Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

RustSmith

A Randomized Program Generator for Rust

Author:
Mayank Sharma

Supervisor:
Prof. Alastair Donaldson

Second Marker:
Prof. Cristian Cadar

June 20, 2022

Abstract

Rust is a modern, popular programming language designed specifically with performance and reliability in mind, and is now being rapidly adopted both in industry and academia. Rust aims to guarantee memory safety through "a rich type system and ownership model" which enable developers using Rust to "eliminate many classes of bugs at compile-time" [1]. As such, ensuring Rust's compiler (`rustc`) is reliable and bug-free is critical.

This project presents RustSmith, a novel Rust compiler fuzzer which aims to test the Rust compiler by randomly generating valid, interesting and diverse Rust programs. RustSmith takes inspiration from previous successful techniques for compiler testing, and adapts them to the context of Rust. In addition, RustSmith utilizes novel techniques that enable it to effectively test *Rust-specific features* such as ownership, borrowing and lifetimes.

RustSmith breaks new ground as the first fuzzer for the Rust programming language [2] capable of generating valid programs that exercise a number of Rust-specific concepts and features. RustSmith is capable of generating files that cover over half of the optimizations module in Rust's compiler source code, along with successfully producing bug-inducing files that detect both historic and current bugs in the Rust compiler. Additionally, we find that RustSmith successfully covers "blind-spots" (which are areas the official test suite is unable to cover) in 9 different optimizations used by the Rust compiler.

Acknowledgements

I would like to thank my supervisor, Professor Alastair Donaldson, for their invaluable guidance, constant enthusiasm and insightful thoughts and ideas throughout the development of the project.

I would also like to thank my second marker, Professor Cristian Cadar, for not only giving feedback for the interim report, but also teaching the Software Reliability course that provided me with the basis to complete this project.

Finally, I would like to thank my friends and family for their immense support throughout the degree and this project, providing the support, encouragement and motivation I needed to complete this project.

Contents

1 Introduction

1.1 Motivation

Rust is a new, general-purpose programming language built specifically with performance and safety in mind [1]. It is syntactically similar to C++ but has unique features that guarantee memory safety, along with lifting concepts from functional programming languages such as pattern matching. Since its first release, Rust has gained a lot of traction in industry with widespread use in companies such as Amazon, Dropbox, Meta, Google and Microsoft [3] and has been voted "most loved programming language" every year since 2016 [4]

Rust is a compiled language, and therefore comes with a compiler called `rustc` which compiles Rust programs into a binary executable. Whilst producing a rudimentary compiler for a small language is fairly trivial, most popular compilers are complex pieces of software, as the source language is usually feature-rich and complex, therefore requiring the compiler to optimize programs before producing an executable.

Despite this complexity, compilers are very heavily relied upon by most modern software systems and are often "blindly" trusted as being completely bug-free. Developers also have the expectation that programs will always be compiled into an executable that will exhibit the expected behaviour. Naturally however, compilers are themselves pieces of software and are therefore prone to bugs as well. For example, the developer community for LLVM, a widely used compiler toolchain, fixes around 150 bugs per month [5]. The consequences of a single bug in a production compiler can be severe as arguably any application built using it could be affected by the underlying issues present in the compiler. The worst type of bug is an "unnoticed miscompilation" which occurs when a compiler throws no errors, produces an executable, but generates the wrong executable for the provided program. For example, in 2011, Java 7 was released with a miscompilation bug, and several popular Apache applications crashed unexpectedly as a result of it [6].

Motivated by the crucial role compilers play in software development, and the potential severity of the consequences of bugs being present in them, ensuring compilers are reliable is an active research area. Multiple approaches have been taken to try to improve compiler reliability, such as the formal verification of compilers and the development of extensive manual test suites. Among these approaches, there has been a recent and active interest in *compiler fuzzing*, which involves automatically and randomly producing large, interesting and diverse test programs to validate the compiler. Crucially, compiler fuzzing helps discover unexpected input programs to a compiler that can cause crashes or miscompilation, where a standard test suite merely confirms that known test-programs behave as expected.

Our work aims to take inspiration from existing techniques to produce a system that can explore how compiler fuzzing can help ensure the reliability of the Rust compiler through a new tool we have developed called RustSmith.

1.2 Objectives & Challenges

The aim of this project is to create a fuzzer for Rust called RustSmith, a random program generator built with the purpose of testing the Rust compiler. Specifically, the overall research objectives of this project are:

1. Create a novel fuzzer using existing compiler testing techniques for Rust by developing a Rust program generator, which produces interesting, valid and diverse programs
2. Devise and implement new techniques which allow the generator to effectively generate programs that exercise unique Rust features such as ownership and lifetimes
3. Evaluate RustSmith's capacity to test the Rust compiler in order to evaluate the impact a compiler fuzzer can have in testing the Rust compiler

Given that (to the best of our knowledge [2]), no fuzzer for the Rust programming language has been created before, completing these objectives requires first taking inspiration from existing and successful compiler testing techniques and then adapting them to produce Rust programs that are both valid and interesting. Additionally, novel techniques will have to be developed to allow advanced, Rust-specific features such as ownership, borrowing and lifetimes to be effectively tested, whilst still ensuring generated programs are semantically and syntactically valid.

1.3 Contributions

The contributions of this project are summarized as follows:

1. We create a novel (and to our knowledge, the first fully-fledged [2]) generator to produce valid and interesting Rust programs. We introduce RustSmith's design and techniques used in Chapter 3, before describing the implementation in Chapter 4.
2. We develop new techniques that allow RustSmith to produce valid programs which exercise Rust-specific features. These features are conceptually different to those of other programming languages (e.g ownership and lifetimes). These techniques are also described in Chapter 3 (specifically Section 3.4 and Section 3.5).
3. We present a design to handle decision weightings in generators that can be extended to other fuzzers through a composable and fully configurable selection manager. The design of these selection managers is described in Section 3.2 and the effectiveness of selection managers is evaluated in Chapter 5.
4. We demonstrate the effectiveness of RustSmith by:
   (a) Detecting bugs across different versions of the Rust compiler, including the latest released version, v1.61.0
   (b) Covering over 50% of the optimization module of `rustc` using just 1000 randomly generated programs
   (c) Covering "blind-spots" in 9 optimizations within `rustc`, which are areas the official test suite is unable to cover
5. The project is also fully open-source and available at `https://github.com/rustsmith`, which contains the generator along with the additional tools developed in the process.

2 Background

This chapter introduces some of the core concepts of compiler testing along with a brief introduction to Rust and the language features that are unique to Rust. Sections 2.1 to 2.4 explain the different known methods of compiler testing, along with the techniques used in fuzzers to aid code generation and ensure validity of test programs. Section 2.5 and 2.6 finish off the introduction to compiler fuzzing by describing the well-known test oracle problem in compiler testing, and explaining the methods used to reduce an interesting test-program down to simpler and smaller programs. Finally, Section 2.7 gives a quick introduction to the Rust language and specifically Rust's most unique features.

2.1 Compiler Testing

Compilers are pieces of software that translate a computer program written in a specific language into a low level language to create an executable of the program. Compilers are very complex by design, and typically involve a pipeline of interacting modules such as performing lexical analysis, parsing the source language into an abstract syntax tree, performing semantic analysis, optimizing the code, and finally generating the executable.

Most programming languages are *compiled languages* which mean they have at least one compiler built for the language's specification. As compilers are such important pieces of software, there is an active and growing interest in ensuring compilers are well tested in a way that allows bugs to be found, ideally before its release.

As such, compiler testing is an active research area that focusses on methods and techniques to pieces of software designed to test compilers and their implementation. The automated testing technique called "fuzzing" is the most common automated technique to test compilers, and involves producing random inputs for the compiler to try and compile.

Fuzzing is actively used for compiler testing for a number of reasons. Firstly, compilers are complex pieces of software, as they provide developers a plethora of features, support for different optimization levels and the ability to compile code for different target platforms. This results in a large configuration space for tests to try and cover, which makes it very difficult to ensure that all configurations are exhaustively explored in manually written unit test suites.

Secondly, due to the number of features a compiler provides, compilers translate and transform programs in ways that are typically not specified through a formal specification. Usually, there is no specification for compilers indicating when certain optimizations should be performed on certain expressions or statements. For example, the LLVM native compiler, Clang, has 58 different optimization passes [7], but there is no specification in C that tells compiler developers which optimization should be performed at which point.

Whilst compiler fuzzers vary in source language, the compilers being tested, and the input domain (with some fuzzers focusing specifically on a small subset of the language such as arith-

metic expressions [8]) they all follow the general design shown in Figure 2.1.

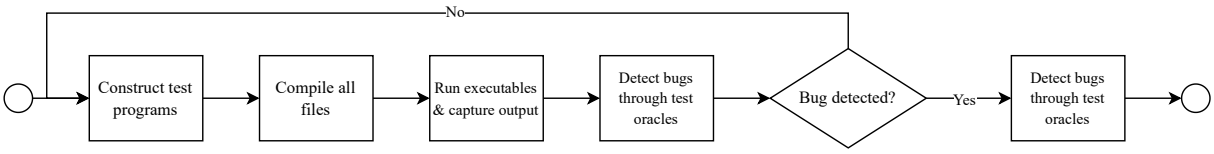

Figure 2.1: General outline for compiler fuzzers

Testing compilers through fuzzing usually begins with generating the inputs, which, in the case of testing compilers, involve constructing test programs in the source language itself. Typical approaches for constructing test programs are described in Section 2.2. The test programs constructed have to be both valid and diverse, and approaches for ensuring these properties are explained in Section 2.3 and Section 2.4 respectively. Once test files have been created, the files are then compiled into executables and run, with their outputs captured. The next stage involves taking the outputs and determining whether they indicate a bug or not, which involves solving the test-oracle problem, described in Section 2.5. If a bug is detected, test case reduction is performed to find the smallest possible program that still triggers the bug, which is described in Section 2.6. A reduced test program can then be reported to the compiler developers for them to ivestigate.

Types of Compiler Bugs

There are two main types of compiler bug that can occur: compiler crashes and wrong-code (or miscompilation) bugs. Compiler crashes can be discovered by generating syntactically and semantically valid test programs, and then checking whether the return code of compiling them is non-zero. Miscompilations however, are trickier to detect; in this case, the compiler succeeds in compiling a program but produces the wrong code for it. As optimization phases in compilers are the most complex phases in the pipeline, and involve the most transformations to the provided source code, miscompilations are more likely to occur and are most likely to be due to an optimization pass bug. This can be seen in other compiler fuzzers such as Csmith [9] and YARPGen [10] which have found most bugs to be in the "wrong-code" category when testing C compilers.

RustSmith focusses on finding miscompilation bugs and there therefore focussing on testing the optimizations in `rustc`. It has the capability to detect compiler crashes and flag them as bugs in the compiler, but is not specifically be designed to trigger crashes themselves.

2.2 Constructing Test Programs

Constructing fully-formed test-programs that are valid and "interesting" presents a variety of challenges in compiler testing. Compiler fuzzers have to make sure that test programs are valid, diverse and fit the requirements of the testing method being used. The two approaches that are commonly used to achieve this are known as *program mutation* and *program generation*.

2.2.1 Program Mutation

Program mutation involves taking in an existing program as an input and producing a mutated program from it through a series of transformations. Within program mutation, there are two broad categories: *Semantic preserving mutation* and *Non-semantic preserving mutation*.

Semantic preserving mutation aims to mutate the input program in a way that does not change the behaviour of the program. Most semantic preserving mutations are based on the general idea of equivalence modulo inputs (EMI) [11]. EMI is a concept that guarantees that, under a set of inputs, two test programs in the same language exhibit the same behaviour and therefore *should* produce the same output.

A good example is presented by Donaldson et al. called GLFuzz [12], which looks at applying semantic-preserving mutations to test OpenGL, a shading language for rendering vector graphics. Taking in an original shader, GLFuzz applies many semantically preserving transformations randomly resulting in a transformed variant shader. This variant shader may look programmatically completely different to the original program passed in, but *should* still result in a visually identical image being rendered. A bug may be detected when the two images rendered are significantly different. Interestingly, some of the transformations that could be chosen involve actually adding in new "dead code" (code that does not impact the final image in any way but still is valid code) showing that both program mutation and program generation techniques can be implemented together.

Alternatively, **non-semantic preserving mutations** mutate programs more freely as there is no requirement for the final program to have the same semantics as the original program. The main reason for doing this is to ensure that the final program is suitable for testing compilers. For example, mutations can occur to ensure the final program is free from undefined behavior. An example of this is Nagai et al.'s [8] method for testing arithmetic optimizations in C compilers. As the method has to guarantee the arithmetic expressions generated are free from undefined behavior, they heavily restrict the size arithmetic expressions can grow to prevent arithmetic overflow. In a later paper [13], this is then improved by applying some heuristics which mutate the expressions and use them to test C compilers. These mutations involve, for example, flipping the operands of an operation or inserting a new operation to ensure the denominator of a divide expression is not 0. This allowed the same tool to generate much larger arithmetic expressions which still had the guarantees of no integer overflow or floating point errors, allowing for many more bugs to be detected.

2.2.2 Program Generation

Program generation involves creating programs from scratch in order to test compilers. Chen et al. categorize program generation into two broad categories: grammar-directed and grammar-aided approaches [14].

Grammar-Directed Approaches typically take in the language's grammar as the input, and produce files by walking through the grammar via a top-down approach and generating literal strings to satisfy grammar rules. By recursively going through the language's grammar, syntactically valid code can be generated and used for testing. This is an effective approach for testing the lexical analyser and parser of a compiler, but falls short when trying to test compilers past those stages. For example, given a subset of a grammar of the form presented in Figure 2.2, selecting a statement randomly from the grammar could mean choosing the `break` statement. While this is a valid choice syntactically, break statements inherently only make sense in certain situations (e.g. inside loops).

```
statement := break | declaration | assignment | forloop
```

Figure 2.2: Example Grammar for statements

As a result of this, all files generated would be syntactically valid but many of them would not be semantically valid. While there has been some more work to try and make grammar-directed

approaches more context-sensitive in an attempt to widen test coverage [15], the grammar-aided approach presented below provides a cleaner way to handle context sensitivity when generating programs.

Grammar-Aided Approaches instead take a grammar and use some heuristics to address the context sensitivity. Most of these approaches work by having a skeleton in the format of a valid file for the given language and then generate code to fill in the gaps left in the skeleton.

A good example of the grammar-aided approach is Csmith [9], a successful compiler fuzzer for C, which, for each test program generated, starts by creating some top-level definitions such as types, global variables and functions. This stage also includes creating a random collection of struct types by randomly deciding on the number of struct members and their respective types (which can include previously defined structs). Once this is complete, the `main` function is then generated and the fuzzer continues to produce code with the exception of function calls. When the function call expression is randomly chosen, and the decision is made to generate a new function for this call (instead of calling an existing function), the generation of the main function is suspended until the new function has been fully created. Throughout generation, complex heuristics are used to avoid C programs that have undefined behavior (Section 2.3).

2.2.3 Comparing test program construction techniques

Both program mutation and program generation have proven useful in research; mutation based fuzzing relies on an initial program to then mutate through transformations, whereas program generation attempts to build up programs from scratch. While both methods are interesting in principle, Csmith's [9] success in finding C compiler bugs has motivated the decision to start by developing a generator for Rust programs.

2.3 Ensuring validity of test programs

Programs must first-and-foremost be valid pieces of code. Languages have requirements and constraints on when certain expressions, statements or constructs can be used, depending on the context. For example, as discussed in Section 2.2.2, it does not make sense for "`break`" to exist outside a loop of some kind. If that were the case, the generated output would simply be considered as syntactically invalid. Programs that are either semantically or even syntactically invalid can be effectively used to test the front-end of compilers such as the parser and semantic checkers. However, these programs provide little benefit when aiming to test the whole compiler, as with syntax errors, the compiler would never reach further than syntax and semantic checking (skipping optimization runs, translations, and writing to executables).

Creating code that is free from undefined behaviour is also critical. Undefined behaviour is the result of invoking erroneous operations within a language. For example, dereferencing a null pointer in C is deemed an undefined behaviour and so it's not safe to have code that dereferences a null pointer [16]. Execution of programs that contains code with undefined behaviour has no value when testing compilers as different compilers or even the same compiler with different optimizations levels may handle these erroneous operations in different ways which causes problems when trying to solve the test-oracle problem. This is why a large proportion of bug reports looking at results of different compilers as the difference is due to an undefined behaviour action.

Similar to undefined behaviour, unspecified behaviour should also be avoided when generating programs. Unspecified behaviour are parts of the language that have no specific behaviour defined, and therefore compilers are free to decide how to handle these operations. This is an issue in the area of compiler testing because when comparing outputs between compilers or

across different optimization levels, a difference in output could trigger a false positive, therefore advertising that a bug has been found when in fact it was due to the unspecified behaviour of a certain operation.

Most test generators have built in the checks to ensure code generated is free from undefined behaviour. For example, when generating an array access, Csmith [9] uses the modulo operator to ensure the index for the array access stays within bounds. These kind of checks are done throughout the generators for all expressions.

An alternative called "Program Reconditioning" has been proposed by Donaldson et al. [17] which aims to decouple the process of generating and ensuring validity of programs as shown in Figure 2.3. This means that expressions are created freely during the generation stage, but then the code is transformed in a way that ensures validity in a second pass. For example, all divide expressions must ensure that the denominator is not 0, to avoid divide by 0 errors. To achieve this, the divide expression the numerator and denominator can be freely generated in the generation phase, as the generation phase isn't responsible for this. Instead, the reconditioned phase will traverse the code and ensure that all division expressions first check that the denominator is not 0 before going ahead with the division.

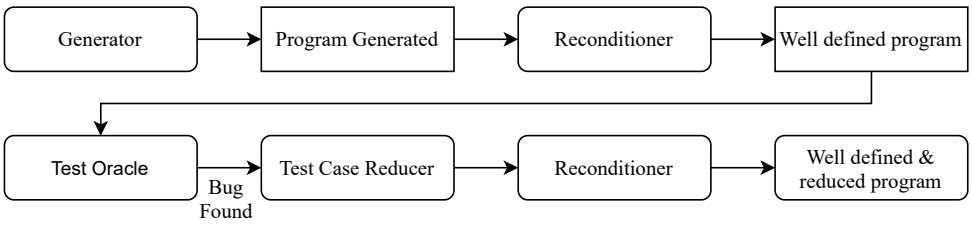

Figure 2.3: Decoupling program generation from avoiding undefined behavior through reconditioning [17]

This decoupling is especially useful later on as well during test case reduction which is discussed in more depth in Section 2.6. RustSmith incorporates a reconditioner (instead of ensuring UB-freedom at generation time), with the future intention to use it for test case reduction.

2.4 Improving code generation diversity

Given compilers are such important pieces of software, they are typically well tested and so detecting bugs using a fuzzer can be tricky to find. Compiler testers sometimes have to generate millions of random programs before any success. This is simply because with a language with so many features and options for expressions and statements, there are so many combinations to produce a final output that the probability of creating a bug-inducing file can be fairly low, depending on the types of bugs present in the compiler.

Ensuring test-programs are diverse is especially important due to the low probability of producing a file that produces a bug. Producing thousands of file that touch similar parts of the compiler provides very little value, and therefore code generators need to find ways to produce programs that can trigger more of the compiler such as triggering more optimizations.

Swarm testing [18] is specifically designed to improve the diversity of the test programs generated by a compiler tester. Swarm testing breaks the norm that all features are included and available when generating programs. For example, when generating an expression, most generators simply look at all possible expressions and pick one at random. Swarm testing instead uses a large set of randomly generated configurations, where each configuration specifies which features of the language should be omitted in the generation process.

