Automated Refactoring of Rust Programs

Garming Sam
VUW, NZ

Nick Cameron
Mozilla Corporation, NZ
ncameron@mozilla.com

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

ABSTRACT

Rust is a novel programming language developed by Mozilla to replace C as the implementation choice for its entire codebase. Rust supports novel constructs such as lifetimes, ownership, controlled garbage collection, macros and a large number of systems programming features while maintaining the performance requirements of a modern browser.

We describe a new refactoring tool that helps developers using Rust including discussing the issues and unusual decisions encountered due to the complexities of modern systems languages. We outline the lessons learned and hope our paper will help inform future programming languages design and enable refactoring to be integrated into the IDE's. The resulting refactoring tool is written in Rust and available from Github under an MIT license [11].

1. INTRODUCTION

The Rust programming language reached the 1.0 milestone back in May 2015. Backed by Mozilla and Mozilla Research, Rust is open source and generates a number of contributions from the community. 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 interoperability with no overhead with C. Rust enforces an ownership system to restrict the duplication of references through borrowing and lifetimes, while still aiming for 'zero cost abstractions'. Using these techniques, Rust prevents dangling pointers and a whole class of related issues concerning iterator invalidation, concurrency and more.

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 or C++ [1]. Manual transformations, including editor search-and-replace are potentially prone to error and so performing tool-assisted refactoring which guarantees some measure of correctness can provide much greater confidence in changes.

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. The idea was to produce a tool which may be run to provide a set of refactorings, which may continue to be extended in the future.

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 The Rust Programming Language

The following is a simple Hello World Rust program with a few additions. Notice how variables have to be annotated with 'mut' in order to be mutated, and how this applies to any references.

```
// 'use' statement imports items into the namespace
use std::collections;
use std::*;

fn main() {
  let a = 2; // variable declaration
  let mut b = 3; // mutable variable declaration
  let c = &a; // 'borrow' or reference to a
  let d = &mut b; // mutable 'borrow' or reference to b
  println!("HELLO WORLD!"); // println! is a macro
}
```

In Rust [19], 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. Alternatively, there is the option of performing cloning and the ability to default to copying behaviour, much like primitives within the Rust language

While straightforward ownership with cloning probably provides a good deal of the necessary functionality for general purpose programming, it would likely be tedious. For in-

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

stance, passing in a variable to a function would simply 'eat' the variable and it would have to be part of the return so that the original variable can regain ownership (fortunately, Rust supports a return tuple). Rust supports references and *borrowing*, where an object can be borrowed instead of being owned. Borrows must conform to certain constraints: either there are only read-only borrows, or there is a single mutable borrow.

Borrows must also be nested correctly so that a borrow does not exceed the *lifetime* of (the borrow of its) 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 does potentially add complexity during coding; when returning references for instance, the compiler might need additional help to infer the borrowing lifetimes of the different parameters and returns. In order to do so, much like generics (in Java or C++) which parameterise functions or types over some generic type, there is the addition of *lifetime parameters* (which precede any normal generic types and annotated using a prefixing apostrophe) to describe the lifetime of the input and output variables:

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

Lifetime parameters act as part of the type information of a variable (or signature). They denote information about a particular 'borrow' and are used by the compiler to determine whether or not the ownership rules have been followed. Once compiled, they don't serve any real purpose.

In most cases, descriptions of lifetime parameters follow quite simple patterns, and so Rust introduced the concept of lifetime elisions to reduce code verbosity. Elisions allow omission of lifetime parameters (based on some rules), requiring them to be specified only when they are actually required (by compiler inference). Based on a survey, the elision rules covered around 87% of uses in the standard library [17]. Lifetimes appear in either the input or output positions, again, denoted by a leading apostrophe. For functions, input positions refer to the types for the formal arguments, and the output refers to the result type. 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.

A *trait* is a collection of methods defined for an unknown type which is denoted by 'Self'. Traits define interfaces, but do not follow the standard notion of inheritance. Traits can be composed of other traits, and any conflicting methods must be resolved if they occur. Traits allow definition of default implementations and most importantly, traits are implemented for data types – specifically structs and enums. Basically, where a type implements a trait, you can be expected to call any of the methods present in that trait on any object of that type. Rust allows limited operator overloading using traits. The '+' operator for instance, can be overloaded by creating an implementation for the corresponding trait 'Add'.

Finally, *modules* provide a logical unit of code organisation that can be used to hierarchically divide code while managing visibility. The appearance of the 'use' declaration allows binding of a module path to a shorter name and utilisation of code stored within such a module namespace. Modules can be nested within the same file, created as a file with the name of the module or as a subdirectory with the name of the module.

2.2 The Rust Compiler

As one of the most prominent examples of Rust code, the self-hosted Rust compiler (http://www.rust-lang.org/), rustc, has on the order of hundreds of thousands of lines of code. In order to generate machine code, Rust relies on an LLVM backend and their associated LLVM IR (or intermediate representation).

In Rust, a 'crate' forms a compilation unit. To compile a crate, a crate root Rust file is supplied to rustc, which merges the contents of referenced modules before the actual compiler is run over it. As a consequence, modules are not compiled individually, only crates. In order to link an external crate, the 'extern crate' declaration must be used. This will both link the library and import all items under a module named the same as the library.

A major feature provided by the compiler to enable refactoring, is the ability to produce a *summarised analysis*. This output in the form of a comma separated (csv) file and collates a number of useful artefacts for annotating source code and performing code analysis. In order for rustc to generate this output, a flag must be supplied to the command line '`-Zsave-analysis`'. Besides the refactoring tool produced as part of this work, the save-analysis output is also used by a Rust plugin for DXR, which is a source code browser developed and used frequently by Mozilla [6].

