Adventure of a Lifetime: Extract Method Refactoring for Rust

SEWEN THY, Yale-NUS College, Singapore
ANDREEA COSTEA, National University of Singapore, Singapore
KIRAN GOPINATHAN, National University of Singapore, Singapore
ILYA SERGEY, National University of Singapore, Singapore

We present a design and implementation of the automated "Extract Method" refactoring for Rust programs. Even though Extract Method is one of the most well-studied and widely used in practice automated refactorings, featured in all major IDEs for all popular programming languages, implementing it soundly for Rust is surprisingly non-trivial due to the restrictions of the Rust's *ownership* and *lifetime*-based type system.

In this work, we provide a systematic decomposition of the Extract Method refactoring for Rust programs into a series of program transformations, each concerned with satisfying a particular aspect of Rust type safety, eventually producing a well-typed Rust program. Our key discovery is the formulation of Extract Method as a composition of naïve function hoisting and a series of *automated program repair* procedures that progressively make the resulting program "more well-typed" by relying on the corresponding *repair oracles*. Those oracles include a novel static intra-procedural *ownership analysis* that infers correct sharing annotations for the extracted function's parameters, and the lifetime checker of rustc, Rust's reference compiler.

We implemented our approach in a tool called REM—an automated Extract Method refactoring built on top of IntelliJ IDEA plugin for Rust. Our extensive evaluation on a corpus of changes in five popular Rust projects shows that REM (a) can extract a larger class of feature-rich code fragments into semantically correct functions than other existing refactoring tools, (b) can reproduce method extractions performed manually by human developers in the past, and (c) is efficient enough to be used in interactive development.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; *Software evolution.*

Additional Key Words and Phrases: Rust, automated code refactoring, program repair

ACM Reference Format:
Sewen Thy, Andreea Costea, Kiran Gopinathan, and Ilya Sergey. 2023. Adventure of a Lifetime: Extract Method Refactoring for Rust. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 245 (October 2023), 28 pages. https://doi.org/10.1145/3622821

1 INTRODUCTION

Code refactoring—a set of techniques for restructuring programs without changing their meaning (Fowler and Beck 1999; Opdyke 1991)—is a well-studied research topic in software engineering with a long history (Mens and Tourwé 2004; Roberts et al. 1997; Schäfer 2010). As of early 2023, implementations of automated code refactorings form a large part of the functionality of modern integrated development environments (IDEs), such as Visual Studio Code (Microsoft 2023), Eclipse (Fuhrer et al. 2004), and the family of tools based on JetBrains' IntelliJ IDEA (Jemerov 2008). Common refactorings are well-understood (Schäfer and de Moor 2010), and re-implementing them for new popular languages poses little challenge for IDE developers. This is not the case for Rust.

Authors' addresses: Sewen Thy, Yale-NUS College, Singapore, s.thy@u.yale-nus.edu.sg; Andreea Costea, National University of Singapore, Singapore, andreeac@comp.nus.edu.sg; Kiran Gopinathan, National University of Singapore, Singapore, kirang@comp.nus.edu.sg; Ilya Sergey, National University of Singapore, Singapore, ilya@nus.edu.sg.

2475-1421/2023/10-ART245
https://doi.org/10.1145/3622821

```
1  const W: [i32; 1] = [5];
2  pub fn foo() {
3    let x: [i32; 1] = [1];
4    let xref = &x;
5    let mut z : &[i32];
6    {
7      let y: [i32; 1] = [2];
8      z = &y;
9      z = if z[0] < xref[0] {
10           &y
11         } else {
12           &W
13         };
14     println!("{:?}", z);
15   }
16 }
```

(a) A program before the refactoring

```
1  const W: [i32; 1] = [5];
2  pub fn foo() {
3    let x: [i32; 1] = [1];
4    let xref = &x;
5    let mut z : &[i32];
6    {
7      let y: [i32; 1] = [2];
8      z = &y;
9      z = bar(z, xref, y);
10     println!("{:?}", z);
11   }
12 }
13 fn bar(z: &[i32], xref: &[i32], y: [i32; 1])
14         -> &[i32; 1] {
15   if z[0] < xref[0] { &y } else { &W }
16 }
```

(b) Result by VSCode Rust Analyzer (doesn't compile)

Fig. 1. Example of a non-trivial Extract Method refactoring instance for a Rust program.

Rust (Rust Team 2017) is a relatively new systems programming language designed for writing low-level code that enjoys high-level correctness guarantees, such as type- and thread-safety, as well as runtime performance of traditional "unsafe" systems languages, such as C/C++. A distinct feature of Rust is its *ownership-based* type system that supports automated reasoning about most common patterns of low-level manipulation with pointers and references, and makes it possible to prove, at compile time, the absence of memory leaks or data races, while also eliminating the need for manual memory management in a runtime that does not rely on a garbage collector. Despite its advantages for safety and performance (Ayooluwa et al. 2020; Qin et al. 2020), the type system of Rust poses unique challenges to the development of language tooling support. In particular, in the case of Rust, implementations of automated refactorings need to take the intricacies of its type system into account in order to produce code that is type-correct.

Consider the popular automated code refactorings "Extract Method", which is used to hoist a lexically well-scoped piece of code into a separate function with suitable parameters and result type (Murphy-Hill and Black 2008). Fig. 1a shows a Rust program with a global array constant `W` and a function `foo` that manipulates several *references*, `xref` and `z`, to 32-bit integer arrays *owned* by local variables `x` and `y`, respectively. Let us assume the user's intention here is to extract the conditional `if-else` expression at the lines 9–13 to a separate function `bar` by invoking the refactoring.

Fig. 1b demonstrates the result of invoking such a refactoring from a popular Visual Studio Code plugin Rust Analyzer. Unfortunately, this code is rejected by the compiler for a number of reasons. The first one has to do with the way the extracted function handles its parameter `y`: its type `[i32; 1]` indicates that `bar` takes the *ownership* of `y` from its caller. Amongst other things, an owner can *drop*, (*i.e.*, deallocate) its content, thus making any future reference to it (*e.g.*, the one the caller makes at the `println!` statement) invalid—clearly not an outcome we desire. At the same time, the result type `&[i32; 1]` suggests that a *shared* reference will be returned—and it has to be this way since the function can also return the reference `&W` to the global constant `W` that is not owned by the function `bar`.

```
pub fn foo() {
  ...// same definitions as before
  {
    let y: [i32; 1] = [2];
    z = &y;
    z = bar(z, xref, &y);
    println!("{:?}", z);
  }
}
fn bar<'a> (z: &[i32], xref: &[i32],
    y: &'a [i32; 1]) -> &'a [i32; 1] {
  if z[0] < xref[0] { y } else { &W }
}
```

Fig. 2. Correctly extracted method bar

Even changing the parameter type for `y` to a reference type `&[i32; 1]` and passing the reference `&y` at the call site of `bar` in `foo` does not fix the code completely. The new issue is: the function's return type is a reference to some value, and all values in Rust have statically determined *owners* that define the *scope*, in which references to those values remain valid (*i.e.*, not dangling). To further correct this issue as per Rust's type system, that is to determine the scopes of the references manipulated by its caller that the result depends upon, the ascribed types must capture the intended scope of the reference returned as the result of `bar`. The main mechanism to syntactically track scopes of references in Rust is *lifetimes*. A valid version of `bar` shown in Fig. 2 exercises this mechanism by explicitly annotating the types of one of its parameters and result with a *lifetime parameter*, `'a`, stating that the scope of its resulting reference will be as large the one of the parameter `y`. This way, the contents of the caller's variable `y` will *not* be dropped as long as `z` is being used; the same holds trivially for the global constant `W`.

The outlined example provides the motivation for this work, demonstrating that, unlike code transformations for Java, Go, and Python, refactorings for Rust programs need to be aware of the semantics of ownership, sharing, and lifetimes, to produce code that compiles.

Our Approach and Key Ideas. The goal of this work is to design a robust and efficient Extract Method algorithm for Rust programs overcoming the challenges posed by its type system:

C1 Extracting a fragment of code requires granting *ownership* or ascribing suitable *sharing* type constructors to its free variables that are abstracted as the extracted function's parameters.

C2 Variables turned function parameters might also need to be annotated with *lifetimes* that soundly indicate how far the scope of their returned reference(s) should last beyond the call to the extracted function.

The reader could notice that both of these challenges have to do with inferring correct type information rather than with generating the "actual" code of the function. This is *almost* the case (we will elaborate on some intricacies below), and we exploit this observation by phrasing our refactoring as a series of *program repairs* (Le Goues et al. 2021) that elaborate code obtained by performing a "naïve" (and often incorrect) method extraction first.

Given a "functionally-correct" (but not type-correct) solution to an Extract Method task, we start by addressing Challenge **C1**. One can think that a solution to this task can be obtained by simply piggy-backing on the type checking algorithm of rustc, the standard Rust compiler. While having access to the type information definitely helps to estimate the level of ownership that can be ascribed to a parameter (*e.g.*, owned, shared, *etc*), it does not always provide the most suitable bound. For example, notice that the contents of `y` in Fig. 1a is fully owned in the scope of the body of `foo`, but are only read (and not mutated) by the fragment being extracted, and hence `y` can be turned into an *immutable borrow* (*i.e.*, shared reference)[1] when it becomes a parameter of `bar`.

This hints at the idea of a tailored analysis for inferring the most precise ownership annotations for the parameters based on their usage *within* the fragment being extracted. To solve Challenge **C1**, we define an ordered abstract domain for ownership annotations and implement an intra-procedural analysis for their inference. We use the results of the analysis to ascribe correct annotations to the introduced parameters and to adapt arguments of the call.

Unlike ownership annotations that introduce obligations on what is required *before* an argument can be passed into a call to the extracted code, depending on their use within the extracted code, correct lifetimes obligations may introduce obligations on how long references should stay alive for *after* the call to the extracted method. While it is not unreasonable to expect that inferring correct lifetimes for the extracted method's parameters might be achieved by developing another

[1] In Rust terminology, *borrow* and *reference* are often considered synonymous and we will be using them interchangeably.

intra-procedural analysis, in practice this task proved to be much more difficult, especially given the need to align the results of such an analysis with the logic of rustc's own lifetime analysis which does not have a formal specification at the time of this writing.

Our solution for Challenge **C2** follows the *refactoring-as-program-repair* line of thinking. For this, we use the compiler's feedback (*i.e.*, the error messages regarding the lifetime constraints) as a *repair oracle* to "fix" the code by introducing suitable lifetime parameters and corresponding annotations. There is an obvious technical performance issue with this approach: invoking a compiler on a sizeable project even once, just to fix method annotations, might render the refactoring prohibitively slow—even more so in the presence of complex dependencies maintained by a build tool, such as Rust's `cargo`. To avoid this potential performance issue, we solely rely on the cargo check utility as a repair oracle since it compiles the code without the expensive operation of generating code. It works remarkably well in practice, allowing us to perform Extract Method in large real-world projects in a matter of one-two seconds on a commodity laptop.

While we identify Challenges **C1** and **C2** as genuinely novel aspects of designing Extract Method algorithm for Rust, our implementation had to address several other technical obstacles, such as, *e.g.*, extracting code with non-local control flow (*i.e.*, `break` and `return` statements), and adapting the arguments at the call site of the extracted method. For the sake of making our exposition self-contained, in this paper we also present our solutions to those obstacles.