The reason for moving towards a set of configurations over always having all features available is that some bugs are more likely to be found when looking intensively in a smaller subset of the language features, over trying to test all features of a language in one go. A good example described in [18] is simply testing a stack implementation involving two basic methods: `push` and `pop`. If a bug exists after 30 push operations chained together, testing the implementation with both operations together would mean that bug would be detected with a probability of $\frac{1}{2^{30}}$ (assuming equal probability of picking an operation). However, if swarm testing was adopted, four configurations are possible for the stack and the configuration including only the `push` operation would give a very a high chance of triggering the bug.

Swarm testing has been found to improve the coverage of compilers and detect up 42% of bugs in C compilers compared to a heavily hand-tuned default configuration including all features. Swarm testing has been introduced into RustSmith in an attempt to improve the diversity of test programs produce.

2.5 Test Oracles

A common concept in software testing are test oracles, first introduced by W.E Howden [19], which are simply ways to determine whether a test has passed or failed.

Oracles are used in test scenarios by comparing the output(s) of the software being tested to the output(s) the oracle determined the software should have produced. A "perfect" oracle would be able to determine what the output should be for any test-case inputs for the piece of software. However, determining the output for a given input is a very hard problem and is known as the "test oracle problem".

With regard to compiler testers, there are two main approaches developed that attempt to address the test-oracle problem: *metamorphic testing* and *differential testing*.

2.5.1 Metamorphic Testing

Metamorphic testing is an approach invented by Chen et al.[20] as one way to solve the test-oracle problem. It involves finding *metamorphic relations*, which specify how particular transformations to the input affect the output for the program under test.

A common example is testing the `sin` function available in most programming languages to 100 significant figures. It is difficult to determine what the correct value of sin(1) should be to 100 significant figures, but the mathematical property $sin(x) = sin(2\pi + x)$ can be used. Therefore, the test simply needs to make sure that $sin(1)$ produces the same value (to 100 significant figures) as $sin(2\pi + 1)$.

Several approaches for metamorphic relations have been proposed but by far the most popular approach is equivalence relations, which establish that two programs are equivalent under some assumptions after mutation.

One interesting form of equivalence relation testing is Equivalence Modulo Inputs (EMI) introduced by Le et al. [11]. The process is as follows. For a given program and it's test inputs, the EMI method produces variants of the program that should produce the same output. The compiler will then compile the original program along with it's variants. Because the variants should behave in the same way as the original program under the same test inputs, the EMI method detects a bug if any one of the outputs are not the same.

2.5.2 Differential Testing

Differential testing was proposed by McKeeman [21] as a way to solve the test-oracle problem for complex pieces of software. Within the field of compiler testing, differential testing involves compiling test-programs in different ways and comparing the outputs to look for discrepancies. There are typically three strategies for compiler differential testing: cross-compiler, cross-optimization and cross-version strategy.

Cross-compiler strategies detect compiler bugs by compiling the same source test-program across different compilers built for the same specification and language. A bug has potentially been detected when the outputs of the test-program are not all the same across all the compilers tested. A drawback of this strategy is that it requires multiple compilers for the same language and specification version, which is an issue for newer programming languages that only have one compiler.

Cross-optimization strategies involve using **one compiler** but compiling the test program with different optimization flags and then again testing to see if the outputs of the test-program are all the same or not across the different optimization levels. This strategy is the most widely used strategy current used in the field of compiler testing.

Similarly, cross-version strategies involve using different versions of the same compiler as the varying parameter to detect bugs. The method of differential testing is demonstrated in Figure 2.4.

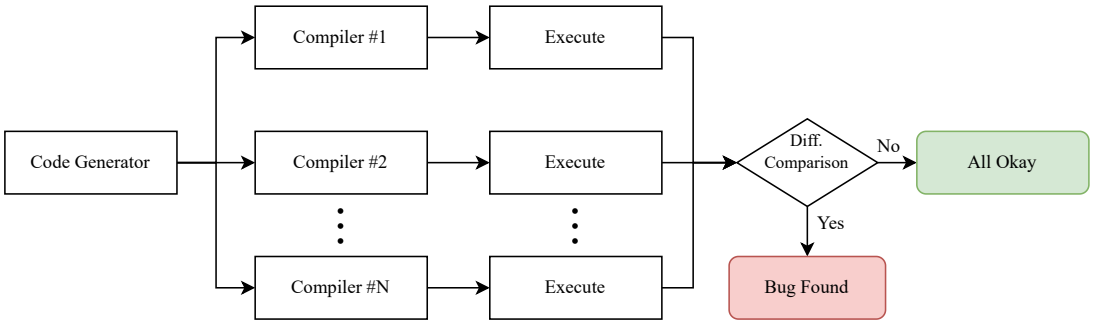

Figure 2.4: Differential Testing (cross-compiler)

All three strategies along with Equivalence Modulo Inputs (EMI) have been used in different compiler-testing research [14] and therefore it's important to understand which strategy should be used when and their advantages and disadvantages.

2.5.3 Comparing Test Oracle Techniques

J.Chen et al. [22] studied cross-compiler (referred to as randomized differential testing (RDF) in the paper), cross-optimization (referred to as different optimization levels (DOL)) and EMI techniques to better understand their uses and advantage. This was done by using Csmith [9], a successful random program generator for C and then testing both approaches on GCC and LLVM, recording the compiler bugs found during 90 hours of testing. When comparing the 3 methods, they looked into how many programs can be tested in a given period of time (giving an idea of efficiency of the method), as well as which technique can detect more bugs (giving a metric for the strength of them as test oracles).

It concludes that using the cross-optimization technique was more effective at detecting optimization-related bugs and cross-compiler bugs are more effective at detecting optimization-irrelevant bugs. With regards to efficiency, DOL is the most efficient technique and EMI is the

least efficient. EMI is also the weakest oracle (RDT was the strongest). Finally, it concludes that efficiency has the largest impact on the effectiveness of compiler testing and therefore the DOL approach is best suited for this.

As there are parallels in the way Csmith generates programs and the way RustSmith does, and because currently the only Rust compiler available is rustc (developed by Rust themselves), the sensible approach would be to go with the DOL approach as it is the most efficient, and therefore has the biggest impact on the effectiveness of the compiler-tester.

There has been some work in combining multiple differential testing strategies together, such as Sun et al.'s [23] technique which uses all 3 strategies together. As right now this is not possible for Rust because there aren't multiple compilers to compare results with, this is not explored but is considered as future work.

2.6 Test Program Reduction

Unfortunately, the test programs that find bugs tend to be fairly large and complex. For example, Chen et al. found that in Csmith [9], the largest number of bugs were found in a program of size 81KB. Therefore, if/when a bug has been detected through a generated test program, it's critical to find the smallest block of code that triggered the bug detected. This is important because developers of these compilers will not be able to fix the bug found by looking at a large and complex file, they instead want reporters "*...to narrow down the bug so that the person who fixes it will be able to find the problem more easily...*" [24].

To tackle this, a common post-processing step for the results is "Test Program Reduction" which aims to take in a test program, and attempt to make the program both simpler and smaller, whilst still inducing the bug initially detected. There's currently a range of test-case reducers available from very language-specific such as glsl-reduce and Fast-reduce which are for GLSL and C programs generated by Csmith respectively to C-reduce and Picire which are language-agnostic. This means that to tackle test program reduction for RustSmith there are 2 choices: create a Rust specific test program reducer and build it around RustSmith or use language-agnostic test program reducers.

The reason language-specific reducers are made is the fact that reducing programs can reintroduce undefined behaviour into the program. To combat this, Fast-reduce for example leverages domain specific knowledge to combat any undefined behaviour that may be introduced during reduction. Therefore, when language agnostic reducers are used, the output has to then be run through a "undefined behaviour analyser" which is a lot of work to fully analyse the program.

However, with the introduction of reconditioning by Donaldson et al. [17] as described first in Section 2.3, language-agnostic reducers can be used freely. As the reconditioner has the freedom to physically change the program to work around any possible undefined behaviour, any analysis of the program is not required. Moreover, the reconditioner built for ensuring validity of programs during generation can be used exactly as-is, meaning that with no extra work, the same code can be used during generation and in test-case reduction.

RustSmith therefore aims to build a reconditioner for the programs it generates from the start so that the same reconditioner can be used for test case reduction of RustSmith generated programs, alleviating the need to build a custom test-case reducer for RustSmith that can still preserve validity of programs.

2.7 The Rust Language

Rust is an upcoming and popular language used by large corporations. Matsakis [25] explains which features of Rust make Rust so unique from other languages, and so this section aims to provide the relevant background to Rust specific features that are implemented in the generator. This section provides a quick introduction to these concepts along with examples describing these features in action.

2.7.1 Ownership

Ownership [26] in Rust is one of the unique features which enables Rust to provide strong memory safety guarantees without the need for any garbage collector or any memory freeing through manual calls to `free()` (like in C).

Ownership in Rust enforces that each value in reach has a variable attached to it called the *owner* inside a *context* such as a scope. Values can only ever have one owner at a time, and when the owner goes out of scope, the value will be dropped, and its location in memory freed.

This applies both to variables placed on the stack and on the heap where the memory is automatically returned as soon as it goes out of scope. For heap stored variables, a `drop` function is called automatically as soon as the scope exits. While this basic concept may seem simple, more complicated situations with multiple variables and assignments may make the reasoning less clear.

When passing variables to functions during a function call, the variable's context moves from the current scope to the function's scope, therefore changing the owner. This means that (unlike most other languages) the caller cannot continue using the variable it passed to the function as its ownership has changed. While this is a simple model that can be checked on compile time to produce extremely safe code, it is extremely impractical from a development perspective and so Rust provides 2 exceptions to the context-switch rule.

One exception is for any type that implements the `Copy` trait. Types either implement the `Move` trait or a `Copy` trait (the default trait is `Move`). These types are copied for an argument of a function call instead of moving the ownership, allowing the original variable to be used after the function call. By default, primitive types like `i32` implement the `Copy` trait as they are very cheap to copy, but programmers can make any type they wish implement the `Copy` trait too. By default, heap allocated types do not implement the `Copy` trait.

Figure 2.5 (taken inspiration from Rust documentation [26]) shows examples of how ownership changes for `Copy` types and `Move` types. The string§ `heap_string` is a heap allocated variable of type `String` (which by default does **not** implement the `Copy` trait). Therefore, when passing `heap_string` to `takes_ownership` the owner for this String is moved from the main scope to `takes_ownership`'s scope, meaning that from line 12 onwards, `heap_string` simply isn't valid and any usage of it would cause `rustc` to throw a compiler error. By contrast, `x` is of type `i32` which has the `Copy` trait implemented, which means that `some_integer` has the copied value of `x` and the *owner* of `x` stays within the main scope. Therefore, `x` is valid from line 15 onwards and can be used freely.

```rust
fn takes_ownership(some_string: String) {
    println!("{}", some_string);
}

fn makes_copy(some_integer: i32) {
    println!("{}", some_integer);
}

fn main() {
    let heap_string = String::from("Heap Allocated String");
    takes_ownership(heap_string);
    // Referencing heap_string after this would lead to a compile error
    let x = 5;
    makes_copy(x);
    // x is valid here as x was copied for makes_copy
}
```

Figure 2.5: Rust Ownership Example (inspired by [26])

2.7.2 References and Borrowing

The second exception to the ownership rules defined above are references which can be created like in C++ for example, where instead of the resource itself explicitly, the reference is passed as the argument to the function. Rust also distinguished between mutable and immutable references and therefore references must be explicitly declared as mutable if the resource needs to be changed by the function. By default, references are immutable.

Rust calls the action of creating and passing a reference to a function *borrowing* as the function borrows the variable, and its value and returns it when the function call completes. During compile time, `rustc` checks for reference borrowing through a *borrow checker*, which will throw an error at compile time if references are being used wrong or if variables are being used without owning them.

```rust
fn main() {
    let s1 = String::from("Heap Allocated String");
    change_string(&s1);

    // s1 is valid and useable from even after the function call
    println!("The updated String is '{}'.", s1);
}

fn change_string(some_string: &mut String) {
    // Without "mut" this line would not succeed as it mutates some_string
    some_string.push_str(", updated!");
}
```

Figure 2.6: Rust References Example (inspired by [26])

Figure 2.6 illustrates how references allow borrowing of variables to occur. `s1` starts by being owned by the main scope, and a then **mutable** reference is created to pass into the `change_string` function. This does not change the ownership of `s1`, it simply means it is now "mutably borrowed". It is important that the `mut` keyword was added to the function definition of `change_string` as without that, `push_str` (on line 11) would not succeed as references by default are read-only. A standard reference (of type `&String`) would be able to access the string

and read properties from it such as `s.len()` but mutations to the variable are not allowed. The *borrow checker* implemented by `rustc` also checks whether immutable references are being used correctly during compilation.

Semantic rules for Mutable and Immutable References

Rust has very strict rules concerning references and how they both work together. The first and most important rule is that a reference cannot live longer than the owner of the resource being borrowed. Whilst this is a known rule within programming languages, as this leads to dangling references or pointers, the Rust compiler will (via the borrow checker) ensure that this isn't the case.

Next, for a given resource, it cannot be referenced both mutably and immutably at the same time. Therefore, only one of the following two options are allowed:

- There is exactly one mutable reference to a resource
- There are one or more immutable references to a resource

Importantly, with mutable references, this actually means exactly one *live* mutable reference at any point.

For example, the following example is valid:

```
1  fn main() {
2      let mut x: i32 = 100i32;
3      {
4          let y: &mut i32 = &mut x;
5      }
6      let z: &mut i32 = &mut x;
7      let a: &mut i32 = &mut x;
8      println!("{:?}", a);
9  }
```

Figure 2.7: Rust Mutable Reference Example

In Figure 2.7, even though there are two mutable references to `x` within the program, because `y` is no longer live after by the end of its scope (at line 5), `x` can again be mutably referenced as is done on line 6. More subtly, a resource can be mutably borrowed in the same scope too, as long as any previous references are no longer live. Therefore, the second mutable borrow within the same scope (on line 7) **is valid as long as z is never used after this point**.

2.7.3 Partial Moves

An extension of Rust's move semantics are **partial moves**. Partial moves occur in container types such as structs and tuples. When an element with a `Move` trait inside these types are accessed through a tuple or ttruct access expression (as in Figure 2.8), the tuple or struct is now partially moved.

```
1  fn main() {
2      let tuple_1 = (100i32, String::from("abc"), String::from("123"));
3      let tuple_2 = tuple_1;
4
5      // This print is INVALID as tuple_1 is now invalid
6      println!("{:?}", tuple_1);
7
8      // This does not change the validity of tuple_2 because tuple_2.0
9      // is an i32 which implements the Copy trait
10     let x = tuple_2.0;
11
12     // This now moves tuple_2.1 out of tuple_2 and into y, making
13     // tuple_2 now partially valid
14     let y = tuple_2.1;
15
16     // This now moves tuple_2.2 out of tuple_2 and into z
17     let z = tuple_2.2;
18
19     // This print is VALID as y owns the string "abc"
20     println!("{:?}", y);
21
22     // This print is INVALID as tuple_2 is partially invalid (although
23     // tuple_2.1 still is valid)
24     println!("{:?}", tuple_2);
25 }
```

Figure 2.8: Rust Partial Move Example

Figure 2.8 shows partial moves in action. A tuple's move semantics are derived implicitly from the move semantics of the elements inside it. For example, if all the elements inside the tuple derive the `Copy` trait, the tuple itself also has `Copy` semantics. However, in the case of `tuple_1` in Figure 2.8, it derives the `Move` semantics as at least one element also derives the `Move` semantics (the String in element position 1 and 2). Therefore, on line 3, the tuple is moved from `tuple_1` to `tuple_2`, making `tuple_1` invalid.

Next, accessing certain elements with different move semantics affects the tuple itself in different ways. When the first element is accessed (on line 7), the validity of `tuple_2` does not change because the first element is copied into `x`. When the second element is accessed (as is done on line 11), the String "abc" is now moved out of `tuple_2` and into `y`. This means that `tuple_2` is now only partially valid as it only owns some elements inside the tuple. Therefore, the tuple in its entirety cannot be borrowed or read, but other elements within the tuple are still accessible, as shown when accessing the string "123".

As RustSmith aims to produce correct and valid programs, it must generate programs with all the concepts presented such as ownership (including respecting copy, move and partial move semantics), immutable and mutable references, and borrowing in a correct way such that the *borrow checker* in `rustc` does not find any issues and successfully compiles the test-program.

2.7.4 Lifetimes and lifetime annotations

Lifetimes

Lifetimes go hand-in-hand with the borrowing semantics described in Section 2.7.2. Lifetimes is the concept used by the borrow checker to ensure rules concerning borrowing are respected in the code being compiled, by checking references are valid for as long as they are needed to be.

With every reference (immutable or mutable) created, a **lifetime** is associated with it, which is the scope that the reference is valid in. The borrow checker then compares the lifetime of a reference with the scope of the resource being borrowed and ensures that the resource lives longer than the reference itself.

The following example shows a small program annotated with the lifetimes of each variable:

```
fn main() {
    {
        let reference;              ---------+-- 'a
        {                                    |
            let x = 5;              -+-- 'b  |
            reference = &x;          |       |
        }                           -+       |
        println!("{}", reference);           |
    }                               ---------+
}
```

Figure 2.9: Rust dangling reference example with variable's lifetimes shown (inspired by [26])

Here, `reference` has the lifetime `'a`, and `x` has lifetime `'b`. As shown on the right, `'b` is "inside" `'a` and therefore the reference is living longer than the resource, violating the first rule described in Section 2.7.2.

RustSmith therefore has to ensure (through lifetimes) that there are no lifetime violations as it generates the code.

Explicit Lifetime Annotations

In some cases, the Rust compiler (specifically, the borrow checker) requires help in understanding the lifetime of certain references and more importantly how they are related to other references.

Lifetime annotations are added to references inside function and struct definitions in the following way:

- The type of an immutable `i32` reference with an explicit lifetime annotation is written as `&'a i32`
- The type of a mutable `i32` reference with an explicit lifetime annotation is written as `&'a mut i32`

This is in contrast with reference types with implicit lifetimes, which are written as `&i32` and `&mut i32` for immutable and mutable references respectively.

Lifetime annotations on their own mean very little, they are only needed and useful when multiple references are using the same annotation. Furthermore, lifetime annotations don't change the lifetime of the variables itself, it simply introduces constraints that the borrow checker ensures aren't broken.

For example, when structs have reference types (both mutable and immutable) within them, explicit lifetime annotations are **required**, as illustrated in Figure 2.10:

```
struct Coord<'a1> {
    x: &'a1 i32,
    y: &'a1 i32
}
```

Figure 2.10: Rust explicit lifetime annotations in structs

Here, both `x` and `y` have a lifetime of `'a1`. What this indicates is that the struct `Struct1` should live at least as long as both `x` and `y` and that `x` and `y` live at least as long as the shortest true lifetime that `'a1` is instantiated as.

To explain this further, Figure 2.11 illustrates cases where the lifetime annotation impact relationships between references inside a struct.

```
fn main() {
    let x: i32 = 10i32;
    let y: i32 = 1i32;
    let p = Coord { x: &x, y: &y };
}
```

(a) Valid program using `Coord` from Figure 2.10

```
fn main() {
    let x = 3;
    let r;
    {
        let y = 4;
        let p = Coord { x: &x, y: &y };
        r = p.x;
    }
    println!("{}", r);
}
```

(b) Invalid program due to using the same lifetime parameter in both `x` and `y`

Figure 2.11: Example illustrating the relationship between lifetimes

Note that both examples in Figure 2.11 use the `Coord` struct defined in Figure 2.10.

