m, o \rangle <: v \in \mathcal{C}\})$$

Least upper bound (LUB) checks for the least permissible permission in the set of permissions using the partial-order we define.

Algorithm 2 shows how we collect the constraints that determine how much permission to give the extracted function for each parameter and then generate patches for the program. We first collect the aliasing constraints so we can propagate any aliased permissions from algorithm 3 and algorithm 4.

Then we check that the permission combinations from our constraint actually gives a valid solution that we can derive patches for the extraction function signature. An invalid solution could be that the extracted function mutates the value x so requiring $\langle \text{imm}, _ \rangle <: x$ but the caller only have $\langle \text{imm}, _ \rangle$ for x . To ensure that the constraints we collected are satisfied, we check:

$$\text{SAT}(\mathcal{C}, v) \stackrel{\text{def}}{=} \perp \text{ iff } \exists v <: \langle m_1, o_1 \rangle \in \mathcal{C}, \langle m_2, o_2 \rangle <: v \in \mathcal{C} : \langle m_1, o_1 \rangle <: \langle m_2, o_2 \rangle \wedge \langle m_1, o_1 \rangle \neq \langle m_2, o_2 \rangle$$

SAT checks that within our constraints there is no such contradicting, invalid constraints—within our constraints \mathcal{C} , there are no two unique least permissive constraints.

Algorithm 2: FIXOWNERSHIPANDBORROWING

```

Input : call expression  $E$ , extracted function  $EF$ , original function  $F$ 
Output: a set of patches  $PS$ 
1  $Aliases \leftarrow$  alias analysis on  $F$  /* maps variables to their aliases */
2  $Mut \leftarrow$  MUTABILITYCONSTRAINTS( $EF, Aliases$ )
3  $Own \leftarrow$  OWNERSHIPCONSTRAINTS( $EF, Aliases, F$ )
4  $PS \leftarrow []$ 
5 for  $param \in EF.params$  do /* derive patches for the signature of  $EF$  */
6    $v, m, \tau, l \leftarrow param.var, param.mut, param.type, param.loc$ 
7   if  $LUB(Mut \cup Own, v) = \langle mut, ref \rangle$  then  $PS \leftarrow (l, m\ v : \tau \rightsquigarrow m\ v : \&mut$ 
8      $\tau) :: PS$ 
9   if  $LUB(Mut \cup Own, v) = \langle imm, ref \rangle$  then  $PS \leftarrow (l, m\ v : \tau \rightsquigarrow m\ v : \&\tau) :: PS$ 
10  if  $SAT(Mut \cup Own, v) \neq \perp$  then raise RefactorError
11 for  $param \in EF.params$  do /* derive the patches for the body of  $EF$  */
12   if  $param.var \in Borrows \wedge v \notin Own$  then
13      $Exps \leftarrow$  collect from  $EF.body$  all the occurrences of  $param.var$ 
14     for  $e \in Exps$  do  $PS \leftarrow (e.loc, e \rightsquigarrow (*e)) :: PS$ 
15 for  $arg \in E.args$  do /* derive patches for the call to  $EF$  */
16    $v, e, l \leftarrow arg.var, arg.exp, arg.loc$ 
17   if  $LUB(Mut \cup Own, v) = \langle mut, ref \rangle$  then  $PS \leftarrow (l, e \rightsquigarrow \&mut\ e) :: PS$ 
18   if  $LUB(Mut \cup Own, v) = \langle imm, ref \rangle$  then  $PS \leftarrow (l, e \rightsquigarrow \&e) :: PS$ 

```

Furthermore, in [algorithm 2](#), for the parameters that we make into references, that IntelliJ’s Rust plugin initially extracted as an owned value, we derive patches for the extracted function body to dereference them and for the call site of the caller to reference them.

Algorithm 3 checks if the value is used on the left-hand-side of an assignment or passed as mutable method call i.e. `&mut self` and [algorithm 4](#) simply checks for whether the variable is used after the call to the extracted method. If it’s not, we want to make the extracted function owns the value because that means the value is dropped at the same space it would have in the caller rather than after the extracted function returns—preserving more semantics.

Algorithm 3: MUTABILITYCONSTRAINTS**Input** : extracted function EF, an alias map *Aliases***Output**: a set *Mut* of mutability constraints

- 1 $MV \leftarrow$ collect all the variables in EF which are part of an *lvalue* expression
- 2 $MV \leftarrow$ add to *MV* all the variables in EF which are function call arguments with mutable requirements
- 3 $MV \leftarrow$ add to *MV* all the variables in EF which are mutably borrowed
- 4 $Mut \leftarrow \{\text{imm} <: p.\text{var} \mid p \in EF.\text{params} \wedge \forall v' \in \text{Aliases}(p.\text{var}) : v' \notin MV\} \cup$
- 5 $\quad \{\text{mut} <: p.\text{var} \mid p \in EF.\text{params} \wedge \exists v' \in \text{Aliases}(p.\text{var}) : v' \in MV\}$