Contributions and Outline. To summarise, in this work, we make the following contributions:

- We identify and systematically characterise key challenges of implementing the automated Extract Method refactoring for Rust, explaining them from the perspective of Rust's type system.
- We describe the design of an Extract Method refactoring that is guaranteed to produce well-typed Rust code. Our key insight is to consider the refactoring as a composition of naïve extraction, as could be done in a Java-like language, with a series of subsequent program repair procedures that rely on static analyses and compiler errors. As one of such oracles, we formulate a novel constraint-based analysis for inferring the most specific yet sufficient ownership annotations.
- We implemented our approach on top of the Rust plugin for IntelliJ IDEA, a popular IDE.
- We evaluated our implementation, assessing its practicality and efficiency, by performing fourty method extractions in several popular large open-source projects. Our experiments included a number of scenarios designed by us, as well as scenarios reproducing changes in the code made manually by Rust developers in the past. Our experiments demonstrate that our approach subsumes the existing state-of-the-art implementations of Extract Method for Rust, and is fast enough to be used in the course of the IDE-assisted development.

In Sec. 2, we provide a more detailed intuition about the hindrances posed to Extract Method by Rust's ownership and lifetimes disciplines, as well as by non-local control-flow operators. Sec. 3 outlines the main components and algorithms of the Extract Method pipeline. In Sec. 4, we report on our experience of evaluating it on real-world projects. We compare our approach to related efforts in Sec. 5, discuss its possible extensions in Sec. 6, and conclude in Sec. 7.

2 OVERVIEW

We start by taking a walk through a series of examples each aimed to showcase a specific challenge posed to the implementation of Extract Method by Rust's type system and its other features.

2.1 Ownership and Mutability

Among the most unique aspects of Rust are its strict *ownership* and *borrowing* disciplines statically enforced by the compiler. Each value in Rust has a *unique owner* and zero or more *references* that *borrow* from this unique owner permissions to access the said value with one crucial safety

```
1  let y = String::new();                  //  y owns the new string
2  let mut x = y;                          // move occurs here
3  // an attempt to use y here would lead to a compilation error
4  x.push('a');                            // mutable borrow occurs here
5  let r = &x;                             // immutable borrow occurs here
6  println!("x: {}", x);                   // immutable borrow occurs here
7  println!("r: {}", r);                   // immutable borrow occurs here
```

Fig. 3. Ownership, moving, and borrowing (mutable and shared).

proviso: in any state, a value either (a) can be *mutated* by being accessed *exclusively* through a *single* reference or (b) it can be *immutably shared* by multiple references, but not both at the same time. Moreover, the owner of a value can transfer the value's ownership to a new owner—a process known as a *move*. To understand these ideas better, let us look at the code fragment in Fig. 3.

Variable `y` in the code is the *owner* of a newly allocated value of type `String`. After the move at line 2, `x` becomes the new owner, invalidating any access permission to this value via `y` from this point onwards. Next, `r` borrows from `x` an immutable access to the `String` value, *i.e.*, `r` is a borrower or an immutable reference to the value owned by `x` and `x` is a lender in relationship with `r`. A couple of salient points are worth noting: first, the signature of the `push` function is `pub fn push(&mut self, ch: char)`, hence the actual call at line 4 is desugared to `String::push(&mut x, 'a')`—in other words, `push` only borrows `x` via a *mutable* reference; second, `println` is a macro which, similar to `push`, only borrows `x`, but it does so with *immutable* permissions.

Now, consider a refactoring that extracts line 4 from the code in Fig. 3 into a separate function `bar`. Naïve Extract Method implementations would extract the call to `push` into a method that expects an argument of type `String`, since that is the type of the variable `x`, thus producing the code on the right.[2] The Rust compiler rejects the resulting code with a compilation error because the move resulted from calling `bar` on `x` implies that `x` has lost its ownership permissions over the value it previously owned, thus making the immutable borrow in the subsequent line illegal. Notice, that even though the produced definition of `bar` and the call to it are not well-typed, they can be made so, provided we know how the extracted fragment uses `x`. To achieve that, we could pre-emptively analyse the code fragment being extracted, collecting *constraints* about the uses of `x`, and then translate those constraints into *type and expression patches* (*i.e.*, fixes to the type ascribed to *x* and updating accordingly the expressions affected by this type fix), before finally applying them to the parameters and variables of the extracted function.

```
// ...
let mut x = y;
bar(x);     // move occurs here
let r = &x; // fails compilation
// ...
fn bar(s: String) { s.push('a'); }
```

Following this idea, our approach performs an intra-procedural analysis on the original code, collecting constraints that describe (*i*) how the extracted code uses `x`, *e.g.*, whether it needs to mutate the value it owns, and (*ii*) whether `x` or any of its references are being used after this call. In this example, the extracted function is going to mutate the value owned by `x`, hence it requires mutable permissions. Furthermore, `x` is still required after executing the extracted code, that is, `x` cannot lose its ownership over the value but can lend its access permissions. Following these constraints, we repair the function signature and its invocation at the call site as follows:

```
fn bar(s: String)   ⇝   fn bar(s: &mut String)
        bar(x)      ⇝   bar(&mut x)
```

[2]We used IntelliJ Rust plugin version 0.4.186.5143-223, released on 27 March 2023.

```
1    let mut y = String::new();
2    let mut x = y;              // 'a: scope of x STARTS here
3    x.push('a');                // 'b: mutable borrow STARTS and ENDS here
4    let r = &x;                 // 'c: lifetime of r STARTS here
5    println!("x: {}", x);       // 'd: immutable borrow STARTS and ENDS here
6    println!("r: {}", r);       // 'c: lifetime of r ENDS here
7                                // 'a: scope of x ENDS here
```

Fig. 4. Rust code from Fig. 3 annotated with lifetimes.

What we have seen is the first example of our program-refactoring-as-program-repair methodology. Specifically, we employed a static data flow analysis to enable a program transformations that fixed type-incorrect signature of an extracted function and its invocation.

Continuing with our example, consider the extraction of line 6 from Fig. 3 into a separate function as shown on the right. This refactoring—you've guessed it—would also be rejected by the compiler.[3] The reason for this is a bit more subtle and a full explanation requires the understanding of Rust lifetimes detailed in Sec. 2.2. For now it suffices to understand that once the owner of a value (in this case, of `s`) goes out of scope, the memory it owns is automatically freed, *i.e.*, the value is *dropped*.

```
//...
let r = &x;
baz(x);                 // move occurs here
println!("r: {}", r); // fails compilation

fn baz(s: String) {  println!("x: {}", s); }
```

In this concrete example, the string value `x` would be dropped upon return from the call to `baz`, because `s`, the new owner of the string value, goes out of scope without returning the value back to the caller. The subsequent attempt of the caller to access it via the previously created immutable reference `r` would lead to a *use-after-free* bug, hence a strong reason for Rust to reject such code. Instead of moving the value into `baz`, we could borrow it with read-only access. Notice that this is in fact the only possible solution which does not break the aforementioned invariant: at any one point only one mutable reference can exist or multiple immutable ones, but not both. The only solution accepted by the compiler is, thus, to use a shared reference:

```
fn bar(s: String)   ⇝   fn bar(s: &String)
        baz(x)      ⇝           baz(&x)
```

Although simple, these examples highlight that it is not trivial to refactor Rust code and there is no conservative solution (*e.g.*, give as much or as little ownership as possible over parameters' values) that would generally work. Instead, a detailed analysis of the value accesses is needed to find a fix that would be accepted by the borrow checker. We will present such an analysis in Sec. 3.2.

2.2 Lifetimes of References

In Fig. 3 we have seen a mutable reference being created on the invocation of `push`, and two immutable ones created afterwards—all having access to the very same `String` value. This seems to contradict the proviso we stated earlier on Rust's alias discipline stating that a mutable reference to a value cannot co-exist with any other reference to that same value. However, this invariant actually holds true due to Rust's treatment of references that are implicitly assigned *lifetimes*—named regions of code that a reference must be valid for.[4] Fig. 4 shows the same example as in Fig. 3, but now with explicitly shown lifetimes that are ascribed by the compiler to each reference.

[3]This code was produced by VSCode Rust Analyzer, version 0.3.1451, available as the latest stable release as of April 2023.
[4]The lifetime of a reference (*i.e.*, the span of time in which it can be used) should not be confused with the *scope* of the value that the reference might refer to (*i.e.*, the span of time before that value gets deallocated). In most of the cases, the latter is much larger, as a value can outlive multiple references that refer to it (Matsakis 2017a).

What makes it possible to safely allow for multiple aliases to the same value while avoiding dangling references is careful tracking done on the basis of the following main rules, which we synthesised from the Rust RFC Book (Matsakis 2017a):

L1 The lender needs to *outlive* all of its (alive) references: a reference lives from the place it started borrowing until its last use.
L2 A mutable borrow can only be created if its lender has no other borrows living at that time.
L3 The lender cannot be used as long as one of its mutable borrowers still lives.
L4 The lender cannot be modified as long as one of its (shared) borrowers still lives.

As an example for **L1**, notice that in Fig. 4 variable `x` *outlives* `r` since the region `'a` corresponding to its scope contains the region `'c` corresponding to the lifetime of `r`, generally denoted by `'a:'c`. Assigning lifetimes and checking whether they follow these rules is generally done automatically by the compiler. However, there are situations where the compiler would statically reject semantically correct programs unless lifetimes and their constraints are explicitly provided. For instance, consider the code snippet in Fig. 5. The aim of this example is to show that, to avoid dangling references by also respecting rule **L1**, rustc is implicitly extending the life span of the value owned by `z` from its last use at line 6 to the end of `y`'s lifetime at line 7. Requesting a refactoring of this code where lines 5–6 are extracted into a separate function without providing any extra hints to the compiler, would get us into trouble, as Rust is not capable of automatically extending lifetimes *inter-procedurally*, as shown in the code below:

```
1 let mut x: &i32 = &0;
2 let mut y: &i32 = &1;
3 let mut z;
4 y = {
5     z = *x + *y + 1;
6     &z };
7 println!("{}", *y);
```

Fig. 5. Extending lifetimes

```
1     let mut x: &i32 = &0;
2     let mut y: &i32 = &1;
3     let mut z;
4     y = bar(x, y, &mut z);
5     println!("{}", *y);
6     ...

10  fn bar(x: &i32, y: &i32, z: & mut i32) -> & i32  { // type error: missing lifetime specifiers
11    *z = *x + *y + 1; z
12  }
```

To help the compiler understand that this code is correct, the extracted function requires *explicit* lifetime annotations, and even more, it requires the reference passed to parameter `z` to *outlive* the returned reference. To infer such specifiers and constraints automatically during the refactoring, we could either design an intra-procedural analysis to track how the borrowing occurs and then add the lifetime specifiers based on the analysis's results, or we could leverage the power of the rustc compiler to guide us into correctly fixing the extracted function's signature. In this work we choose the latter for the following reasons. First, rustc is known for offering accurate recommendations whenever it reports a static error. Second, while less theoretically exciting than building our own lifetime analysis, explicit reliance on rustc for inferring lifetimes is more resilient to changes in the de-facto lifetime logic that is implemented by the reference compiler.