At the top of the csv there is metadata associated with the crate for which the analysis was generated. Following crate definitions, there are lines of the output which correspond individually to declarations and references for variables, types, functions etc. The first value of each of these lines is the type of information described by the line. In the cases described by Figure 1, they are `variable` and `var_ref` for variable declaration and variable reference respectively. The remaining attributes are mostly shared between the different types of declarations and references. File name indicates the file in which an item is declared or referenced. The next attributes (`file_line`, `file_col`, `extent_X`) concern where in the source file the code in question appears in terms of file lines, file columns, byte indices relative to the file and byte indices relative to the entire crate. In variable, there is the `id` attribute which corresponds to the node id of the corresponding node in the abstract syntax tree built by the compiler. In `var_ref`, this attribute is replaced by `refid` which corresponds to the node id which the reference is referencing. Here they both refer to "13", which indicates they concern the same variable. In static situations, like variable usages, the `refid` corresponds directly to the associated node for the declaration. However, this is not always the case with method calls and dynamic dispatch. In these cases, it is necessary to refer to an additional attribute called `declid`. With types, the node id may be aliased with the node id of the constructor instead of the node id of the declaration of the type, and this must be resolved within the refactoring program.

In Rust, a path (within the compiler) corresponds to the

Variable declaration:

```
variable,file_name,"basic_rename.rs",file_line,3,file_col,8,extent_start,38,
extent_start_bytes,38,file_line_end,3,file_col_end,9,extent_end,39,
extent_end_bytes,39,id,"13",name,"y",qualname,"y$13",value,"y = 20",
type,"i32",scopeid,"0"
```

Variable reference:

```
var_ref,file_name,"basic_rename.rs",file_line,10,file_col,20,extent_start,
178,extent_start_bytes,178,file_line_end,10,file_col_end,21,extent_end,
179,extent_end_bytes,179,refid,"13",refidcrate,"0",qualname,"",scopeid,"4"
```

Figure 1: Example csv output from *-Zsave-analysis*

concept of a name formed by a sequence of individual identifiers and other supporting information, e.g. std::cmp::PartialEq. Each of these segments might have type or lifetime information, but more importantly, they have a name and a syntax context. The names are stored, or interned, in an interner table and the syntax context provides information to specify where this name might be located. The appearance of a syntax context is due to Rust macros, which allow usage of variables and variable names introduced in a macro without conflicting or aliasing against any names once the macro is expanded or inlined. This behaviour is only part of the functionality provided by Rust macros, which allows manipulation of the AST and more powerful and (type) safe transformations than text replacing macros [7].

In many ways, the functionality of Rust macros parallels the functionality aiming to be achieved by refactoring, particularly in the case of renaming. However, the goal of each is divergent enough that the two systems are not compatible. One of the goals of hygienic macros which work with the compiler, is not to introduce a name that will conflict with any existing names [7]. This is done with the use of a syntax context, but the (textual) name itself does not actually change. Essentially, it can produce an AST which would not compile within the Rust compiler, however, when converted to source text where the differences in syntax contexts are unavailable, the code could compile just fine. In Figure 2, we can see how this can be visualized by striping code with colours. The 'a' in red might refer to `a#24` while the 'a' in blue might refer to `a#30`, placing each in different syntax contexts. Renamings performed on source text need to understand where the textual names conflict, and reusing a macro based system as currently implemented would only provide any significances internally in the compiler. Extensions to the macro system could be possible and parts of the system made to the advantage of refactoring, however, it appears fundamentally they are not compatible enough for one to be used as a replacement for the other.

```
let four = {
    let a = 42i;
    a / 10
};
```

Figure 2: Example of syntax contexts and identifier striping

Paths form an important part of name resolution within the compiler. Forming the first part of the analysis phase in the compiler, name resolution walks the entire AST and attempts to check that the different paths exist at the contexts in which they were used or their lexical scope [18]. In the case of refactoring, this is an important piece of functionality which needs to be used: First to ensure that a proposed name for a refactoring (or renaming) doesn't already exist and will not conflict, and secondly, that once any references to a name are changed, they do not refer to a completely different name. This is despite potentially being spelt the same and the consequence of different scoping or shadowing rules.

2.3 Refactoring

Martin Fowler's definition in 1999 [5] 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 [9]. Although taken from a C++ perspective, the definition continues to be used more widely [12].

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 3: Renaming local variable *a* to *b*

Figure 3 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 duplicated variable bindings under the same name, unlike other languages like Java. Such examples (of shadowing) are particularly worrying from a programmer's perspective since compilation failures clearly indicate issues but here successful compilation would provide no indication of preserved behaviour. The program would simply function differently without the programmer necessarily noticing any changes. Although this might be caught with testing, having such a comprehensive test suite for every such eventuality is impractical.

One of the first key ideas that the classical Fowler [5] 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 uses bad code smells such as code duplication, long methods and large classes, as rationale for the various refactorings and when they should be performed.

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, and one which could be approached with straightforward text-replace. 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. The examples here are not tailored for any specific language and the naming convention is taken from the comments of the gorename tool [15].

