Automated Refactoring of Rust Programs

Garming Sam
Catalyst IT, NZ
garming@catalyst.net.nz

Nick Cameron
Mozilla Research, NZ
ncameron@mozilla.com

Alex Potanin
VUW, NZ
alex@ecs.vuw.ac.nz

ABSTRACT

Rust is a modern systems programming language developed by Mozilla Research and the Rust community. Rust supports modern constructs such as ownership, lifetimes, traits, and macros, whilst supporting systems programming idioms with low-cost abstractions and memory safety without garbage collection.

We describe a new refactoring tool for Rust developers, including discussing the issues and unusual decisions encountered due to the complexities of modern systems languages. We outline lessons learned and hope our paper will help inform design of future programming languages and refactoring tools. The resulting refactoring tool is written in Rust and available from Github under an MIT license [8].

1. INTRODUCTION

Rust is developed by Mozilla Research and a very active, open source community; it reached the 1.0 milestone in May 2015. As a modern, memory-safe systems programming language, the aim for the language is to provide reliable and efficient systems by combining the performance of low level control, with the convenience and guarantees of higher level constructs. All of this is achieved without a garbage collector or runtime and allows very low-overhead interoperability with C programs. Rust enforces an ownership system to restrict the duplication of references through borrowing and lifetimes. Using these techniques, Rust prevents dangling pointers, iterator invalidation problems, concurrent data races, and other memory safety issues.

Refactoring is the act of performing functionality preserving code transformation. Traditionally, these transformations needed to be performed manually, but in recent years, a number of tools to aid and automate refactorings have arisen in many programming languages such as Java and C++ [1]. Manual transformations, including editor search-and-replace are potentially prone to error, and so performing tool-assisted refactoring (which guarantee some measure of correctness) can provide much greater confidence in changes.

ACSW '17, January 31-February 03, 2017, Geelong, Australia

ISBN 978-1-4503-4768-6/17/01…$15.00

DOI: http://dx.doi.org/10.1145/3014812.3014826

The main purpose of this work is to produce a proof-of-concept refactoring tool which utilises existing infrastructure made available by the Rust compiler.

Our refactoring contributions are as follows: (1) Renaming local and global variables, function arguments, fields, functions and methods, structs, and enumerations; (2) inlining local variables; (3) lifetime parameter reification and elision. In particular, the third item was not implemented in any other language until this point. Finally, we outline the challenges involved and evaluate our tool.

2. BACKGROUND

2.1 Rust

The following is a Hello World program with a few additions:

```
fn main() {
  let a = "world"; // variable declaration
  let b = &a;      // 'borrow' or reference to a
  println!("Hello, {}!", b); // println! is a macro
}
```

In Rust [14], variable bindings have *ownership* of whatever they are bound to. When an owning variable moves out of scope, any resources (including those on the heap) can be freed and manual cleanup is unnecessary. By default, Rust has move semantics where by assigning the value of one variable to another, the object is moved to the new variable and the old variable cannot be used to reference this object anymore. Rust supports *borrowing* references, where an object is borrowed instead of moved. Borrows may be either multiple or mutable, never both.

A borrow must not exceed the *lifetime* of owner. By adding these restrictions, memory can usually only be modified by one place and it allows compile-time abstractions (without runtime penalty) to ensure memory safety.

This system can add complexity during coding; when returning references for instance, the compiler might need additional help to infer the lifetimes of the different parameters and returns. In order to do so, functions (and types) can be parameterised by lifetime variables, in a similar manner to type varibles.

```
fn foo<'a>(x: & 'a Debug)
```

Many uses of lifetime parameters follow common and simple idioms. To improve ergonomics, Rust allows many actual lifetime parameters to be elided. These elision rules cover around 87% of uses in the standard library [13]. The rules are described as follows: Each elided lifetime in input position

becomes a distinct lifetime parameter. If there is exactly one input lifetime position (elided or not), that lifetime is assigned to all elided output lifetimes. If there are multiple input lifetime positions, but one of them is &self or &mut self, the lifetime is assigned to all elided output lifetimes. Otherwise, it is an error to elide an output lifetime. Reification of lifetimes refers to performing the opposite of elision, i.e. reintroducing a lifetime parameter where one did not exist previously.

2.2 Refactoring

Martin Fowler's definition in 1999 [4] defines refactoring as the following: *"Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure."* Bill Opdyke in Refactoring Object-Orientated Frameworks defined behaviour preservation in terms of seven properties [6]. Although taken from a C++ perspective, the definition continues to be used more widely [9].