We proceed to fix the result of the refactoring from the example above (*i.e.*, the function `bar`) by (1) annotating each parameter and return type with a unique lifetime, (2) using the compiler as an oracle to collect lifetime constraints, and (3) elaborating the function's signature and go back to step (2) if the compiler still does not accept the refactored code. We call this method *loosest bounds first*, because of our adopted lifetime annotations strategy, where each parameter's lifetime is *distinct* and *unconstrained* at first and is being subsequently constrained down incrementally following the compiler's feedback. Applying this approach to our example (extracting lines 5–6

```
1  fn foo(x: i32) -> String {
2
3   let y =
4     if x > 2 {
5       x
6     } else {
7       return String::from("end")
8     };
9
10    y.to_string()
11  }
```

(a) Before extracting the lines 4–8

```
1  fn foo(x: i32) -> String {
2    let y = bar(x);
3    y.to_string()
4  }
5  fn bar(x: i32) -> i32 {
6    if x > 2 {
7      x
8    } else {
9      return String::from("end")
10   };
11 }
```

(b) Result by IntelliJ IDEA Rust plugin

Fig. 6. Extracting a code with non-local control flow operators into a separate function.

in Fig. 5) results in the code below, which is accepted by the compiler since the reference flowing into the third argument is constrained to live at *least as long as* the output reference, *i.e.*, where `'a:'b`.

```
    let mut x: &i32 = &0
    let mut y: &i32 = &1;
    let mut z;
    y = bar(x, y, &mut z);
    println!("{}", *y);
    ...
fn bar<'a,'b>(x: &i32, y: &i32, z: &'a mut i32) -> &'b i32  where 'a:'b {
  *z = *x + *y + 1; z
}
```

To simplify the signature of an extracted function, our approach additionally applies the *lifetime elision* rules recommended by the Rust Reference. This is why the parameters `x` and `y` in the final version of `bar` don't have explicit lifetime annotations.

Our method works well even in the presence of indirect references as per the code below:

```
    let p: &mut &i32 = &mut &0;  // a reference to a reference to an integer
    let x = 1;                   // a constant reference to an integer
    *p = &x;                     // a value of p is now an immutable borrow
    println!("{}", **p);
```

Although both `p` and `x` have the same lifetimes in the original code above, when extracting line 3 into a separate function, the compiler needs to know that the reference assigned to the parameter `x` lives at least as long as the reference living in `p`:

```
    let p: &mut &i32 = &mut &0;
    let x = 1;
    bar(p, &x);
    println!("{}", **p);
    ...
fn bar<'a, 'b>(p: &mut &'a i32, x: &'b i32) where 'b: 'a { *p = &x; }
```

2.3 Non-Local Control Flow

Finally, we are going to discsuss the aspect that is not specific to Rust's type system yet makes code extraction tricky in fairly common scenarios: those involving non-local control flow operators, such as **`return`**, **`break`**, **`continue`**, and, specifically, their arbitrary combinations.

Rust is primarily an expression-oriented language (Matsakis 2017b), with statements mainly used for sequential composition of computations. This allows the developers to write nested expressions, while adding more challenges for automatic refactoring. Consider the code in Fig. 6a. A naïve

```rust
fn foo(x: i32) -> String {
  let y = match bar(x) {
    Ok(value) => value,
    Err(value) => return value,
  };

  y.to_string()
}
```

```rust
fn bar (x: i32) -> Result<i32, String> {
  let result = if x > 2 {
    x
  } else {
    return Err(String::from("end"))
  };
  Ok(result)
}
```

Fig. 7. Extracting code with a non-local control via VSCode Rust Analyzer.

```rust
fn foo(x: &mut i32) -> String {

  loop {

    if *x > 2 {
      *x = *x - 1;
    } else if *x == 1 {
      *x = *x - 2;
      return String::from("42");
    } else {
      break;
    }

  }

  x.to_string()
}
```

(a) A program with both **break** and **return**

```rust
fn foo(x: &mut i32) -> String {
  loop { match bar(x) {
      Fig6::Ok(x) => x,
      Fig6::Return(x) => return x,
      Fig6::Break => break,
    }
  }
  x.to_string()
}
fn bar(x: &mut i32) {
  let result = if *x > 2 { *x = *x - 1; }
  else if *x == 1 {
    *x = *x - 2;
    return Fig6::Return(String::from("42"));
  } else { return Fig6::Break; };
  Fig6::Ok(result)
}
```

(b) Extract Method result by our approach

Fig. 8. Extracting code with a combination of non-local control flow operators.

extraction of the **if**-**else** expression in the lines 4–8, as by IntelliJ Rust plugin, would fail to produce code that even type-checks (*cf.* Fig. 6b). The extraction produces the function `bar` whose return type indicates an **i32** as return value, but the body returns a `String` instance in the **else**-branch. Even if we were fixing the type, the code does not preserve the initial semantics, since, upon calling `bar` it transfers the control back to the binding expression at line 3 of Fig. 6b upon executing the **return** at line 8, instead of passing the control to the caller of `foo` like in the original program.

The solution to this issue is well-known amongst the IDE developers: simply "wrap" the result of the extracted function into instances of a *tagged* data type, such as Rust's **enum**, and use different instances to indicate either "normal" termination of the function or a "special" case that needs to be handled appropriately at the call site. This solution is adopted by VSCode Rust Analyzer, which for the code in Fig. 6a produces in Fig. 7, using the library **enum** type `Result`.

Such solutions[5] turn out to be rather ad-hoc: it is capable of handling the non-local control-flow scenarios with **return** statements by using `Result` type as shown above, and with **break** *or* **continue** by using the `ControlFlow` data type from the standard library. However, it refuses to extract a function from the **if**-**else** statement between the lines 5–12 of the code shown in Fig. 8a.

Our approach proposes a more general solution to this problem by identifying the non-local control flow operators used in the fragment to be extracted and by constructing a corresponding **enum** data type whose constructors are used as indicators for the corresponding call-site logic, while carrying the respective results of the extracted function's call. For instance, for the program

[5]We tried an implementation of Rust Analyzer's Extract Method as available in early April 2023.

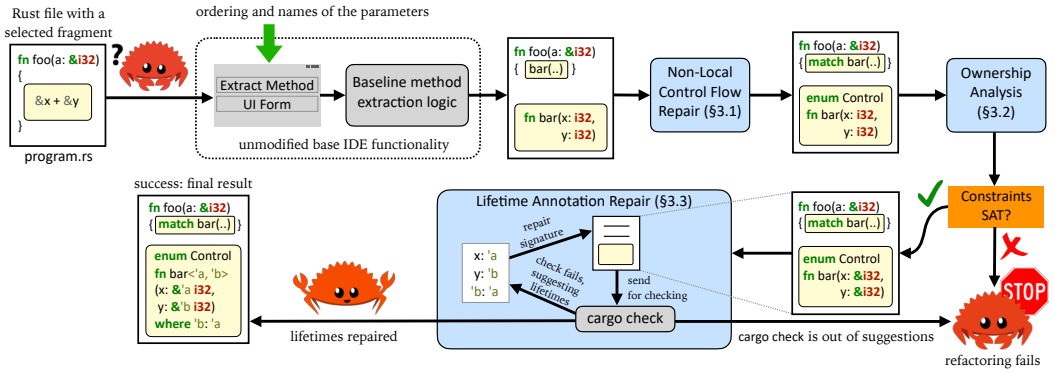


Fig. 9. High-level design of REM. Light-blue boxes are the stages of the refactoring contributed by this work. Grey boxes are the software components reused by us as-is. Some common graceful failure modes (*e.g.*, due to inability of the underlying IDE to perform baseline extraction) are elided to avoid clutter.

in Fig. 8a our approach generates the auxiliary data type `Fig6` shown on the right. This type is then used to represent the possible outcomes regarding the non-local control-flow resulting from a call to the extracted function `bar` and use them for branching within the caller, as shown in Fig. 8b.

```
enum Fig6<A, B> {
    Ok(A),
    Return(B),
    Break,      }
```

Arguably, we could have solved this problem without introducing new data-types such as `Fig6` at all, but rather by constructing a suitable composition of existing library data types, such as `ControlFlow<B, C>`. We believe, our current solution offers a better readability, and note that the two approaches are easily interchangeable for each particular combination of control-flow operators via a simple conversion between two isomorphic data types.

2.4 Putting It All Together

Our approach to Extract Method for Rust combines the outlined solutions to the challenges posed by the language aspects described in Sec. 2.1–Sec. 2.3. Each solution corresponds to a specific refactoring pass that tackles a particular aspect in a program repair-inspired fashion. Going back to our initial example from Fig. 1a: the correctly extracted function from Fig. 2 is obtained by first determining that the ownership type of `y` should be an immutable borrow via our novel ownership analysis and, second, by deriving the correct lifetime annotations by running several plausible lifetime annotation schemas against rustc and adapting the result accordingly.

In the subsequent sections, we will formally introduce the components of our refactoring and demonstrate its utility and efficiency by using it on large open-source Rust projects.

3 THE EXTRACT METHOD ALGORITHM

Fig. 9 presents the high-level of design of our refactoring tool dubbed Rusty Extraction Maestro (REM). REM extends the basic functionality of an IDE, such as IntelliJ IDEA with Rust plugin, by applying three algorithms for generating patches that fix each of the previously outlined shortcomings of the existing baseline Extract Method implementations: handling the *non-local control flow* of the extracted code (Sec. 3.1), passing arguments to the extracted method such that the refactored code respects Rust's *ownership and mutability disciplines* (Sec. 3.2), and fixing the signature of the extracted function to specify the *lifetime constraints* of related references (Sec. 3.3). In several scenarios, the REM fails to perform the refactoring, in which case it fails gracefully by reverting the changes to the original state. We discuss those failure modes in Sec. 3.4.

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 $\quad RTY \leftarrow \textsc{BuildReturnType}(R, B, C)$
6 $\quad PS \leftarrow \textsc{UpdateReturnType}(EF, RTY) :: PS$
7 $\quad$ **for** $l_r \in R$ **do** $PS \leftarrow (l_r,$ `return` $e \rightsquigarrow$ `return Ret(`e`)`$) :: PS$
8 $\quad$ **for** $l_b \in B$ **do** $PS \leftarrow (l_b,$ `break` $\rightsquigarrow$ `return Break`$) :: PS$
9 $\quad$ **for** $l_c \in C$ **do** $PS \leftarrow (l_c,$ `continue` $\rightsquigarrow$ `return Continue`$) :: PS$
10 $\quad l_E \leftarrow$ find location of the final expression of EF
11 $\quad PS \leftarrow (l_E, E \rightsquigarrow$ `Ok(`E`)`$) :: PS$
12 $\quad \overline{CS} \leftarrow \textsc{BuildCasesForReturnType}(RTY)$
13 $\quad l_{\text{caller}} \leftarrow$ location of E
14 $\quad PS \leftarrow (l_{\text{caller}}, E \rightsquigarrow$ `match` E `with` $\overline{CS}) :: PS$
15 **return** PS

3.1 Handling Non-Local Control Flow

If an expression being extracted contains `return`, `break`, or `continue` statements, then the code of the callee and caller may need additional changes to preserve the semantics of the original code. For the callee, problems arise as the behaviours of control-flow operations were defined in the context of the caller, and do not make sense within the newly extracted function. For the caller, as the callee can not affect the control flow outside of its own definition, modifications will be needed to recover the original control-flow from the results of the callee. Both of these problems relate to *propagating* the effects of the non-local control flow between function calls, which, as it turns out, can be solved in an idiomatic way by *reifying* control flow operators.