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.

```
int A = 1;                      int A = 1;
int B = 2;                      int B = 2;
{                               {
    int A = 3;                      int B = 3;
    print B; // 2                   print B; // 3
}                               }
```

Figure 4: 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.

```
int B = 1;                      int A = 1;
{                               {
    int A = 2;                      int A = 2;
    print B; // 1                   print A; // 2
}                               }
```

Figure 5: Sub-block conflict: Renaming outer B forces block local A to shadow outer A

In other languages, *same-block conflict* normally occurs with local variables which appear in the same scope. However, as described earlier, let bindings allow the redeclaration of variables under the same name in the same scope. In Rust, this allows mutability to be modified while retaining the original name and is generally considered good practice. While this conflict doesn't occur in Rust in the context of local variables, they still occur with global static variables, fields, and works similarly with other constructs like methods and types.

```
int A = 1;                      int A = 1;
int B = 2;                      int A = 2;
```

Figure 6: 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

All of this information can be found in the save-analysis data; however, it is completely static and simplified. In order to be able to perform these operations in the general case, the compiler has to be run again. For a refactoring to succeed, all names in a refactored program must bind to the same declaration as the original program [12]. 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 [12] describe the act of inlining a variable in the most specific detail. At the time, they also note the existing scarcity of indepth 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.

For this analysis, in order to reduce the problem space (as well as applying an accurate quantification), a number of assumptions have been made. First of all, only inlining of standard local variables will be discussed and not variable-like items such as function parameters. There is also the assumption that any code marked as unsafe (which does not follow the usual Rust ownership rules) should not interfere with the refactoring. Furthermore we assume that there only exists sensible destructors and operator overloads (or in other words, implicit behaviour in other locations) e.g. no actions that somehow modify a global variable which may affect our inline. We also assume that there is at least one usage of the variable and the manual equivalent inline actually holds meaning.

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 refers 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 description is optimistic in that an invalid refactoring should fail, but it also means that some valid refactorings may also 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 [12], 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 [20]. In Figure 7, we can see how the inlining of a database call which might insert a single record will suddenly be repeated if it is inlined. Now an interesting question is the presence of constructors or factory methods. In some cases, where there is only a single usage the inline could be valid, but the code in the constructor might violate purity. When there are multiple usages, being immutable and overriding the correct equality operators may suffice for inlining over most situations, but requiring strict singletons might be necessary for others. In any case, constructors appear to be an additional level of difficulty much beyond the current level of analysis.

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

Figure 7: 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 instance, 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 optimistic.

```
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 8: 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 8 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 [10], or some form of recursive analysis of origin of memory appears to be required, but this is outside of the scope of this work.

There is one other edge case without mention yet. In Figure 9 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];    // a has been inlined
// i is an iterator    // i is an iterator
let i = v.iter();      let i = vec![1, 2].iter();
```

Figure 9: Inlining causes compilation error: borrowed value does not live long enough

3.3 Lifetime elision and reificaion

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 10, 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 less similarly equivalent to '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 forget to explicitly annotate them. They follow common patterns so that in most cases, you will never need to include any lifetime parameters in your function declarations. If no pattern is matched, then those lifetimes cannot be omitted. In the below figure, none of these lifetimes are actually needed.

The rules essentially boil down to the following:

1. Any lifetimes as input (red or green) which are not marked become distinct lifetime parameters i.e. they will each use a fresh name like 'x, 'y, 'z etc.
2. If there is only a single red (or green) lifetime, or there should be a single red (or green) lifetime, that lifetime is assigned to all blue output lifetimes.
3. If there are multiple red or green lifetimes, the green self lifetime takes precedence and will be assigned to all blue output lifetimes.
4. Any other case is an error.

Now the idea is to build a tool to annotate these lifetimes where they have been omitted (reification) or to remove them where they are unnecessary due to compiler inference (elision). Despite being called the elision rules in the

```
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 10: 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 11: Examples of rule 1

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

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

Figure 12: Examples of rule 2

RFC [17], 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.

Here is an list of constraints that will ensure that only valid elisions may occur (but not necessarily allowing all valid elisions):

- Do not elide if there are multiple output lifetimes
- Do not elide if the return is not parameterized by the function i.e. in the `<...>`
- Do not elide if an input lifetime is used more than once
- If there is an output lifetime, either it follows the self lifetime parameter, or it follows the only input parameter
- Do not elide an input lifetime if it is not parameterized by the function
- Do not elide if there are bounds on a lifetime i.e. there are defined relationships between lifetimes, like `'a` must live as long as `'b`.

3.3.1 Discussion

As mentioned earlier, lifetimes follow standard patterns a significant proportion of the time. Considering no one has really thought about the automation of elision and reification (being a Rust specific behaviour), it is important to be clear here about the motivations in design.

For reification, we envision a developer who stumbles upon a piece of code involving lifetimes that they wish to change. The lifetimes were originally elided to reduce noise so that

```
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 13: Examples of rule 3 and rule 4

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

Figure 14: Partial reification – *'a* exists, *'b* missing

anybody using a function, for instance, could more easily grasp its underlying purpose. In modifying the code, the developer now wishes to visualize exactly which lifetimes are in use where. The developer could manually reinsert the lifetimes themselves, possibly erroneously or tediously when there are many lifetimes. 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.

Despite the general idea that elision and reification are complete opposites, the reality is not quite so simple. In particular, we note that the operations are not necessarily inverses of each other. This creates additional difficulty in defining constraints for validating elisions when attempting to reverse the elision rules. Part of the reason the two operations are not completely opposite is due to the ability to only partially annotate the expected lifetimes. In Figure 14, we can see an example of a partial reification. If we were to apply elision on the result, there would be no lifetimes remaining and the partially specified lifetimes would no longer be present. This raises another question: if the existing lifetime was specified for a reason (to clarify some detail for instance), would we ever want to preserve the lifetime with the elision? While not considered here, it is an interesting design decision that remains consistent with the workflow specified earlier.