1. Unique superclass – After refactoring, a class should have at most one direct superclass, which is not one of its subclasses.
2. Distinct class names – After refactoring, each class name should be unique.
3. Distinct member names – After refactoring, all member variables and functions within a class have distinct names.
4. Inherited member variables not redefined – After refactoring, an inherited member variable from a superclass is not redefined in any subclass.
5. Compatible signature in member function redefinition – After refactoring, if a member function in a superclass is redefined in a subclass, the two function signatures must match.
6. Type safe assignments – After refactoring, the type of each expression assigned to a variable must be an instance of the variable's defined type.
7. Semantically equivalent references and operations.

The first six properties can be verified by a single run of the compiler correctly succeeding; however the seventh property cannot be ensured with the same action. Essentially, the last property is to ensure that two runs of a program with the same inputs will always produce the exact same outputs as the original, unchanged program. There are a number of very simple cases where compilation may succeed while the functionality of a program has changed, one major cause of which is shadowing.

```
let a = 2;          let b = 2;
let b = 4;          let b = 4;
let c = a + b;      let c = b + b;
```

Figure 1: Renaming local variable *a* to *b*

Figure 1 describes an example in Rust. In the original code `c` evaluates to 6, but in the renamed case, after renaming `a` to `b`, `c` evaluates to 8 due to shadowing. The situation here is slightly unique with local variables since Rust allows variable names to be re-used, unlike other languages like Java. Shadowing is worrying from a programmer's perspective since it easily allows a program to compile, but produce incorrect behaviour.

One of the first key ideas that the classical Fowler [4] book asserts is that before refactoring occurs, a solid suite of tests, particularly unit tests should be present to ensure that functionality is never modified.

The book describes three key details in providing a practical refactoring tool, using the Smalltalk Refactoring Browser – one of the earliest and most comprehensive refactoring tools – as a guideline. The first is speed: If it takes too long, a programmer would likely prefer to just perform a refactoring by hand and accept the potential for error. The second is the ability to undo: Refactoring should be exploratory, incremental and reversible assuming it is behaviour preserving. The last is tool integration: with an IDE that groups all the tools necessary for editing, compilation, linking, debugging etc., development is more seamless and reduces the friction in adopting additional tools into a workflow.

3. REFACTORING RUST

3.1 Renaming

Performing renaming without any of the necessary checks is not a particularly difficult task. What should be considered when performing an accurate refactoring is the potential to change behaviour and cause conflicts. Fundamentally, there are three different conflict types that occur with lexically scoped items.

Super-block conflicts occur when a new name coincides with one declared in an outer enclosing block. In this situation, any references to the name in the outer block could be shadowed by the new name.

```
let a = 1;                    let a = 1;
let b = 2;                    let b = 2;
{                             {
  let a = 3;                    let b = 3;
  println!("{}", b); // 2      println!("{}", b); // 3
}                             }
```

Figure 2: Super-block conflict: Renaming block local a to shadow outer b

Sub-block conflicts occur when a new name coincides with one declared in an inner sub-block. In this situation, any references to the name in the outer block when changed to the new name might be shadowed by the existing declaration in the sub-block.

```
let b = 1;                    let a = 1;
{                             {
  let a = 2;                    let a = 2;
  println!("{}", b); // 1      println!("{}", a); // 2
}                             }
```

Figure 3: Sub-block conflict: Renaming outer b forces block local a to shadow outer a

In other languages, *same-block conflict* occurs with local variables which appear in the same scope. However, let

bindings in Rust allow redeclaration of variables in the same scope. Conflicts among redeclared variables can be regarded as super- or sub-block conflicts, where the block is implicit. While same-block conflicts cannot occur in Rust for local variables, they can still occur with global variables, fields, types, etc.

```
const a: i32 = 1;          const a: i32 = 1;
const b: i32 = 2;          const a: i32 = 2;
```

Figure 4: Same-block conflict: Renaming b to conflict with a in the same scope

When performing a renaming, there are two main operations that need to be performed:

- Finding all accesses of a declaration
- Finding the declaration of an access

in the general case, the compiler has to be run again to find this information. For a refactoring to succeed, all names in a refactored program must bind to the same declaration as the original program [9]. All original uses should be updated to bind to the renamed declaration and any other usages binding to a different declaration, remain binded to a different declaration.

3.2 Inlining

Of the available literature, it appears that the authors of the JRRT [9] describe the act of inlining a variable in the most specific detail. At the time, they also note the existing scarcity of in-depth documentation for specific refactorings. Working with Java in particular, they note that due to the limitations of Java, it is impossible to absolutely ensure 100% correctness under even common circumstances. In this section, a description as best as possible within the context of Rust will be shown and how despite promising additional guarantees such as mutability, absolute correctness is still quite out of reach.

We limit our analysis to safe Rust code (c.f., code explicitly marked as unsafe) and assume that there is at least one use of the variable being inlined.

There are a number of factors to be considered when inlining a variable. The first is the purity of any function calls in the composing expression. The second is the mutability of the local variable to inline. The third is the number of usages of the local variable. The last is whether or not any identifiers used to initialise the variable now refer to something else.