Algorithm 4: OWNERSHIPCONSTRAINTS**Input** : extracted function EF, an alias map *Aliases*, original function F**Output**: a set *Ownership* of ownership constraints

- 1 $FV \leftarrow$ free variables of the subtree(s) of block *F.b*
- 2 $PBV \leftarrow$ collect all vars in *EF.params* declared as pass-by-value
- 3 $Borrows \leftarrow PBV \cap \{p.\text{var} \mid p \in EF.\text{params} \wedge \exists v' \in \text{Aliases}(p.\text{var}) : v' \in FV\}$
- 4 $Own \leftarrow$ collect all the vars in EF which are moved into or out of
- 5 $Ownership \leftarrow \{v <: \text{ref} \mid v \in Borrows\} \cup \{\text{own} <: v \mid v \in Own\}$

4.3 Lifetime Repairs

Since errors related to lifetimes are quite complex and there are no formal proof regarding the type soundness, the approach we took is similar to [Emre, Schoroeder et al.](#), in using the compiler as an oracle to help us correct our errors ([Matsakis and Klock II, 2014](#), [Emre, Schoroeder et al., 2021](#)).

The key insight is that some errors occur in the borrow checker which runs on rustc's MIR, those information can analyzed to repair the program. To ensure that we can reach borrow checker's analysis, we need to annotate all the lifetime slots. There are 2 possible strategies for this:

1. annotating all the lifetime slot in the function signatures with the same lifetime parameter: create the tightest possible bounds for their lifetimes;
2. or annotating all the lifetime slot in the function signatures with the different lifetime parameter: create the loosest possible bounds for

Algorithm 5: FIXLIFETIMES

```

Input : a Cargo manifest file CARGO_MANIFEST, 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).error
6   if err.length is 0 then break           /* refactoring is complete */
7   suggestions ← collect lifetime bounds suggestions from err
8   if suggestions.length is 0 then raise RefactorError /* refactoring failed
   */
9   EF' ← apply suggestions to EF'
  // readability optimizations:
10 EF' ← collapse the cycles in the where clause of EF'.sig
11 EF' ← apply elision rules

```

their lifetimes.

After both of these annotation strategies, the compiler can be used to repair the program until it can compile. However, what the borrow checker does is annotating all the unannotated lifetime slot with unique lifetime (except some special cases where `self` exists) first then perform its analysis (Matsakis and Klock II, 2014, Klabnik and Nichols, 2019). This is similar to the second strategy which is the loosest possible bounds. Hence, we decided to go with the second strategy.

The extracted function signature is checked for any argument and output that has a lifetime slot and annotate each of those argument with the a different lifetime `'lt0`, `'lt1`, etc.

We create the patches for those lifetime repairs using [algorithm 5](#). The program is compiled, if there are bounds errors that the compiler can fix, those repairs are accepted. There are only ever a limited amount of constraints that can be added by the compiler i.e. `lt_slotcount!` which would simply have all bounds being the same as the fixpoint (for 2 lifetimes,

'l_{t0} : 'l_{t1}, 'l_{t1} : 'l_{t0}, meaning 'l_{t0} is alive exactly as long as 'l_{t1} and vice versa). Hence, this algorithm will terminate.

4.4 Implementation

We need to start the extraction process from IntelliJ's IDE which does the initial extraction using our version of the Rust plugin. Our modified plugin calculates the input variables, then using IntelliJ's type inferences, we infer types for the inputs and the method calls, and does the initial extraction i.e. lifting the body from the caller and make a new function.

Our tool is named Rusty Extraction Maestro or REM. REM is entirely built using Rust. We implemented simple constraints checking using syntactic traversals with the `syn` crate using the visitor pattern. More complex constraints analysis are done using SWI-Prolog. Since the size of the inputs are relatively small in terms of lines of code and are always localized to within one function, these two constraint analysis methods are sufficient and effective.

We split REM into three separate components each modularly addressing our motivating issues:

- controller: traverse the caller to collect the facts regarding the loops within the caller and where the callee is. Then we traverse the callee to check whether there is any `return`, and if the callee is within any loops we check for loop controls i.e. `continue`, and `break`. From those constraints, we create patches using [algorithm 1](#), and we traverse the caller and callee to applies those patches. If any of those traversal fails, we exit controller and fails the extraction.

- borrower: collects the facts from the caller, then using SWI-Prolog, check the aliases for the inputs. We also collect the facts for the mutability and ownership constraints (using [algorithm 3](#) and [algorithm 4](#) respectively) using syntactic traversals of the caller and the extracted function. Then applying [algorithm 2](#) we create patches for the program and then apply them to the caller and callee using another syntactic traversal. If any of those traversal or constraint analysis fails, we exit borrower and fail the extraction.
- repairer: annotate the lifetime slot of the extracted function using syntactic traversal. If the annotation fails, we fail the extraction. The repair loop runs `cargo check`, and using [algorithm 5](#), we create patches and apply the repairs using syntactic traversal again. We gather the bounds suggestions from the borrow checker using regular expression on the error messages.