While the only lifetime parameters seen here accompany an explicit borrow using '&', they are not the only positions where a lifetime can occur. There is also the case where a trait might be boxed, or in rough terms wrapped by some pointer. In Figure 15, we can see how a function declaration with a boxed trait is reified. While this appears to be known behaviour of the compiler, the RFC does not explicitly mention this case and the actual semantics of the behaviour is defined completely implicitly with the current compiler implementation. More to the point, the term 'input position' in the RFC definition is still open to interpretation and it is unknown if there are any other cases which should have been considered here, but were not.

4. IMPLEMENTATION

4.1 Renaming

```
trait SomeTrait<'a> {...}

fn foo(x: Box<SomeTrait>) -> &i8
```
reified to: `fn foo(x: Box<SomeTrait<'a>+'a>) -> &'a i8`

Figure 15: Reification of a boxed trait

Given a node id, a new name, the save-analysis file and the crate root file, a rename refactoring then has enough information to begin. Loading in the csv analysis, there are two separate pieces of information that need to be identified: the declaration and the references. Once they are ascertained, we run the compiler API to invoke the compiler. Using name resolution within the compiler, we can attempt to resolve the new name at the declaration site in the AST to ensure that it does not cause any conflicts. By doing so, this would avoid same-block conflicts and prevent all super-block conflicts. Consequently, this also prevents a number of valid renamings where there is no eventual usage of the shadowed item.

Referring back to the conditions listed in Section 2, resolution at the declaration site for super-block and same-block conflicts force usages binding to different declarations to remain binded to their different declaration. By addressing sub-block conflicts, at each renamed usage, name resolution would check the remaining condition that the binding was made only to the renamed declaration. Ideally, name resolution would run with both the declaration renamed and the usages renamed and within a single pass of the compiler.

Unfortunately, limitations imposed by the structure of name resolution and the internal representation mean that this is not possible. In order to provide functionality for detecting the missing sub-block conflicts, recompilation of the entire crate with a single use renamed is necessary. Of course this provides significant overhead; however, hopefully name resolution can provide the required functionality in the future. Apart from compilation, there does not appear to be any straightforward way to checking if a name already exists in the context for a usage. The full name resolution approach is one which appears to be adopted by gorename [15] and is much more efficient in general due to the fact that only one compiler run should be necessary to check every modification point. The additional choice of employing the full compilation approach for declarations indicates further complexities in providing valid expression constructions (to test the presence of an existing name). A generic approach could not be used and so constructions of different forms for variables and variations of types and functions would be necessary – which might not be compatible with simple ad-hoc replacement at the source code level.

Adopting the compilation approach, each reference is renamed to the new name one at a time and compiled to ensure that it fails. If a compilation succeeds, then a super-block or sub-block conflict would have occurred in this location and the refactoring must be halted. Care must be taken to ensure that the compilation fails due to a name resolution problem and not one which is due to other failures. If all the compilations fail correctly, the refactoring proceeds and performs all renamings of the occurrences of a variable/function/type.

4.2 Inlining

Following the description given in Section 2, the feasi-

ble implementation of inlining a local variable is relatively straightforward. This is especially the case when considering that Steps 1 and 2 listed are effectively impossible given the current language constraints as well the current compiler implementation.

4.2.1 Addressing Steps 3 to 7

In order to provide the functionality for the remaining steps, the compiler provided essentially all of the necessary constructs. By reappropriating the save-analysis module that typically outputs a csv file that includes all the variable usages, the tool goes and counts the number of usages of the variable you are about the inline. This information is enough to satisfy Step 3.

By using the node id that will be supplied to the tool at the beginning to identify which variable to inline, the tool can use the compiler to reconstruct the AST and determine all the mutability information. In order to get this information though, the tool needs to run the compiler to the end of the analysis phase which forms a significant proportion of the time spent compiling. This is required to check that a 'mut' declaration was actually required and that the variable does not have interior mutability (satisfying Steps 4 and 5).

To replace the usages of the local variable with the initializing expression, the Rust compiler offers a useful 'folder' trait which allows manipulation of the AST. This 'folder' trait is used to expand or replace nodes in the tree and is how macros or syntactic sugar are generally handled in the compiler. The idea is to first walk the AST of the initializing replacement expression to determine which identifiers are being used to compose it. Then you walk the tree with the folder looking for any references to your local variable. If you find one, go through all the identifiers you found earlier and use name resolution to see if they resolve to their original declarations. If not, abort the refactoring. This satisfies Step 6.

Step 7 is actually fairly trivial to implement because all that needs to be done is to add an additional check during the folding just described to remove the affected declaration nodes. From here, the inbuilt pretty printer in the compiler is used to format the modified AST.

4.2.2 Concrete example with order of operations

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

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

Figure 16: Correct inlining with order of operations

In Figure 16, you can see the general result of running the tool on the given input. In particular, you can notice that the order of operations is preserved due to the fact that the pretty printer correctly identifies where parentheses are required. Without the pretty printer handling this case, the identified expression would evaluate to 4 instead of 6. Originally, this was not the behaviour given by the pretty printer, and contributions to the compiler were required to ensure that this case (as well as other similar cases) were correctly handled.

4.2.3 An alternative approach

In the original list of steps, there is no transformation between some abstract representation, like the AST, to concrete code. Using the pretty printer was a relatively straightforward choice since walking the AST was required for the checks in Step 6. By performing the replacements at this step, there would be no need to do any secondary walks and by pretty printing, there would be no need to determine and translate the locations in each file of the variable usages. There are some obvious disadvantages to using the pretty printer. One of them is bugs in the compiler, which was found to happen with parentheses. Another is the fact that the pretty printer and replacement operations function on the expanded AST, where macros no longer exist. Although the expanded code of the macro might compile and function just fine, there is the chance that it doesn't due to the syntax contexts distinguishing identifiers only within the compiler. Furthermore, the expanded code is often just ugly, which is why it was replaced with a simple macro.