`Coord` is defined with the same lifetime parameter and therefore any instance of `Coord` must live as long as the longest living reference between `x` and `y`. The program in Figure 2.11 (a) depicts a basic program where this condition holds. Both `x` and `y` live for the same amount of time (as they are instantiated in the same scope), and therefore the instance of `Coord` is valid as it lives as long as both `x` and `y`.

However, in Figure 2.11 (b), `x` is defined in the top-level scope, and `y` is defined in a block scope later. When the `Coord` is instantiated on line 6, it requires both `x` and `y` to have the same lifetime (due to the relationship enforced in Figure 2.10). However, when `r` is assigned `p.x`, the reference `&x` created on line 6 is being forced to live as long as `r`, which is also in the main scope. Forcing `&x` to be live in a higher scope, in turn forced `&y` to be live in the same (higher) scope, which is impossible as `y` is killed on line 8. This renders the program invalid.

This can be fixed by simply allocating different lifetime annotations to `x` and `y` in `Coord` as shown in Figure 2.12.

```
struct Coord<'a1, 'a2> {
    x: &'a1 i32,
    y: &'a2 i32
}
```

Figure 2.12: Rust explicit lifetime annotations in structs

Note that without the variable `r`, the code would have been valid as Rust allows for *lifetime coercion* [27], so that a longer lifetime (in this case `x`'s lifetime) can be coerced into a shorter one so that the lifetime rules work.

3 Design of RustSmith

This section provides an in-depth insight into design decisions made in the project that form the core of the RustSmith generator. Section 3.1 describes the overall project setup, including the overarching generation and scope management techniques. Sections 3.2 and 3.3 then describe two techniques implemented in RustSmith to solve problems with probability weightings and generation failure respectively. Generation of Rust-specific features such as ownership, lifetimes and borrowing are described in Sections 3.4 and 3.5, and finally all other notable generation techniques are described in Section 3.6.

3.1 Project Setup and Approach

Just like most languages, Rust is built up from three building blocks:

1. **Types**: such as `i32`, `String`, `struct Struct1`, `(i32, String)`, etc.
2. **Expressions**: such as literals (`100i32`), add expressions (`exp + exp`), if-else statements (`if (bool) { exp } else { exp }`), function calls (`fun10(100i32, String::from("abc"), x)`) etc.
3. **Statements**: such as declarations (`let var1:type = exp`), expressions as statements (`exp;`), return statements (`return exp`) etc.

It is important to note that the vast majority of Rust is simply expressions (including conditionals and loops) as all expressions can form statements with the addition of the semicolon. Therefore, the bulk of the generation is concerning generation of expressions.

The RustSmith generator is tasked with producing valid code using these three building blocks.

3.1.1 Generating types, expressions and statements

The generator at its core simply tries to build an abstract syntax tree (AST) in a top-down approach using these three building blocks. It constructs programs by starting at the top node (known as the `ProgramNode` in the implementation) which triggers the creation of a main function and its body. Whilst generating the `main` function, it may "step out" and create structs or other functions, producing a full program. Once the main function's generation has completed, the AST is simply "pretty-printed" by recursively converting each AST node into its Rust equivalent syntactical form.

The generation of these blocks can be described at a high level with the following pseudocode methods which will be utilized throughout:

- `generateType()` generates a random type. This method is recursive and therefore can produce types such as `(i32, String, (f64, Struct1))` (which features a tuples within another tuple)

- `generateExpression(type)` generates a random expression for the given `type`. Again, this function is recursive and therefore may internally call `generateExpression` to produce the chosen expression.

- `generateStatement()` generates a random statement. The generation of a statement usually requires the generation of a type or expression first, and therefore may call `generateType()` or `generateExpression(type)` respectively.

All three methods internally go through the same process. First, a list of available *productions* are found for the particular situation the generator is in, both contextually and based on the grammar. For example, within `generateExpression`, a list of all the available expression productions are collected, which are all the kinds of expressions available in the current context. Mostly, the list will contain all the relevant kinds of types, expressions or statements RustSmith provides, but in certain scenarios some options may be omitted. For example, unless the generator is generating expressions inside the body of a loop expression, the `break` expression is omitted from the available expressions. Next, RustSmith randomly selects one production from the list and finally the chosen production is generated and added to the AST.

Expression generation is directed by the `type` parameter passed in, and therefore only the kinds of expressions that can produce that particular type will be included in the list of available expressions. For example, the option to generate an add expression should not be considered if the type to be generated is a boolean.

Finally, RustSmith also includes an internal node known as `StatementBlock`. This is simply a linear list of statements that are used to produce bodies of statements for functions, if-else blocks, block expressions etc. When generating statement blocks, `generateStatement()` is called for each statement to be generated one at a time.

3.1.2 Keeping track of variables and scopes

While generating the program top down, information needs to be stored "on the fly" about variables, functions and structs so that they can be used throughout the program. Therefore, inspired from how compilers work, a symbol table is created to hold information about existing variables, functions and structs throughout generation.

Symbol Table

The symbol table is implemented as a tree to handle scoping in programs. The top of a function's symbol table houses all the arguments passed in along with their relevant type.

After the root node, each node in the tree holds information for the given scope in a map from variable name to variable information. Variable information includes information about the type, whether the variable is mutable, and the current validity state of the variable (see Section 3.4). Queries for variable information in the symbol table are made from the leaf node up to the root, following the scopes back up to the function's scope.

An example of the symbol table for a given program is shown in Figure 3.1.

```
1  fn main() {
2      let var1: i32 = 100i32;
3      let mut var2: f64 = 0.123f64;
4      let var3: String = String::from("abc");
5      {
6          let var4: bool = true;
7          let var5: i64 = 150i64;
8          let mut var6: bool = false;
9      }
10     if (var1 > 112i32) {
11         let var7: String = String::from("123");
12         let mut var8: i32 = var1;
13         let var9: bool = false;
14     } else {
15         let var1 : bool = true;
16         let var10: String = String::from("123");
17         let mut var12: bool = var1;
18     }
19 }
```

Figure 3.1: Rust Scoping & Symbol Table Example

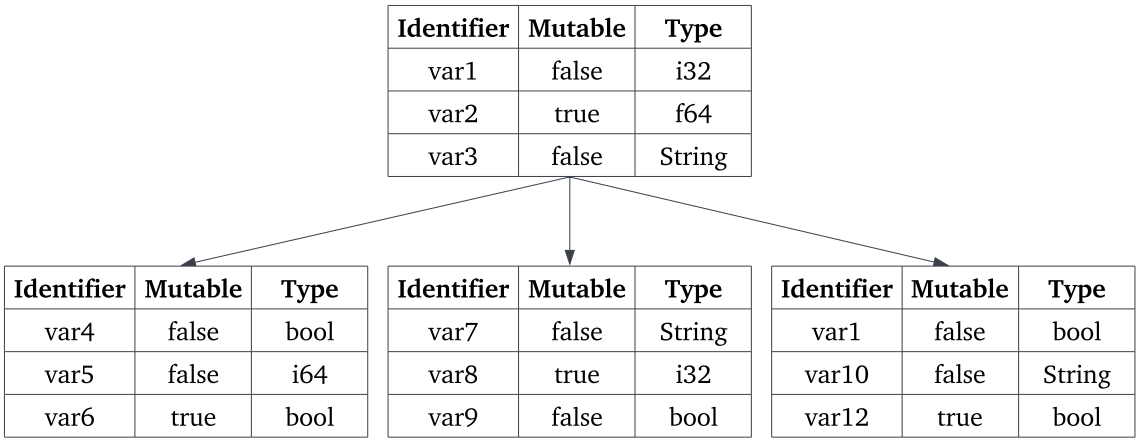

Figure 3.2: Associated Symbol Table for Rust Program

The root of the tree represents information about variables in the `main` function's top scope. From then on, every new scope creates a new node to hold information about variables for the new scope. The symbol table holds information about each variable such as its mutability and the type of the variable.

When generating the RHS for the declaration of `var12` (on line 17), `generateExpression(boolean)` chose the `Variable` Expression node to be generated. The variable `var1` was found as a valid option for a boolean variable due to the bottom up approach of querying the symbol table. The `var1` on line 15 was found before the `var1` on line 2 meaning `var1` was a valid candidate to produce a variable node.

One common alternative design for symbol tables makes use of a stack instead of a tree. On entering a scope, an empty symbol table is pushed onto the stack, and popped on every exit of scope as the symbol table is not queried after exiting scope. However, as seen in Section 4.1.1 later, another pass of the AST is required during reconditioning, and therefore symbol tables

may be queried again to determine the type of variables. This is the motivation for using a tree for the symbol table in RustSmith.

Global Symbol Table

The root of each symbol table is connected to a single parent known as the "global symbol table". The global symbol table holds information globally available such as function and struct definitions, along with any globally-defined constants. When generating function calls or struct instantiations, the global symbol table is queried to find a relevant function or struct type respectively.

In addition to the function and struct definitions, the global symbol table also holds information about tuples. While this may seem unnecessary as tuples do not require any definition above the `main` function in a Rust program, they are stored during generation to restrict unbounded generation for the tuple type.

Given all the types available in Rust, the number of tuples that can be generated randomly is infinite. If generation of the tuple type produced a random combination of the types available, the chances of two distinct variables of the tuple type having the exact same internal types would be incredibly small. This causes issues when struct members or function arguments are tuples, as instantiating structs or generating function calls respectively would mean no tuples are available from existing variables in scope.

Therefore, the global symbol table also houses a list of tuples that have been generated so far. This allows the generator to select from one of the available tuple types when generating the tuple type.

3.2 Weightings & Selection Managers

3.2.1 Decision-making during generation

The generation of programs can be seen as a decision tree where, at every point, a decision from a selection of options has to be made. The decision taken guides the generation of the program and ultimately dictates the types of programs generated.

Decisions are made throughout such as:

1. Deciding whether to make a variable mutable or not during its declaration
2. Deciding whether to create a new struct or tuple, or use an existing struct or tuple respectively
3. Deciding whether to create a new function or use an existing one
4. Choosing randomly from a list of available kinds of types, expressions or statements
5. Choosing the number of elements to include in a struct or tuple, or the number of arguments when generating a function's signature
6. Choosing whether to create a new statement when generating a `StatementBlock`

These decisions were initially built in as a **uniform choice** across all the possible options. This meant each binary decision was simply a "coin-toss", and every node available had an equal chance of being chosen at any situation.

This approach, however, has its shortcomings. Deciding between all options with an equal weighting meant that program generation could enter infinite recursion. For example, when

generating an add expression, both the left-hand side (LHS) and right-hand side (RHS) require expressions too. If either of those expressions chose another "recursive expression" such as the subtract expression from the available options, it in turn also requires two expressions. With no changes to the probability of choosing recursive expressions, there is a chance that recursive expressions are chosen over and over again, resulting in infinite recursion. Instead, recursive expressions should become increasingly less likely as the depth of the generating expression increases.

The opposite is also required in certain scenarios. When all options are equally likely, program generation sometimes terminates very quickly without creating many (or in some cases, any) statements in the `StatementBlock`. Therefore, at the start of the program, statements such as declarations and if-else expressions should be more likely, to increase the size of programs being generated.

After experimenting with the uniform random distribution, the decision was made to design a more sophisticated system to fully control and configure these decisions with parameters, allowing independent **strategies** to be made. Specifically, the system was required to:

1. Have a data-structure to hold information about the **context** of which part of the code is being generated. Information from this data-structure will be used as parameters for the weightings of expressions, types and statements.
2. Fully control the weightings of expressions, types and statements through a **configurable and parameterized framework** along with any decisions to be made during generation.
3. Be capable of **composing these strategies together** to form different strategies. This will allow generation to be targeted in certain ways. For example, "aggressive" strategies may build on-top of an "all-round" strategy and then simply configure one expression to become a lot more likely to be chosen, allowing that particular expression to be aggressively tested. This will also make it possible to implement swarm testing [18] (mentioned in Section 2.4) by building on top of the all-round approach and simply setting the weightings of some expressions to 0.

3.2.2 Keeping track of context

To accomplish the data-structure that stores the context for generation, an immutable (read-only) object is passed around throughout all generation methods. This `Context` object holds certain statistics about the AST generated so far, along with specific information relating to where in the AST the generation is taking place. As `Context` is immutable, any updates are made by returning a new instance of `Context` with the required updates made.

Some of the information `Context` keeps track of is:

1. Number of functions, structs and tuples defined
2. Number of declarations in current scope
3. And most importantly, the **depth** of statements, expressions and types

The structure is chosen to be immutable because generation of an AST node is mostly done in a depth-first fashion. For example, when generating an add expression, the generation of the entire LHS of add is done first and then the generation of the RHS begins. Using the same `Context` object would mean that after generating the LHS node, any changes to depth would have to be removed as the LHS and RHS are generated from the same level.

Most of `Context`'s information is derived from the symbol table (including the `GlobalSymbolTable` for function, structs and tuple information). `Context`'s main purpose is

to store and provide **depth information**.

Depth is tracked within the context by storing counters for AST nodes in a map describing how "deep" the generator is with respect to different nodes. These counters are incremented with an `incrementCount` method. As mentioned above, `Context` is an immutable object, and therefore `incrementCount` returns a new `Context` object.

Finally, `Context` also holds extra information that is important to be passed down during generation. For example, when generating the body of a function, `Context` will store the return type required for the function so that if a `return` statement is chosen, an expression of the correct return type is generated.

3.2.3 Selection Managers

The information `Context` gathers during generation paves the way towards creating a fully configurable and easy-to-use method to control the generator's decisions, which in turn control the types of files generated themselves.

To achieve this, "Selection Managers" are created to implement these different strategies. Each selection manager is a single implementation of a strategy, which simply implements all the decisions the generator is required to make throughout. The context is passed into all these decision-making functions, although there is no requirement to use information the `Context` object has been collecting.

Decisions are made by assigning weights to all the possible options and then randomly selecting one of them by using those weights as a weighted probability. These weights can be constants, or they can be in the form of equations by using one or more pieces of information from `Context` to construct it.

Another important property required by Selection Managers was "composability". Creating a composable system allows for Selection Managers to build up from one another to produce more interesting strategies.

Some of RustSmith's Selection Managers are shown in Figure 3.3 in Figure 3.3.

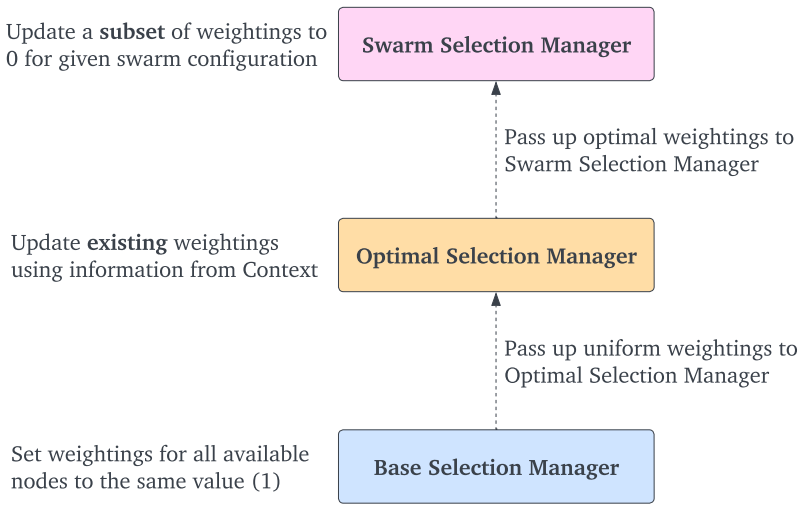

Figure 3.3: Diagram depicting selection managers in RustSmith

A uniform random selection manager is first created as the base of the composable system

(known as the `BaseSelectionManager`). This selection manager sets the weighting for every option to be 1, meaning that all options are equally likely to be chosen. It also implements any filtering of the nodes based on the context. For example, it will only present the `BreakExpression` to be an option to be chosen when inside a loop expression. This is detected by using the depth counters for loop expressions in the `Context`.

Next, an all-round selection manager (known as `OptimalSelectionManager`) is created that builds on top of the `BaseSelectionManager`. This strategy aims to thoroughly test out all aspects of the expressions, types and statements implemented in RustSmith, by changing the weightings passed up from `BaseSelectionManager`. Importantly, any strategy that builds upon another strategy cannot make a new option available - it can only change weightings. For example, if the `OptimalSelectionManager` wishes to set the weighting of `BreakExpressions` to 10, it will only be set if it is available in the first place, which `BaseSelectionManager` decides based on the context.

Finally, to implement swarm testing [18], a selection manager is created that builds on top of the `OptimalSelectionManager`. By looking at a subset of expressions available, the swarm testing selection manager picks a random subset to remove as options and therefore simply sets the weightings to 0 for those expressions, regardless of what weightings any underlying selection manager assigned to that expression.

3.3 Directed vs. Fail-Fast node generation approach

During generation of certain nodes, there will be scenarios where the generation of the chosen node may not be immediately possible. A simple example of this is generating an assignment statement. If an assignment is chosen as the first statement in the `main` function, there are no variables declared and therefore no variables in the symbol table, resulting in no variables available to assign to.

To solve this, two approaches are implemented and available for the user: a directed approach and a fail-fast approach.

3.3.1 Directed generation approach

The premise of the directed approach is that when a certain node is chosen for generation, that particular node is guaranteed to be created and will not fail to be created.

To achieve this, the generator is able to suspend the current generation, and step out to generate any dependencies. Once they are completed, the original generation resumes and the node can be completed.

For example, when generating an assignment node in the scenario described (as the first statement in the `main` function with an empty symbol table), the generator suspends current generation and first creates a declaration to inject above the current statement. The assignment node can then be completed by assigning to the variable just made. This is done in multiple different situations; for example, when a function call expression is chosen with no functions defined with the required return type, the generator again steps out of generating the function call expression, generates a full function, and then returns to complete the function call.

One limitation of injecting statements such as declarations above is that these statements can only be injected within the same scope. This is because injected statements in higher scopes cause issues with the semantics of some Rust features such as mutable borrows as illustrated in Figure 3.5.

```
1 fn main() {
2   let mut x: i32 = 100i32;
3   /*let mut_x: &mut i32 = &mut x;*/ -----+
4   let y: &&mut i32 = {                   |
5       x = 10i32;                         |
6       &<&mut i32 variable required>-----+
7   }
8 }
```

Figure 3.4: Directed generation approach limitation when requiring variables from previous scopes. `<&mut i32 variable required>` denotes the position where a mutable reference to `i32` is required, and the arrow shows where the potential injected declaration would be included

This simple program illustrates where the directed approach can produce invalid code. First, `x` is created as a mutable `i32` variable, and then y's type is decided to be `&&mut i32`. The block expression is then chosen for the RHS of the declaration, which contains a reassignment to x's value. When the final expression is chosen for the block expression (on line 6), first a reference expression is chosen, and then a variable node is chosen.

As there are no variables with the type `&mut i32`, a declaration for such a variable has to be created. Crucially, the variable must be declared before y, along with being declared in the same (or higher) scope `y` is in to avoid any lifetime issues. If the declaration in line 3 is injected into the existing code ,however, this makes the program invalid and results in a compilation failure. This is because looking at the program from top-down, it shows `x` being mutably borrowed at line 3, and then assigned to at line 5, which is illegal for mutably borrowed resources as explained in Section 2.7.2.

While there are different ways of handling this problem, most methods would lead to a requirement to eventually "give-up" and try a different expression. Therefore, to handle these situations, the directed approach will temporarily fall back on to the fail-fast approach, described below, until the expression generation is over.