1. Check the initialising expression for the variable. If there are any non-pure function calls, abort the operation.
2. If the initialising expression has any references to mutable memory, abort.
3. If the variable is only used once and never used as a left-hand side, skip to step 6.
4. If the variable is declared 'mut' and the 'mut' declaration was required, abort.
5. If the variable has interior mutability, abort.
6. Visit each usage of the local variable, replacing the variable but also checking that any identifiers used in the initialising expression refer to the same variables. If not, abort.
7. Remove the declaration of the local variable.

This algorithm is conservative: some valid refactorings may fail. Our first point of interest is the requirement for pure function calls which have no side effects. Although it appears to be a reasonable requirement, the function actually need only be conditionally pure for the code section of interest for the inline. This appears to be a very difficult analysis, when even regular purity cannot be predicted in Rust. Much like the case in Java for JRRT [9], the issue of identification of these functions cannot be solved in Rust. Pure functions were part of the language definition earlier on in the development of Rust but due to difficulty in producing an exact definition, they were abandoned [15]. In Figure 5, we can see how the inlining of a database call which might insert a single record will be repeated if it is inlined.

```
let a = insert_into_db(); // After inlining a
println!("{}", a);        println!("{}", insert_into_db());
println!("{}", a);        println!("{}", insert_into_db());
```

Figure 5: Functions violating behaviour preservation with inline local

For Step 3, if there is exactly one usage of a local variable in an inline, then due to uniqueness constraints in Rust, there really is just a single usage without any aliases. This is unlike C++ for example, where some other pointer could still refer to the same section of memory. The check for the left-hand side is to ensure that the variable was not being assigned some value. In general, mutating the value of a local variable that is about to be inlined is invalid since the inline converts a single long-lived state into transient ones. This reasoning applies exactly the same for steps 4 and 5, noting that interior mutability should be considered unsafe. The interior mutability may be unused and so, this is somewhat conservative.

```
let b = 1;              let b = 1;
let a = 2 + b;          // a has been inlined
let b = 4;              let b = 4;
println!("{}", a);      println!("{}", 2 + b);
```

Figure 6: Inlining changes behaviour: Prints 6 instead of 3

Step 6 makes sure that if any variable composing the initializing expression has been redeclared with a new let binding, then the inline should not work. Rust is special here since it allows redeclaration of variables with the same names. Looking at Figure 6 we can see how the inline of the variable `a` is incorrect due to the fact that `b` has been redeclared in the meantime. Now, this step is actually a superficial version of Step 2 which queries the 'inner' mutability of the memory referred to by the variable. We find that the identification of mutable parts of an expression (Step 2) is practically impossible given the current Rust compiler implementation. It is unknown if compiler work alone would be sufficient

to remedy this issue or language tweaks would be required unless the actual work was carried out. In particular an 'effect' system [7], or some form of recursive analysis of origin of memory appears to be required, but this is outside of the scope of this work.

Finally, in Figure 7 we can see the inlining of a vector. The problem with the resulting code is that despite calling `iter()` on the inlined vector, the vector should be disposed. As a local variable, a valid borrow normally occurs, but without it, the iterator has no proper parent and causes a violation of lifetimes. Besides running through compilation (analysis) again, it is unclear how this case should be handled or if they can be resolved in a simpler way. As such, no further considerations are made.

```
let v = vec![1, 2];   // v has been inlined
// i is an iterator   // i is an iterator
let i = v.iter();     let i = vec![1, 2].iter();
```

Figure 7: Inlining causes compilation error: borrowed value does not live long enough

3.3 Lifetime elision and reification

Although the concepts of lifetimes and ownership are not trivial, the effect of reification and elision is actually quite simple and relatively easy to understand. In Figure 8, we can see input lifetimes marked in red or green for a number of function declarations. Green lifetimes belong to the self parameter (much like Python for object orientation or 'this' in Java). Output lifetimes are marked in blue which appear in the return type. The elision rules in Rust essentially describe which lifetime will be inferred if you elide a lifetime. They follow common patterns so that in most cases, you will never need to include any lifetime parameters in your function declarations. In the below figure, all lifetimes may be elided.

The rules essentially boil down to the following:

1. Where a type has a formal lifetime parameter, but there is no corresponding actual parameter, a fresh one (using an unbound name) is introduced.
2. If there is one fresh input and one fresh output parameter, they are unified.
3. Otherwise, if there is fresh input parameter on 'self' and a fresh output parameter, they are unified.
4. Otherwise, it is an error to have multiple fresh input parameters and a fresh output parameter.

```
fn foo<'a>(x: & 'a Debug)
fn foo<'a, 'b>(x: & 'a Debug, y: &'b Debug)
fn foo<'a>(x: & 'a Debug) -> &'a Point
fn foo<'a>(& 'a self)
fn foo<'a, 'b, 'c>(&'a self,
                   x: &'b Debug, y: &'c Debug)
fn foo<'a, 'b, 'c>(&'a self,
                   x: &'b Debug,
                   y: &'c Debug) -> &'a Point
```