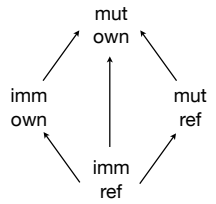
Suppose we have an extracted function that makes use of non-local control flow and so now behaves incorrectly; in this case, we can reify its non-local operations to restore the original semantics. In particular, we can augment the extracted function with a wrapper type that reifies the control flow of its body in each of its constructors, capturing whether the function terminated normally (*e.g.*, `Ok(_)`) or performed some non-local control flow operation (*e.g.*, `Ret(_)`, `Break`, `Continue`). With this transformation, it is possible to restore the original behaviour of the program by modifying the extracted function's call site to pattern-match on the wrapper result and perform the corresponding jump-like operation. While such transformation introduces an additional runtime cost with the wrapper type, those overheads most likely can be removed by an inlining compiler.

Algorithm 1 presents the formal description of our repair technique, which is based on the high-level intuition just described. The algorithm starts by collecting any non-local control flow operations in the extracted code (line 2–3) and tests the number of such operations (line 4). If the extracted expression does make use of such operations, then further changes will be needed to restore the original semantics of the program, and our algorithm initiates its repair process. The algorithm begins by using BuildReturnType to construct a new data-type to represent the control-flow operations that the extracted code makes (line 5), including a new constructor for each such operation and a distinguished constructor `Ok(_)` for the normal control flow. The algorithm then emits a patch to update the result type of the function via UpdateReturnType (line 6). The patches emitted at the lines 7–9 update the body of the extracted function, replacing any use of non-local control flow operators with returning corresponding constructors, including any values

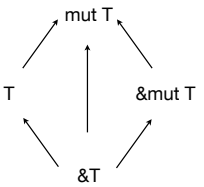
(*Mutability*) m := mut | imm
(*Ownership*) o := own | ref
(*Permission*) p := $\langle m, o \rangle$

(*Less − than*) $p_1 <: p_2$ (see Fig. 10b)

(a) Mutability and ownership capabilities



(b) Permissions lattice



(c) Permission subtyping

Fig. 10. A combination of mutability and ownership permission maps to a Rust type constructor

to be passed to the caller. To complete the update to the extracted function code, the algorithm wraps the final expression of the extracted body into the `Ok(_)` constructor, to capture the normal control flow (lines 10–11). Finally, the algorithm updates the caller (lines 12–14), replacing the call site of the extracted function with a **`match`** statement that scrutinises the returned wrapped value and enacts the corresponding effect in the context of the original function.

3.2 Inferring Ownership Annotations

As demonstrated with the examples in Sec. 2.1, producing a well-typed function as the result of Extract Method boils down to phrasing the constraints imposed by Rust's type system in terms of the output of the refactoring. At the high-level, the well-typedness of the produced code, when adapted to our cause, is a consequence of satisfying the following three requirements: (*i*) the extracted function must have sufficient permissions to type-check, (*ii*) the caller must have sufficient permissions for the references it is passing to the call to the extracted method, and lastly, (*iii*) the caller must also have sufficient permissions left for its operations after the call to the extracted method has taken some of its permissions. To infer the fixes that results in the code satisfying these three requirements, firstly, we need to understand what exactly these *permissions* are, and secondly, we also need to have a clear definition of what is the meaning of *sufficient*.

3.2.1 Constraints. We define permissions as a pair of capabilities, denoted $\langle m, o \rangle$: the mutability capability m (a variable is either mutable, mut, or immutable, imm) and the ownership capability o (a variable is either an owner, own, or a reference, ref). Fig. 10a shows the space of the capabilities. To understand what it means for permissions to be sufficient, let us revisit an example from Sec. 2.1:

```
1  bar(&x);
2  println!("x: {}", x);
3  // end of the enclosing function for the statements above
4  fn bar(s: &String) { println!("x: {}", s); }
```

Adapted for this code snippet, the above listed requirements (*i*)–(*iii*) can be stated as follows:

(*i*) Assume the parameter `s` has permission $\langle m, o \rangle$. Since `bar` performs no mutation via `s`, it is sufficient for `s` to be immutable, a constraint denoted as imm <: s.m (or, shortly, imm <: s). Furthermore, since no move occurs within this code (*i.e.*, no values are stored), it is sufficient for `s` to be a reference, denoted as ref <: s.o. Therefore, to satisfy type-checking, it is sufficient for `s` to have the ⟨imm, ref⟩ permission, denoted ⟨imm, ref⟩ <: ⟨s.m, s.o⟩ (or ⟨imm, ref⟩ <: s).

(*ii*) For `bar` to be called with permission ⟨imm, ref⟩ on the borrow `x`, it is sufficient for the caller to have it at least with permission ⟨imm, ref⟩. In other words, if the caller would have any of the ⟨mut, ref⟩, ⟨mut, own⟩, or even ⟨imm, own⟩ permission, it would still type-check, since all these permission would allow the caller to *lend* `bar` an immutable access to `x`. We observe that ascribing the least permissive permissions to a callee's parameters makes its caller less restricted with regards to the permissions it needs to have to support the call.

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 ← alias analysis on F                                   /* maps variables to their aliases */
2  Mut ← CollectMutabilityConstraints(EF, Aliases)
3  Own ← CollectOwnershipConstraints(EF, Aliases, F)
4  PS ← [ ]
5  for param ∈ EF.params do                                        /* derive patches for the signature of EF */
6      v, τ, l ← param.var, param.type, param.loc
7      if UNSAT(Mut ∪ Own, v) then raise RefactorError
8      if LUB(Mut ∪ Own, v) = ⟨mut, ref⟩ then PS ← (l, v : τ ⇝ v : &mut τ) :: PS
9      if LUB(Mut ∪ Own, v) = ⟨imm, ref⟩ then PS ← (l, v : τ ⇝ v : & τ) :: PS
10 for param ∈ EF.params do                                        /* derive the patches for the body of EF */
11     if LUB(Mut ∪ Own, param.var) = ⟨_, ref⟩ then
12         Exps ← collect from EF.body all the occurrences of param.var
13         for e ∈ Exps do PS ← (e.loc, e ⇝ (* e)) :: PS
14 for arg ∈ E.args do                                             /* derive patches for the call to EF */
15     v, e, l ← arg.var, arg.exp, arg.loc
16     if LUB(Mut ∪ Own, v) = ⟨mut, ref⟩ then PS ← (l, e ⇝ &mut e) :: PS
17     if LUB(Mut ∪ Own, v) = ⟨imm, ref⟩ then PS ← (l, e ⇝ &e) :: PS
```

(*iii*) To type-check the usage of x right after the call to bar, it is sufficient for x at line 2 to have at least the permission ⟨imm, ref⟩. Since bar only borrows immutable access to x via &x (but doesn't own it), the borrow x at line 2 still has the same permission it had before the call to bar.

We define a *less-than* relation $<:$ on all possible permissions as a partial order between capabilities in the style of (Boyland et al. 2001), as depicted in the lattice in Fig. 10b. The lattice represents a set of rules defining the subsumption of capabilities. The bottom of the lattice, ⟨imm, ref⟩, is the least permissive permission corresponding to an immutable borrow. The top of the lattice, ⟨mut, own⟩ is the most permissive combination of capabilities, corresponding to a mutable variable with full ownership permissions. In other words, a caller with a permission ⟨mut, own⟩ on a reference, should be able to offer the extracted method any permission to use it, while a caller with permission ⟨imm, ref⟩ can only lend an immutable borrow. Each permission in Fig. 10b corresponds to a Rust type constructor as depicted in Fig. 10c.

Given a set of capability constraints C, we next define a *sufficient* permission for a variable v, as the least upper bound in the lattice that satisfies all the constraints on v:

$$\text{LUB}(C, v) \triangleq \text{LUB}(\{\langle m, o\rangle \mid \langle m, o\rangle <: v \in C\})$$

Contradictions in the set of constraints C in relation to a variable v are detected as follows:

$$\text{UNSAT}(C, v) \triangleq \exists\ \{v <: \langle m_1, o_1\rangle, \langle m_2, o_2\rangle <: v\} \subseteq C\ .\ \langle m_1, o_1\rangle <: \langle m_2, o_2\rangle \wedge \langle m_1, o_1\rangle \neq \langle m_2, o_2\rangle$$

The latter case corresponds to one of the failure modes of REM, which we discuss in Sec. 3.4.

3.2.2 Constraints-Based Patch Generation. Postponing the description of the constraint generation until Sec. 3.2.3, we present Algorithm 2 that shows how, once collected, these constraints guide the inference of program patches to fix the refactored code. The algorithm expects as input the extracted function *EF*, an expression *E* representing the call to *EF*, and the original caller *F* in the form preceding the refactoring. It returns a set of patches aimed at fixing the refactored code of both the caller and the callee to respect the ownership and borrowing disciplines.

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 $Mut \leftarrow \{\mathsf{imm} <: p.var \mid p \in EF.params \wedge \forall v' \in Aliases(p.var) : v' \notin MV\} \cup$
5 $\{\mathsf{mut} <: p.var \mid p \in EF.params \wedge \exists v' \in Aliases(p.var) : v' \in MV\}$

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 $Ownership \leftarrow \{v <: \mathsf{ref} \mid v \in Borrows\} \cup \{\mathsf{own} <: v \mid v \in Own\}$

The algorithm starts by inspecting the body of the extracted function *EF* for the purpose of collecting mutability and ownership constraints. Given the set of constraints, deriving the patches for the signature of the extracted function (lines 5–9) follows the one-to-one correspondence of capability combination and Rust type constructor shown in Fig. 10. If the constraints contain contradictions (line 7), an error is raised indicating failure of the refactoring (*cf.* Sec. 3.4). For the parameters whose types are constrained to be references (lines 10–13), the algorithm produces patches to fix their accesses in *EF* to be dereferences. Finally, where applicable, the arguments at the call site of the extracted function are updated to be borrows instead of moves (lines 14–17).

3.2.3 Constraints Collection. Algorithm 3 describes the collection of the mutability constraints. For each variable in the extracted function *EF* it checks whether (a) it takes part in a store operation (line 1) or, by inspecting the type signatures, whether (b) it is an argument in a function call that requires mutability capabilities (line 2), or, whether (c) it is used to create a mutable borrow (line 3). A parameter is constrained to be at least mutable if it falls into any of the cases (a)–(c), otherwise it is constrained to be at least immutable (lines 4–5).

Algorithm 4 is in charge of collecting the ownership and borrowing capabilities. Aiming to infer the least permissive ownership capabilities for the extracted method makes it very simple: it constrains all the parameters of the extracted function to be references (line 5, $v <: \mathsf{ref}$). There is only one exception: if a parameter originates from a variable in the refactored code that is never used (directly or indirectly via aliases) after the call to the extracted function, then we do not constrain this variable to be a reference, hence the set intersection at line 3. The rationale behind this exception is that a borrow implies keeping the memory it refers to occupied throughout the call, instead of freeing it during the call itself, when it's safe. In other words instead of retiring a borrow to a lended value that is not going to be used, the callee might as well just drop that value. Ownership constraints, $\mathsf{own} <: v$, are collected for variables involved in a move operation (line 4).

These two algorithms also account for how the collected constraints are propagated through the borrows. This propagation is achieved by making the alias information extracted from the original code available to the algorithms. We implemented an Andersen-style intraprocedural

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

context-insensitive data-flow analysis (Andersen 1994) that captures may-aliases as constraints and solves them using SWI-Prolog. The example on the right demonstrates one of its applications. Assume we want to extract lines 3-4 (ignoring the `foo` function for now). In a first step, we collect a constraint stating it is sufficient for `b` to be immutable, *i.e.*, imm <: b. Knowing that `a` and `b` are aliased, this information is then propagated to `a`, *i.e.*, imm <: a. This way, although `a` is declared as a mutable borrow in the original program, it is sufficient to be declared as immutable borrow in the refactored code when used as a parameter of the extracted function `foo`.