3.3.2 Fail-Fast generation approach

The fail-fast [28] approach for software is an approach built upon the idea that software should fail as soon as possible, throwing an exception if something is wrong during object instantiation.

The fail-fast approach within RustSmith uses this idea by throwing exceptions as soon as a situation arises where the generation of a node is not possible. Instead of suspending generation to generate a node's dependencies, the fail-fast approach quickly fails when scenarios arise where nodes cannot be created.

For example, for the situation with the assignment node with no available variables in the `main` function, the fail-fast approach simply throws an exception. When this occurs, the assignment node is removed from the list of available statements and another statement is chosen at random. This will continue until one of two situations occur: either a valid statement is found from the selection, or no valid statement can be generated. If no valid statement can be found, then it means the program cannot be generated any further and therefore an exception is thrown and the program is regenerated. Note that while this example concerned statements, the same logic applies for both expressions and types.

The absence of a valid expression can cause a statement to not be generated too. For example, if the declaration statement is chosen, but no valid expression can be found for the RHS of the

declaration, the declaration statement itself throws an exception and removes itself from the available statements.

Rolling back changes after failing

This fail-fast approach, however, requires a system to *rollback* changes when exceptions are thrown due to Rust's ownership rules (described in Section 2.7.1). Expressions can change the state of certain resources by invalidating them due to, for example, movement to another variable. However, if that expression is then part of a rejected expression or statement, the change to the resource state should be reverted to its original state. This is illustrated in the example in Figure 3.5.

```
1 fn main() {
2     let mut x: String = String::from("abc");
3     let z = {
4         let y = x;
5         <EXCEPTION THROWN>
6     }
7 }
```

Figure 3.5: Fail-fast rollback example (`<EXCEPTION THROWN>` denotes the case where no expression could be found for the returning value of the `BlockExpression`)

When the declaration of `y` occurs at line 4, it changes the ownership of string `"abc"` from `x` to `y`. However, when the returning expression of the block expression could not be generated (on line 5), the `BlockExpression` is removed from the list of expressions and another expression is attempted for generation. However, the issue is that within the symbol table, `x` was invalidated when line 4 was generated, and therefore a rollback mechanism is required so that `x` becomes valid again.

To achieve this, inspiration is taken from database snapshots from an SQL database [29] which takes an independent, read-only copy of data in case rollbacks are required. Similarly, before any generation of an AST node, a snapshot of the whole symbol table is taken and stored. The generation then proceeds as before, but when an exception is thrown, before trying another available node, the symbol table is rolled back by using the data from the snapshot. This sets the ownership states for every variable to the state before the attempted generation of the node.

3.4 Handling Rust's ownership semantics

The following two sections describe the design decisions taken for ensuring the generator respected the Rust specific semantics that make it different to other programming languages, such as ownership, borrowing and lifetime parameters.

To respect the ownership semantics for objects as described in Section 2.7.1, extra information must be stored in the symbol table in order to track the ownership state for each variable.

The state of each variable's ownership is tracked using an enum with the options `VALID`, `PARTIALLY_VALID` and `INVALID`. Two additional options are later added to handle borrowing semantics (namely `BORROWED` and `MUTABLY_BORROWED`), however these semantics and states are discussed in more detail in Section 2.7.2.

As expected, declarations of variables set the variables to `VALID` and therefore are available from the declaration's point onwards. Next, ownership semantics for variables are determined by their

types, and so for any given type, a mapping is present from its type to one of `MOVE` or `COPY`. For a variable with `COPY` semantics, the variable's validity stays at `VALID` regardless of how many times the variable is used. However, when a variable is being used that has move semantics, its validity will shift to either `INVALID` or `PARTIALLY_VALID`, depending on the scenario.

When a variable is used inside a "partially moving expression" such as a struct or tuple element access, the variable itself is set to `PARTIALLY_VALID`. In any other case, it must be used in its entirety and therefore its state moves to `INVALID`.

Keeping track of the ownership at only a variable level as described above would ensure valid generation of Rust programs regarding ownership. However, this would be quite a conservative approach since the example with partial moves shown in Figure 2.8 could never be generated. Only keeping track of ownership states at a variable level won't let the generator know specifically which parts of the tuple or struct are partially moved over others and therefore after one partial move, none of the other elements would be available for access. Therefore, for structs and tuples, information about each element is also stored to know specifically which elements have been partially moved to allow for others to still be accessed.

When finding possible variables to use during generation, variables with different ownership states are available for different uses. When moving a variable or an element inside a struct or tuple, the object must be `VALID`, but when a variable or element is being **assigned to** it can be any of `VALID`, `PARTIALLY_VALID` and `INVALID`.

3.5 Borrowing and Lifetimes

The next major Rust-specific language feature concerns borrowing and lifetimes. Due to the number of rules and constraints required to produce valid Rust code with references, novel techniques have been developed to address each rule.

3.5.1 Immutable and Mutable Reference Rules

To begin with, the rules for handling mutable and immutable references are considered, which are the following:

1. Either there is exactly one mutable reference to a resource
2. Or there are one or more immutable references to a resource

To implement this, the states for resources described in Section 3.4 are extended to include 2 more states: `BORROWED` and `MUTABLY_BORROWED`. Once a resource is moved to either of these states, the resource is simply no longer available for mutable borrows. However, if the resource has been immutably borrowed, the resource is still available when generating immutable borrow expressions.

While leaving it at this would produce valid Rust programs regarding these two rules, it would again be a fairly conservative approach. As shown in Figure 2.7, once the scope ends for a reference, the resource is available to borrow again. However, the current approach would not *revert* the changes to the resource's state when the reference's scope ends, thus never making the resource available for referencing again.

To implement this, a resource's state is stored in the symbol table, using scope depths implicitly as a stack. The symbol table, as described in Section 3.1 is designed as a tree, but can be perceived as a stack when looking from a leaf node up to the root, where the leaf is the top of the stack, and the root is the bottom of the stack. The stack represents the resources' state in different scopes, where the top of the stack represents the "current" state of the resource.

The state of the resource in higher scopes are underneath the top of the stack so that when the scope ends, the stack is popped and the previous state is available. However, when a resource becomes `INVALID` (or `PARTIALLY_VALID`) in any scope, the resource then becomes invalid in **every scope**. Therefore, for "overriding states" such as `INVALID` and `PARTIALLY_VALID`, once a resource's state moves to these states, the whole stack is overridden with the particular state, as even after exiting any number of scopes, the state must still be `INVALID` or `PARTIALLY_VALID`.

To make this clearer, an example is shown in Figure 3.6, along with a representation of `x`'s stack at different points in the program in Figure 3.7.

```
1  fn main() {
2      let mut x = String::from("abc"); // (1)
3      {
4          let y = &mut x;              // (2)
5      }
6      {
7          let z = &mut x;              // (3)
8      }
9      {
10         let a = x;                   // (4)
11     }
12     println!("{:?}", x);             // (5)  -- COMPILATION ERROR
13 }
```

Figure 3.6: Example Rust program to showcase how RustSmith handles borrowing

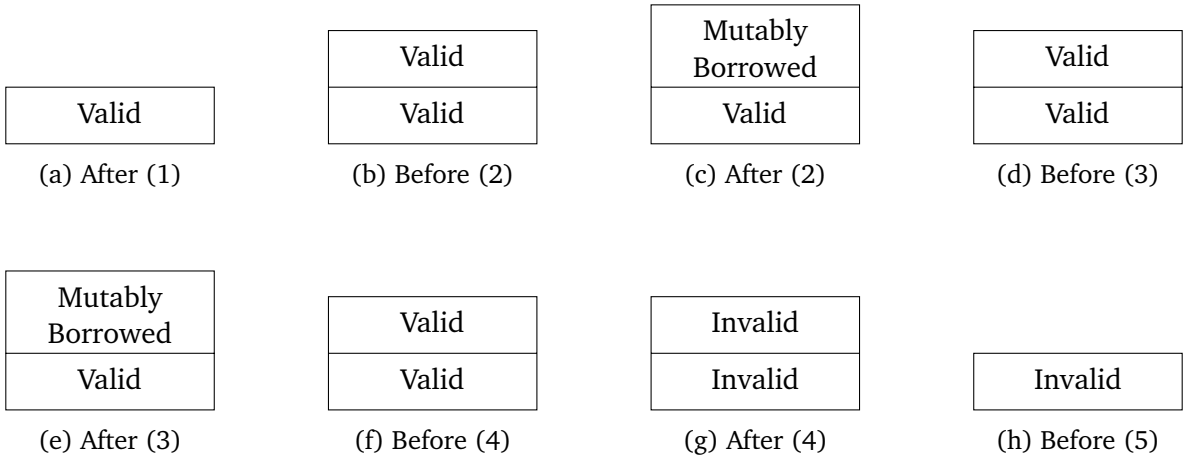

Figure 3.7: The ownership stack for `x` while generating the program. (n) denotes the different checkpoints at which the stack representation is shown

After `x` is declared on line 2, as expected, the validity state for `x` is set to valid and therefore the stack has one value with the state `VALID` on it. Next, entering a new scope at line 3 causes the stack to have the top of the value copied over and pushed onto the top (shown in (b)). After the mutable reference expression of `x`, the state of resource `x` changes **only in the current scope** to `MUTABLY_BORROWED` (shown in (c)). Once this scope is over however, the top of the stack is popped off and `x` is back to simply being `VALID` again. Therefore, mutably borrowing `x` is possible again, therefore allowing (3) to be possible and changing the state to `MUTABLY_BORROWED`.

Exiting the scope on line 11 makes `x` once again valid, and therefore allows for `x` to be moved. When this occurs inside another scope on line 10, the stack for `x` changes from (f) to (g). This is

because once the move is complete (g), `x` is no longer valid in any scope, which is why the stack has been overwritten with `INVALID`. This is why, even after exiting the final block expression on line 11, `x` is still `INVALID` causing the print statement on line 12 to be a compilation error.

Note that while this method enables generation of mutable references to be less conservative than never mutably referencing a resource again, it still has shortcomings. The rules for mutable references actually concerns liveness of mutable references more than scoping itself, and therefore two mutable references expressions to the same resource can occur in the same scope as long as the initial mutable reference is no longer live (as shown in Figure 2.7). Modifying the generator to handle this case proved to be tricky as, when generating "top-down", the liveness of resources are never determined, as in the future the generator may randomly decide to use a particular resource. Therefore, the compromise was made to use scopes instead of explicitly the liveness of references.

3.5.2 Generating lifetime-respecting programs

The second important rule regarding lifetimes involves ensuring the constraints of lifetimes are respected in the generator. This essentially means ensuring both dangling references never occur, and that lifetime annotations are included when required and respected in the code generated.

Ensuring references don't outlive the resource

To ensure references don't outlive the resource themselves, a **lifetime counter** is implemented. This lifetime counter essentially represents the depth of a given scope w.r.t. the root top level body of a function. Each scope therefore is assigned a lifetime value, which are assigned to variables during their declaration. Therefore, the generator needs to ensure that any reassignments to a variable with a reference type ensures that the variable is available from the variable's lifetime value or above. An example should help illustrate this further, as shown in Figure 3.8.

```
1  fn main() {
2      a1': {
3          let mut a = 100i32;
4          a2': {
5              let mut reference = &mut a;
6              let mut b = 150i32;
7              a3': {
8                  reference = &mut b;
9                  reference = &mut a;
10                 let mut c = 10i32;
11                 reference = &mut c; // INVALID CODE
12             }
13         }
14     }
15 }
```

Figure 3.8: Example demonstrating RustSmith's generation technique for generating lifetime respecting programs

Figure 3.8 is a standard Rust program with extra syntax before the start of block expressions to label scopes with lifetime parameters. Going through each declaration at a time, the declaration of variable `a` is declared inside the `a1'` scope block and therefore has the lifetime value of 1. After the next block is created (with lifetime value 2), the `reference` variable is declared which is given a lifetime value of 2.

To find the RHS for the declaration, the `Context` object is used to specify what the **lifetime value requirement** is for the expression being generated. The value passed in is 2, which means that for the expression generated, any resource being used for the mutable reference expression must have a lifetime value of 2 or less. Therefore, the variable `a` is found as a valid expression as `a` has a lifetime value of 1, which is valid given the constraint.

When `a3'` is created, `reference` can be reassigned again to `&mut b` as `b` has a lifetime value of 2, and `reference` (still) has a lifetime value of `2` as well. Therefore, they live as long as each other which is still valid for references. Importantly, in reassignments, the lifetime value of `reference` does not change, and therefore `a` can still be a valid RHS for `reference`. Finally, the final assignment on line 11 is invalid as it doesn't fit the constraints required. The RHS has a lifetime value of `3` (due to its declaration occurring in `a3'`) but `reference`'s lifetime value is still 2.

Note that there will be cases in which there simply are no resources available in the program that will satisfy the constraints required from lifetimes, in which case an exception is thrown through the fail-fast mechanism described in Section 3.3.2.

Adding explicit lifetime annotations when required

Adding explicit lifetime annotations now becomes a fairly simple task with the *lifetime value requirement* presented above in place. Structs are created "on demand" (i.e. when the requirement is there for a struct with specific types to be created) and therefore is created as a by-product of creating a struct instantiation. Structs however require explicit lifetime parameters of any references to be included in its definition. To provide these lifetimes, the *lifetime value* assigned to every reference is used, and therefore, after generation of a Struct's first instantiation, the lifetime values of the references inside it are used.

Crucially, the next time the same struct is used to instantiate another struct object, the lifetime requirements already in place from the first definition is used. If no references can be created for the lifetime values present in the struct, the struct instantiation fails. This ensured that structs are only ever instantiated when the lifetimes of the references match the depths they were first created with. While this method ensures validity for struct instantiations, it is conservative as it doesn't exercise Rust's *lifetime coercion* [27] that allows longer lifetimes to temporarily shorten so that requirements match as explained in Figure 2.11.

Functions can also optionally have lifetime parameters added onto them based on the arguments' lifetime values, but this is currently omitted, and the compiler therefore performs *lifetime elision* [30] on the parameters to infer the lifetimes of each reference passed in.

3.6 Other Generation Techniques Used

This section describes some remaining interesting techniques used to generate different parts of Rust programs.

3.6.1 Functions

Given the techniques already described, implementing functions was a fairly trivial task. Each function is provided with a new symbol table tree, simply with the arguments populated into the root table. A single parent is then created from which the body of the function can be created. This therefore keeps the function arguments and any other variables created in the function body in separate tables.

This decision was chosen mainly for handling lifetime calculations as described in Section 3.5.2. If both function arguments and variables within functions had the same **lifetime value**, function arguments that were mutable references could be reassigned to variables that live only inside the body of the function. This would mean the reference outlives the reference, throwing a compilation error.

Mutable references passed in as function arguments hold onto the references that they were called from, therefore ensuring that lifetimes will match when reassigning to mutable references.

Additionally, *inline attributes* (such as `#[inline(never)]` and `#[inline(always)]`) are added randomly to the top of functions which indicate to the compiler whether to inline functions. This is so that the inline.rs optimization, specifically tasked on inlining parts of the code in rustc, is exercised in as many different ways as possible.

3.6.2 Declaration and type inference

The generation of declarations follows the expected approach. First, the type of the declaration is decided by generating a random type. Next, the variable name is decided, and finally a random expression is generated for the given type. The pretty-printed Rust equivalent is therefore produced in the following form:

```
let [VAR]: [TYPE] = [EXP]
```

However, to test `rustc`'s type inference capabilities, sometimes, the type annotation is omitted and is therefore left for the compiler to infer what the type of the variable is. This technique has recently proven useful in finding numerous type inference bugs in popular JVM languages such as Java, Kotlin and Groovy [31].

3.6.3 Volatile variables through command line arguments

Finally, as done with functions as well, forcing the compiler to explicitly not optimize portions of code can prove useful as it can trigger special cases in the compiler which may be more subtle and potentially less tested. CSmith [9] does this by utilizing `volatile` variables in C. However, Rust has no equivalent of volatile variables and therefore instead, command line arguments are used. Command line arguments are not available to the compiler at compile time and therefore cannot be considered for optimization. Additionally, using command line arguments in larger blocks such as for the predicate of if-else statements would mean that no dead-code elimination could occur for the branches.

To achieve this, the generator can decide to generate a "command line argument access" expression which essentially extracts a particular argument from the command arguments and converts the string into the type required. Generation of these arguments also generates the required literal to a list of literals that need to be passed in as command line arguments when executing the file.

4 Implementation and Deployment

This chapter describes details of RustSmith's implementation (Section 4.1), along with the additional tools used to validate programs (Section 4.2), view results and statistics (Section 4.3), and deploy RustSmith to perform large-scale testing (**??**).

4.1 Implementing RustSmith

The core generator (which simply generates Rust programs independently) is implemented in Kotlin [32]. Kotlin was chosen specifically for it's ease of use, rapid development capabilities, along with its null-safety capabilities.

Additionally, for RustSmith's use cases in particular, Kotlin's object-oriented capabilities allowed for a heavy use of subclasses which is required in the AST nodes, along with the advantage of inspecting these class hierarchies during runtime. For example, for the selection managers described in Section 3.2, it allowed for the capability to adjust the probability for "Recursive Expressions" as a whole to a new weighting by adjusting all subclasses of the `RecursiveExpression` interface through inspection of the class hierarchy. Inspection of the hierarchy is also used to build the `BaseSelectionManager` so that all the available types, expressions and statements can be found, and their weighting can be set to 1.

More concretely, RustSmith can generate programs with the following features:

- Most basic expressions and statements used in Rust programs
- Global functions with function body and function arguments and return types, along with *inline attributes* such as `#[inline(never)]`
- Control flow expressions such as: if/else expressions, function calls, loops, break and return expressions
- Signed and Unsigned integer and floats types with different widths such as `i8`, `i32`, `i64`, `f32`, `f64` etc.
- Arithmetic, bitwise and logical expressions on integers and floats such as `a > b`, `a << 2`, `a + b` etc.
- Struct and Tuple types, including nested structs and tuples
- Arrays of the types available (again including nested arrays)
- Command line arguments to mimic `volatile` variables in C, forcing Rust to not optimize them
- References (and dereferencing), both mutable and immutable along with lifetime annotations on structs
- Reading and writing of heap allocated data using `Box` expressions

An example generated file can be found at https://gist.github.com/mayanksharma3/4c5eba5bdaa05dedb5e11c39bbf92b4b

Usage

RustSmith, once built, simply produces an executable that can produce files randomly for testing the Rust compiler.

The user can decide whether to write the generated files to stdin or to a file. In addition to the Rust files generated, an additional file called inputs.txt is created which is a space separated list of literal expressions generated alongside the file. When validating the files, `inputs.txt` is "piped" in to the executable after compilation.

Programs are produced with a custom `Random` instance, which has a specific seed set. Therefore, each file is produced with a random seed, which can then be passed in to generate the same file again.

The choice of selection managers can also be passed into `rustsmith` as arguments. The default selection manager is the `OptimalSelectionManager` (as described in Section 3.2), but this can be changed and multiple can be passed in so that files are generated with different managers in a round-robin fashion.

4.1.1 Reconditioning

Reconditioning, as described in Section 2.3, is implemented in RustSmith as a second pass of the generated AST to ensure programs are free from undefined behaviour. The "reconditioner" traverses the AST, and wraps expressions with special AST nodes where the expressions require some form of reconditioning as they may exhibit undefined behaviour. For example, when an add expression of 2 integers is present, the reconditioner will simply wrap the add expression node with a `ReconditionedAdd` expression.