Figure 8: Examples of lifetime parameters

```
fn foo(x: &Debug)
becomes:
  fn foo<'a>(x: & 'a Debug)

fn foo(x: &Debug, y: &Debug)
becomes:
  fn foo<'a, 'b>(x: & 'a Debug, y: &'b Debug)
```

Figure 9: Examples of rule 1

```
fn foo(x: &Debug) -> &Point
becomes: fn foo<'a>(x: & 'a Debug) -> &'a Point
```

Figure 10: Example of rules 2 and 5

5. All other fresh parameters are left unique.

Now, the idea is to build a tool to annotate these lifetimes where they have been elided (reification) or to remove them where they are unnecessary due to compiler inference (elision). Despite being called the elision rules in the RFC [13], they actually specify exactly what steps to take in order to reify, not elide. The rules describe basically how the compiler performs reification of missing lifetime parameters internally and so all a tool needs to do is follow the rules. In order to build an elide tool, the steps have to be taken in reverse.

3.3.1 Discussion

For reification, we envision a developer who encounters a piece of code involving lifetimes that they wish to change. The lifetimes were originally elided to reduce noise, e.g. so that anybody using a function could more easily grasp its underlying purpose. In modifying the code, the developer now wishes to visualize exactly which lifetimes are in use where, or debug a type error involving lifetimes. The developer could manually reinsert the lifetimes themselves, or they could use a tool for automating the reification of lifetimes.

For elision, we envision a situation where a developer has a piece of code with all the lifetimes specified, where they were either provided from scratch while performing the implementation or by reification (ideally through a tool). The lifetimes make the code more verbose and harder to comprehend, especially to others, and so, the developer wishes to elide as many lifetimes as possible. This could be done manually, but allows the possibility of errors and missed opportunities to remove a lifetime parameter; or they could use a tool to automate the elision of lifetimes.

As you might see, the inclusion of both elision and reification in an automated refactoring tool is quite important since the use of reification might often imply the use of elision. Using the two together in this fashion, they might form a standard workflow and so pursuing these refactorings has been a point of interest.

```
fn foo(&self, x: &Debug) -> &Point
becomes:
fn foo<'a, 'b>(& 'a self, x: &'b Debug) -> &'a Point

fn foo(x: &Debug, y: &Debug) -> &Point
does not compile
```

Figure 11: Examples of rules 3 and 4

4. IMPLEMENTATION

Our refactoring tool relies on the compiler for semantic information about programs to be refactored. The compiler can be requested (with the '`-Zsave-analysis`' flag) to dump information from its analysis of a program. In the cases where we need more refactoring-specific information, we can also use a modified version of the compiler with callbacks to our refactoring tool from the name resolution pass.

4.1 Renaming

To evaluate a potential renaming, our tool starts with the save-analysis information. This allows the tool to identify the declaration of a variable (or other item) and all of its uses (in contrast with a syntactic search, the compiler can differentiate between different bindings with the same name). If the renaming is valid, then this is enough information to perform the rename. To check validity, we must re-run the compiler in order to try and resolve the name name at the declaration site. If the name does resolve, then there would be potential name conflicts. Our check prevents all super- and same-block conflicts. However, it is conservative and some valid renamings are rejected (where the existing name is not in fact used in the program after the declaration).

To guard against sub-block conflicts, we must further try to resolve the new name at every use of the variable being renamed. Only if every attempt at resolution fails can we be sure that the renaming is safe.

Whilst in theory, all these checks could be done with a single pass of the compiler, in practice Rust's name resolution is not flexible enough to check an arbitrary name, it can only check a name from the source text. Furthermore, we could at the time only observe success or failure of name resolution, not the reason why (the compiler has improved considerably since we implemented this tool). That means that to be safe, we must re-compile once for each use of the variable being renamed. This is clearly expensive.

A much better approach (but outside the scope of this project) would be to modify name resolution to allow checking for arbitrary names. This approach is taken by Gorename [12].

4.2 Inlining

Again, inlining starts with the save-analysis data. This data allows finding the number of uses of a variable to be inlined and the mutability of its type. However, this is not enough to complete our analysis. In particular, in Rust objects can have *interior mutability* which is not reflected in that object's type. However, it is tracked by the compiler, so our tool can query this information. We also take account of a mutability annotation being egregious by relying on the compiler identifying such unneccessary annotations. Unfortunately, this requires running the compiler to a late stage of its analysis and thus is fairly time-consuming. Finally, we rely again on name resolution to ensure that any variables which are substituted in still resolve to their original binding.