```
1  let mut x = String::from("Hi");
2  let a: &mut String = &mut x;
3  let b = a;
4  println!("b: {}", b);
5  ...
6

fn foo(a: &String){
  let b = a;
  println!("b: {}", b);
}
```

3.3 Lifetime Annotation Repair

It is common for Rust programs to create new references within a scope of a function body, thus, exercising the mechanism of *re-borrowing*: borrowing a reference to a value that has already been borrowed. Re-borrowing takes place implicitly when passing a borrowed value to a function that requires a reference, but the original borrow would still need to be scoped after the call to the function. In this case, re-borrowing creates a new reference that has a shorter lifetime than the original borrow. In case a function returns a borrow, the relation between that result's lifetime and those of the parameters must explicitly be captured in that function's signature by means of a lender-borrower lifetime relation as demonstrated, *e.g.*, in Fig. 2. To this purpose, all returned references must bear explicit lifetime annotations, as do some of the input parameters.

Ascribing correct relations to the lifetimes of parameter and return references is the last stage of our refactoring pipeline (*cf.* Fig. 9). Determining such relations can be achieved, by *e.g.*, developing a custom data-flow analysis. However, since this type repair procedure is already applied to a "nearly type-correct" function definition, we observed that the simplest and the most robust technical solution by far is to use rustc as an oracle for deriving correct lifetime relations. Specifically, our key insight is that if we start by assigning each reference a unique lifetime, we can next rely on the Rust type checker to guide the repair of the extracted function signatures which require explicit lifetime constraints. Below, we describe Algorithm 5 that does just that.

The algorithm first annotates each reference type in the signature of an extracted function (including the result type, if necessary) with distinct lifetimes (lines 2–3). This annotation strategy

enables what we call the *loosest bounds first* approach, *i.e.*, each lifetime is different and there is no relation between any two of them. The algorithm proceeds by constraining down these lifetimes to accommodate the type checker's feedback (lines 4–9). At this stage, any such feedback, *e.g.*, *consider adding the following bound:* `'a: 'b`, is added to the `where` clause of the function signature:

```
fn bar<'a,'b>(v: &'a i32) -> &'b i32 where 'a: 'b
```

Given a finite number of reference parameters, this loop is guaranteed to terminate, since the compiler will not be suggesting redundant lifetime constraints. This repair-and-check loop repeats until either the refactoring successfully completes with no more compilation errors (line 6) or it fails with no more suggestions of lifetime constraints by `cargo check` (line 8).

Collapsing circular lifetime constraints. In the case of successful refactoring, we identify and simplify the lifetimes constraints (line 10) before applying the elision rules (Rust Reference 2023) to remove the annotations for which the compiler requires no explicit lifetime parameters in a function signature (line 11). The constraint simplification is made possible by observing that any circular dependencies between lifetime parameters can be removed by replacing all the parameters in a cycle by a single variable (*i.e.*, by collapsing the corresponding cycle into just one lifetime). For instance, consider the following signature featuring two lifetime constraints:

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

This signature is the result of rustc's lifetime checker having detected that reference `x` must live at least as long as reference `y` and vice versa: both facts reflected in the `where` clause as `'a: 'b, 'b: 'a`. We detect such cycles in the constraints by constructing a directed graph of lifetime dependencies and check it for cycles. We collapse these cycles by generating lifetime substitutions for every two lifetimes within the cycle-induced equivalance class. The signature of `bar` can thus be simplified to:

```
fn bar<'a>(x: &'a i32, y: &'a i32) -> &'a i32
```

3.4 Correctness, Extensions, and Failure Modes

Correctness. It is important to ensure that a local refactoring, such as Extract Method, is correct is the sense that it preserves the original behaviour of the program. We do not provide a fully formal correctness proof of our implementation, as it would require us to develop a formal semantics of safe Rust, a complete one currently does not exist,—an endeavour well beyond the scope of this work.[6] That said, we note that our implementation implicitly assumes type correctness (akin to Java types) but not ownership correctness of the underlying refactoring machinery inherited from the base IDE (we implemented REM on top of IntelliJ IDEA Rust plugin), modulo the shortcomings that we explicitly fix in Sec. 3.1. By such correctness we mean that the result of Extract Method is correct in a "dialect of Rust" that *does not enforce* the ownership/lifetime discipline via types, and relies on a garbage collector instead.

Tackling full Rust. For clarity, our exposition so far has presented the essentials of REM's algorithm as it relates to the core features of Rust, however, in practice, there were a number of additional language features which could interact with the repair procedure that also had to be handled in order to produce a tool suitable for refactoring real code. Most such features were straightforward to support, although one that posed slightly more challenge was in properly handling functions making use of `struct`s. In particular,

```
struct A<'a> {x: &'a i32}
struct B<'a> {x: &'a i32,
              y: A<'a> }
pub fn foo() {
  let m = A {x: &0};
  let n = B {x: m.x, m};
  /* do stuff with n */ }
```

Fig. 11. Structs with lifetimes

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

```
1 fn foo1() {
2   let a = String::new(); // extraction start
3   let b = &a[..];        // extraction end
4   println!("{}", b);
5 }
```

(a) Extending ownership scope beyond the callee

```
1 struct S { a: String, b: String }
2 fn foo2() {
3   let s = S {a: "a".to_string(), b: ...};
4   drop(s.a); // extract this line
5   drop(s.b);
6 }
```

(b) Moving out a `struct`'s field

Fig. 12. Two examples of scenarios in which REM fails to extract a method.

our implementation needed to take special considerations when extracting programs making use of such constructs because the extracted function might borrow values that are structs which themselves have fields that require lifetime annotations (*cf.* Fig. 11). When assigning lifetimes to such references, our implementation had to be careful to remain consistent with Rust's borrowing rules: a reference to a struct must live at most as long as any of the references within the struct.

Reasons why REM might fail. As mentioned above, in some scenarios our approach can fail to produce a result, in which case the refactoring reverts all its changes back to the original code. We outline those scenarios below, and demonstrate that they are uncommon in our experiments.

(1) REM fails to refactor programs that require ownership permissions both *within* the extracted code as well as *after* the call to the extracted function. Consider the following example:

```
// x is the owner of a String value
x = foo(x); // with the signature of foo being foo(s: String) -> String
println!("{}", x);
```

An attempt to extract `x = foo(x)` into a separate function generates the following constraints: $x <: \mathsf{ref}$ (because `x` is used both within the extracted function and after its call by the `println!` macro), and $\mathsf{own} <: x$ (as required by type signature of `foo`). According to the UNSAT relation in Sec. 3.2.1, these constraints form a contradiction, and our algorithm aborts this refactoring. We made this design choice since such scenarios do not represent Rust idioms that encourage manipulation with references rather than the tedious operation of moving ownership "in and out" of a function through parameters and return values, respectively. For the same reason, REM does not perform refactorings that require significantly changing the structure of a program to extend the scope of values created and owned by an extracted callee to outlive its call. A characteristic example of this situation is shown in Fig. 12a. Extracting lines 2–3 into a function would require us to introduce a new variable in the caller, passing a reference to it as an argument to the callee, to serve as a recipient for a value allocated by `String::new()`.

(2) REM does not perform extractions that require identifying ownership capabilities for separately moved fields of `struct`s that are not explicitly owned by a particular variable. An example of such manipulation is shown in Fig. 12b, in which case an acceptable extraction of line 4 would require to make a function of type `String -> ()` taking `s.a` as an argument. This limitation can be addressed by (*i*) making our ownership analysis (Sec. 3.2) type-aware so it would infer ownership permissions for structure fields, not just variables, and by (*ii*) enhancing our repair procedure (lines 14–17 of Algorithm 2) to pass separately moved-out `struct` fields as arguments (*e.g.*, `s.a` instead of `&s`) to the call to an extracted function.

At the time of this writing, none of the scenarios (1)–(2) were supported by IntelliJ or VSCode.

Lastly, we have to note that, as the last stage of REM pipeline relies on a successful run of cargo check (Sec. 3.3), it will fail if invoked on a project that did not compile to begin with—a situation that is not too uncommon in a daily development workflow of a Rust programmer. A possible

trade-off to support this scenario would be to forego the last stage of our approach entirely, settling for code that might require additional fixing later—perhaps done in the same way as we described.

3.5 Implementation

We have implemented REM as an extension to IntelliJ IDEA's plugin for Rust. REM is broadly implemented as two parts: an editor-agnostic executable implementing the repair procedure described in Sec. 3 written in around 10k LOC of Rust, and separately, a modified version of the IntelliJ Rust plugin that invokes our executable (requiring an additional 1k LOC of Kotlin). Our implementation relies on InteliiJ's Rust plugin to perform the initial extraction and provides partial type information for any arguments to the refactored code and the methods it calls, supplying this information when invoking our executable with the broken Rust program. Our executable uses Rust's `syn` library to perform manipulations of the AST, and calls out to the Rust compiler to solve lifetime constraints. We use SWI-Prolog to solve any other constraint systems required for repair. We have found that since the sizes of the refactoring changes are generally small relative to the codebase and are always localised to a single function, the combination of these two solvers are sufficient to produce a fast and effective refactoring tool.

4 EVALUATION

In this section we present the results of our evaluation of REM, comparing its performance against the state of the art Rust refactoring tools available at the time of writing and demonstrating our tool's efficacy for refactoring real-world Rust codebases.

4.1 Evaluation Design and Experiments

In order to demonstrate the utility of our tool, we conducted an empirical evaluation of REM's capabilities on a large corpus of real world Rust programs. In the evaluation, we placed an emphasis on testing our refactoring tool in terms of both the end-to-end performance of the plugin as seen by an end user and also the diversity of Rust features that could be successfully handled. We designed our evaluation to answer the following research questions:

- **(RQ1)** Is REM practical? Can it extract code from real codebases to produce extracted functions of reasonable quality, including those making use of non-trivial combinations of language features?
- **(RQ2)** Is REM efficient? Can it extract code in a reasonable amount of time?
- **(RQ3)** How does REM compare to existing refactoring tools, both in terms of performance of the refactoring, and the classes of language features that it can support?

To answer these research questions, we first selected a corpus of five widely-used Rust projects to serve as real-world codebases on which we could evaluate refactorings. The 5 projects were, (1) `petgraph`,[7] a graph library, (2) `gitoxide`,[8] a Rust implementation of git, (3) `kickoff`,[9] a GUI program launcher, (4) `sniffnet`,[10] a network packet sniffer and (5) `beerus`,[11] a lightweight client for the Starknet decentralised network. All projects are actively developed, having received commits within a month at the time of writing, and meet a minimum degree of popularity, with all projects having at least 100 stars on their respective pages on GitHub. Using these collected projects, we then devised a series of 40 different experiments to evaluate the capabilities of our refactoring tool.

[7] https://github.com/petgraph/petgraph, 2.1k stars on GitHub.
[8] https://github.com/Byron/gitoxide, 6k stars on GitHub.
[9] https://github.com/j0ru/kickoff, 200 stars on GitHub.
[10] https://github.com/GyulyVGC/sniffnet, 3.5k stars on GitHub.
[11] https://github.com/keep-starknet-strange/beerus, 100 stars on GitHub.

Table 1. Statistics for the experiments on five projects with their size in lines of code. The types of each experiment 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 each refactoring attempt are 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-elidible lifetimes (NEL), struct has lifetime slot (SHL). The types of refactoring outcomes for the unmodified IntelliJ IDEA Rust plugin (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 cycles required to infer correct lifetime annotations, and measure the total time (in seconds) taken to extract each refactoring.

#	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	1.70
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	1.20
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	0.70
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	1.00
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

Our experiments were designed to be representative of real refactorings that might be conducted by developers in practice. To this end, when possible, we selected real refactorings that could be found in the version controls of each of the projects, although, as such transformations were rarely committed alone, we also constructed a number of artificial transformations. Broadly speaking, our experiments can be classified into three different classes by their origin: (1) a refactoring based on a real commit in the project history, (2) a partially-synthetic inline-and-extract based refactoring where we manually inline an existing function and then attempt to extract it, and (3) an artifical extraction, where we select an arbitrary section of code from a method and extract it.

Additionally, in order to investigate the effect of different language features on the refactoring process, for each experiment we also recorded the types of features that the refactored code made use of, identifying 6 different classes of features that complicate the refactoring process: (1) non-local loops (NLL) for refactorings requiring extracting code within a loop, (2) non-local returns (NLR) for refactoring code using returns, code making use of (3) immutable borrows (IB) and (4) mutable borrows (MB), (5) code that contains non-elidable lifetimes (NEL) and requiring explicit annotations, and finally (6) code involving structs having lifetime parameters (SHL) (*cf.* Sec. 3.4).

The left side of Tab. 1 presents the full listing of all experiments and their properties.

4.2 Results

The right half of Tab. 1 presents the full results of our evaluation. We compare REM against IntelliJ's Rust plugin (IJR) (0.4.186.5143-223) and Visual Studio Code's Rust Analyzer plugin (VSC) (v0.3.1451) with latest versions as of February 2023. All experiments were performed on a commodity laptop with an AMD Ryzen 7 4800HS CPU and 16GB of memory, running Fedora Linux 37.

4.2.1 RQ1: Practicality. For REM to be practical, it must be able to extract code from real-world programs and produce usable refactored artefacts. As such, in our evaluation we considered an extraction successful, if it produced code that compiles, *i.e.*, it is well-typed, adheres to Rust's borrowing discipline using rustc as an oracle, and is comparable to a manually written refactoring by a developer if one exists. All refactorings were manually confirmed to be correct, *i.e.*, semantics-preserving (including preservation of lifetimes), by inspection. The sixth column of Tab. 1 (Outcome/REM) lists the results for our tool, and shows that it was successful on most experiments.

Notably, Experiments #16, #17, and #40 correspond to real refactorings found in the version controls of their respective projects, and for all such cases, REM was able to produce a result comparable to the developer-written version. For example, consider Experiment #16, taken from the `gitoxide` library, where a developer manually extracted a section of code from a function into a separate helper (*cf.* Fig. 13a). Using REM to automatically perform this transformation produced a similar refactoring (*cf.* Fig. 13b), modulo some superficial differences. One such difference is in the encoding used for non-local control flow. While both refactorings introduce a new type to restore the original control flow, REM introduces a simpler generic type to capture normal and early returns, whereas the developer constructs a more bespoke type that hard-codes the logic of the original program into its constructors. In this way, although marginally more verbose than the developer-written one, by opting for a simpler transformation, REM produces more generic code.

Generally speaking, REM prioritises the generation of code that is usable in a larger number of contexts at the cost of its conciseness. An extreme example of this trade-off can be found in Experiment #19, where we first inlined and then extracted a function, `decode`, that happened to take a number of arguments containing lifetime parameters as per the following signature:

```
pub fn decode<'a, E: T1<&'a [u8]> + T2<&'a [u8]>>(i: &'a [u8]) -> Result<&'a [u8], B<'a>, E>
```

When we extract this, REM's algorithm initially produces a function with the following signature:

```rust
pub fn hex_prefix_dev(data: &[u8; 4])
 -> Result<LineOrWanted, Error> {
  for ... in ...
  { if ... { return Ok(Line(...)); } }
  ... if ... {return Err(...)}
  Ok(Wanted(...))
}
enum LineOrWanted {Line(...),Wanted(usize)}

fn streaming(...) -> Result<...,...> {
 ... match hex_prefix_dev(...){
  Wanted(...) => ...,
  Line(...) => return Ok(...)
 } ... }
```

(a) Developer's extraction

```rust
fn hex_prefix_REM(data: &[u8; 4])
 -> RetHex<usize, Result<..., Error>> {
  for ... in ...
  {if ... {return RetHex::Return(Ok(...))}}
  ... if ... {return RetHex::Return(Err(...))}
  RetHex::Ok(...)
}
enum RetHex<A, B> { Ok(A), Return(B) }

fn streaming(...) -> Result<...,...> {
 ... match hex_prefix_REM(...){
  RetHex::Ok(...) => ...,
  RetHex::Return(...) => return ...
 } ... }
```

(b) REM's extraction

Fig. 13. Developer-written extractions with non-local control flow versus code produced by REM

```rust
fn decode_extracted<'lt0, 'lt1, 'lt2, 'lt3, 'lt4, E: T1<&'lt1 [u8]> + T2<&'lt2 [u8]>>
  (i: &'lt0 [u8]) -> Result<(&'lt3 [u8], B<'lt4>), Err<E>>
where 'lt0: 'lt1, 'lt1: 'lt2, 'lt2: 'lt1, 'lt0: 'lt3, 'lt1: 'lt3, 'lt0: 'lt4, 'lt1: 'lt4
```

Observe that this signature, while correct, contains some circular lifetime constraints that are redundant: `'lt1: 'lt2` and `'lt2: 'lt1` require that `'lt1` and `'lt2` are equal. As a post-processing step, REM will simplify these bounds further by collapsing any such cycles into a single lifetime:

```rust
fn decode_extracted<'lt0, 'lt1, 'lt3, 'lt4, E: T1<&'lt1 [u8]> + T2<&'lt1 [u8]>>
  (i: &'lt0 [u8]) -> Result<(&'lt3 [u8], B<'lt4>), Err<E>>
where 'lt0: 'lt1, 'lt0: 'lt3, 'lt1: 'lt3, 'lt0: 'lt4, 'lt1: 'lt4
```

It is worth noting that, while REM's result is more complex than the original function even after simplification, the resulting code places fewer constraints on its caller, and thus is more reusable.

Let us motivate the benefits of having such permissive lifetime constraints by examining a slightly simpler example. Consider two functions, `f_strict` and `f_perm`, which both have the same body, but different type signatures, with `f_strict` *strictly* enforcing a single lifetime for all its references, and `f_perm` instead *permissively* using additional lifetime constraints to relate its parameters:

```rust
fn f_strict<'a>(x: &'a mut i32, y: &'a i32) -> &'a i32 { *x = *y; y }
fn f_perm<'a, 'b, 'c>(x: &'a mut i32, y: &'b i32) -> &'c i32 where 'b: 'a, 'b: 'c { *x = *y; y }
```

The function `f_perm` allows for a super-set of all callers of `f_strict` by assigning all references distinct lifetimes and then constraining these lifetimes such that `'b` lives at least as long as `'a` and `'b` lives at least as long as `'c`. As an example of how this can affect the caller, consider the code in Fig. 14 where we have a variable `x` that lives within an inner scope and is dropped when it ends while `y` and `z` live in the outer scope and outlive `x`. Using `f_perm`, the borrow checker simply requires that the value that `y` references will live at least as long as `x` and `z`—which is respected. However, when we replace `f_perm` with `f_strict` the borrow checker raises an error that `x` does not live as long as its borrow `z`. There is no particular reason why this caller should not be allowed as `z` is always a reference to whatever `y` borrows. In this way, the typing discipline adopted by REM produces code that is more reusable.