Reconditioned expressions are then added to the final program as *macros* which are then reused throughout the program. An example macro for a safe division is shown in Figure 4.1.

```
macro_rules! reconditioned_div {
    ($a:expr,$b:expr, $zero: expr) => {{
        let denominator = $b;
        if (denominator != $zero) {
            ($a / denominator)
        } else {
            $zero
        }
    }};
}
```

Figure 4.1: Reconditioned Division Macro

This macro simply checks whether the denominator of the division expression is zero or not. If it is, then the division simply returns 0, and if not, the division is performed as expected.

The reconditioner is specifically made as a separate module that is independent of the generator itself. Decoupling this from the generator is done so that the same module can be used in the automated reduction process (as reducing a program can (re)introduce undefined behaviour), as described in Section 2.6. Note that while reconditioning is implemented, Rust has a fairly

limited set of undefined behaviours and therefore no rust-specific reconditioning methods were required.

4.2 Validating Rust programs against `rustc`

In addition to generating standalone programs, a tool was created (called `rustsmith-validator`) to test the generated program against the rust compiler across different optimization levels.

Rust has 5 different optimization levels that fall under 2 categories, optimizing for speed, and optimizing for size. Optimization levels 1, 2 and 3 optimize for speed with optimizations applied more aggressively the higher the number. Optimization levels `s` and `z` optimize instead for size. Note that there is the default optimization level 0 which applies no optimizations.

Therefore, the tool simply compiles the program 6 times (1 time for each of the 6 optimization levels, and then once with no optimizations) and the compiler `stdout` and `stderr` output (if any) is written to a `compile.log`. Each of the 6 executables produced (assuming they all compiled) are then executed and the output from `stdout` and `stderr` is written to an `output.log` file.

Finally, for a given file, the 6 produced `output.log` files are compared for any differences and the result of the run is reported. The validator is capable of processing through a folder with many files in them, and utilizes thread pools to process the files in parallel (the default is to use 4 threads).

4.3 Viewing and debugging results

While not essential, a helpful tool to visually understand and view the results of multiple files easily was developed, called `rsmith-viewer`. At its core, it simply is a web server that reads from a folder of files generated by RustSmith and `rustsmith-validator` and combines them to show results and files in a clear and easy to understand way. The tool is invoked by `rsmith-viewer` and is shown in Figure 4.2.

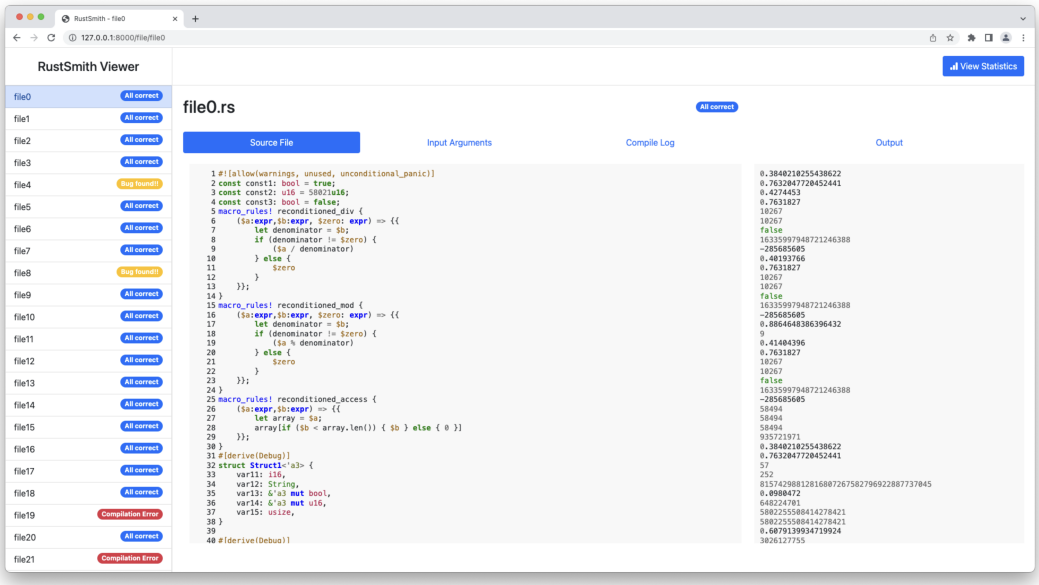

Figure 4.2: RustSmith Viewer Screenshot

All of the generated files can be seen on the left, with the status of each file shown (i.e. whether a bug was detected in the output, or a compilation error occurred etc.). Selecting a program from the panel on the left allows the user to view the generated source file in full, and can switch through viewing the source code, the input arguments passed in (if any) and the compilation log (if any). Finally, on the right-hand side, the output produced when the source file was compiled and run is shown so that the outputs can be compared. This allows the user to clearly see, across a large list of generated files, the status of all the files.

Additionally, the `rsmith-viewer` also has a separate page which allows users to view statistics about the files generated by RustSmith. The screen show a variety of statistics and graphs about the files generated, along with the execution time. It shows statistics such as the average file size, average number of lines, average execution time, and a couple of graphs showing histograms for the size of programs and execution times. Below the histograms, it also gives statistics about how often different AST Nodes were used along with other information such as standard deviation, quartiles etc.

A screenshot of this page is shown in Figure 4.3.

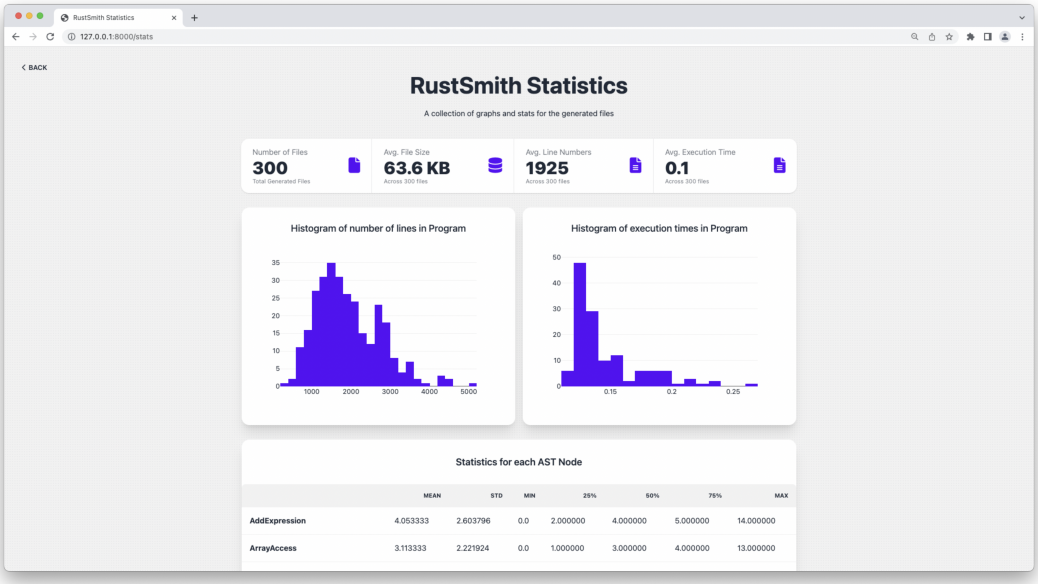

Figure 4.3: RustSmith Statistics Screenshot

These statistics have been actively used throughout the development of RustSmith to fine-tune selection managers such as `OptimalSelectionManager`, along with being used during evaluation for different generation policies including swarm testing.

4.4 Deploying RustSmith for large scale testing

Finally, fuzzing in compilers requires a large-scale testing campaign and therefore there was a requirement for an infrastructure to deploy RustSmith and test in a large-scale and distributed method. The requirements for such a system were as follows:

1. The tester must be able to **test files in parallel**, with a clear and easy method to scale-up testing in parallel
2. The tester must be able to test **multiple versions at the same time**. This is required to

evaluate the performance of RustSmith as it should be capable of finding more bugs in older versions of `rustc`

3. Finally, bug inducing files should be **put aside and collected** after the large-scale run so that they can be reduced and further inspected.

Therefore, the final tool, called `rustsmith-tester` was created to orchestrate the setup and deployment of this system. The architecture for the tester is shown in Figure 4.4.

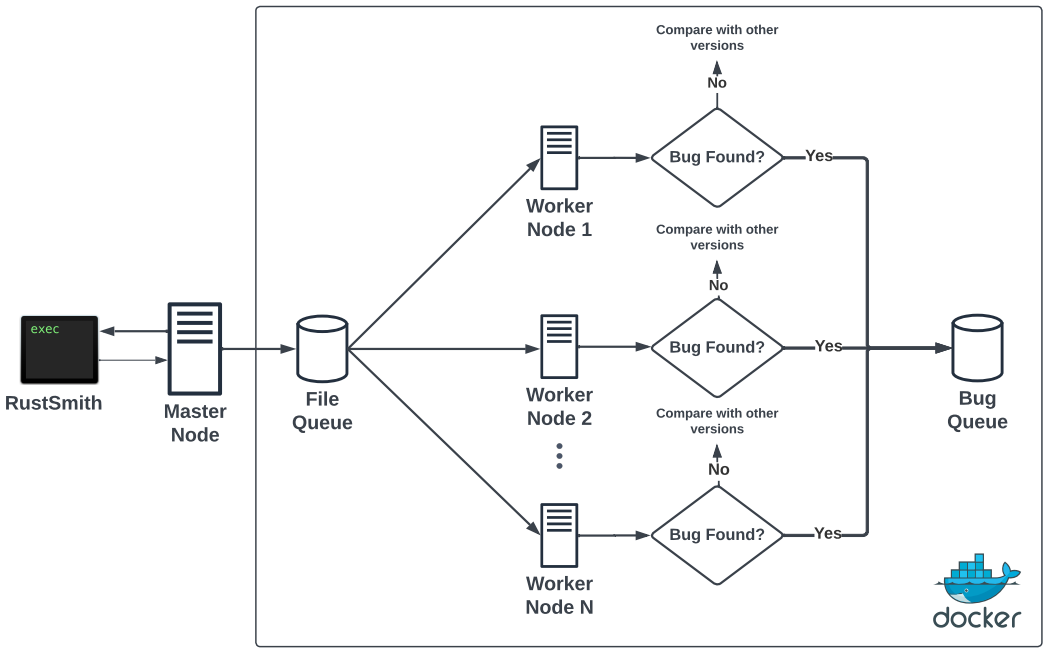

Figure 4.4: RustSmith Tester System Architecture Diagram

Figure 4.4 illustrates how this large-scale system works. It follows the leader-follower design pattern [33] which usually involves one leader node delegating its tasks to many follower nodes so that the tasks can be completed in parallel. In `rsmith-tester`, the **leader node** uses the latest RustSmith binary to generate Rust programs. These files are then put onto a separate queue outside the follower. For this, beanstalk is used [34] as the queue as it's known for its speed, ease of use, and concurrency guarantees (which is essential due to the parallel nature of the system).

Within an instance of a beanstalk, different "tubes" can be created which represent different queues. Therefore, each version to be tested is given a dedicated tube, and the leader node simply enqueues each generated file into all the different tubes.

Follower nodes then poll from the required node, based on which version of `rustc` has been assigned to them. The file is then validated (in the same way `rustsmith-validator` validates file as explained in Section 4.2), and then, if a bug inducing file is found, it is simply put into a separate bug queue. If no difference in output was found, the output is compared with other versions of `rustc` being run, so that cross-version checking can occur too.

The startup of such a system is governed by a simple configuration file of the form shown in Figure 4.5.

```
{
    "1.56.0": 3,
    "1.61.0": 3,
    "latest": 2,
    "nightly": 2
}
```

Figure 4.5: RustSmith tester configuration example

This config simply indicates how many follower nodes are required for testing each version of rust. Versions can be concrete numbers or simply the "latest" or "nightly" version of `rustc`. This config is also used to create the different "tubes" in the file queue for the leader to enqueue files in.

The majority of this system was deployed in practice using Docker [35] and docker containers. Beanstalk queues were deployed using the `schickling/beanstalkd` image and the followers were deployed using modified rust images that were first custom-built before running. These images were built up from `rust:[VERSION]` images (the official rust images with the tag obtained from the config) and simply included the logic for pulling from the queue, validating the file, and putting the result in the bugs queue (if needed).

Whilst this is currently deployed through docker containers, the same system is capable of running follower nodes across different physical servers too, as long as the queue is available across the network for reading and writing from.

5 Evaluation

This chapter focusses on evaluating both RustSmith's techniques along with RustSmith's overall performance. The evaluation of RustSmith attempts to answer the following research questions:

RQ1 How effective is RustSmith in its ability to find bugs (both current and historic) in the Rust compiler?

RQ2 To what extent of the Rust source code is RustSmith capable of covering (specifically the optimization module within `rustc`)?

RQ3 How has the addition of novel techniques and features impacted RustSmith coverage capabilities of `rustc`?

RQ4 How effective are RustSmith's selection managers in impacting the kinds of programs generated?

RQ5 How do RustSmith's failure approaches affect the performance of RustSmith and the kinds of programs generated?

The following evaluation techniques are performed in an attempt to answer these questions. Section 5.1 describes and analyses the bugs RustSmith across different versions of `rustc` to answer **RQ1**, and Section 5.2 evaluates the coverage of the compiler's source code to answer **RQ2**. **RQ3** is investigated in Section 5.3 by evaluating the coverage of RustSmith by using different "checkpoints" in the development of RustSmith to mimic ablation studies, which help identify the impact different techniques, improvements or features make to the performance of software. Finally, the last two research questions are answered by investigating two major internal techniques used within RustSmith. Section 5.4 evaluates selection managers to answer **RQ4** and Section 5.5 evaluates the two failure approaches described in Section 3.3.2 to answer **RQ5**.

5.1 Evaluating bugs discovered (RQ1)

The following section highlights some historic and current bugs RustSmith was able to detect as a result of generating files and using the large-scale testing system explained in Section 4.4.

5.1.1 Bug 1: Runtime crash due to non-inlined recursive function

The first bug analysed is one found in the version 1.61.0, which, at the time of writing, is the latest version of `rustc`. The bug results in a runtime crash when the program is optimized on any optimization level. The original bug-inducing file generated by RustSmith was ≈1200 lines (the original file can be found at https://gist.github.com/mayanksharma3/4c5eba5bdaa05dedb5e11c39bbf92b4b), and then was manually reduced to the snippet in Figure 5.1.

```rust
#[inline(never)]
fn fun25(var574: Box<i32>) -> ! {
    fun25(var574)
}

fn main() {
    let var574 = Box::from(1745183449i32);
    fun25(var574);
}
```

Figure 5.1: Non-inlined recursive function bug in `rustc` 1.61.0

Compiling the code in Figure 5.1 with no optimizations enabled correctly exits with a stack overflow error. However, when compiled on any optimization level (by running: `rustc -C opt-level=s bug1.rs` for example), running the produced binary will throw a `Trace/breakpoint trap ./bug1` error.

The assembly for the program in Figure 5.3 is inspected using Godbolt [36] on an optimization level in an attempt to work out what has occurred. The resulting assembly is shown in Figure 5.2 using the GNU Assembly Syntax.

```
example:fun27:
    pushq   %rax
    callq   *example:fun25@GOTPCREL(%rip)
    ud2

example:main:
    ud2
```

Figure 5.2: Assembly for bug 1 when compiled with `opt-level=3`

The assembly in Figure 5.2 includes an `ud2` instruction on lines 4 and 7, which is an instruction specifically made to generate an invalid opcode. It is the `ud2` instruction that causes the runtime crash when executed.

This issue was first detected by `rustsmith-tester` on the latest version of Rust (at time of writing), and further investigation found that the bug wasn't always present in previous `rustc` versions. It is present in versions 1.59 to 1.61, but the bug was not present in `rustc` before then. A GitHub issue [37] was created to report the bug, although it was linked to a duplicate issue someone else reported two weeks before it was detected by RustSmith. The issue has since been fixed in the nightly version of `rustc`, although the latest released version of rustc still has the bug present. The bug was fixed by fixing a bug in LLVM, which was then pulled into the rust source code.

5.1.2 Bug 2: Runtime crash due to LLVM loop optimizations

The second bug analysed originates from infinite loops, which, when used in a certain way, results in a runtime error on any optimization level (compiling and executing with no optimizations turned on does not crash). After generating the ≈3000 file program that detected this bug (which can be found at: https://gist.github.com/mayanksharma3/71b2f0fc6b837cde6336f9a361f0e562), the program was then manually reduced to the code in Figure 5.3.

```rust
fn fun1() {
    loop {
        loop {
            let mut var184: i32 = 90807803i32;
            var184 = 123123912i32;
        }
    }
}

fn main() -> () {
    fun1();
}
```

Figure 5.3: LLVM Loop Optimization Bug

Running the code in Figure 5.3 through `rustc` with no optimizations results in an infinite loop (as expected). However, when compiled on any optimization level (by running: `rustc -C opt-level=2 bug2.rs` for example), running the produced binary will throw a `Trace/breakpoint trap ./bug2` error.

The assembly again is investigated in an attempt to work out what has occurred. The resulting assembly is shown in Figure 5.4 in the GNU Assembly Syntax.

```
example:fun1:
  jmp   .LBB0_1
.LBB0_1:
  jmp   .LBB0_1

example:main:
  pushq  %rax
  callq  *example::fun1@GOTPCREL(%rip)
  popq   %rax
  retq
```

(a) Assembly for unoptimized program

```
example:fun1:
.LBB0_1:
        jmp     .LBB0_1

example:main:
        ud2
```

(b) Assembly for optimized program (optimized on opt-level 2)

Figure 5.4: Unoptimized and optimized assembly comparison for bug 2

The assembly of the unoptimized program in Figure 5.4 shows a program working as expected, the `callq` instruction calls `fun1` which includes a jump back to `.LBB0_1` which therefore results in an infinite loop. However, when optimized, the resulting assembly still has `example:fun1`, but the main method simply has the `ud2` instruction again, which results in the runtime error.

This issue was detected by `rustsmith-tester` in version 1.40 of `rustc`, but further investigation found that the bug was initially present in version 1.23 of `rustc`, and was present until version 1.48, until it finally got fixed in 1.49. The associated GitHub issue [38] was also found which indicates that this bug was attributed to LLVM optimizing out non side-effecting loops, as in C and C++, this is considered as undefined behaviour. However, as this is not undefined behaviour in Rust, this never should be optimized out. The bug was eventually patched and released in January 2021 when `rustc` was upgraded to LLVM 12.

5.1.3 Bug 3: Miscompilation between Rust versions

The last bug analysed is a true miscompilation bug concerning constant propagation and borrowing. In this instance, the discrepancy in output was not across optimization levels,

but across different `rustc` versions. The outputs produced between the different Rust versions were substantially different, which indicated potentially some incorrect branching when run (for example one followed the true branch of an if, while the other followed the false branch). The original file (which can be found at https://gist.github.com/mayanksharma3/ec22972bed44e927bc42977dc7dc5f35) was again manually reduced to file the smallest file which is shown in Figure 5.5.

```
1  fn main() {
2      let mut var1: (bool, f64, i32) = (false, 0.5f64, 100i32);
3      let var2: &mut bool = &mut var1.0;
4      *var2 = true;
5      let var3: (bool, f64, i32) = var1;
6      if (var3.0) {
7          let var4 = 10i32;
8          println!("{:?}",  ("var4", var4));
9      } else {
10         let var5 = 1i32;
11         println!("{:?}",  ("var5", var5));
12     }
13 }
```

Figure 5.5: Small program that triggers the miscompilation bug

The bug was initially discovered as a difference between the latest version of rust (v 1.61.0) and version 1.45.0. The first element in the tuple (defined on line 2) is reassigned to the value `true` by dereferencing a mutable reference (on line 4). Therefore, when the first value of the tuple is used as the predicate of an if-else expression (on line 6), the expectation is that execution follows the true branch and prints out `("var4", 10)`. However, on version 1.45.0, the output instead prints `("var5", 1)` as execution follows the false branch.