We perform the actual inlining on the Abstract Syntax Tree (AST). Following that change, we must ensure that the fragment of the AST is properly printed back into the source text. In particular, parentheses may need to be added to ensure the correct ordering of operations due to precedence. See Figure 12 for an example.

```
Input:
fn main() {
    let a = 2 + 1;
    let _ = a * 2;
}

Output:
fn main() { let _ = (2 + 1) * 2; } // rather than 2 + 1 * 2
```

Figure 12: Correct inlining with order of operations

4.3 Lifetime elision and reification

Reification of lifetime parameters was based on the implementation of error reporting for missing lifetimes in the compiler. This was somewhat complicated by the compiler's representation of lifetimes (a combination of explicit binder structures and de Bruijn indices); converting into fresh lifetime variables again required interaction with name resolution (although note that name resolution for lifetimes is a simpler case in the Rust compiler and is handled by its own code).

In contrast, the fundamentals of elision were more complex - there is no help from the compiler here, and we only implemented for very simple cases. However, since we are only removing lifetime parameters from the source code, there is no difficulty with names.

For a problematic example, see Figure 13. Here, 'b can be elided, but 'a cannot because if it were, the compiler would treat x and y's types as having unique lifetimes.

```
fn foo<'a,'b>(x: &'a Debug, y: &'a Debug, z: &'b Debug)
becomes:
fn foo<'a>(x: &'a Debug, y: &'a Debug, z: &Debug)
```

Figure 13: Partial elision – only `'b` removed

5. EVALUATION

5.1 Validity of refactorings

We wrote 85 tests to ensure that the refactoring tool [8] functions as expected. We test that either refactoring aborts or gives the expected output.

5.1.1 Validity of renamings

The test suite tests both the cases where a renaming should occur, and cases where it should not due to the variations in conflict types as outlined in Section 2. Currently there are around 60 tests specifically written to test renaming, spread across the different classes of renaming. In particular, tests try to cause conflicts between the different classes, e.g., variable names with type names. We can examine a generic rename in Figure 14. When running any of the renaming refactorings, name resolution will run to find any super-block conflict. Afterwards, any sub-block conflicts will be caught in a compilation run.

Methods and functions.

There are several tests for renaming methods defined with a trait and/or overridden by an implementing struct. Tests address both static and dynamic dispatch. One known failure

```
let a = 2; // 1. Super-block conflict: caught by name resolution
    {
        let a = 3;
        let a = 4;
        let b = a; // 2. Sub-block conflict: caught by a compilation run
    }
```

Figure 14: Examining a tentative rename in red

```
// Before refactoring
let Point{x, y} = Point{x:1, y:2}
// After refactoring (invalid):
let Point{foo, y} = Point{x:1, y:2}
// Manually user-corrected (valid)
let Point{x:foo, y:y} = Point{x:1, y:2}
```

Figure 15: Invalid rename of x to *foo* which is easily fixed manually

mode of the refactoring tool is when a function (or trait) is declared within a function scope.

Concrete types – structs and enums.

There are tests for both renaming of structs and enums with detection of namespace collisions (which are not in local scopes). The checking of namespace collisions also extend to the usage of 'use' imports which allow a specific namespace to be imported and no need to additionally qualify some names. The renaming of concrete types does not extend to type aliases, although the extent of partial support is unknown.

Examining a particular edge case.

Figure 15 shows an edge case identified during this project. A struct `Point` with fields `x` and `y` can be used to initialize two corresponding local variables `x` and `y`. Without intervention, a local variable renaming might attempt to change `x` to `foo` but `Point` has no corresponding field `foo`.

5.1.2 Examining inline-local

The inline-local refactoring is at proof-of-concept stage. The majority of the work on this refactoring has been in exploring and describing the knowledge gained. While there is some obvious checking, without an 'effect' system and pure functions, there is a good deal of missing validation. Arguably, the situation is not much better than any other languages that don't bother with mutability or ownership at all.

5.1.3 Examining elide-reify

We are confident (supported by tests) that the lifetime reification refactoring is correct and complete.

Lifetime elision is mostly correct but incomplete. It fails in complex cases, such as when only some lifetimes may be elided or lifetimes used as bounds on traits. Since this is a new refactoring, exactly how useful the reification and elisions are, is unknown.

5.1.4 Formal correctness and alternative approaches

Formal foundations for refactoring in general appears incredibly weak as raised by the JRRT [9] paper. Even the most trivial of refactorings often lack written specification and are based solely on implementation. One related case study includes a graph rewriting approach to refactoring simple programs [5] which achieves reasonable success, but they note in their discussion the difficulties of handling language specific features (highlighting a massive limitation). Another recent paper from the Haskell community reiterates the difficulty of implementing a concrete refactoring tool or even understanding what should and should not be considered a "valid refactoring" [10]. It appears that, the act of implementing a reasonably powerful refactoring tool is no easy feat on its own, leave alone stating its specifications or formal properties it might guarantee.