```rust
fn foobar() { // outer scope
  let y = &0; let z;
  { // inner scope
    let mut x = 1; z = f_perm(&mut x, y);
  } // end inner scope
  println!("{}", z) // end outer scope
}
```

Fig. 14. A discriminating caller

Finally, of the 40 experiments we considered, there were 3 refactorings that REM could not handle, and refused to operate to avoid producing ill-typed extracted code. The reason for these failures

were the same and related to handling more complex features of Rust, such as generics with trait constraints, unrelated to borrowing that were not supported by IntelliJ's Rust plugin. For example, REM refused to perform a refactoring for Experiment #7 from the `petgraph` library, because IntelliJ was unable to infer appropriate trait bounds for the generic parameter `G` in code making use of the `Dot` data type: `impl<'a, G> Dot<'a, G> where G: A + B + C + D`. While we could apply the same repairs we did for lifetime here (Sec. 3.3), without further constraint collection and reliable type inference this is also non-trivial. Instead, in such cases, REM informs the user that it failed to perform the extraction and offers to restore the code back to its original state—this was not the case with IntelliJ's Rust plugin or Rust Analyzer, which would both silently extract ill-typed code.

4.2.2 RQ2: Efficiency. The last column of Tab. 1 lists time taken by REM to perform a refactoring for each experiment—as can be seen from the results, REM is quite efficient: it is able to complete all refactorings in less than 2.5 seconds, with most refactorings being done within 1.5 seconds, and an average repair time across all experiments being around 1 second. If these refactorings were to be done manually, they would probably take much longer since user studies show that a manual Extract Method refactoring takes 35 seconds *on average* for mainstream languages with less complex type systems such as Java (Negara et al. 2013).

The efficiency of REM may seem surprising, especially considering that we invoke the compiler during each repair iteration—the reason for this relates primarily to our choice to invoke cargo check rather than cargo build. Though compiling a Rust project, as done by cargo build, is typically an expensive operation, we have found that a large portion of this cost actually arises from the process of simply emitting and linking the final binary. In contrast, if we only run the type-checking and borrow-checking phases of the compiler, as is done by cargo check, then the execution times of the compiler are short enough to be used in the loop as part of an IDE plugin. Furthermore, since the changes of the extract function refactoring are always localised to a single file, Rust's incremental compilation techniques can further amortise the cost of compilation by caching results for unchanged files, leading to shorter refactoring times, even for very large projects.

Another interesting observation from our table is that the size of the source code has minimal effect on REM's performance and that the main determinant of refactoring time for REM is primarily the number of calls to cargo check (listed in the seventh column of Tab. 1). In particular, within our experiments we consider projects of varying sizes, between 300 and 20,000 LOC of Rust, however, our results find very little difference in the performance of REM across these projects, with no clear trend between the size of the project and the time taken for the refactoring. Even looking at the sizes of the functions being extracted from or into, there is no clear relation between source or extracted code size and duration of the refactoring. In contrast, while almost all extractions complete within 2 sec, the only one that takes longer, #19, also requires the largest number of iterations of cargo check to complete. This extraction, which was discussed in Sec. 4.2.1, involves several lifetimes and takes 3 cycles to repair, as REM must refine its lifetimes constraints several times before it obtains a valid type signature for the extracted function. Each call to cargo check invokes the compiler, re-analysing the modified file, and thereby incurs relatively higher cost. Hence, the number of repair cycles produces the most impact on REM's extraction time.

4.2.3 RQ3: Comparisons. The sixth column of Tab. 1 presents a comparison of REM against the other main Rust refactoring tools available in the ecosystem: IntelliJ's Rust plugin and Visual Studio Code's Rust Analyzer plugin. As can be seen from the table, our tool surpasses the state of the art: REM is able to handle most experiments and only fails to handle 3 cases, while all other tools fail on a significantly larger fraction of experiments (IntelliJ's Rust plugin produces 19 ill-typed refactorings out of 40, while Visual Studio Code's Rust Analyzer plugin produces 16 ill-typed refactorings and fails to handle 2 refactorings). We do not record the times taken for the other tools

in our evaluation as they all completed near instantaneously on all experiments ($\leq 0.1s$), however, while our tool is slower than the original plugin by IntelliJ and Rust Analyzer, we argue that as REM completes within a few seconds, *i.e.*, is not unreasonably large, the marginal increase in refactoring time is offset by the benefit of being able to automatically refactor a larger class of programs, and thereby saving more time than annotating all the lifetimes and borrows manually.

The code produced by REM is idiomatic and mostly of comparable quality to the other state of the art refactoring tools. We observed that the most significant differences in REM's output primarily related to its use of lifetime parameters, wherein REM prioritises assigning the most *permissive* signature to its extractions. Indeed, while REM can emit code with excessive parameterisation in extreme cases (*cf.* Sec. 4.2.1), we found that most Rust programs in our experiments did not experience similar blow-ups as their code only involved a few lifetimes at any single point. Furthermore, while other tools would sometimes produce more concise code by eliding lifetimes, we found that the implementation of such elision is often ad-hoc and best-effort, sometimes even causing the refactoring itself to fail. One example of this occurred in Experiment #4 from `petgraph`, where VSCode extracted a function that elided all lifetimes: `fn helper<EF>(&self, f: &mut T, edge_fmt: &EF, g: G)`. While Rust's lifetime elision rules would normally assign unique lifetimes to all arguments for such references, the use of `&self` causes Rust to adopt a different heuristic and instead assign the *same* lifetime to all references, thus causing the resulting code to fail to borrow-check. The IntelliJ Rust plugin happens to have heuristics for this rule, and so manages to produce a well-typed program for this particular case. In contrast, REM avoids the problem entirely due to its prioritisation of permissive signatures. Overall, we find REM to be robust and applicable in a diversity of contexts.

For the few cases where REM is unable to perform a refactoring, generally both other refactoring tools also struggle to execute correctly. As mentioned in the previous subsection, REM refused to operate on certain programs in our experiments because they made use of language features that were not well supported by IntelliJ Rust plugin, such as generic trait bounds—surprisingly, while VSCode's Rust Analyzer plugin uses an entirely different implementation, it also faces difficulties refactoring these programs and often produces ill-typed code for them. The one apparent counter-example to this trend is Experiment #32 where only VSCode is able to extract a well-typed function, however, interestingly, this discrepancy arises for an orthogonal reason. This program in Experiment #32 happens to be challenging because it makes use of Rust's "struct punning" language feature–using the variable name as a struct's field name—and as this feature is usually overlooked in Rust analysis tools, it results in extracted programs failing to type-check (most tools, including IntelliJ that REM is built upon, produce `A {*x}` while `A {x: *x}` is required). In this case, VSCode avoids the issue because its analysis detects that the parameter being punned is of a type that can be copied (in Rust parlance, it implements the `Copy` trait), so it instead produces an extracted function that takes full ownership of the parameter rather than a borrow. We observed that if we removed the `Copy` trait on this parameter, then VSCode's result becomes ill-typed for the same reason.

5 RELATED WORK

A Brief History of Automated Refactoring Tools. The notion of refactoring traces back to Opdyke's PhD thesis (1991) that investigated automated techniques for software maintenance in object-oriented programming (OOP), and has been followed by a large body of research on this topic.

Refactoring in OOP is primarily challenging because of the class hierarchy, which complicates tasks such as name resolution. Building on Opdyke's work, Roberts (1999) developed a strategy to optimise the performance of refactoring for Smalltalk programs. Kniesel and Koch (2004) decomposed refactorings into sub-components and introduced the idea of conditional transformations that may not be behaviour-preserving, but produce correct program transformation when composed

together. Juillerat and Hirsbrunner (2007) observed that tracking only simple properties is sufficient to implement a correct extract method for many common patterns in Java code.

Another line of works investigated how these common refactorings could be applied in functional languages. In his PhD thesis, Griswold (1992) considered refactorings for Scheme, and constructed program dependence graphs as a lightweight analysis to ensure that refactorings preserve program behaviour. Li et al. (2005) reported on their development of a refactoring engine for Haskell. Subsequent works have considered refactorings for other modern functional languages such as Scala (Sergey et al. 2010), Erlang (Li and Thompson 2012) and OCaml (Rowe et al. 2019).

While there are non-trivial challenges by the various language features considered by these prior works, *e.g.*, dynamic binding, implicit arguments, type classes *etc*, their approaches are not amenable to handling the unique ownership and lifetime constraints imposed by Rust's type system.

Refactoring Tools for Rust. Owing to the relative youth of the language, Rust's tooling ecosystem still remains in a nascent stage, however some initial work has been done on building automated refactoring tools for it (Emre et al. 2021; Ringdal 2020; Sam et al. 2017).

Sam et al. (2017) were the first to investigate refactoring for Rust programs, where they constructed an automated algorithm to handle simple transformations such as renaming variables or functions, inlining expressions or adding or removing lifetime parameters; refactorings such as extract method were not investigated. More relevant to our work, in his masters thesis, Ringdal (2020) reports on the development of a refactoring tool that partially supports extract method and unbox field transformations. While Ringdal's extract method can handle non-local control-flow, it explicitly does not incorporate lifetime information in its analysis, and cannot handle cases where lifetime annotations are required. Furthermore, Ringdal's implementation only tackles a restricted subset of Rust, disallowing generics in refactored code; it also suffers from a large variation in runtimes, taking 7 seconds on average to perform a refactoring when operating on larger projects.

Moving beyond standard refactorings, recent works considered more bespoke program transformations that are unique to Rust ecosystem. Most notably, the C2Rust (Immunant Inc. 2022) program provides an automated tool that translates C to unsafe Rust, and has served as the basis of a number of follow-up works. Emre et al. (2021) developed a tool that automatically translates code written in the unsafe fragment of Rust to equivalent versions using only safe operations, using the error messages emitted by the compiler to repair lifetime constraints much like our tool.

Constraint-Based Refactoring. Constraint solving was first brought to refactoring research through works that investigated typing-related transformations, where the refactoring process could be framed as solving a set of type constraints (Tip et al. 2003). Later, Tip (2007), observed that the transformations considered in Opdyke's work were also primarily related to types, and thereby presented a constraint-based implementation of those operations. Subsequent works investigated using typing-related transformations for substituting classes with alternative implementations (Balaban et al. 2005), introducing type parameters for mutually recursive types (Kiezun et al. 2007), replacing inheritance for delegation (Kegel and Steimann 2008) and moving methods and variables between classes (Steimann and Thies 2009). These works were primarily concerned with handling the nuances of the complex sub-typing relations introduced by object-oriented type systems, and did not consider more complex typing disciplines such as the linear type system used in Rust.

Most recently, Steimann (2018) presented a generic framework for reasoning about constraint-based refactoring, framing refactoring as a constraint-based repair much like we do. Steimann's approach requires encoding the entire refactoring operation as a constraint problem and "repairing" the constraint set to determine a transformation, which would require re-implementing existing refactorings in this exotic format, whereas our repair procedure operates on almost correct refactoring results and transparently serves to enhance an existing baseline refactoring tool.

```
1  struct Point {
2    x: i32,
3    y: i32,
4  }
5
6
7  fn foo() {
8    let mut p = Point { x: 0, y: 0, };
9    p.x = p.x + 10;
10   p.y = p.y + 10;
11   println!("The new position is ({},{})",
12                 p.x,p.y);
13 }
```

(a) Original code