Instead of using the pretty printer and pursuing the issue much like the Scala refactoring tool, we can perform the replacements one by one which requires all the location information for each of the usages to be recorded. The only caveat is the removal of the actual local variable declaration. This is because although we can delete the entire declaration, it may not be the case that we can remove the blank line left in its place without additional analysis. The fact that compatibility with macros has not been a strong point of the tool and the relative ease of implementation, both contributed to the decision to opt for the pretty printing approach.

4.3 Lifetime elision and reification

As Section 2 described, implementation based on the RFC rules makes reify relatively straightforward. The reintroduction of lifetime parameters was based on the implementation of error reporting of missing lifetimes within the compiler. The original hope was that the compiler could simply output the reified function declarations, but it appeared that all that information is encoded in a different format (possibly for LLVM) and impossible to translate back to the AST. In general, this has been a recurring problem that after losing an abstraction level, it is impossible to raise it back up an abstraction level. Macros are another good example of this. While it is logical and useful for a compiler to perform these abstraction changes, for a tool, it is important to know which level to operate on, which steps are reversible and how your approach should be accommodated.

The idea in general is to count the amount of lifetimes in the various positions: in, out, as well as noting the position of self. The idea is to do a walk the AST, looking specifically at the lifetimes present within a function declaration. In order to rebuild the function declaration with the correct lifetimes using the lifetime error reporting system, a vector describing a partition of the parameters is necessary. The vector contains a list of the different equivalence classes of lifetimes. Once rebuilt, the pretty printer can be used to replace the old function declaration.

Compared to reify, elide was not quite so simple. The list of constraints identified and shown in Section 2 have still

not been completely implemented. In particular, the tool bails out in cases where a partial elision could still occur, like Figure 17 below. Even the constraints themselves are quite conservative and more work can definitely be done to improve them, particularly with parameters with bounds. Again, the idea is to walk the AST, counting the amount of lifetimes in various positions or situations. If all the constraints are met, then we use a 'folder' much like inline-local to fold away the unnecessary parameters and simply pretty print the result.

```
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 17: Partial elision – only *'b* removed

5. EVALUATION

5.1 Validity of refactorings

Included in the code repository are a number of tests to ensure that the refactoring tool functions as expected, all written in Rust. Each test consists of a csv dump file, a Rust source file and if the refactoring is designed to be successful, an additional output Rust source file. In the case of success, the output of the tool is found to be equivalent to the expected output source code and in the case of failure, the test code confirms that the tool aborts the refactoring. Currently there are a total of 85 different tests in the test suite.

5.1.1 Validity of renamings

The testing suite attempts to test 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 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 18. When running any of the renaming refactorings, name resolution will run to find any super-block conflict or some cases same-block conflicts. Afterwards, any sub-block conflicts will be caught in a compilation run. In some cases, the 'use' import graph is then rebuilt to identify same-block conflicts which were missed by name resolution. By convincing ourselves that all the conflict types are covered, in addition the having test cases to show that it works, there can be confidence that the overall approach works.

Local and global variables (and generally function arguments and fields).

At the moment there exists several tests for const, static and normal local variables, both in successful and non-successful situations. In terms of the ability for this class of renamings to fail unexpectedly, the chances appear slight since variables lack the most dynanism and complexity (no dynamic dispatch for instance), particularly for local variables. As long as name resolution works correctly and the compilation process, few of these renamings should cause issues. During the process of testing, it was found that 'static mut' global variables did not record their spans correctly due to an error in the save-analysis code. In particular, the span was recorded for the 'mut' identifier as opposed to the actual name of the variable. Fixing this required a minor patch to the Rust compiler and this patch was upstreamed prior to the release of Rust 1.0.

Methods and functions.

At the moment there are several tests for renaming methods defined with a trait and/or overridden by a inheriting struct. Tests address both cases of static dispatch and dynamic dispatch, addressing all currently known issues with the csv file description and the reliance on the use of `declname` as opposed to a proper id for dynamically dispatched methods. One known failure mode of the refactoring tool is when a function (or trait) is declared within a function scope. Prevented renamings include dynamically dispatched methods, handled by the compiler runs on each usage.

Concrete types – structs and enums.

At the moment 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' statements which allow a specific namespace to be added to the default 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 19 shows an edge case identified during this project. A struct `Point` consisting of an `x` and a `y` can be used to initialize two corresponding local variables `x` and `y`. Without intervention, a renaming might attempt to change `x` to `foo` but `Point` has no corresponding field `foo`. The general difficulty in solving these issues is not fixing the tool itself, but finding the issues. In many cases, outliers like these should simply provide warnings since simple manual correction would solve all the issues (as all the other usages are necessarily correct). From this point of view, the goal is not always pure correctness but ease of use.

5.1.2 Shared refactoring benefits and limitations

All of the refactorings rely on the fact that the save-analysis output is correct, offloading this burden. Although errors might have been found during this project, none of them were major. Minor errors in processing has meant that little known edge cases have occurred and would not have been found without the current testing. In terms of current testing of the save-analysis functionality, it is relatively minimal and would be difficult to implement considering the different possible combinations of expressions and items. Beyond errors in the save-analysis translation, errors in the compiler would also affect the ability to function correctly, but the compiler should be much more tested. On the other hand, lesser used and documented API like name resolution could be affected and this is a problem especially from the context of a third party tool and the first of its kind to use various APIs. This makes adoption of this code important by those who use Rust, and want to see a tool flourish.

With the save-analysis, there is also the limitation of the missing macros in the output. As a pending issue in the Rust compiler, the necessary plumbing required for macros could

```
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 18: 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 19: Invalid rename of x to *foo* which is easily fixed manually

be implemented given enough resources and should eventually function together with save-analysis. Unfortunately, until then, any considerations for validity must exclude the use of macros. On the other hand, reduction of the problem space makes it easier to ensure that refactorings are valid. Ensuring tests are up to date prepares us for this eventual change.