5.2 Performance evaluation

With only single file tests, the amount of time spent performing each refactoring is negligible. Compilation times for single files, particularly trivial programs do not provide sufficient evidence of practical timing for performing a refactoring. Therefore we investigated real world code from crates.io, the Rust package repository [3].

5.2.1 Relative crate sizes

Figure 16 lists four crates from crates.io that have been chosen to help evaluate our tool. The crates are four of the five most downloaded crates by the Rust community [3]. The 'winapi' crate was omitted due to less relevance on a Linux platform. The lines of code metric counts only Rust source (.rs) files, but does not discount comments or test code. The purpose of the comparison is only to generate a rough idea of the size of the crates.

Rust crate	Lines of code
libc	6547
rustc-serialize	5741
rand	5187
log	1449

Figure 16: Size of crates compared

5.2.2 Comparing the timings between the types of refactorings

Timings were generated for the different crates using the Linux perf tool: `perf stat -r 10`. Timings were averaged over 10 runs and use of the perf tool gave much smaller unaccountable variations in results compared to other tools such as `time`. The machine used was a dual core 2.0 GHz machine running Ubuntu 12.04 with Rust Nightly 23/09/2015. The tool was compiled in release (i.e., optimised) mode which ensures increased speed, around 10x based on observation (relative to debug, un-optimised, mode). Timings represent elapsed time, not system time or CPU time.

The classes of refactorings measured are: renaming variables, renaming functions, renaming types, lifetime reification and lifetime elision. Inline local has been omitted due to lack of sufficient examples in the given crates, particularly without mutation. In each case, examples were picked with effectively a single usage, access or equivalent (with minimal modifications made) so that the difference in timings between each of the individual refactorings could be highlighted. Figure 17 shows how, regardless of refactoring, the time taken is generally comparable within a crate. Looking at Figure 16, the blind code size metric does not appear to be a good indicator of the base refactoring time and so comparisons between crates are limited. In general, the complexity of a crate is not necessarily tied to crate size. Function renaming takes noticeably longer, while lifetime refactorings are noticeably quicker. This is linked to the basic compile time. While a rename refactoring requires checking every usage of a declared item using the compiler, in the 'happy path' every usage should fail the compiler check during the early stages of name resolution. Only in more unlikely or unfortunate cases will a refactoring require any additional processing in analysis.

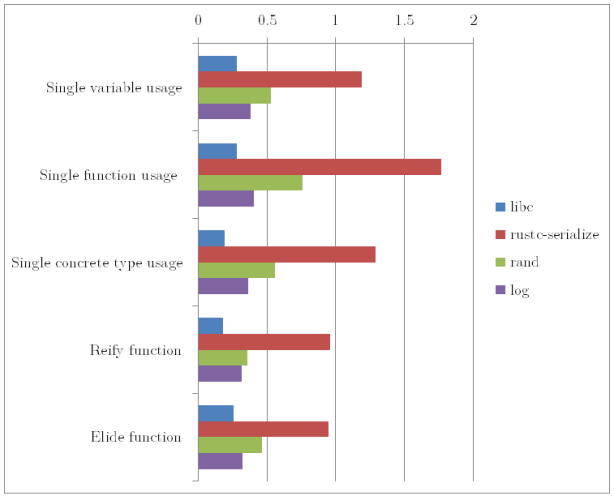

Figure 17: Graph displaying timing in seconds for the different refactorings

5.2.3 *Varying the number of refactoring locations or uses of type name*

From 'libc', a number of different concrete types were targeted for performing a rename. 'libc' was chosen for having a wider variety of occurrence counts specifically with concrete types. Figure 18 shows how concrete type renaming appears to scale linearly (as expected from Sect. 4) with the number of places a type is used in a program (type usages). Ideally, this analysis would have been done with other refactorings, but finding a reasonable amount of variation on the number of usages was difficult and searching was mostly manual. All the rename refactorings generally follow similar code paths and so the idea is that they should all scale in the same way (with the same general relationship). In particular, investigating function renaming would have been insightful as they take inherently more time. Although we expect to always scale linearly with the number of usages, the entire compiler is invoked each time instead of running what is actually necessary, like name resolution. As such, improvements can likely be made to the multiplying factor. As for the lifetime refactorings, the selection for the earlier timings did not strongly consider the number of visible '&' and from observation, the time they took did not generally vary significantly. This is probably because they do not use additional passes of the compiler. Investigating scalability allows us to predict the time required for refactoring larger codebases. Referring back to Fowler, this is important for a tool since taking too long means that a programmer would simply prefer to do the refactoring by hand.