We combine our three components together with the Rust plugin in [figure 4.2](#). If any of the component fails, the extraction stops and outputs a failure message then reverts the transformations. Since dumping of the type inferences can be offload to the compiler, this design is very modular in terms of responsibility. If we want change the analysis for propagating the non-local control flow or borrowing values, we can also swap out the controller or borrower components respectively.

4.4.1 Failure modes

If any of the component fails, the extraction stops and outputs a failure message then reverts the transformations.

1. if IntelliJ Rust plugin fails to infer the types of all expressions in the caller or fails to perform any of its initial extraction, then the

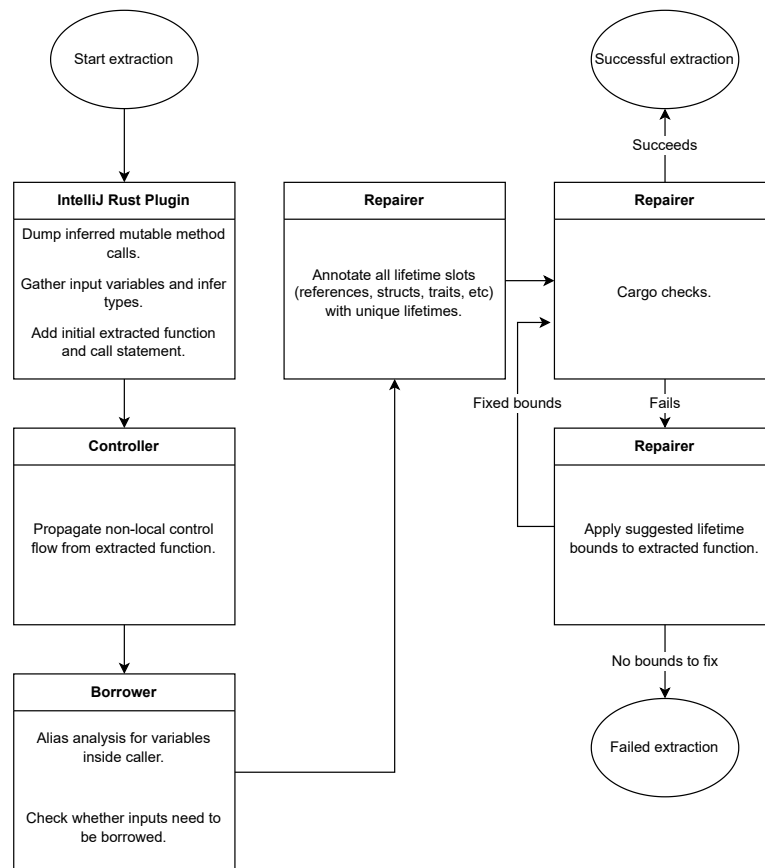


FIGURE 4.2: Implementation flow for REM

extraction does not happen and nothing is changed in the user's code. This can happen when the plugin could not figure some types through its analysis as it is not using `rustc` directly. Since we are building on IntelliJ's type inference we cannot recover from this failure.

2. if the non-local control flow transformation or the borrow and mutability transformation fails, then the extraction does not happen and nothing is changed in the user's code. Since Rust is still an evolving language, this can occur when there are extractions in features that we don't fully support and we cannot find the caller or callee or gather enough details about them.
3. if repairing the lifetime using Cargo fails, then the user can either choose to keep the refactoring with possibly failing lifetime or undo any extraction. There are two reasons for a failure here:
 - the borrow checker could potentially bounds all the lifetimes to be the same length but could still not compile it (our fixpoint of repairs). We did not encounter this failure in our experiments.
 - since we only repair bounds, any failure that is undetected in the previous steps (such as an incorrect type inference that missed a generic trait bound), will be detected here too so we will stop any bad extraction. This is the most frequent source of failure in our experiments.

All these failure modes ensure that even when REM fails, it fails gracefully and have no risk of changing the semantics of the original program.

Chapter 5

Evaluation

To re-iterate the tools we to evaluate is IntelliJ Rust plugin (0.4.186.5143-223) and Visual Studio Code's Rust Analyzer plugin (v0.3.1451) with latest versions as of February 2023. For the implementation, I am using SWI-Prolog version 8.4.3. The experiment is run on my laptop with CPU AMD Ryzen 7 4800HS on 16GB of memory, running Fedora Linux 37.

5.1 Effectiveness

To evaluate our implementation we ran 40 experiments on 5 different projects (Table 5.1). We did 3 different kind of experiment: arbitrary extraction where we selects a random chunk of code and extract it; inline and extract where we inline a function that the developer created then re-extract it; and commit-based extraction where we find a commit in the project history that does extract method refactoring and then extract it.