We can consider Bill Opdyke's preservation constraints and adapt them for Rust (like to traits). Like usual, in Rust there should always be distinct member names and distinct 'class' names. Both of these should be generally checked using name-resolution. Ensuring compatible signatures and type safe assignment should not cause any issues since the compiler (and save-analysis output) should generally prevent this. As for his last constraint, semantic equivalence is hard in any case and at best we use our testing to gain confidence.

5.1.3 Examining inline-local

The entire inline-local refactoring is definitely still a proof of concept. 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 checks. Arguably, the situation is not much better than any other languages that don't bother with mutability or ownership at all, but what is possible at all has been implemented as best as possible (along with tests). One major problem that was correctly tackled with the use of the pretty printer was the preservation of comments. Although not necessarily location preserving, dropping comments altogether represents a serious issue. Comments can be quite important and help to explain complicated pieces of code. Having tests to document this ensures that conversion to a non-pretty printer approach will not cause an oversight. Although it is possible that there are flaws in the inline description, it is more likely that they materialize as flaws in the language understanding. In any case, hopefully, this work provides contributions to the refactoring community.

5.1.4 Examining elide-reify

At least in regards to the situations and prerequisites identified, confidence in the reify functionality is quite high. In particular, the RFC is straightforward, there does not seem to be any unknown edge cases, and tests written cover parts of the RFC. Many atttempts at reification which do fail, fail with the premise that the function was invalid in the first place. This is usually impossible since the csv needs the program to have compiled in the first place.

Elide is likely correct within the given constraints given, although they are more restrictive. In most cases, it outright fails if there is anything too complex. This has advantages and disadvantages of course. Because it only handles a subset, testing should cover more ground. Compared with reify, it is probably more incomplete (although both still lack the boxed trait case). Since this is a new refactoring, exactly how useful the reification and elisions are, is unknown. With Rust users, determining what is the 'right' behaviour is, should be done before validity is truly questioned.

5.1.5 Formal correctness and alternative approaches

Formal foundations for refactoring in general appears incredibly weak as raised by the JRRT paper. Even the most trivial of refactorings rarely have huge amounts of published detail and are based solely on implementation. Aiming to create any reasonable formalism appears to be an arduous task, and not one met with many rewards, particularly when the goal here has been specifically the production of a Rust refactoring tool. Not only that, but the amount of time available and the necessary experience appeared thoroughly inadequate for producing anything meaningful for others. Looking at some attempts, at best they appear to be attempting case studies for different approaches of formalization. One attempt used graph rewriting [8] to reasonable success, however they note in their discussion the difficulties of handling language specific features (reiterating a massive limitation). From looking at a Haskell refactoring paper, the general attitude appears to be that if we cannot even implement a concrete tool or understand what should and should not be passing, there is no way that a formalization holds any specific meaning [13]. Furthermore, the act of implementing a reasonably powerful tool is no easy feat on its own.

Where possible, informal arguments have been given as to how different refactorings work, but to address any shortcomings, the a major focus has been on implementing test cases for each of the individual refactorings. While unit testing would provide more assurance that the code written is correct, a major part of the underlying code is actually in the compiler and the refactoring tool has to trust the correctness of this external code. To continue to write unit tests in the compiler was an option, but one that would likely be less effective in ensuring validity of the actual refactorings implemented. Certainly this project has succeeded in its goal of a proof of concept, but the end goal should be real world testers utilizing this code to identify issues that genuinely cause people problems.

5.2 Steps necessary to perform a refactoring

Looking at what is necessary to invoke the refactoring tool we can determine a general description of steps required:

1. Compile a program with `-Zsave-analysis` to produce a csv analysis file.
2. Either inspect the csv manually or run the refactoring library to determine the node id of the element you wish you alter.
3. Run the refactoring tool, choosing your desired refactoring. With a rename, the new name must be supplied, the node id and optionally, the file.
4. Wait for the refactoring to occur and the compiler to run all the proper checks.
5. Process the result and save the result to disk, if desired.

Ideally, the flow of behaviour a user would want would be to simply identify the row and column and the refactoring to happen automatically. Not involving node id at all would be good, but this is necessary to adequately treat the tool as a library as opposed to a full fledged tool. Being able to easily identify row and column still requires some form of GUI tool and integration with such a tool would be desirable to reduce the amount of different parts required to perform a refactoring. Having to regnerate a csv file every time a refactoring has been made is definitely not desirable, although using the same csv file could only ever allow renamings to new names of the same length and renaming would have to occur on separate variables. Even though the compiler would probably generate the same AST with the same node ids (being deterministic) having different lengths in a new name would cause all the indexes into the code maps to be at the wrong offsets. Implementing some form of analysis cache could reduce user friction and reduce the amount of time spent compiling, but unless it managed all the changes in offsets correctly, as soon as a refactoring which significantly changed the output began, any existing analysis would still be out of date. Likely the best solution would be to correctly implement incremental compilation within the Rust compiler, which would allow less significant compile times and faster regeneration of analysis – functioning basically like a cache. The lack of such behaviour is a shortcoming of the compiler and does appear to require significant structual changes but the benefits would extend further than just enabling better refactoring.