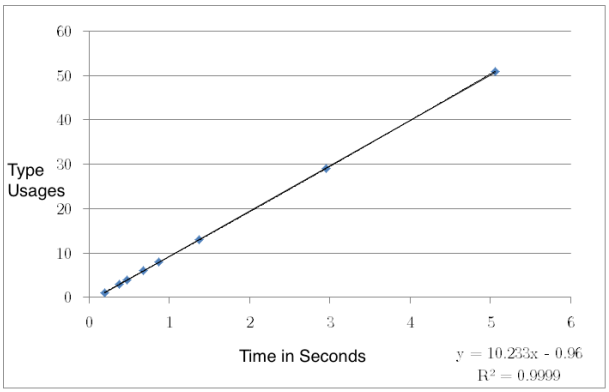

Figure 18: Graph displaying results of varying the number of usages

6. DISCUSSION

General progress in the field of refactoring appears quite slow, with many opting to simply produce implementations rather than tackling the problem at large. In terms of missing formalisms, refactoring is not the only one at blame it appears. Much in the same way, compilers encode otherwise undescribed aspects of programming languages. Rust is no different, and the case with the relatively informal elision RFC (which was descriptive but definitely not complete) was a reminder of this. Being well-defined likely would have allowed reversal of the elision rules with much more ease.

Compiling Rust programs is slow relative to other languages. Work is underway to implement incremental compilation and other compiler performance improvements. However, based on the current design of the tool and the need for multiple runs with modified source, the issue of performance would be better addressed by improvements to name resolution in the compiler and our tool's interaction with it. If name resolution could be re-designed to allow an 'interactive' mode, renaming could require a single compiler run, rather than one for every use of the variable being renamed.

Our tool cannot handle macro uses in the source text. Since Rust's macros operate on the AST rather than plain text, and the compiler is aware of them to a great extent, a tool should be able to work around macros to an extent impossible in C/C++. Whilst it is unlikely that code with macros can be refactored as easily as code without, we should at least be able to give a good estimate of whether the refactoring is safe or not.

The relationship between macro hygiene and refactoring

is interesting, but from initial analysis, does not appear to provide any particular benefits. Further research into the relationship might provide some unique insights and a system which is able to incorporate both would be of significant interest.

7. FUTURE WORK

One area of particular interest would be to investigate the overall effect of refactorings across multiple crates and how warnings (or patching scripts) should be presented when changes occur that might affect a public API. Examining the evaluation, this appears to be a major shortcoming in how not only this tool functions, but how other tools function across codebases. An investigation of Crates.io [3], the Rust package repository might yield interesting findings on crate interaction. One goal might be to describe the best steps to take when an API must change and how that should be dealt with from a community perspective.

7.1 Testing and current test failures

Going forward, the refactoring tool needs more testing to ensure that the current set of refactorings are correct. In adding functionality, tests ensure that existing refactorings are not broken with further changes to the tool. Furthermore, the refactoring tool evolves independently of the Rust compiler and changes to the compiler inevitably affect the outcome of the tool. Already, a number of existing compiler bugs have been found and reintroducing them by accident does not appear to be difficult with current limitations in the amount of testing and documentation available for some areas of the compiler. In this manner, expanding testing does not necessarily need to be confined to the refactoring tool.

With limited experience with Rust in general, using real-world code to attempt refactorings would help test for more obscure corner cases which may have been overlooked. Some initial testing with Crates.io [3] has been done along with initial analysis of its efficiency and usefulness, but more should be done in the future.

As of right now, the test cases identify roughly 10 incorrectly handled, but somewhat exceptional cases. A few of them likely require the addition of further investigation to determine their underlying source of error. On the other hand, a few of them are relatively subjective, in that they could be solved by simply restricting the inputs to the tool. One clear issue is that function local definitions currently cause name resolution issues. They appear to be at least solvable within the current tool architecture though. Elision is another source of failures, being only partly complete, having holes which were noted in Chapter 4.

7.2 Future refactorings

In terms of future refactorings, there are a number of next-step refactorings which should use most of the same infrastructure of the existing tool. For instance, trait renaming should use most of the same conventions provided by concrete types like structs or enums. This is because of the output of save-analysis and the simplifications made in process of generating the csv output. Renaming and modifying type aliases should present interesting problems due to deviations from typical object-orientated languages, along with type parameters and associated types. Furthermore, as previously described, renaming types declared within some function introduces issues to do with path resolution that still need to be resolved. To do so requires determining which parts of the path are 'private' to that scope and should not be used in the general case. The treatment of types in Rust generally complicates matters, and this is already with the simplifications made from save-analysis.

In the context of Rust, there are a number of refactorings which would be more unique. Extraction of a method, as described by Fowler, presents a number of intricacies tied with lifetimes and the ownership system in Rust. Ensuring borrows are made correctly (or move semantics if a constructor is present in the extracted code) presents difficulties specific to Rust refactoring. While other refactoring changes are semantics preserving, extraction of a method could introduce new lifetimes and verification of correctness may not be so trivial. Similarly, extraction of local variables appears to provide difficulties with the ownership system and should be tackled first to provide insights for approaching method extraction.