Consider an extraction for `gitoxide` (Table 5.1 #16), where the developer does bytes checks, decodes the hex prefix of a slice and then do a conditional control flow.

```
1 pub fn hex_prefix(data: &[u8; 4]) -> Result<..., Error> {
2     for (...) in &[...] { if ... { return Ok(...); } }
3     if wanted_bytes == 4 { return Err(Error::DataIsEmpty); }
```



```

4     Ok(PacketLineOrWantedSize::Wanted(wanted_bytes))
5 }

```

REM extracted bar.

```

1 ... let wanted_bytes = match bar(hex_bytes)
2 { RetBar::Ok(x) => x, RetBar::Return(x) => return x }; ...
3 fn bar(hex_bytes: &[u8]) -> RetBar<usize, Result<..., Error>> {
4     for (...) in &[ ... ]
5         { if ... { return RetBar::Return(Ok(...)); } }
6     if wanted_bytes == 4 { return RetBar::Return(Err(...)); }
7     let result = wanted_bytes; RetBar::Ok(result)
8 }
9 enum RetBar<A, B> { Ok(A), Return(B) }

```

The borrow needed was that of `hex_bytes`, which both REM and the developer use the immutable borrow. The lifetime are elided in both the developer and REM. Our propagation of the non-local control flow is more verbose than the developer's but with the `enum` usage, we are more adaptable to the different kind of control flow (such as loop controls which you can see is in [Table 5.1](#)).

Another more complex extraction requires using 2 structs that need lifetime bounds. In `gitoxide` ([Table 5.1](#), #17), modifies some trait references in a vector—we examined this back in [subsection 3.2.3](#).

```

1 fn _<'a>(_: &mut File<'a>, _: &mut Vec<values::Path<'a>>,
2     id: SectionId)

```

The developer extracts the function and rightly identifies that `File` requires a lifetime and that it needed to be bounded to `values::Path` and a possible solution is to bound them to the same lifetime `'a`. REM extracted bar.

```

1 fn bar<'lt0, 'lt1>(_: &mut File<'lt0>,
2   _: &mut Vec<values::Path<'lt1>>, _: &SectionId )
3   where 'lt0: 'lt1

```

The use of two lifetimes and bounds rather than one with that of the developer extraction, makes the memory use more flexible for `bar` because `'lt1` can live as long as it needs to then die without waiting for `'lt0` to die as well.

This extraction has some small idiosyncracies with the dereference and re-reference such as `&*id`. Since `id` was demoted to a reference meaning we need to dereference it if we need to use it but in the extraction `id` is being referenced again. The code is still readable and can be corrected by simply deleting `&*`.

A slightly more idiosyncratic version of the multiple lifetime bounds is an extraction (Table 5.1, #19) where we have a generic that requires lifetime annotations in the signature. We inline a method with the following signature:

```

1 pub fn _<'a, E: TA<&'a [u8]> + TB<&'a [u8]>>(i: &'a [u8])
2   -> R<&'a [u8], SR<'a>, E>

```

When we extract our inline method we got the following signature:

```

1 fn bar<'lt0, 'lt1, 'lt2, 'lt3, 'lt4,
2   E: TA<&'lt1 [u8]> + TB<&'lt2 [u8]>>
3   ( i: &'lt0 [u8]) -> R<(&'lt3 [u8], SR<'lt4>), Err<E>>
4   where 'lt0: 'lt1, 'lt1: 'lt2, 'lt2: 'lt1, 'lt0: 'lt3,
5     'lt1: 'lt3, 'lt0: 'lt4, 'lt1: 'lt4

```

It is worth noting that while REM's version is more complex, it is less restrictive. There are simplification to be done like `'lt1: 'lt2` and

'lt2: 'lt1 means 'lt1 and 'lt2 are the same lifetime. REM can simplify these bounds further by representing them as a graph and collapse the cycles into a single node. With these simplification the above signature becomes:

```

1 fn bar<'lt0, 'lt1, 'lt2, 'lt3,
2   E: TA<&'lt1 [u8]> + TB<&'lt1 [u8]>>
3   (i: &'lt0 [u8]) -> Result<(&'lt2 [u8], SR<'lt3>), Err<E>>
4   where 'lt0: 'lt1, 'lt0: 'lt2, 'lt1: 'lt2,
5         'lt0: 'lt3, 'lt1: 'lt3,
```

More permissive: we can demonstrate this less restrictive aspect by examining these two functions `foo` and `bar`, which has the same body of assignment and returning a reference which requires a bound such that the assignee lifetime of `y` lives at least as long as the assigned slot lifetime within `x` and that `y` lives at least as long as the output reference.

```

1 fn foo<'a>(x: &'a mut i32, y: &'a i32) -> &'a i32 { *x = *y; y }
2 fn bar<'a, 'b, 'c>(x: &'a mut i32, y: &'b i32) -> &'c i32
3   where 'b : 'a, 'b : 'c { *x = *y; y }
```