Looking at the list given in the Chapter 3 for the ideal steps for a refactoring tool, we can see we generally follow the guidelines. One point of distinction is the ability to generate patch files, but there is no reason this cannot be done (even externally from the current tool). The same argument applies for undo functionaltiy. As of right now, there is no reason to think that we cannot just plug our library into an Emacs plugin for instance, or build a simple GUI (although how robust and convenient Rust FFI is, now comes into play).

5.3 Performance evaluation

With only single file tests, the amount of time spent performing each refactoring is quite negligible. Compilation times for single files, particularly trivial programs do not provide sufficient evidence of practical timings for performing a refactoring. Therefore the choice was made to investigate real world code from the Crates.io, Rust package repository [4].

5.3.1 Relative crate sizes

Figure 20 lists the four crates from Crates.io that have been chosen to help evaluate the produced tool. The listed crates form four of the top five most downloaded crates by the Rust community [4]. The 'winapi' crate was omitted due to less relevance on a Linux platform. A fifth crate 'bitflags' was originally going to be used for analysis; however macro incompatibility made this an impossible task. The lines of code metric only concerns Rust source (.rs) files and does not take into account comments or test code. The purpose of the comparison is only to generate a rough, high level contrast and to gather any overall insights.

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

Figure 20: General figure for the relative size of crates compared

5.3.2 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 less unaccountable variations in results compared to other tools such as `time`. The machine used was a dual core 2.0 GHz virtual machine running Ubuntu 12.04 with Rust Nightly 23.09.2015 (along with the latest version of the refactoring tool). The tool was compiled in release mode (not debug) which ensures increased speed, by at least 10 times based on observation. Timings represent elapsed time, not system time or CPU time.

The classes of refactorings measured are: renaming variables (or variable-like constructs), renaming functions (or methods), renaming concrete types, reification and 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 21 shows how regardless of refactoring, the time taken is generally comparable within a crate. Looking at Figure 20, 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 e.g. 'libc' being mostly header declarations. Although we have some spread in crate size for this analysis, this could likely be improved. Function renaming takes noticeably longer while lifetime refactorings are noticeably shorter. This base refactoring time is likely linked to the basic compile time and how at worst, we need the analysis information to check the validity of a refactoring. 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 parsing or name resolution. Only in more unlikely or unfortunate cases will a refactoring require any additional processing in analysis. This likely makes the different renamings relatively comparable.

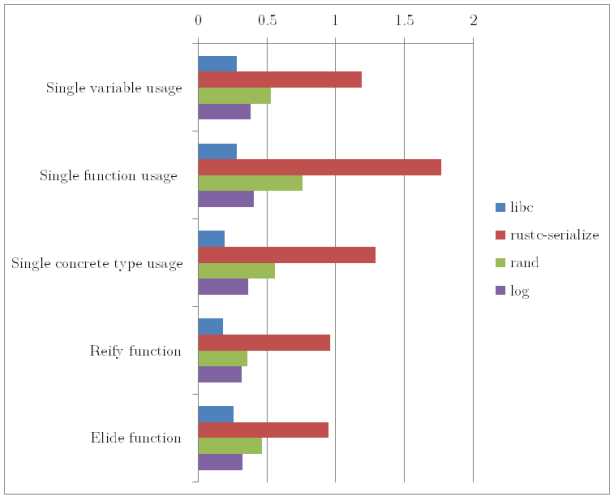

Figure 21: Graph displaying timing in seconds for the different refactorings

5.3.3 *Varying the number of refactoring locations or usages*

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 23 shows how concrete type renaming appears to scale linearly (as expected from Chapter 4) with the number of 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.

Rust generally encourages the use of Crates.io and package management to form modular systems. This means that the amount of code implemented by you personally is as minimal as possible, making scalability potentially a non-issue. On the other hand, more modular systems mean more API and shared functions, types etc. While this might not affect performance it affects the overall difficulty of refactoring because once an API is exposed, it is usually not liable to change. Changes across different code-bases are problematic, not just for Rust, but for programmers in general (due to different owners, licenses, etc.).

Number of replaced occurrences	Time in seconds
1 type usage	0.1925
3 type usages	0.3821
4 type usages	0.4749
6 type usages	0.6737
8 type usages	0.8739
13 type usages	1.373
29 type usages	2.960
51 type usages	5.059