```
1  fn foo() {
2    let mut p = Point { x: 0, y: 0, };
3    p.move_pt(10,10);
4  }
5
6  impl Point{
7    fn move_pt(self: &mut Point, x: i32, y: i32) {
8      self.x = self.x + x;
9      self.y = self.y + y;
10     println!("The new position is ({},{})",
11                   self.x,self.y);
12   }
13 }
```

(b) Extracting as a method of the `Point` struct

Fig. 15. Towards more idiomatic refactoring with data structures

Ownership Types for Object Confinement. Rust's ownership model draws inspiration from prior work on object capabilities (Boyland et al. 2001) having a strong correspondence to ownership types (Boyapati et al. 2002; Clarke and Drossopoulou 2002; Clarke et al. 2013, 1998; Sergey and Clarke 2012). For example, mutable borrows in Rust are a similar concept to that of so-called external uniqueness enforced via a form of onwership types (Clarke and Wrigstad 2003). Our work exploits the similarities with prior type systems by building an analysis for ownership inference (Sec. 3.2) that is reminiscent to the inference of ownership types (Huang et al. 2012b; Ma and Foster 2007; Vakilian et al. 2009). In particular, inferring ownership types through a combination of points-to analysis and constraint solving has been studied in the work of Huang et al. (2012a) that required the programmer to provide partial annotations to guide the inference towards "best typing". Automated refactoring does not need any extra annotations beyond what's already available in the program; instead, our approach strips the parameters in the extracted method of the ownership information and uses the solutions to the constraints collected from the extracted code fragment to ascribe valid ownership annotations.

6 DISCUSSION

In this work, we observed that phrasing ownership and lifetimes as two *independent aspects* of Rust type system allows for designing a refactoring that is compositional in its implementation and that produces code of good quality. The compositionality of our approach manifests in the ability of its components to be replaced or extended with alternative inference procedures without affecting the rest of its pipeline. For example, one can further extend our ownership analysis (Sec. 3.2) to support complex features, such as generics with traits constraints. While our current implementation exercises a pragmatic approach to lifetime annotation inference by relying on rustc (Sec. 3.3), a more principled methodology could use tools such as Polonius,[12] thus making the refactoring more robust by removing its dependency from the compiler's error messages.

Another avenue for improvement is towards producing more idiomatic Rust code, possibly with additional hints from a user. For instance, when extracting a code fragment that manipulates the payload of one specific data structure, *e.g.*, `Point` in lines 8–11 of Fig. 15a, a reasonable scenario would be to hoist it into a separate **`impl`**emenetation for that data type rather than a standalone application-specific function, *e.g.*, `move_pt` in Fig. 15b. Further enhancement would be to make use of `&mut self` as a parameter when dealing with mutating a data structure via an extracted method

[12] https://github.com/rust-lang/polonius

instead of passing its fields one by one. An additional challenge would be to redefine the structs to use generic type parameters for its fields and corresponding function implementations.

By and large, the space of such improvements is too vast for us to predict all of them at this stage. From our past industrial experience we envision these changes to be more likely introduced in response to concrete feature requests coming from the users of the tool.

7 CONCLUSION

Designing an efficient and effective automated refactoring is a balancing act between "pessimistic" and "optimistic" approaches. The former approach is more common in mainstream IDEs and works well for languages with relatively "simple" type systems, such as that of Java. It achieves "best-effort" soundness via static analyses run *prior* to the code modification. The latter approach is less explored in practice and recasts refactoring as a *repair* procedure. We found it more viable for a language with a "complex" type system (*i.e.*, Rust), as it allows for a conceptually simpler refactoring formulation and implementation by making extensive use of existing checkers as repair oracles.

We believe that the presented Extract Method design provides an informative case study in combining these two approaches and paves the way to bringing more results from the decades-long research in tooling support for software development to the rapidly growing Rust ecosystem.

ACKNOWLEDGMENTS

We thank Vitaly Bragilevsky, Matthew Flatt, Son Ho, and Alex Potanin for their feedback on drafts of this paper. We also thank the OOPSLA'23 PC and AEC reviewers for their constructive and insightful comments. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001.

DATA AVAILABILITY

The software artefact accompanying this paper is available online (Thy et al. 2023). It contains the source code and the case studies for REM that can be used to reproduce the experimental results described in Sec. 4, a self-contained Docker file to automate setting up the development environment, and a `README` file that provides detailed step-by-step instructions.

REFERENCES

Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. University of Copenhagen, Denmark.

Isaiah Ayooluwa, Ledru Sylvestre, and Ivy Roy. 2020. Rewriting the GNU Coreutils in Rust. https://lwn.net/Articles/857599/

Ittai Balaban, Frank Tip, and Robert M. Fuhrer. 2005. Refactoring support for class library migration. In *OOPSLA*. ACM, 265–279. https://doi.org/10.1145/1094811.1094832

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*. ACM, 211–230. https://doi.org/10.1145/582419.582440

John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In *ECOOP (LNCS, Vol. 2072)*. Springer, 2–27. https://doi.org/10.1007/3-540-45337-7_2

Dave Clarke and Sophia Drossopoulou. 2002. Ownership, Encapsulation and the Disjointness of Type and Effect. In *OOPSLA*. ACM, 292–310. https://doi.org/10.1145/582419.582447

Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. LNCS, Vol. 7850. Springer, 15–58. https://doi.org/10.1007/978-3-642-36946-9_3

Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness Is Unique Enough. In *ECOOP (LNCS, Vol. 2743)*. Springer, 176–200. https://doi.org/978-3-540-45070-2_9

David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*. ACM, 48–64. https://doi.org/10.1145/286936.286947

Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating C to safer Rust. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29. https://doi.org/10.1145/3485498

Martin Fowler and Kent Beck. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.
Robert M. Fuhrer, Frank Tip, and Adam Kiezun. 2004. Advanced Refactorings in Eclipse. In *OOPSLA Companion*. ACM, 8. https://doi.org/10.1145/1028664.1028669
William G. Griswold. 1992. *Program Restructuring as an Aid to Software Maintenance.* Ph. D. Dissertation. University of Washington, USA.
Wei Huang, Werner Dietl, Ana L. Milanova, and Michael D. Ernst. 2012a. Inference and Checking of Object Ownership. In *ECOOP (LNCS, Vol. 7313)*. Springer, 181–206. https://doi.org/10.1007/978-3-642-31057-7_9
Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. 2012b. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *OOPSLA*. ACM, 879–896. https://doi.org/10.1145/2398857.2384680
Immunant Inc. 2022. c2rust. Available at https://c2rust.com.
Dmitry Jemerov. 2008. Implementing refactorings in IntelliJ IDEA. In *Second ACM Workshop on Refactoring Tools (WRT)*. ACM, 13. https://doi.org/10.1145/1636642.1636655
Nicolas Juillerat and Béat Hirsbrunner. 2007. Improving Method Extraction: A Novel Approach to Data Flow Analysis Using Boolean Flags and Expressions. In *1st Workshop on Refactoring Tools (WRT)*. 48–49.
Hannes Kegel and Friedrich Steimann. 2008. Systematically refactoring inheritance to delegation in java. In *ICSE*. ACM, 431–440. https://doi.org/10.1145/1368088.1368147
Adam Kiezun, Michael D. Ernst, Frank Tip, and Robert M. Fuhrer. 2007. Refactoring for Parameterizing Java Classes. In *ICSE*. IEEE Computer Society, 437–446. https://doi.org/10.1109/ICSE.2007.70
Günter Kniesel and Helge Koch. 2004. Static composition of refactorings. *Sci. Comput. Program.* 52 (2004), 9–51. https://doi.org/10.1016/j.scico.2004.03.002
Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. 2021. Automatic Program Repair. *IEEE Softw.* 38, 4 (2021), 22–27. https://doi.org/10.1109/MS.2021.3072577
Huiqing Li and Simon J. Thompson. 2012. A Domain-Specific Language for Scripting Refactorings in Erlang. In *FASE (LNCS, Vol. 7212)*. Springer, 501–515. https://doi.org/10.1007/978-3-642-28872-2_34
Huiqing Li, Simon J. Thompson, and Claus Reinke. 2005. The Haskell Refactorer, HaRe, and its API. In *Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA) (ENTCS, Vol. 141)*. Elsevier, 29–34. https://doi.org/10.1016/j.entcs.2005.02.053
Kin-Keung Ma and Jeffrey S. Foster. 2007. Inferring aliasing and encapsulation properties for java. In *OOPSLA*. ACM, 423–440. https://doi.org/10.1145/1297027.1297059
Nicholas D. Matsakis. 2017a. The Rust RFC Book: Non-Lexical Lifetimes. https://rust-lang.github.io/rfcs/2094-nll.html
Nicholas D. Matsakis. 2017b. The Rust RFC Book: Statements and expressions. https://doc.rust-lang.org/reference/statements-and-expressions.html
Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *IEEE Trans. Software Eng.* 30, 2 (2004), 126–139. https://doi.org/10.1109/TSE.2004.1265817
Microsoft. 2023. Refactoring in Visual Studio Code. https://code.visualstudio.com/docs/editor/refactoring.
Emerson R. Murphy-Hill and Andrew P. Black. 2008. Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method. In *ICSE*. ACM, 421–430. https://doi.org/10.1145/1368088.1368146
Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A Comparative Study of Manual and Automated Refactorings. In *ECOOP (LNCS)*. Springer, 552–576. https://doi.org/978-3-642-39038-8_23
William F. Opdyke. 1991. *Refactoring object-oriented frameworks.* Ph. D. Dissertation. University of Illinois at Urbana-Champaign, USA.
Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *PLDI*. 763–779. https://doi.org/10.1145/3385412.3386036
Per Ove Ringdal. 2020. *Automated Refactoring of Rust Programs.* Master's thesis. University of Oslo.
Don Roberts, John Brant, and Ralph E. Johnson. 1997. A Refactoring Tool for Smalltalk. *Theory Pract. Object Syst.* 3, 4 (1997), 253–263.
Donald Bradley Roberts. 1999. *Practical Analysis for Refactoring.* Ph. D. Dissertation. University of Illinois at Urbana-Champaign, USA.
Reuben N. S. Rowe, Hugo Férée, Simon J. Thompson, and Scott Owens. 2019. Characterising renaming within OCaml's module system: theory and implementation. In *PLDI*. ACM, 950–965. https://doi.org/10.1145/3314221.3314600
The Rust Reference. 2023. Lifetime Elision. https://doc.rust-lang.org/reference/lifetime-elision.html.
The Rust Team. 2017. The Rust programming language. http://rust-lang.org.
Garming Sam, Nick Cameron, and Alex Potanin. 2017. Automated Refactoring of Rust Programs. In *Proceedings of the Australasian Computer Science Week Multiconference, ACSW*. ACM, 14:1–14:9. https://doi.org/10.1145/3014812.3014826
Max Schäfer. 2010. *Specification, Implementation and Verification of Refactorings.* Ph. D. Dissertation. University of Oxford, UK.

Max Schäfer and Oege de Moor. 2010. Specifying and implementing refactorings. In *OOPSLA*. ACM, 286–301. https://doi.org/10.1145/1869459.1869485

Ilya Sergey and Dave Clarke. 2012. Gradual Ownership Types. In *ESOP (LNCS, Vol. 7211)*. Springer, 579–599. https://doi.org/10.1007/978-3-642-28869-2_29

Ilya Sergey, Dave Clarke, and Alexander Podkhalyuzin. 2010. *Automatic refactorings for Scala programs*. CW Reports CW 577. KU Leuven. http://lirias.kuleuven.be/1652327.

Friedrich Steimann. 2018. Constraint-Based Refactoring. *ACM Trans. Program. Lang. Syst.* 40, 1 (2018), 2:1–2:40. https://doi.org/10.1145/3156016

Friedrich Steimann and Andreas Thies. 2009. From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility. In *ECOOP (LNCS, Vol. 5653)*. Springer, 419–443. https://doi.org/10.1007/978-3-642-03013-0_19

Sewen Thy, Andreea Costea, Kiran Gopinathan, and Ilya Sergey. 2023. *Reproduction Artefact for Article "Adventure of a Lifetime: Extract Method Refactoring for Rust"*. https://doi.org/10.5281/zenodo.8124395

Frank Tip. 2007. Refactoring Using Type Constraints. In *SAS (LNCS, Vol. 4634)*. Springer, 1–17. https://doi.org/10.1007/978-3-540-74061-2_1

Frank Tip, Adam Kiezun, and Dirk Bäumer. 2003. Refactoring for generalization using type constraints. In *OOPSLA*. ACM, 13–26. https://doi.org/10.1145/949305.949308

Mohsen Vakilian, Danny Dig, Robert Bocchino, Jeffrey Overbey, Vikram Adve, and Ralph Johnson. 2009. Inferring Method Effect Summaries for Nested Heap Regions. In *ASE*. IEEE Computer Society, 421–432. https://doi.org/10.1109/ASE.2009.68

Received 2023-04-14; accepted 2023-08-27