Further investigation found that this bug was only ever in 1.45.0, and was fixed immediately after. A similar GitHub issue [39] relating to this bug indicates a constant propagation bug. It seems that the constant propagation module is unable to track constants through references, and therefore constant propagation of values references should be avoided. Version 1.45.0 allowed for constant propagation of resources that had been referenced, which was the reason the miscompilation was detected.

5.1.4 Summary

In answer to **RQ1**, RustSmith was successfully able to detect bugs that presented different symptoms (runtime errors and miscompilation) in different versions of `rustc`. Each bug presented exercised different parts of Rust such as inlining, loops and borrowing, which illustrates RustSmith's ability to test different parts of Rust well. Note that in total, RustSmith detected five distinct bugs, but the remaining two bugs were so historic they provided little value in being analysed, as they were from a time Rust was fairly unstable.

Of course, there must be more bugs present throughout different versions of Rust that RustSmith either isn't able to detect (due to missing features), or didn't detect due to decision-making through selection managers or a limited testing campaign. Given the testing campaign lasted around a month, and given RustSmith currently generates programs with a fairly small subset of Rust, extending both the testing campaign with `rustsmith-tester` (see Section 4.4) and adding more Rust features into RustSmith *should* increase the chances for finding more bugs.

5.2 Evaluating rustc coverage (RQ2)

One of the most crucial pieces of evaluation concerns analysing the coverage of `rustc` after compiling RustSmith's generated files on it. The overall aim for RustSmith is to test the Rust compiler and therefore measuring how much of the Rust compiler is tested (and specifically which parts well) is crucial.

To achieve this, coverage instrumentation of the rust compiler was required. This itself was a tricky process, as the Rust compiler is built with a subset of rust, so modifying the bootstrapping process to include `-C instrument-coverage` flags would not work for every module. Therefore, the build process was modified to include the profiling and instrumentation flags only when the stage of the compiler being built was **not the standard library** (which relied on the `profiler_builtins` crate, the package that included instrumentation code). A coverage instrumented fork of the rust compiler has been created and published as part of the RustSmith project at: https://github.com/rustsmith/rust.

Once a rust binary could be built with coverage enabled, coverage data would be produced on every file compiled (stored in a `.profraw` file) which are collected for every file produced (across each optimization run). Grcov [40], a tool for producing readable coverage information for rust programs, is then utilized to get detailed information about line and function coverage overall, along with a breakdown for each file. A screenshot of the information Grcov provides is shown in Appendix A.

1000 files were randomly generated through RustSmith, and then compiled through an instrumented version of rustc. The overall results of the latest version of RustSmith at the time (`v1.60.0`) are shown in Table 5.1 after 3 runs:

	Run 1	**Run 2**	**Run 3**	**Average**
Line Coverage	33.75%	33.76%	33.69%	33.73%
Function Coverage	12.34%	12.31%	12.32%	12.32%

Table 5.1: Line and Function Coverage of the optimization module in rustc compiler source code

As we can see in Table 5.1, the coverage of rustc is (on average) 33.7% line coverage and 12.32% function coverage. Given Rust is a fairly large language with a lot of features including concurrency, generics, traits etc. RustSmith seems to be performing well in covering this much of rustc.

However, simply viewing coverage information quantitively across the whole compiler codebase bears little value. RustSmith's main purpose is to be testing the optimization passes implemented in `rustc` (as bugs are detected through miscompilations between optimization levels). Therefore, the coverage specifically for the optimization files (found in `compiler/rustc_mir_transform`) are analysed for coverage. The results are then compared with the official Rust test-suite handwritten to test the optimization passes within the compiler. First, the overall results for the whole module from both runs (1000 RustSmith generated files and Rust's official handwritten optimization tests) are displayed in Table 5.2.

	Line Coverage	**Function Coverage**
RustSmith (1000 files)	51.01%	26.89%
Official mir-opt test-suite	68.84%	33.76%

Table 5.2: Line and Function Coverage of the **whole** rustc compiler source code

As expected, the official test-suite covers more of the `rustc_mir_transform` than 1000 randomly generated files through RustSmith. It is interesting to note however that with just 1000 files, RustSmith **can cover over 50% of the same package** (w.r.t line coverage), and RustSmith's files are completely randomly generated and not hand-written to test specific optimizations (as done in the official test-suite). RustSmith therefore is generating interesting enough programs to exercise optimizations well.

Again however, simply looking at overall statistics will not highlight which optimizations RustSmith test better over others. Table 5.3 shows the line coverage for each optimization file in the `compiler/rustc_mir_transform` module for the same 2 runs from above.

File	Line Coverage		Function Coverage		Line Coverage		Function Coverage	
abort_unwinding_calls.rs	69.32%	61 / 88	25%	1 / 4	79.55%	70 / 88	25%	1 / 4
add_call_guards.rs	100%	40 / 40	50%	4 / 8	100%	40 / 40	50%	4 / 8
add_moves_for_packed_drops.rs	37.74%	20 / 53	37.50%	3 / 8	96.23%	51 / 53	50%	4 / 8
add_retag.rs	2.54%	3 / 118	4.17%	1 / 24	2.54%	3 / 118	4.17%	1 / 24
check_const_item_mutation.rs	33.33%	29 / 87	13.33%	4 / 30	33.33%	29 / 87	13.33%	4 / 30
check_packed_ref.rs	33.33%	25 / 75	27.78%	5 / 18	33.33%	25 / 75	27.78%	5 / 18
check_unsafety.rs	51.14%	225 / 440	19.74%	15 / 76	73.86%	325 / 440	30.26%	23 / 76
cleanup_post_borrowck.rs	82.35%	14 / 17	33.33%	2 / 6	82.35%	14 / 17	33.33%	2 / 6
const_debuginfo.rs	5.17%	3 / 58	12.50%	1 / 8	5.17%	3 / 58	12.50%	1 / 8
const_goto.rs	98.18%	54 / 55	50%	3 / 6	96.36%	53 / 55	50%	3 / 6
const_prop.rs	79.94%	574 / 718	33.93%	57 / 168	84.54%	607 / 718	34.52%	58 / 168
const_prop_lint.rs	59.20%	383 / 647	25.33%	38 / 150	72.80%	471 / 647	32%	48 / 150
dead_store_elimination.rs	97.01%	65 / 67	50%	6 / 12	98.51%	66 / 67	50%	6 / 12
deaggregator.rs	96.55%	28 / 29	50%	4 / 8	96.55%	28 / 29	50%	4 / 8
deduplicate_blocks.rs	97.37%	111 / 114	47.06%	16 / 34	95.61%	109 / 114	47.06%	16 / 34
deref_separator.rs	94.29%	66 / 70	37.50%	3 / 8	95.71%	67 / 70	37.50%	3 / 8
dest_prop.rs	1.38%	7 / 507	1.25%	1 / 80	1.38%	7 / 507	1.25%	1 / 80
dump_mir.rs	40%	4 / 10	33.33%	2 / 6	40%	4 / 10	33.33%	2 / 6
early_otherwise_branch.rs	1.41%	3 / 213	5.56%	1 / 18	1.41%	3 / 213	5.56%	1 / 18
elaborate_drops.rs	86.19%	362 / 420	42.55%	40 / 94	85.48%	359 / 420	46.81%	44 / 94
function_item_references.rs	27.48%	36 / 131	16.67%	4 / 24	41.22%	54 / 131	20.83%	5 / 24
generator.rs	0.70%	7 / 1005	1.43%	2 / 140	87.26%	877 / 1005	36.43%	51 / 140
inline.rs	75.48%	431 / 571	38.57%	27 / 70	90.89%	519 / 571	41.43%	29 / 70
instcombine.rs	66.14%	84 / 127	35%	7 / 20	74.02%	94 / 127	40%	8 / 20
lib.rs	88.60%	342 / 386	36%	18 / 50	93.01%	359 / 386	44%	22 / 50
lower_intrinsics.rs	13.79%	16 / 116	33.33%	2 / 6	68.97%	80 / 116	33.33%	2 / 6
lower_slice_len.rs	66.67%	46 / 69	50%	4 / 8	98.55%	68 / 69	50%	4 / 8
marker.rs	100%	7 / 7	50%	3 / 6	100%	7 / 7	50%	3 / 6
match_branches.rs	81.73%	85 / 104	37.50%	3 / 8	97.12%	101 / 104	37.50%	3 / 8
multiple_return_terminators.rs	96%	24 / 25	33.33%	2 / 6	96%	24 / 25	33.33%	2 / 6
normalize_array_len.rs	60.41%	119 / 197	50%	6 / 12	91.88%	181 / 197	50%	6 / 12
nrvo.rs	91.91%	125 / 136	40%	12 / 30	89.71%	122 / 136	40%	12 / 30
pass_manager.rs	88.24%	90 / 102	45.35%	39 / 86	88.24%	90 / 102	45.35%	39 / 86
remove_false_edges.rs	80%	8 / 10	50%	1 / 2	100%	10 / 10	50%	1 / 2
remove_noop_landing_pads.rs	95.71%	67 / 70	50%	6 / 12	95.71%	67 / 70	50%	6 / 12
remove_storage_markers.rs	93.33%	14 / 15	50%	3 / 6	93.33%	14 / 15	50%	3 / 6
remove_uninit_drops.rs	0%	0 / 114	0%	0 / 20	0%	0 / 114	0%	0 / 20
remove_unneeded_drops.rs	68%	17 / 25	25%	1 / 4	96%	24 / 25	25%	1 / 4
remove_zsts.rs	82.76%	48 / 58	40%	4 / 10	91.38%	53 / 58	40%	4 / 10
required_consts.rs	100%	9 / 9	50%	2 / 4	100%	9 / 9	50%	2 / 4
reveal_all.rs	80.95%	17 / 21	37.50%	3 / 8	85.71%	18 / 21	37.50%	3 / 8
separate_const_switch.rs	72.58%	90 / 124	50%	6 / 12	70.16%	87 / 124	41.67%	5 / 12
shim.rs	56.06%	319 / 569	23.96%	23 / 96	69.42%	395 / 569	32.29%	31 / 96
simplify.rs	85.71%	306 / 357	43.75%	28 / 64	87.11%	311 / 357	45.31%	29 / 64
simplify_branches.rs	92.59%	25 / 27	50%	3 / 6	96.30%	26 / 27	50%	3 / 6
simplify_comparison_integral.rs	76.19%	96 / 126	42.86%	6 / 14	78.57%	99 / 126	42.86%	6 / 14
simplify_try.rs	2.19%	11 / 502	2.27%	2 / 88	2.19%	11 / 502	2.27%	2 / 88
uninhabited_enum_branching.rs	28.57%	28 / 98	20.83%	5 / 24	93.88%	92 / 98	45.83%	11 / 24
unreachable_prop.rs	63.51%	47 / 74	41.67%	5 / 12	97.30%	72 / 74	41.67%	5 / 12

Table 5.3: Line and Function Coverage comparison for 1000 RustSmith generated files against the official optimization test-suite

With these results, analysis is performed to identify where RustSmith performs well in, and

where (and for what reasons) RustSmith fails to perform as well.

5.2.1 Well covered optimizations

RustSmith seems to be performing quite well across a wide variety of optimizations. Whilst not all are thoroughly explained here, some of the most covered and interesting optimizations are explained.

simplify.rs (85.71% vs. 87.11%)

This optimization aims to remove redundant blocks from the control-flow as much as possible. For example, it attempts to remove any blocks in the intermediate-represent that (either immediately or due to another optimization pass occurring before this one) have been found to be redundant. It also detects any local variable decorations that are not used and therefore can be removed. Whilst not every line of code is covered using RustSmith, it covers almost as many lines and functions as the official test suite, showing that RustSmith is exercising this particular optimization pass well. This makes sense as this particular optimization does not rely on specific program features past control flow nodes or local declarations, and therefore as long as interesting enough programs are creates randomly, the optimization should be tested well throughout. The only parts of the simplification process missed by RustSmith at this time is an optimization check to do with pointers, which is missed simply because raw pointers are not implemented in RustSmith.

dead_store_elimination.rs (97.01% vs. 98.51%)

This optimization performs dead store elimination (DSE) which is an optimization that eliminates any assignments that are unnecessary simply because the content on the RHS of an assignment is never read by any other instruction. RustSmith covers all but 1 line which is under a check which, again, concerns a check to do with pointers. The difference between the 2 coverages is due to a provenance_soundness.rs test file which checks whether a casted pointer should be eliminated or not (it shouldn't).

match_branches.rs (81.73% vs. 97.12%)

Match branches aims to eliminated branching in an interesting way. This optimization aims to detect blocks inside control flow expressions with the same code **other than potentially assignments relating to the control flow**. The code presented in Figure 5.6a shows an original piece of code, and the optimization aims to produce code similar to Figure 5.6b.

```
let mut y = false;
if (x == 42) {
    y = false;
    fun1();
} else {
    y = true;
    fun1();
}
```

(a) Original code before match branches optimization

```
let mut y = false;
y = if (x == 42) { false } else { true };
fun1();
```

(b) Optimized code after match branches optimization

Figure 5.6: Example illustrating the effect match branches can have to particular parts of a program

In a step towards exercising this optimization, when creating branching code blocks, sometimes the same block of code is chosen so that for example, both the `true` and `false` branches of

an if statement contain the exact same code. However, the parts of the optimization that are missed are to deal with the cases where the boolean assignments are performed in each branch **differently** so that the inline switch of the predicate can be inserted into the intermediate representation.

This is a limitation of RustSmith as there currently is no realistic chance this situation occurs, although in the future, selection managers could be made to target specific optimizations that could try and exercise these edge cases.

5.2.2 Optimizations covered better by RustSmith

RustSmith was also able to generate files that **covered more code than the official test suite**. The line coverage data was compared with the line coverage for each optimization in the official test suite, and it was found that RustSmith successfully covered lines not covered in the test-suite in 9 different optimizations. The results are shown in Table 5.4.

Optimization File	Extra Lines Covered	Lines in Optimization
const_goto.rs	1	103
const_prop.rs	8	1143
deduplicate_blocks.rs	2	190
elaborate_drops.rs	15	589
instcombine.rs	2	203
normalize_array_len.rs	2	285
nrvo.rs	3	235
separate_const_switch.rs	9	339
simplify_comparison_integral.rs	2	224

Table 5.4: Optimization files RustSmith can cover lines in that the official test suite cannot

As shown in Table 5.4, RustSmith's generated files were able to cover lines not covered by the test-suite in **9 different optimizations** with varying degrees of extra coverage. For example, the `const_goto.rs` is simply covering the closing bracket of an if statement, which indicates that all the tests from the official test suite cause the optimization to early return from the if statement, whereas RustSmith's files don't early return.

However, there are some optimizations RustSmith covers a significant number of lines that the test suite does not. This is further analyzed below:

elaborate_drops.rs

The `elaborate_drops` optimization is one of the most successful optimizations that RustSmith covers extra lines in. It's able to cover 15 more lines than the official test suite, which, given the size of the file accounts to 2.55% of the codebase. Comparing the line coverage indicated that the case with resources' parents and resources inside arrays weren't being covered.

Drop elaboration is an optimization specific to Rust that attempts to efficiently place `Drop` terminators in the intermediate representation so that the resource can be destroyed (just like developers manually call `free` in C). However, `Drop` should only be called on initialized values or partially initialized values. For the partially initialized resources, only the internal elements that are initialized should be dropped. Drop elaboration attempts to wrap `Drop` terminators in conditions to check (using a "drop flag") whether to actually call the drop destructor or not. More information about drop elaboration can be found here in the official developer guide for Rust [41].

One of the files that covered these extra 15 lines were taken and then subsequently manually reduced, ensuring that after every reduction the extra coverage still held. The original file can

be found at https://gist.github.com/mayanksharma3/25aafb271b456496e27a023ce851c4f7 and the reduced code is shown in Figure 5.7.

```
1 fn main() -> () {
2     let mut var562: Vec<bool> = vec![true, false, false];
3     let mut var561: &mut Vec<bool> = &mut (var562);
4     (*var561) = vec![true, true, true, true, true, true];
5 }
```

Figure 5.7: Reduced Code fragment from RustSmith generated file that covers more of elaborate drops than the official test-suite

The simple program in Figure 5.7 displays the smallest example that covers extra lines in `elaborate_drops.rs`. It seems that no test-suite file covers a specific case where arrays are dropped due to a de-reference assignment shown on line 4.

There is code concerning lookups to parents and traversing up, but it is only covered with this fragment of code.

const_prop.rs

The `const_prop.rs` optimization is one of the most common optimization passes present in compilers. Constant propagation identifies expressions that can be interpreted at compile time, and replaces them with the result of the expression itself. For example, constant propagation will replace `1000 * 12` with `12000` so that the CPU doesn't have to do calculate this expression at runtime.

As before, one of the generated files that covers more of `const_prop.rs` than the official test suite was taken and manually reduced, ensuring that the extra coverage was still present after every reduction. The original file can be found at https://gist.github.com/mayanksharma3/a184a8c8b2e54a81b48131e6ebb0216d and the reduced code is shown in Figure 5.8.

```
1 fn main() -> () {
2     let var379: i32 = -692353821i32;
3     let var380: i32 = -351168514i32;
4     let var378: i32 = var379.wrapping_mul(var380);
5
6     let var285: i32 = 1630372709i32;
7     let mut var284: i32 = -833350497i32.wrapping_sub(var285);
8 }
```

Figure 5.8: Reduced code fragment from RustSmith generated file that covers more of `const_prop.rs` than the official test-suite

The trivial program in Figure 5.8 performs a multiplication and subtraction of `i32` values. As this was produced through the reconditioner, the arithmetic operations are done through `wrapping_*` methods, which intentionally producing wrapping arithmetic so that any overflows wrapped around to 0. These overflowing arithmetic operations are the reason why extra lines are covered by RustSmith over the official test-suite. Looking into the optimization file itself, overflowing operations are not fully tested, as it seems overflowing operations require additional checks and different substitutions when performing constant propagation.

Additionally, further experimentation was performed on the program in Figure 5.8 by removing `wrapping_mul` and `wrapping_sub` with `*` and `-` respectively. As expected, this threw a compila-

tion error as it flagged the overflowing operations. Interestingly however, comparing line coverage against the test suite showed that the file covered 10 more lines in `const_prop_list.rs`, another file in the optimization folder. This indicates that producing intentionally invalid code can be beneficial in coverage the Rust compiler, even the optimization module.

5.2.3 Optimizations covered better by official test suite

As expected, the majority of optimizations are covered better by the official test suite. Along with the reasons covered in Section 5.2.1, the reasons why other optimizations perform better with the test suite are explained below.

generator.rs (0.7% vs 87.26%)

The `generator.rs` file is almost completely uncovered by RustSmith's generated files. The only parts of the file covered is the visitor implementation that detects for specific AST nodes. The reason RustSmith's file don't cover any of this particular optimization is because `generator.rs` optimizes a Rust feature that exposes what they call a "resumable function" through generators. As RustSmith does not create generators in the programs it produces, this low coverage is expected.