Figure 22: Timings for varying usage counts of concrete types in libc

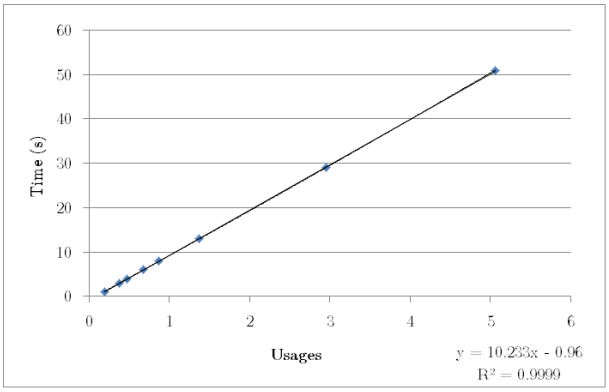

Figure 23: Graph displaying results of varying the number of usages

5.3.4 *Limitations in the performance evaluation*

Perhaps the most obvious shortcoming in these tests is the lack of sufficient data points, especially when trying to justify a trend. Finding the current set of examples was already extremely difficult given the size of the crates, made worse by the lack of macro support. Originally, a fifth crate 'bitflags' was to be analyzed; however, the entire crate formed a macro expansion, preventing any refactoring. 'libc' was almost entirely header declarations, having no 'let' bindings, which made it difficult to generate the full set of refactorings. Performing scaling tests on more than just renaming types would have been helpful to prove worst-case linear scaling, especially for renaming functions as it was noticeably slower. But finding any meaningful variation of occurrences was difficult, and having but a few usages was common (as seen by the concentration of points in Figure 23). The relative size of the crates is still quite small and how the tool functions on much larger codebases is still completely unknown. Missing inline definitely limits the amount of useful conclusions that can be draw here. But the fact that few suitable locations for its usage were found questions the usefulness of the refactoring and the associated checks (given how incomplete they are).

6. DISCUSSION

One of the reasons refactoring is not a straightforward topic is because of the different ways people might interpret a refactoring. In some domains or context, a perceivable behaviour violating change may not even matter. There are some cases where garnering user feedback will offer a clear-cut decision, but there are likely cases that will not. Caring is important because even if a user dislikes the way code is pretty-printed, for instance, they might avoid a tool entirely. Offering choice sounds good, but deciding what is an error, or a warning is still not easy (with better granularity here being a goal). One recent example that has come to light is the existence of conditional compilation within Rust. If performing a refactoring breaks other platforms, but not yours, should that be a problem? In particular, you may not have the required analysis of the remaining code, in which case a failure might be better.

As raised throughout this report, lack of formalism is an ongoing problem in the area of refactoring. 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. This is not a reason to point blame, but to highlight the genuine scale and complexity of the projects and issues at hand.

Given the current architecture of the tool, there are a number of further improvements that could be made to the compiler to improve its efficacy and efficiency. Using `-Zsave -analysis` provides another additional step before a refactoring can be made. Furthermore, it must be generated every time a refactoring is to be undergone. By providing a library and API with which this information can be queried using the existing compiler API, the need for csv parsing and associated unrelated complexity disappears. The actual performance offset of having to run the compiler again to generate this information does not disappear however.

Helping to address performance in the compiler itself in the general case should also help to improve performance of the refactoring tool. 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 the name resolution module in the Rust compiler. In particular, providing name resolution for usages instead of just at the declaration level would turn O(N) runs of the compiler into O(1) by using name resolution multiple times in a single run. Fundamentally this requires heavy changes to the structure of the compiler and the nature of name resolution. As it currently is, not every node gets its individual node id in nested expressions and to resolve this requires significantly more memory usage or novel, significant and well-architected changes, well beyond the scope of this project.

Currently macros are completely ignored by the save-analysis report generated by the compiler. As it is, this appears to be a major shortcoming in the refactoring tool that has yet to be addressed. However, not a lot can be done at this point due to the general limitations regarding macros and how they are typically ignored when generating metadata from the compiler. Without being able to identify any spans associated with macros, it isn't possible to make the necessary code transformations. Implementing the necessary functionality is certainly non-trivial and likely requires much better domain knowledge than that which could be provided here. Resolving this would also fix pretty printer abstraction layer issues.

The relationship between macro hygiene and refactoring is particularly novel, 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 academic interest, and interest to this author.

Efficiency of large-scale refactoring in general needs more attention. Although analysis of individual packages on a service like Crates.io provides some benefits, it would be curious to see how a tool functions on code bases which are much larger, in the order of hundreds of thousands of lines of code or larger. Although the expectations on this particular tool are not quite so high, understanding the general tradeoffs of provably correct refactorings and time taken and developer perception of this tradeoff is likely to produce fascinating observations. Looking at the Google-scale of refactoring, Google have set up a Clang map-reduce to perform refactoring on large codebases over a network of machines [2]. Evidently the utility and associated confidence in such refactorings led them to spend the time to create such a framework, but determining when this happens in the general case provides provocative food-for-thought.

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 [4], 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 [4] 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 [3]. 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 further 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 [14], 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 within the context of 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.

At the moment the provided tool supports renaming of local and global variables, fields, function arguments, structs, enum and functions (or methods). Beyond renaming, it also allows reification and elision of lifetime parameters for functions and methods 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. The presence of bugs in the compiler is a real problem to generality, but without real world use and more contribution to testing, finding these corner cases appears to be difficult.

At the moment, Rust lacks significant refactoring tool-support and evidently requires more work particularly within the compiler to enable further, valuable progress. Although a preliminary tool has been provided, there are many avenues for continuing work and the hope is that this first investigation provides useful insight for future efforts. Understanding the required context and the necessary infrastructure has been a major part of this work. In particular, learning and understanding Rust has been incredibly challenging, as it introduces concepts rarely used elsewhere. Continuation of this work should allow greater focus on implementing a more difficult and comprehensive set of refactorings for Rust.

9. REFERENCES

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

[2] C Carruth. Clang MapReduce-Automatic C++ Refactoring at Google Scale. *Google Inc*, 2011.

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

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

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

[6] Github. Rust DXR plugin. https://github.com/nrc/dxr/blob/rust5/dxr/plugins/rust/__init__.py, 2015.

[7] Daniel Keep. Practical Intro to Macros in Rust 1.0. danielkeep.github.io/practical-intro-to-macros.html, 2015.

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

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

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

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

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

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

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

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

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

[17] The Rust Community. librustc_driver – driver.rs. https://github.com/rust-lang/rust/blob/master/src/librustc_driver/driver.rs, 2015.

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

[19] 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.