Although not implemented here, inlining of functions was considered to some depth. Rust has the fortunate advantage of being able to evaluate a block of code as an expression, returning the result of the final line. This should allow relatively straightforward inlining of function bodies when there is a single return at the end, or no return at all. Where it becomes more complicated is with early returns whose control flow cannot be modelled simply by a block expression (which do not have the concept of early returns). This could normally be mitigated by the use of a labelled goto-like construct, however, Rust does not support goto which makes this a much more difficult problem. Whether or not classic gotos can even be implemented in Rust appears to be an open problem due to the marked increase in difficulty in static analysis (scoping or typing rules). There are labelled loop break-continue statements, so one approach might be to wrap the block in a loop. Further issues include redeclaration of arguments, or alternatively renaming of arguments and identifying when double nesting of blocks is required to achieve the correct scoping and lifetimes.

Extraction, when compared to inlining, adds additional issues such as determining code-spans for expressions or the potential use of declared but uninitialized variables. In the standard case, extracting a local appears to be feasible as long as the same scope can be achieved; however, there is yet to be any actual evidence besides conjecture.

Converting functions to methods or vice versa is also a critical function that would be useful for the Rust community. To alleviate issues with API changes, supporting these changes with a refactoring tool would be incredibly useful, even if it only used a structural search and replace. This would provide functionality similar to gofmt and go-fix as Go originally updated their API [2]. Making changes which break a sizable amount of user-code is unfavourable, however, it usually must be done at some point and having a tool to remedy the stress of such a change would be good for both the developers of Rust and the community. A non-comprehensive list of other refactorings to be considered are:

- Inlining methods
- Creating or inlining modules
- Adding or removing function args
- Changing types to type aliases
- Extracting traits or moving inherited to trait

7.3 Modularity of code

While the evaluation mostly concerned external qualities of the tool, another aspect that could have been considered is the relative difficulty of adding or modifying refactorings. Currently, the compiler is tightly coupled with the refactoring tool and if you compare this to the Scala refactoring tool, you might find the latter uses much more well-established API and is more readily composable. In a similar way, JRRT likely benefits from the 'obliviousness' of aspects [11], being structured in such a way that injecting code is not noticed by the original program code. Whether this should be achieved here, or by working more with the compiler, it appears an important detail to enable wider contributions to the tool.

8. CONCLUSION

This work has explored refactoring and providing tool-support for the Rust programming language. Utilizing existing infrastructure provided by the compiler, this work identifies extensions which help facilitate common automated refactorings. In a more general sense, this work attempts to build upon existing work done on refactoring by documenting specific decisions made in building a refactoring tool (which might normally only be encoded in the source code of an actual tool), and attempting to analyze the decisions made by others.

Currently, the provided tool supports renaming of local and global variables, fields, function arguments, structs, enums, and functions. Beyond renaming, it also allows reification and elision of lifetime parameters and has preliminary support for inlining of local variables. These refactorings in particular highlight the idiosyncrasies of Rust, ensuring that the analyses performed here are some of the first of its kind. The complete limitations of these refactorings are not yet fully known, but there exists a current suite of tests to ensure that there are no obvious flaws in the approach.

9. REFERENCES

[1] Christopher Mark Brown. *Tool Support for Refactoring Haskell Programs*. PhD thesis, University of Kent, 2008.

[2] Russ Cox. Introducing Gofix. http://blog.golang.org/introducing-gofix, 2011.

[3] Crates.io. Cargo: The Rust Community's crate host. https://crates.io/, 2015.

[4] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. AW, 1999.

[5] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *ICGT*, pages 286–301. SV, 2002.

[6] William F Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[7] Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *ECOOP*, pages 258–282. SV, 2012.

[8] Garming Sam. rust-refactor: Rust refactoring project. https://github.com/GSam/rust-refactor, 2015.

[9] Max Schafer. *Specification, Implementation and Verification of Refactorings*. PhD thesis, Oxford, 2010.

[10] N. Sculthorpe, A. Farmer, and K. Beck. The HERMIT in the Tree. In *IAFL*, pages 86–103. SV, 2013.

[11] Friedrich Steimann. The paradoxical success of aspect-oriented programming. *ACM Sigplan Notices*, 41:481–497, 2006.

[12] The Go Authors. Rename: check.go. https://github.com/golang/tools/blob/master/refactor/rename/check.go, 2015.

[13] The Rust Community. RFC - Lifetime elision. https://github.com/rust-lang/rfcs/blob/master/text/0141-lifetime-elision.md, 2014.

[14] The Rust Community. Rust Documentation. https://doc.rust-lang.org/, 2015.

[15] Walton, P. (Mozilla). Why in the Rust language functions are not pure by default? – Email. https://mail.mozilla.org/pipermail/rust-dev/2013-January/002903.html, 2013.