However, `generator.rs` is the largest file in the optimizations module, and therefore adding this feature in the future would be desirable to test this complex optimization pass.

lower_intrinsics.rs (13.79% vs 68.97%)

The lower_intrinsics.rs file attempts to optimize some internal intrinsics within Rust. As RustSmith uses very few of these intrinsics (other than `wrapping_mul` and `wrapping_div` etc. for reconditioning), it is expected that the RustSmith coverage for this file is much lower than the official testsuites as there is a dedicated file called lower_intrinsics.rs in the test folder testing many more intrinsics such as `size_of`, `min_align_of`, `forget` and many more.

5.2.4 Summary

In answer to **RQ2**, the results show that RustSmith is able to coverage both the whole compiler, and the optimization module quite well. Performing poorer than the test-suite was expected given RustSmith implements a fairly small subset of the whole Rust programming language. Therefore, covering over a third of the whole compiler and over half of the optimization module with just 1000 randomly generated files when compared against handwritten tests shows RustSmith is capable of effectively testing a substantial portion of the rust compiler. Additionally, RustSmith was capable of testing lines in optimizations that handwritten test-suites were unable to cover, proving RustSmith's effectiveness in testing the compiler in practice.

We are confident that extending RustSmith to handle to cover more of Rust's language, along with sometimes intentionally producing invalid code, should start covering more of the compiler and its optimizations. Using coverage information such as Table 5.3 to prioritize which features to add next to RustSmith may be an effective strategy to achieve this.

5.3 Evaluating RustSmith's features (RQ3)

The next evaluation strategy involves analysing the impact different features or changes made to the RustSmith generator by comparing the overall, and optimization module code coverage of the Rust compiler. This idea is inspired by an evaluation technique used in machine learning called ablation studies where certain components are removed from generation and the code coverage compared.

To analyse the generator, 10 different configurations are used; the first configuration is a "bare bones" generator which won't generate any functions, structs, tuples, arrays or block statements, but simply generate simple assignments and declarations, and the final configuration is the latest version of RustSmith. Note that with every configuration used, the generator still is producing valid Rust code. The configurations roughly follow major checkpoints in the development process of RustSmith and therefore correspond to features being added or majorly improved.

Results are gathered by configuring the generator to produce code as described in the checkpoints below. 100 files in this configuration are then generated using RustSmith and then the coverage information is gathered. This is repeated 3 times in the same configuration and an average is taken. Note that before testing a set of 100 files, all coverage gathering files from previous runs are deleted and therefore the results don't show cumulative coverage across versions.

The results and explanations of the checkpoints are shown in Figure 5.9.

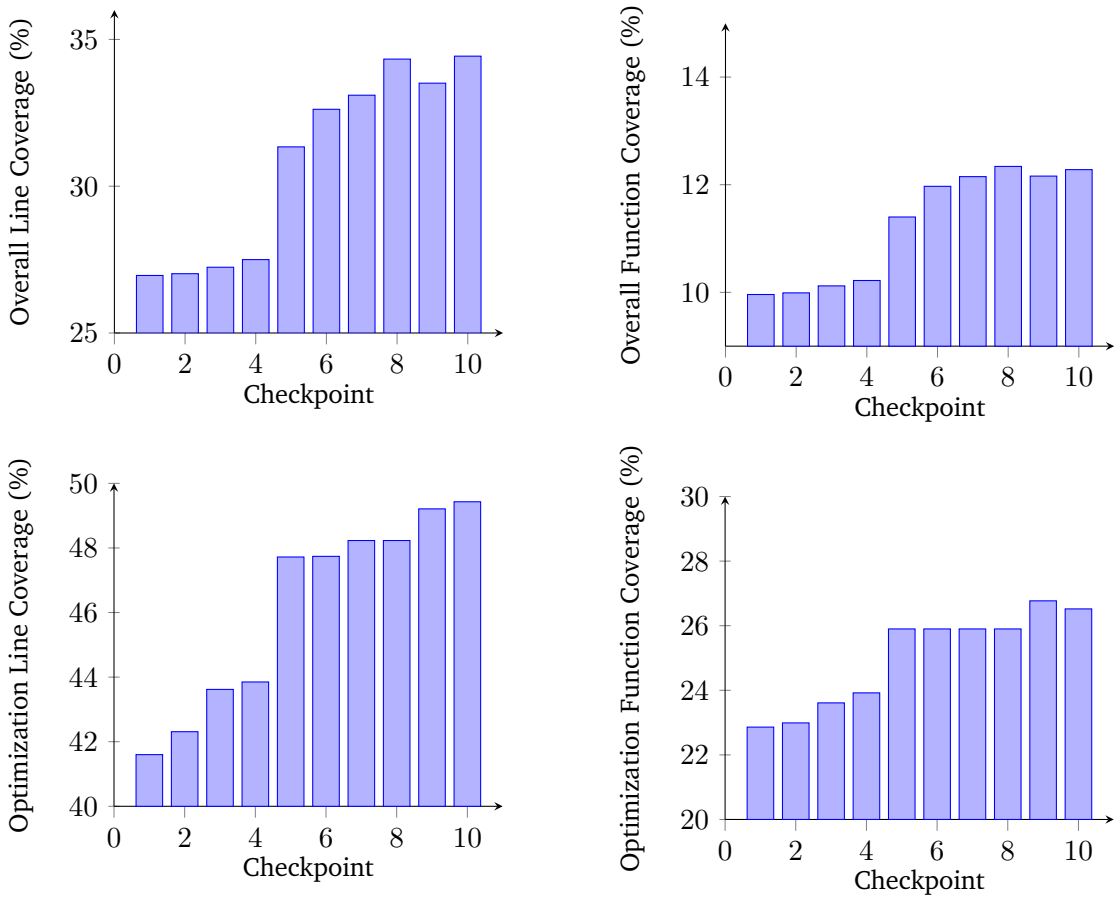

Figure 5.9: Line and function code coverage for different checkpoints for both the whole compiler and the optimization module

1. **Checkpoint 1**: RustSmith will only produce code in a **single scope** (the `main` function) with **only declarations and assignments** to basic types such as integers and booleans
2. **Checkpoint 2**: RustSmith will also now produce **block expressions along with if and else expressions**
3. **Checkpoint 3**: RustSmith is now capable of **producing functions** other than `main`
4. **Checkpoint 4**: `OptimalWeightingSelection` is now introduced and used (up to this point every decision was a uniform choice)

5. **Checkpoint 5**: **Structs and tuples** (along with struct and tuple access is introduced). It also allows for partial moves and handles all the ownership rules described in Section 2.7.1
6. **Checkpoint 6**: **Command-line arguments** are now both generated and retrieved within the compiler to mimic `volatile` arguments in C, along with extending the LHS for assignments to include struct and tuple element assignments
7. **Checkpoint 7**: **Immutable and Mutable references** are included in the generated files, along with lifetime parameters where required.
8. **Checkpoint 8**: The **fail-fast approach** is introduced to handle edge cases in lifetimes and the option is exposed to the user. The default usage is still the directed approach here
9. **Checkpoint 9**: RustSmith's latest version with the fail-fast approach turned on. Compared to checkpoint 8, RustSmith also gained more boolean operators along with arrays and array access, insertion and length operators.
10. **Checkpoint 10**: RustSmith's latest version with the directed approach turned on

Using the results from Figure 5.9, the impact of adding certain features or making certain improvement can be seen. Overall, the general trajectory can be seen that adding features in throughout different points have made a significant difference to the code coverage of both the compiler as a whole and especially the optimization module.

Checkpoints 1 – 4 make little progress in the overall compiler line coverage and function coverage (line and function coverage only increase by 0.6% and 0.3% respectively), although within the optimization module, steady progress is made. The major jump in coverage occurs when Structs and tuples are added in **Checkpoint 5** (along with exercising partial moves on these types). This again makes sense as these two constructs (structs and partial moves) add more complexity to the programs generated and allow for optimizations such as dead store elimination or simplification to occur (as described above). Adding lifetimes and references (in **Checkpoint 7**) made an impact to the both the overall compiler and the optimization module, which makes sense as borrow checking in Rust is performed on the MIR, and therefore any clean-up after borrow checking is an optimization phase that is now covered.

Finally, **Checkpoints 9 and 10** compare the effectiveness of the fail-fast approach on coverage, and the results produced are fairly interesting. Using the fail-fast approach results in the overall compiler coverage to dip substantially for both line and function coverage. The coverage does increase in the optimization module for both line and function coverage, but this may be attributed to other changes along the way, such as including generation for arrays and array access. Therefore, to understand the fail-fast approach's true impact, the results should be compared with the directed approach. The directed approach improves coverage of the overall compiler with compared to the fail-fast approach but only improves the line-coverage in the optimization module. The increase may be due to more language features being tested overall. For example, the reference expression can only be generated if a resource is available of the exact right type to borrow. This means that in the fail-fast approach, the chances for a reference expression to be created without an exception being thrown is very small, but with the directed approach, if nothing is found, the right resource can simply be created.

5.3.1 Summary

In answer to **RQ3**, incrementally improving and extending RustSmith has made a significant impact on how much of `rustc` RustSmith can cover. Specifically, generating programs that excercised Rust-specific features such as ownership and borrowing made one the biggest advances in RustSmith's coverage capabilities on both the Rust source code and specifically the optimization module too.

5.4 Evaluating Selection Managers (RQ4)

This section aims to evaluate the built-in selection managers (`OptimalSelectionManager`, `BaseSelectionManager` and `AggressiveSelectionManager`). Different experiments are performed that allow us to verify and evaluate how well each selection manager is performing to complete the purpose it was built for.

5.4.1 Understanding `BaseSelectionManager`

The `BaseSelectionManager` is largely present as a method to build upon to create more meaningful selection managers. However, the manager is still an option for generation and still is capable of generating valid Rust programs. To understand the impact other selection managers make, the base selection manager's generation patterns need to be understood so that they can be quantitively compared.

First, the distribution of the size of the programs are analysed. The following graph displays the distribution of the size of programs (looking at the number of lines of code) for 300 randomly generated files.

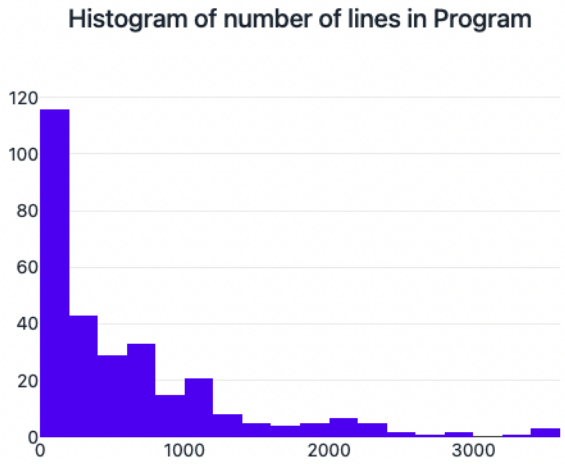

Figure 5.10: Distribution of program size for `BaseSelectionManager`

As shown in Figure 5.10, the majority of programs were generated to be under 200 lines, and the number of programs decrease as the size of the program decrease fairly consistently.

This is expected given what `BaseSelectionManager` attempts to do, which is to provide (as much as possible) an equal weighting to all selection options. To prevent infinite recursion however, hard limits have to be set as otherwise situations where recursive expressions are always chosen can occur. Therefore, given the majority of expressions are literal expression such as an `i32` literal, most expressions end up not requiring more statements inside them. The probability therefore for creating incredibly long programs is very low (after the restriction on recursive expressions) explaining the sharp drop off and continued decline in program size.

After understanding the pattern of files generated by the `BaseSelectionManager`, more useful selection managers can now be evaluated and analysed in depth.

5.4.2 Evaluating the Optimal Selection Strategy

The optimal selection strategy aims to be an all-rounded approach on testing different parts of the language well. It also aims to make larger programs than the base selection manager,

however still with the chance of creating particularly large or particularly small programs. The program size is mostly controlled by two decisions that the `OptimalSelectionManager` attempts to balance to produce this particular distribution.

The first decision decides whether a statement block should grow (by generating another statement) or terminate. Instead of a static probability to grow the statement block, the optimal selection manager calculates the probability of growing with the following formula:

```
30.0 / (ctx.statementsInCurrentScope + 1.0)
```

This formula means that the probability for creating statements in the block are very high to begin with, but with every statement created, the probability decreases with an inverse relationship.

The next decision involves adjusting the weighting of `RecursiveExpressions` by using depth information about `RecursiveExpressions`. The `BaseSelectionManager` houses a very strict cut-off for all recursive expressions (concereteley, the depth of recursive expressions cannot ever exceed 3), but the OptimalSelectionManager tries to relax this in higher depths with the following formula:

```
1.0 / (recursiveExpressionDepth * 4 + 1.0)
```

This is very similar to the formula for growing statement blocks, but stricter. This is because there are multiple recursive expressions in the Rust language implemented, and therefore must be fairly strict. However, it still allows recursive expressions to be chosen at higher depths but simply rapidly decreases the probability as the depth of recursive expressions increase.

To evaluate this selection manager, 300 files are generated randomly and the histogram for program sizes is compared to Figure 5.10. Additionally, the coverage of `rustc` through RustSmith generated files is compared to validate whether the optimal selection manager tests the compiler better than the base selection manager. The results from the 300 randomly generated files are shown in Figure 5.11.

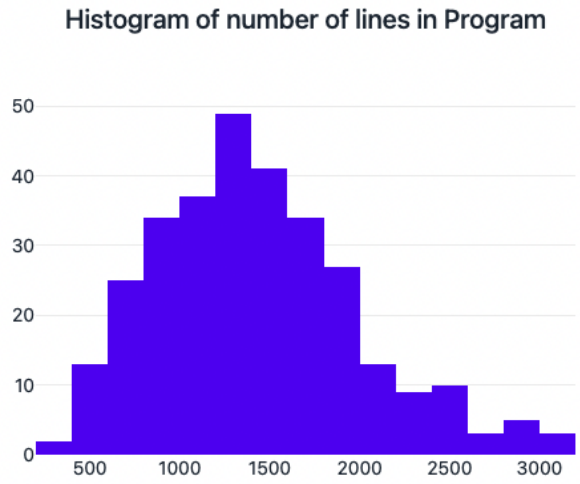

Figure 5.11: Distribution of program size for `OptimalSelectionManager` for 300 programs

As shown in Figure 5.11, the distribution of the 300 generated files are much more desirable than the distribution found in Figure 5.10. The distribution follows roughly a pyramid distribution, with the peak occurring in programs sized between 1200 and 1400 lines. Generation of smaller and large files no longer have a steep decline as in Figure 5.10 and therefore allow for a much more diverse distribution of program sizes.

These results show that the optimal selection manager is creating diverse programs by creating larger programs and a better distribution of program sizes than the base selection manager.

5.4.3 Evaluating the Aggressive Node Strategy

The aggressive node strategy is specifically built on-top of the `OptimalSelectionManager` and sets the chosen node to have a higher probability than other available nodes, therefore aggressively testing that particular node by multiplying the weighting of the particular node by 5.

To evaluate whether the aggressive node strategy works as expected, statistics are gathered from 100 runs of the generator. The statistics include the counts of AST nodes found in the final produced program AST. The mean is then calculated by averaging statistics across 100 files and evaluated to check whether the aggressive strategy did in-fact test specific parts of the language over others.

The aggressive strategy was run on 3 different expressions: `AddExpression`, `CLIArgumentAccessExpression` and `FunctionCallExpression` and the results are shown in Figure 5.12.

Expression Node	OptimalManager	FunctionCallExpr	CLIArgExpr	AddExpr
AddExpression	3.27	3.78	5.09	**49.11**
BitwiseAndLogicalAnd	2.92	2.99	0.73	1.53
BitwiseAndLogicalOr	2.73	3.32	0.80	1.73
BitwiseAndLogicalXor	2.85	3.27	0.96	1.74
BlockExpression	7.42	9.15	2.37	3.38
BooleanLiteral	126.83	**232.11**	18.51	23.49
CLIArgumentAccessExpression	81.74	33.78	**153.59**	64.87
DereferenceExpression	0.09	0.22	1.96	0.12
DivideExpression	3.24	3.93	0.91	2.50
Float32Literal	119.58	208.80	21.92	78.86
Float64Literal	121.05	217.61	27.43	76.94
FunctionCallExpression	107.19	297.36	2.07	79.09
GroupedExpression	6.44	8.44	2.58	3.62
IfElseExpression	6.62	8.30	2.09	3.38
IfExpression	0.85	0.93	0.38	0.21
Int128Literal	118.36	219.07	25.11	73.18
Int16Literal	119.41	186.23	22.11	76.19
Int32Literal	117.84	204.73	24.93	70.77
Int64Literal	119.00	215.45	26.50	91.79
Int8Literal	120.84	179.68	25.45	78.27
LoopExpression	1.03	0.99	0.40	0.26
ModExpression	2.13	2.33	0.71	1.64
MultiplyExpression	3.31	3.85	1.06	2.49
MutableReferenceExpression	0.54	1.16	37.02	0.31
ReferenceExpression	0.98	1.68	42.06	0.60
StringLiteral	118.36	213.03	17.55	22.40
StructElementAccessExpression	2.66	3.57	2.90	1.77
StructInstantiationExpression	79.14	151.95	27.66	21.33
SubtractExpression	3.39	3.81	0.99	2.53
TupleElementAccessExpression	3.05	3.79	3.59	1.58
TupleLiteral	69.33	121.35	28.19	20.72
Variable	19.51	40.11	45.17	25.10
VoidLiteral	72.70	104.61	11.77	14.48

Figure 5.12: Results for comparing the performance of `AggressiveSelectionManager` by calculating the average number of times each node was generated. Note that not all expression nodes RustSmith is capable of producing are shown here

The **OptimalSelectionManager**'s results in Figure 5.12 show a more all-rounded approach compared to the aggressive strategy. While it seems like there is some imbalance (such as when AddExpresions are much less tested than for example the Int8Literal) this is due to the fact that the OptimalSelectionManager penalizes RecursiveExpressions very rapidly. Therefore, non-recursive cases are chosen more often to prevent infinite recursion cases.

Looking at the runs using the aggressive strategy, starting with the **FunctionCallExpr** row, the value for `FunctionCallExpressions` are higher than any other node in the list. Therefore, aggressively testing `FunctionCallExpressions` has successfully resulted in more functions calls being tested than usual (as can be seen in the first column). The same can be seen for the run where **CLIArgumentAccessExpression**s are aggressively tested.

Next, when the **AddExpression** was the chosen node for the aggressive testing, something slightly different occurs. `AddExpression` is not the highest used AST node in this particular run, and instead nodes such as the integer and float literals are must higher. However, given the add expression contains inside it 2 sub-expressions, and importantly, add expressions' sub-expressions must be integers or floats, this makes sense. For every add expression generated, 2 sub-expressions must be created, and therefore the literal expressions are used. Therefore, to understand whether `AddExpressions` are being tested more thoroughly than usual, the value should be compared against other RecursiveExpressions within the same run. Add expressions are generated much more than any other recursive expression such as `BitwiseAndLogicalAnd` or `ModExpression`. Additionally, add expressions are generated much more often in the aggressive run than the run where the `OptimalSelectionManager` was used.

However, there currently is any option to aggressively test multiple different nodes in a round-robin fashion or multiple at the same time. While something like this would be fairly easy to implement and expose through the command line arguments, it would then be desirable to "rank" the chosen nodes to aggressively test so that between the multiple chosen nodes, some are given higher weightings than others. This leads towards the overall current limitation with the selection managers, which concerns **user configurable custom selection managers**. Currently, the selection managers were designed carefully to allow for modular and composable selection managers to be added very easily, but currently are no ways to pass in a configuration file and use formulas from a file as a selection manager. Instead, the source code would need to change and the generator rebuilt for any custom selection manager to be included in. Providing functionality for the user to pass in a configuration file is therefore considered to be future work.