`foo` solves this by having all the references have the same lifetime `'a` and `bar` assign all the references with different lifetimes and then bounds such that `'b` lives at least as long as `'a` and `'b` lives at least as long as `'c`. Consider the listing below where we have `x` lives within the inner scope and is dropped when it ends while `y` and `z` lives in the outer scope and outlives `x`.

```

1 fn foobar() { // outer scope
2   let y = &0; let z;
3   { // inner scope
4     let mut x = 1; z = bar(&mut x, y);
```

```
5   } // end inner scope
6   println!("{}", z) // end outer scope
7 }
```

Using `bar`, we simply check that the value `y` references will live at least as long as `x` and `z`—which is respected. However, when we replace `bar` with `foo` we get an error that `x` does not live as long as its borrow `z`. There is no good reason for this error because `z` is only ever a reference to what `y` borrows. Hence, our extraction is more permissive.

In cases where we use our failure mode we prevent badly refactored code. For [Table 5.1 #7](#), we failed to infer some additional trait bounds for generic `G` for `Dot`.

```
1 impl<'a, G> Dot<'a, G> where G: A + B + C + D
```

While we could apply the same repairs we did for lifetime here, without further constraint collection and reliable type inference this is also non-trivial. However, we simply inform the user that we failed to run cargo check on the extraction, and we offer to restore the code back—this was not the case with IntelliJ’s Rust plugin or Rust Analyzer which simply extracted incorrect code. For the other cases ([#20](#), and [#32](#)) where we could not extract correctly, we give the user this option (see more for [#32](#) in [subsection 5.3.3](#)).

Another interesting case is [#18](#), where IntelliJ’s old Rust plugin succeeds in producing correct code that is closer to what the developer did and simpler than what REM extracted while Rust Analyzer failed to extract. Firstly, Rust Analyzer failed to extract this because it was trying to propagate the non-local control flow but without wrapping the output of the extracted function. IntelliJ’s old Rust plugin simply copy-pasted the

function body and the signature of the caller name. While there were non-local control flow, since we extracted the entire body of the caller, those control flow is propagated correctly without any match-statement. REM extracted a more pendantic version bar.

```
1 pub fn name(path: &BStr) -> Result<&BStr, name::Error> {
2     match bar(path)
3     { RetBar::Ok(x) => x, RetBar::Return(x) => return x, }
4 }
5 fn bar<'lt0, 'lt1, 'lt2>(path: &'lt0 BStr)
6     -> RetBar<Result<&'lt1 BStr, Error>, Result<&'lt2 BStr, Error>>
7     where 'lt0: 'lt1 {...}
```

Since we propagated the non-local control flow here, we actually created two lifetime slot in the output while IntelliJ's old Rust plugin (and the developer's) simply had one namely `Result<&BStr, name::Error>`. This means that their version could have lifetime elided legally because there is one input lifetime slot and one output lifetime slot. However, in REM case, we needed to bound the input lifetime `'lt0` to the first generic of our variant type `RetBar` because that is the output that uses the path reference. Note that while we have a "more permissive" annotation that that of the elided one (which is the same lifetime for both input and output), we will not ever accept any bad code with this exact signature because we only ever takes in one input reference still. This is the only exception to the non-elidable lifetime feature in the extracted code in [Table 5.1](#) where we could extract correctly, the other extraction failed to produce correct code. In these cases (#3, #9, #12, #17-#20), we provide good annotations of the implicit lifetime constraints while IntelliJ's old Rust plugin

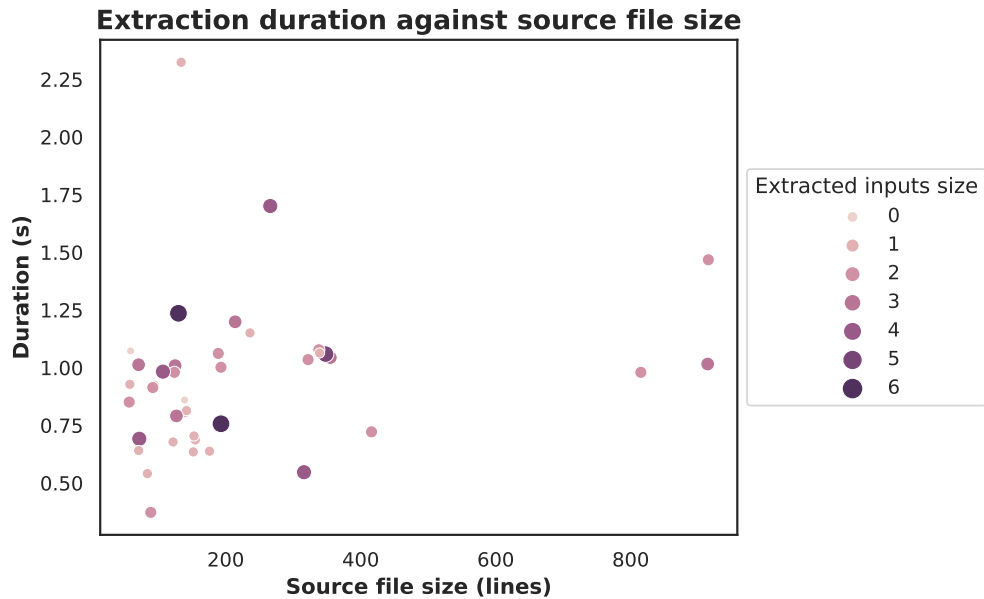


FIGURE 5.1: Extraction duration against source file size

and Rust Analyzer did not.

