Rust's Borrow System in Static Analysis

Exploring Usages and Benefits of Rust's Borrow System Through Static Taint Analysis

Felix Cho Petersen
Mathias Knøsgaard Kristensen
Simon Vinberg Andersen

June 16, 2022

AALBORG UNIVERSITY
STUDENT REPORT

Rust's Borrow System in Static Analysis

Exploring Usages and Benefits of Rust's Borrow System Through Static Taint Analysis

Group Name:

CS-22-DS-10-06

Supervisor:

Danny Bøgsted Poulsen
René Rydhof Hansen

Group Members:

Felix Cho Petersen
Mathias Knøsgaard Kristensen
Simon Vinberg Andersen

Project Page Count:

60

Total Page Count:

75

This report explores potential benefits that Rust's borrowing system might confer to static analyses. Furthermore, this report showcases some of these benefits via application in a static taint analysis.
We did this by first establishing an overview of the borrowing system, the stages of the Rust compiler, and exploring the novel non-lexical lifetimes concept which is the current basis for the borrowing system.
We then established a definition for what 'borrowing' means in a Rust context, and what precisely this entails. Next, we defined predicates in natural language, to limit the broader borrowing definition presented before. After this, a syntax and instrumented semantics for Rust's MIR compilation layer is presented. This is followed by definitions for some of the predicates, written in more precise boolean algebra.
A static taint analysis is presented to ascertain whether the instrumentation and boolean algebra holds. Furthermore, we presented the basis for an analysis based in non-lexical lifetimes. The analysis is never implemented, but we present the theoretical basis for possibly reducing the state space needed to search through for a given program.
The report concludes with an evaluation of the theory presented. We conclude that analyses may benefit from the borrowing system, by leveraging it to possibly reduce the total amount of program that requires analysis. However, nothing conclusive can be drawn from the results due to the lack of proper implementation. We deem the project a partial success, both deserving of and requiring additional work.

Preface

This report is written by the university group 'CS-22-DS-10-06' during the Spring semester of 2022 at Aalborg University. This project is a master's thesis in the field of distributed software which explores the potential benefits that Rust's borrowing system provides for static analysis.

We would like to give special thanks to Emil Njor and Hilmar Gústaffson for their mini-seminar on their work "Static Analysis in Rust"[1], which was used in our master's thesis, in addition to being available for questioning during the creation of this project.

Aalborg University, June 16, 2022.

Felix Cho Petersen
fcpe17@student.aau.dk

Mathias Knøsgaard Kristensen
matkri15@student.aau.dk

Simon Vinberg Andersen
svan17@student.aau.dk

Contents

Chapter 1

Introduction

As technology advances, the complexity of software required to run the technology rises. This leads to a higher potential for errors, which in turn increases developer workload as fixes become necessary.

Common Weakness Enumeration (CWE) is a community-driven information aggregation list maintained by Mitre corporation, which gathers information about common software weaknesses [2]. The CWE defines a weakness as *"flaws, faults, bugs, or other errors in software or hardware implementation, code, design, or architecture that if left unaddressed could result in systems, networks, or hardware being vulnerable to attack."*[3]

Since 2019, CWE has released a top 25 list of weaknesses, ranking them based on both their prevalence and severity [4, 5, 6]. The weaknesses are given a score meant to reflect the relationship between these attributes, such that a very common and very severe weakness will receive a high score. A severe weakness that is very rare, or a very common weakness that has little impact, will gather a lower score [7].

Rank	ID	Name	Score
1	CWE-787	Out-of-bounds Write	65.93
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.84
3	CWE-125	Out-of-bounds Read	24.9
4	CWE-20	Improper Input Validation	20.47
5	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	19.55
6	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	19.54
7	CWE-416	Use After Free	16.83
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.69
9	CWE-352	Cross-Site Request Forgery (CSRF)	14.46
10	CWE-434	Unrestricted Upload of File with Dangerous Type	8.45

Table 1.1: CWE top 25 list for 2021, truncated to top 10 entries [6]

Table 1.1 show the top 10 entries for the 2021 CWE top 25 list. Of note is the fact that 3 of the entries are memory related (rank 1, 3, and 7), and 6 are directly related to user input (rank 2, 4, 5, 6, 8, and 10). We limit the scope of the problem domain to the problems depicted in the aforementioned CWE list. Based on this list, we argue that providing better tools to mitigate memory-related and input-related errors could prove beneficial for mitigating many of the prevalent problems modern programmers are faced with.

We intend to conduct a study on the Rust programming language. The reasoning for picking Rust is twofold: First, by basing our solution in Rust, we automatically solve some of the memory-related issues presented in the CWE via Rust's unique ownership system, which we will explain in more detail at a later point. Second, we postulate that the borrowing system in Rust can help making some types of static program analysis more accurate. In theory, the Rust borrowing system should allow us to make certain assumptions when creating analyses. We postulate that we can circumvent aliasing, making it easier to analyse elements that access the heap.