5.4.4 Summary

In answer to **RQ4**, different selection managers make a substantial impact on the kinds of programs generated. This is most clearly seen when comparing the distribution of the size of programs generated using the `OptimalSelectionManager` instead of the `BaseSelectionManager`, where `OptimalSelectionManager` can easily control how large programs can become. Additionally, the `AggressiveSelectionManager` successfully targets specific AST nodes by increasing their chances of being chosen. Interestingly, the second bug presented in Section 5.1 was identified when using the `AggressiveSelectionManager` on the `loop` expression in Rust.

5.5 Evaluating failure approaches: directed vs. fail-fast (RQ5)

As explained in Section 3.3.2, there are two main mechanisms implemented in RustSmith to handle failure cases: the directed approach and the fail-fast approach. These two mechanisms have their strengths and limitations and this evaluation section aims to determine in which cases one method may perform better than another.

5.5.1 Evaluating generation speed

First, the speed of generation when using different approaches is considered. This is because it is possible that these approaches affect the speed of generation very differently, as the directed approach (mostly) commits to a specific node and generates it, while the fail-fast approach attempts generation every time but "gives up" a lot more by throwing an exception. The fail-fast approach also includes the step where the node is removed from the list of available nodes for selection, so it may be a long time before a *valid* node is found.

To test this hypothesis, 100 files are randomly generated using RustSmith with each approach, and the time for generation is compared. The same selection manager is used and therefore should produce similar sized programs. The time statistics for generation time are gathered using the `time` [42] command in Unix-based systems, which gathers information about the time elapsed along with the CPU time spent. The results after 3 runs for each are shown in Table 5.5.

	Run 1	Run 2	Run 3	Average
Time	31.07	32.38	30.90	31.45
CPU Time	24.16	25.43	24.40	24.66

(a) Directed Approach speed results

	Run 1	Run 2	Run 3	Average
Time	40.94	39.51	39.42	39.96
CPU Time	34.76	33.59	33.20	33.85

(b) Fail-fast Approach speed results

Table 5.5: Comparison of performance when using the directed or fail-fast approach

As we can see, the fail-fast approach takes on average around 8 seconds to generate 100 files than the directed approach. It also uses more CPU time as a proportion of execution time (78.4% and 84.7% for the directed and fail-fast approach respectively) than the directed approach. This may be attributed to the exceptions that are thrown when using the fail fast approach, which costs CPU time, along with resulting in more code being executed overall due to the multiple attempts to generate a node. Therefore, the directed approach performs better in terms of generation speed as expected, simply due to the directed approach not throwing exceptions and requiring re-attempts for generation of nodes.

5.5.2 Evaluating coverage on the programs generated

Whilst more was explained in Section 5.3, the impact of the failure approach on the coverage of the rust compiler is critical. The fail-fast approach should, in theory, produce less optimizable code than the directed approach simply due to the fact that in Rust, after a certain expression is chosen (such as the reference expression), only a very limited set of expressions can be used after it. Therefore, if no valid resource can be found to reference, the whole reference expression can not be created. The directed approach however can combat these situations by (whenever possible) creating the resource to borrow. Therefore, more of the languages features can be exercised in different ways, which should follow through into more optimization coverage.

The results for the coverage of the compiler when different approaches are used are shown in Table 5.6.

	Coverage			
Failure Approach	**Overall Line**	**Overall Function**	**Optimization Line**	**Optimization Function**
Directed	34.43%	12.28%	49.43%	26.52%
Fail-fast	33.51%	12.16%	49.21%	26.77&

Table 5.6: Results comparing the coverage of the rust compiler between the fail-fast and directed approach

As hypothesized, the directed approach does indeed perform better in terms of coverage for both the compiler and the optimization module (except for function coverage in the optimization module).

5.5.3 Summary

In answer to **RQ5**, the failure approaches affect the performance of RustSmith in different ways. The directed approach performs better both in terms of speed and CPU time, along with code coverage. Therefore the default failure approach used in RustSmith is the directed approach (including the fall back to the fail-fast approach described in Section 3.3.1). Note that the fail-fast approach is still exposed as an option to the user, although as explained, generation will be slower than usual.

6 Conclusion and Future Work

Overall in this project we have successfully produced an automated means to test the Rust compiler, by providing a tool that can generate Rust programs to test the compiler with. Existing techniques such as grammar-aided approaches, differential testing, and reconditioning have been successfully adapted to Rust, which make the generator capable of producing rudimentary programs. We then successfully designed new techniques such as lifetime value requirements and ownership and borrow tracking, which effectively test the ownership, borrowing and lifetime rules built within the Rust compiler. We have also designed a clear and easy to extend approach to handle weightings for different decisions within the generator through composable selection managers (see Section 3.2).

Based on evaluation studies that investigated RustSmith's ability to detect bugs (Section 5.1) as well as through thorough analysis of RustSmith's ability to cover the compiler source code (Section 5.2), the final version of the fuzzer is clearly able to produce interesting, diverse and valid programs that can detect bugs in the Rust compiler, along with effectively testing the Rust source code (specifically the optimizations' module). Indeed, one evaluation study found that RustSmith could cover lines in 9 different optimizations that the official optimizations test suite could not.

In summary, the results clearly show that compiler fuzzing can effectively be used as a method to both test `rustc` for bugs, and substantially cover the source code in the Rust compiler. Despite its success, there is a lot of potential for further research in both resolving RustSmith's limitations and further extending its ability to detect distinct new bugs in the Rust compiler.

6.1 Future Work

Some of the limitations within RustSmith that we aim to resolve over time, along with potential avenues for future research, are described below:

- **Extending language features within RustSmith**

  One of RustSmith's key limitations is its ability to effectively test a wider array of Rust's language features. Providing support for features such as generators, pointers, unsafe code, traits, etc. would both increase coverage of the Rust compiler, and increase RustSmith's ability to detect new bugs. Language features that are least covered by RustSmith can be prioritized by using information from Table 5.3.

- **Designing new techniques to allow for less conservative generation**

  Another key limitation within RustSmith is how conservative its approaches are. RustSmith utilizes conservative approaches during generation, such as mutable reference liveness (described in Section 3.5.2), and the inability for lifetimes to be coerced, since the priority was to generate *valid* Rust programs. Designing new techniques that would allow RustSmith's generation in these areas to be less conservative (for example, by allowing

lifetime coercion) should exercise more of the borrow checker and increase the chances to detect new bugs.

- **Integrate automated test-case reduction within RustSmith**

  During the evaluation of bugs detected by RustSmith, as well as during the analysis of RustSmith's extra coverage of specific optimizations, manual reduction of the test-programs was performed. With more time, we would have aimed to create an automated test-case reduction system. This would utilize the reconditioner built within RustSmith (described in Section 4.1.1) along with a language agnostic test-case reducer to automatically reduce a file based on whether a reduced program is "interesting" or not (as described in Section 2.6). In our use case, "interesting" would mean "still produces a bug-inducing file" when analysing bugs detected, and would mean "still covers extra lines in an optimization (when compared to the original test suite)" when analysing the extra coverage RustSmith can achieve on optimizations.

- **Investigating alternative fuzzing techniques**

  RustSmith can be used as the starting point for many different tools that can utilize different fuzzing techniques. For example, programs generated by RustSmith could then be further *mutated* to produce more interesting programs. Additionally, equivalence testing (as described in Section 2.5) could be used instead of differential testing by mutating generated files in *semantically preserving* ways. Given that only one compiler for Rust currently exists, this may produce interesting results.

  Additionally, RustSmith currently performs "black-box fuzzing", as it uses no internal knowledge of the Rust compiler while generating programs. However, given that we have now managed to get detailed coverage statistics about line and function coverage within the Rust compiler, extending RustSmith to perform *coverage-guided fuzzing* may help us achieve higher coverage, which could therefore lead to more bugs being detected. It is important to note, however, that gathering coverage after compiling a ≈2000 line file takes between 30 seconds to a minute, and so this may drastically slow down RustSmith's generation speed.

6.2 Ethical Considerations

While this tool has been created to help find compiler bugs with the end goal of making the compiler it tests more stable and reliable, this may not be how others perceive it. As with any software, finding bugs and fixing them is a big part of its development, but bugs are also a large part of what malicious users will try to make use of. If a malicious person finds a bug in a piece of software, they could attempt to exploit it in different ways.

Furthermore, with such a crucial piece of software like a compiler, where its reliability has a direct impact on all applications that are built upon it, bugs in the hands of a malicious user can cause severe consequences. There are even cases where compiler bugs can be exploited accidentally, such as the Unix sudo tool that was compromised due to a publicly known bug in LLVM [43]. Therefore, it is properly disclaimed that this piece of software should not be used in any malicious way, such as trying to find "compiler backdoors", for example.

A | Grcov HTML outputs

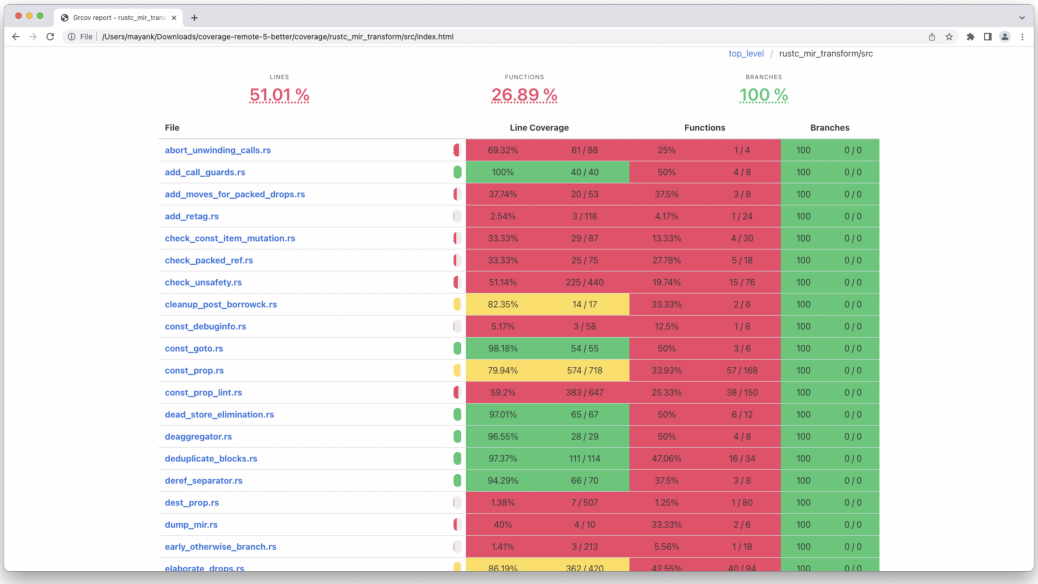

Figure A.1: Top level view of statistics within the optimization module of the Rust source code

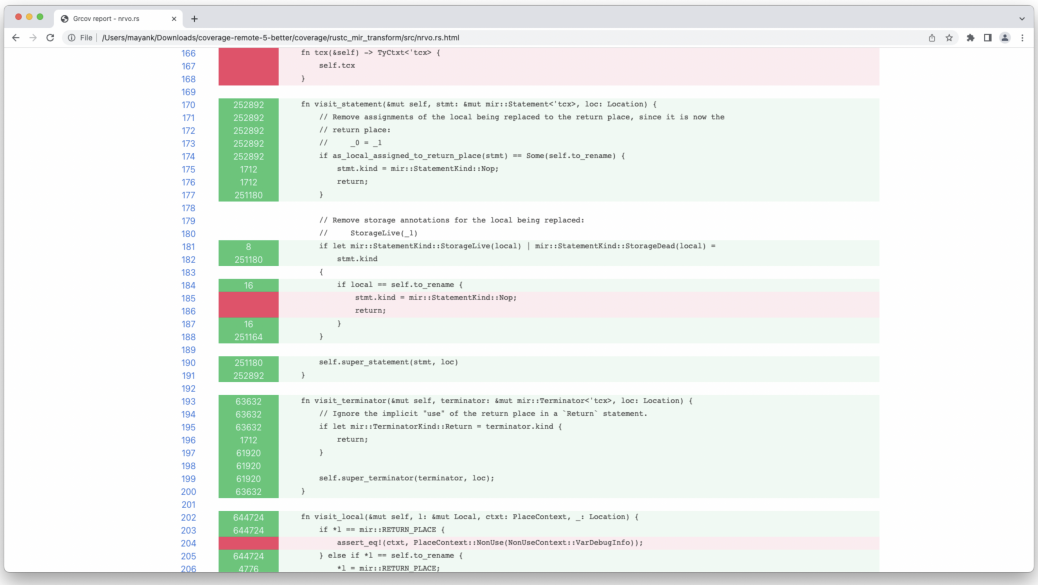

Figure A.2: Specific view of line coverage within the `nrvo.rs` optimization file

Bibliography

[1] Rust Homepage; 2022. Available from: https://www.rust-lang.org.

[2] Groce A, van Tonder R, Kalburgi GT, Le Goues C. Making No-Fuss Compiler Fuzzing Effective. In: Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction. CC 2022. New York, NY, USA: Association for Computing Machinery; 2022. p. 194–204. Available from: https://doi.org/10.1145/3497776.3517765.

[3] 9 Companies That Use Rust in Production; 2020. Available from: https://serokell.io/blog/rust-companies.

[4] Stack Overflow Developer Survey 2021; 2021. Available from: https://insights.stackoverflow.com/survey/2021.

[5] Sun C, Le V, Zhang Q, Su Z. Toward Understanding Compiler Bugs in GCC and LLVM. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ISSTA 2016. New York, NY, USA: Association for Computing Machinery; 2016. p. 294–305. Available from: https://doi.org/10.1145/2931037.2931074.

[6] The real story behind the Java 7 GA bugs affecting Apache Lucene / Solr; 2011. Available from: https://blog.thetaphi.de/2011/07/real-story-behind-java-7-ga-bugs.html.

[7] LLVM Documentation; 2021. [Online; accessed 15-Jan-2021]. https://llvm.org/docs/Passes.html.

[8] Nagai E, Awazu H, Ishiura N, Takeda N. Random Testing of C Compilers Targeting Arithmetic Optimization; 2012. .

[9] Yang X, Chen Y, Eide E, Regehr J. Finding and Understanding Bugs in C Compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '11. New York, NY, USA: Association for Computing Machinery; 2011. p. 283–294. Available from: https://doi.org/10.1145/1993498.1993532.

[10] Livinskii V, Babokin D, Regehr J. Random Testing for C and C++ Compilers with YARPGen. Proc ACM Program Lang. 2020 nov;4(OOPSLA). Available from: https://doi.org/10.1145/3428264.

[11] Le V, Afshari M, Su Z. Compiler Validation via Equivalence modulo Inputs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '14. New York, NY, USA: Association for Computing Machinery; 2014. p. 216–226. Available from: https://doi.org/10.1145/2594291.2594334.

[12] Donaldson A, Evrard H, Lascu A, Thomson P. Automated testing of graphics shader compilers. ACM; 2017. p. 1–27. Available from: http://dx.doi.org/10.1145/3133917.

[13] Nagai E, Hashimoto A, Ishiura N. Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions. IPSJ Trans Syst LSI Des Methodol. 2014;7:91–100.

[14] Kossatchev A, Posypkin M. Survey of Compiler Testing Methods. Programming and Computer Software. 2005 01;31:10–19.

[15] Bazzichi F, Spadafora I. An Automatic Generator for Compiler Testing. IEEE Transactions on Software Engineering. 1982;SE-8(4):343–353.

[16] Wang X, Chen H, Cheung A, Jia Z, Zeldovich N, Kaashoek M. Undefined behavior: What happened to my code? 2012 07;1.

[17] Donaldson A. Program Reconditioning: Avoiding Undefined Behaviour During Compiler Test Case Reduction; 2021. *Under Submission*.

[18] Groce A, Zhang C, Eide E, Chen Y, Regehr J. Swarm Testing. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. ISSTA 2012. New York, NY, USA: Association for Computing Machinery; 2012. p. 78–88. Available from: https://doi.org/10.1145/2338965.2336763.

[19] Howden WE. Theoretical and Empirical Studies of Program Testing. IEEE Transactions on Software Engineering. 1978;SE-4(4):293–298.

[20] Chen TY, Cheung SC, Yiu SM. Metamorphic Testing: A New Approach for Generating Next Test Cases; 2020.

[21] McKeeman WM. Differential Testing for Software. Digit Tech J. 1998;10:100–107.

[22] Chen J, Hu W, Hao D, Xiong Y, Zhang H, Zhang L, et al. An Empirical Comparison of Compiler Testing Techniques. In: Proceedings of the 38th International Conference on Software Engineering. ICSE '16. New York, NY, USA: Association for Computing Machinery; 2016. p. 180–190. Available from: https://doi.org/10.1145/2884781.2884878.

[23] Sun C, Le V, Su Z. Finding and Analyzing Compiler Warning Defects. In: Proceedings of the 38th International Conference on Software Engineering. ICSE '16. New York, NY, USA: Association for Computing Machinery; 2016. p. 203–213. Available from: https://doi.org/10.1145/2884781.2884879.

[24] How to submit a LLVM bug report;. Accessed: 2022-01-22. https://llvm.org/docs/HowToSubmitABug.html.

[25] Matsakis N. What's Unique about Rust?; 2019. Available from: https://nikomatsakis.github.io/rust-latam-2019/.

[26] Rust Documentation - What is Ownership?;. Accessed: 2022-01-23. https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.

[27] Lifetime Coercion; 2021. Available from: https://doc.rust-lang.org/rust-by-example/scope/lifetime/lifetime_coercion.html.

[28] Shore J. Fail fast [software debugging]. Software, IEEE. 2004 10;21:21 – 25.

[29] Database Snapshots (SQL Server); 2021. Available from: https://docs.microsoft.com/en-us/sql/relational-databases/databases/database-snapshots-sql-server?view=sql-server-ver16.

[30] Lifetime Elision; 2021. Available from: https://doc.rust-lang.org/nomicon/lifetime-elision.html.

[31] Chaliasos S, Sotiropoulos T, Spinellis D, Gervais A, Livshits B, Mitropoulos D. Finding Typing Compiler Bugs. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2022. New York, NY, USA: Association for Computing Machinery; 2022. p. 183–198. Available from: https://doi.org/10.1145/3519939.3523427.

[32] Kotlin programming language;. Available from: https://kotlinlang.org/.

[33] Leader and Followers; 2020. Available from: https://martinfowler.com/articles/patterns-of-distributed-systems/leader-follower.html.

[34] Github: beanstalkd; 2021. Available from: https://github.com/beanstalkd/beanstalkd.

[35] Docker; 2022. Available from: https://www.docker.com.

[36] Godbolt; 2021. Available from: https://godbolt.org.

[37] Non-inlined recursive function produced different value when optimised; 2021. Available from: https://github.com/rust-lang/rust/issues/98192.

[38] LLVM loop optimization can make safe programs crash; 2021. Available from: https://github.com/rust-lang/rust/issues/28728.

[39] ConstProp miscompilation; 2022. Available from: https://github.com/rust-lang-ci/rust/commit/dafe18cecf6789ba8b08b0848132cc04a9b4ecf8.

[40] Grcov Homepage;. Accessed: 2022-01-22. https://github.com/mozilla/grcov.

[41] Drop Elaboration; 2021. Available from: https://rustc-dev-guide.rust-lang.org/mir/drop-elaboration.html.

[42] time: OpenGroup; 2021. Available from: https://pubs.opengroup.org/onlinepubs/9699919799/utilities/time.html.

[43] Defending Against Compiler-Based Backdoors; 2015. Available from: https://blog.regehr.org/archives/1241.