5.2 Efficiency

While cargo takes a long time to build application and generate the binary, in repair iterations it does not take that long because we are only running `cargo check` that will tell use the compile errors. Furthermore, since extract function refactoring is localized to one particular file in a project, even if it's for bigger project, cargo will complain while it compiles the edited file (in a possible sub-workspace) and will error out within a second. The only long check time is during the first check when all the metadata are generated. All those metadata are cached and subsequent checks are only done on "dirty" modules. We time our extraction after this first check because any developer will most likely have to compile their project at least once while writing code.

The project then size only has minimal effect on REM. Within our samples size we have projects of size between 1,500 lines and 20,000 lines of Rust code with very little differences between them so our tool is quite scalable (Table 5.1). Our total repair time with all the 3 components combine comes to a mean of less than 1s within our experiments. This is good because REM can be used in big or small project without much worries about extraction time.

As you can see in Figure 5.1, there is no clear relation between source size and duration, while most extraction ends within 1.5s, there are a few that goes over that with one extraction taking up to 2.5s (Table 5.1, #19). This extraction, which we evaluated in the effectiveness section (5.1) above, takes up 3 cycles to repair (+ 1 final check to confirm no more repair is needed) while the next example #25, also within the same project and have the highest source file size but did not take up cargo repair cycles finishes extracting in 0.7s. Hence, the repair cycle counts appear to have more impact towards the extraction time.

Within Table 5.1 and Figure 5.1, we looked closer at the function we extracted from (i.e. the “caller”). We considered whether there were any correlation between the size of the inputs to the extraction duration but there doesn’t appear to be any clear correlation on that as there are extracted functions with bigger input counts across Figure 5.1 too. rustc also provides feedback for all the bounds it can figure out at that time so REM also applies all those suggestions at a time minimizing the number of cycles needed. While there are 7 bounds added for #19, it took only 3 cargo check cycles meaning also that the cargo check cycles does not necessarily scale one-to-one with the extracted function input size or

the bounds needed between them. The size of the function we extracted from does not seem to correlate to the duration either so we can arbitrarily extract large chunks of code from the project itself without worries.

While REM is slower than the original plugin by IntelliJ and Rust Analyzer, it saves more time than annotating all the lifetimes and borrow manually. We can also answer the question whether it'd be tolerable to use `cargo` instead of `rustc` for compiling the project without slicing our source file and creating stumps so we can use `rustc` for compiling just the stumps. It turns out to be good enough to use `cargo check` even on bigger projects and still have an efficient solution.

5.3 Discussions

5.3.1 Type Inferences

Since we depend entirely on IntelliJ's type inference, sometimes we do not get the correct bounding for generics. In our experiment [Table 5.1 #7](#), we examined in [5.1](#) and mentioned that we could also apply the same treatment for this trait bound repair as we did with lifetimes. However, that would also increase our possible transformation space that is difficult to account for. The best solution is querying the compiler and get the type for a particular expression so we have to wait until the `rustc`'s developers offer that.

This limitation is a host to related problems.

Qualified name: getting the correct level of qualified name is not straight-forward as some extraction failed due to not having a use declaration at the top level but IntelliJ does not provide the path for that type.

#	Project (LOC)	Type	Size (LOC)		Code Features						Outcome			Cargo Check	Time (sec)		
			CLR	CLE	NLL	NLR	IB	MB	NEL	SHL	IJR	VSC	REM				
1	petgraph (20,157)	⊙	21	10								✓	⊕	✓	0	0.37	
2		⊙	20	11									✓	✗	✓	0	1.02
3		⊙	8	5									✗	✗	✓	1	1.47
4		⊙	54	26			✓			✓			✓	✗	✓	0	1.7
5		⊙	51	15			✓	✓					✓	✗	✓	0	0.85
6		⊙	21	8			✓						✓	✓	✓	0	0.98
7		⇕	54	49			✓			✓			✗	✗	⊕	1	0.55
8	gitoxide (20,211)	⊙	8	5								✗	✓	✓	0	0.93	
9		⊙	53	35		✓			✓	✓			✗	✗	✓	1	1.24
10		⊙	16	10									✓	✗	✓	0	0.64
11		⊙	17	9									✗	✗	✓	0	0.81
12		⊙	50	13					✓	✓			✗	✗	✓	0	0.81
13		⊙	13	8									✗	✗	✓	0	0.86
14		⊙	30	15				✓		✓			✗	✗	✓	0	0.69
15		⊙	34	7									✗	✓	✓	0	0.68
16		☺	47	21		✓							✗	✓	✓	0	0.54
17		☺	73	11			✓	✓	✓	✓			✗	✗	✓	1	1.2
18		⇕	30	27		✓				✓			✓	✗	✓	1	0.92
19		⇕	60	55						✓	✓		✗	✗	✓	3	2.32
20		⇕	116	6				✓	✓				✗	✗	⊕	1	1.15
21		⇕	50	9									✓	✓	✓	0	0.69
22		⇕	47	6									✓	✓	✓	0	0.64
23		⇕	132	14									✓	✓	✓	0	0.7
24		⇕	38	3			✓						✓	✓	✓	0	0.64
25	⇕	65	17			✓			✓			✗	✗	✓	0	0.72	
26	kickoff (1,502)	⇕	56	16			✓	✓		✓		✓	✓	✓	0	1.03	
27		⊙	53	7		✓							✗	✓	✓	0	1.01
28		⊙	51	17									✓	✓	✓	0	0.91
29		⊙	34	7									✓	✓	✓	0	0.98
30		⊙	21	13			✓	✓		✓			✗	✓	✓	0	0.79
31	sniffnet (7,304)	⇕	71	21			✓					✓	✓	✓	0	1.04	
32		⇕	180	50			✓		✓				✗	✓	⊕	1	0.76
33		⊙	50	14			✓	✓		✓			✓	✓	✓	0	1.01
34		⊙	98	28				✓		✓			✗	✓	✓	0	0.98
35		⊙	27	13									✓	✓	✓	0	1.06
36		⊙	55	20			✓						✓	✓	✓	0	1.0
37		⊙	45	15									✓	✓	✓	0	1.06
38		⊙	20	13				✓					✗	✓	✓	0	1.08
39		⊙	71	17									✓	✓	✓	0	1.06
40	beerus (302)	☺	26	23								✓	⊕	✓	0	1.07	

TABLE 5.1: Statistics for the case studies on five projects with its size in lines of code. The types of case studies include reproducing refactoring from a commit by a human developer (☺), inlining an existing function and extracting it again (⇕), and arbitrary extraction of a code fragment (⊙). The sizes of these cases in lines of code for the caller function (pre-extraction) (CLR), and extracted function i.e. the callee (CLE). Notable language features occurring in the refactored code fragments include: non local loop (NLL), non local return (NLR), immutable borrow (IB), mutable borrow (MB), non elidable lifetimes (NEL), struct has lifetime slot (SHL). The types of refactoring outcomes for IntelliJ IDEA Rust plug-in (IJR), VSCode Rust Analyzer (VSC), and REM include: producing well-typed code (✓), producing ill-typed code (✗), and refusing to perform the refactoring (⊕). For REM, we count the cargo check repair cycles, and measure the total time taken to extract the case study in seconds.

Currently, the implementation solves this by having IntelliJ (through its inferences) dump the fully qualified name and then traversing through the qualified name and drop the part before the current crate name (inclusive). However, when IntelliJ does not give the correct path, it also fails.

Copy trait: if we have access to determine which type implement `Copy` trait, our extraction could be much simpler with regards to ownership and borrows.

Macros: requires much more “hard-coding” of the handling so we are only supporting `vec!` and `*print*!` macros in our extractions since IntelliJ does not infer the type of usages within them. However, if we can query the type of an expression, we can know whether the macro mutates the reference or not and decides our ownership of extracted function.

Slicing: properly allows us to compile only a small subsection of the project using `rustc` and use stumps for dependencies which is much lighter than `cargo`. Currently, using `rustc` is not feasible with REM as we cannot know the dependencies so while we can have some liveness analysis to slice the function and its caller, we cannot properly stump out the dependencies.

While type inference is a non-trivial limitation, we have built REM in such a modular way that we can forgo this limitation to use compiler query quite easily, however, as the query interface still is unstable and we have committed to IntelliJ since the beginning, we will keep the project as such.

5.3.2 Cargo Check Trade Off

While cargo check makes our solution extremely fast (after a first check is done), we have a problem with strict checks over dependencies. Sometimes one dependency of a submodule we are working on, failed the check because of some missing crate in one of its versions.

```
1 error[E0463]: can't find crate for 'unicode_normalization'
2   --> .cargo/registry/src/github.com-_/idna-_/src/uts46.rs:17:5
3 17 | use unicode_normalization::{is_nfc, UnicodeNormalization};
4   |     |\verbatim{^}| can't find crate
```

However, when we run cargo build, it completes successfully. There is no good reason for this stringency on dependencies in an extraction but sacrificing cargo check efficiency for cargo build would be unacceptable.

5.3.3 Technical Detail

Struct Punning: handling the dereferences within the extracted function body such that trait initialization does not fail because of bad naming—it is a technical implementation that can be extended by further traversing the AST to get the labelled fields of the struct. More specifically REM failed when it is initializing a struct A with field name x but we borrowed x so the initialization code in the body which was previously A {x} becomes A{*x} which cause the failure—it can be simply corrected by changing the struct initialization to A{x: *x}. Rust Analyzer succeeds in this respect because it recognize that the required value implements Copy so it can simply takes the ownership in the extraction and not borrowing so that particular initialization succeeded—although, when we remove

the `Copy` trait of the struct, that particular extraction also failed due to the exact same problem.

Non-local control flow syntax sugar: currently, we do not support de-sugaring non-local control flow syntax such as `?>` and we have changed some examples to not use it before running our experiments. This can be fixed by de-sugaring the syntax.

Chapter 6

Conclusion

In conclusion, we recognize that extracting a function in Rust is non-trivial. This stems mostly from the analysis that `rustc` does (especially in the borrow checker) that provides implicit contracts. When we extract a function, we make those implicit contracts, explicit through our extracted function signature and therein lies the challenge—getting the explicit contract right.

Pursing that correct explicit contract, we contribute a constraint-analyze-patch pipeline that can propagate non-local control flow semantics from the extracted function to the caller, and gives the least permissive type to the extracted function parameters. We also contributed a novel use of program repair in refactoring by using `rustc` as an oracle to repair our lifetime annotations.

We evaluated that REM is both effective in producing readable code, and is efficient enough to be usable in arbitrarily large project extracting arbitrarily large chunk of code. We further discuss the limitation of REM on type inference that we rely on IntelliJ for. Ideally, we would be depending on `rustc` interface for this but it is still in development. However, since REM is very modular, we are ready to swap over when `rustc` interface is ready.

Bibliography

- Emre, Mehmet, Schoroeder, Ryan, Dewey, Kyle, and Hardekopf, Ben. Translating c to safer rust. *Proc. ACM Prof. Lang.*, 5(OOPSLA):121, 2021.
- Klabnik, Steve and Nichols, Carol. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019. ISBN 9781718500440.
- Martin, Robert Cecil. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. ISBN 978-0-13-235088-4.
- Matsakis, Nicholas. Non-lexical lifetimes: introduction. 2016.
- Matsakis, Nicholas and Klock II, Felix S. The rust language. 2014.
- Ringdal, Per Ove. Automated refactoring of rust programs. 2020.
- Sam, Garming, Cameron, Nick, and Potanin, Alex. Automated Refactoring of Rust Programs. In *Proceedings of the Australasian Computer Science Week Multiconference, ACSW*, pages 14:1–14:9. ACM, 2017. doi:10.1145/3014812.3014826.
- Schäfer, Max and de Moor, Oege. Specifying and implementing refactorings. In *OOPSLA*, pages 286–301. ACM, 2010. doi:10.1145/1869459.1869485.

Schäfer, Max, Ekman, Torbjörn, and de Moor, Oege. Sound and extensible renaming for Java. In *OOPSLA*, pages 277–294. ACM, 2008. doi:10.1145/1449764.1449787.

Schäfer, Max, Verbaere, Mathieu, Ekman, Torbjörn, and de Moor, Oege. Stepping Stones over the Refactoring Rubicon. In *ECOOP*, volume 5653 of *LNCS*, pages 369–393. Springer, 2009. doi:10.1007/978-3-642-03013-0_17.

Appendix A

Example Code Refactoring

A.1 Zola

commit preview:

```
1 commit 774514f4d4efc65e99a9108a6fc886e9747e567c
2 Author: Peng Guanwen <pg999w@outlook.com>
3 Date: Sat Jan 5 22:37:24 2019 +0800
4
5 refactor markdown_to_html
6
7 this commit contains two refactors:
8 - extract custom link transformations into a function.
9 - separate some trivial markup generation.
```

A.2 Rust

commit preview:

```
1 commit 1f0a96862ac9d4c6ca3e4bb500c8b9eac4d83049
2 Merge: bf242bb1199 48a48fd1b85
3 Author: bors <bors@rust-lang.org>
4 Date: Wed Feb 9 09:41:48 2022 +0000
5
6 Auto merge of #92306 - Aaron1011:opaque-type-op, r=oli-obk
7 ...
```



```
8      * The body of `try_extract_error_from_fulfill_cx`
9      has been moved out to a new function\
10     `try_extract_error_from_region_constraints`.
11     This allows us to re-use the same error reporting code between
12     canonicalized queries (which can extract region constraints directly
13     from a fresh `InferCtxt`) and opaque type handling (which needs to
14     take region constraints from the pre-existing `InferCtxt` that we use
15     throughout MIR borrow checking).
```

A.3 Gitoxide

commit preview:

```
1 commit c0786717c4979810002365a68d31abbf21d90f2d
2 Author: Svetlin Stefanov <s.m.stefanov@gmail.com>
3 Date: Sat May 28 09:59:12 2022 +0200
4
5     Extract include_paths.
```