We intend to employ a type of static analysis to verify whether any benefits gained from the borrowing system holds. One approach that we deem beneficial for verifying these theories would be via taint analysis, which aids in tracking and sanitising user input. Furthermore, we hypothesise that it should be possible to improve upon the taint analysis by marking taint sources in Rusts mid-level intermediate representation layer (MIR) control flow graph (CFG) as irrelevant in a manner akin to program slicing. As such, we intend to explore the following problem:

How does a static taint analysis benefit from the unique borrowing system employed by Rust?

To answer this we intend to:

- Formalise a definition of what 'borrowing' is, and what it allows.
- Create a set of predicates that define how the type system is allowed to use the 'borrowing' concept.
- Formalise semantic rules for the Rust's MIR layer.
- Define a formal set of instrumentation for the semantics.
- Create a taint analysis which employs the instrumentation to ascertain whether the borrowing system improves accuracy in analyses.

Chapter 2

Rust Overview

To formalise semantic rules, predicates, and instrumentation over Rust, it is necessary to understand how Rust is structured and what its novel features are.

Furthermore, Rust does not provide a formal definition of what 'borrowing' is, and instead only alludes to what it is via use-case examples in their documentation. As such, it is imperative that we also define what we mean by borrowing, covering what it is, and how it functions in Rust.

2.1 The Rust Borrow System

One of the defining aspects of the Rust programming language is its ownership-based memory-management system. This ownership system allow Rust to automatically handle memory-management on behalf of users without the need to implement advanced garbage collection as seen in higher level programming languages like Java or C#.

2.1.1 Ownership

To our knowledge, Rust provides no formal definition of ownership, but instead provides an explanation of how the system works, what is allowed, and what is not allowed [8]. We will attempt to give a more concise definition, that is still broad enough to cover all cases. On an intuitive level the Rust ownership system binds a value in memory to a single variable. Borrowing in Rust can be done in two ways - mutably or immutably.

A mutable borrow gives ownership of a value to a variable that is not the original holder. Ownership confers read and write privileges to the owner. In the case of mutable

borrows it also gives exclusivity from all other borrows, such that a value can only be borrowed mutably by a single variable at a time.

```
1  // foo is assigned to vector containing the elements 1, 2, and 3
2  // Note that foo is stated to be 'mut', meaning it is mutable
3  let mut foo = vec![1,2,3];
4
5  // bar made to be a mutable reference of foo
6  let bar = &mut foo;
7  bar[0] = 5;
8
9  // This will now print '5'
10 println!("foo[0] is: {}", foo[0])
```

Listing 2.1: An example of a mutable reference in use.

Listing 2.1 shows an example of a mutable reference. Line 3 declares a mutable variable `foo` which holds a vector with the elements 1, 2, and 3. Ownership and read/write privileges, are passed by reference to a new variable `bar` on line 6. Lastly we reassign the first element of the vector so that it holds 5 instead of 1, and then finally we print the first element of the original variable `foo` showing us that changes to `bar` were conferred to `foo` as intended.

Alternatively, an immutable borrow suspends write privileges from the original owner while the borrow is in effect. Immutable borrows also do not take ownership from the original owner, instead only sharing read privileges of the value in question. This allows us to create any number of immutable references at the same time.

```
1  // foo is assigned to vector containing the elements 1, 2, and 3
2  let foo = vec![1,2,3];
3
4  // bar made to be a reference of foo
5  let bar = &foo;
6  let foobar = &foo;
7
8  // Through the reference bar, we can print the elements of foo, and
9  // foobar as if they have ownership.
10 println!("foo[0] is: {}", foo[0]);
11 println!("foobar[2] is: {}", foobar[2]);
```

Listing 2.2: An example of an immutable reference in use.

Listing 2.2 is an example of an immutable reference. This example resembles the one in Listing 2.1 with the only difference being that `foo` is declared as immutable, and subsequently borrowed by `bar` and `foobar` as such, this also shows that it only passes reading privileges. However if a change would be made to either of the borrowed variables, would result in the compiler throwing an error if `bar` attempts to mutate `foo`.

2.2 Rust Compiler

The Rust compiler is structured into several intermediate representation (IR) layers which help the compiler process and convert Rust source code into LLVM. Each layer of the Rust compiler contributes to the compilation process, by being the stages for type-checking, parsing and borrow checking [9].

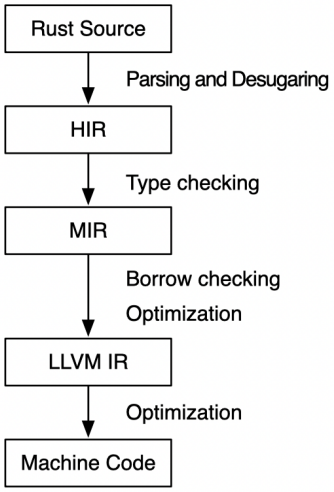

Figure 2.1: Illustration of the Rust Compiler Structure [9].

Figure 2.1 shows the structure of the Rust compiler, and what each layer adds to the compilation process.

All stages of the Rust compiler will be explained to some degree, but work conducted in this report relates almost exclusively to MIR. As such, MIR will receive a more in-depth explanation than the other stages of the compilation process.

2.2.1 High-level Intermediate Representation

The primary purpose of the high-level intermediate representation (HIR) layer is to type check the source Rust code. This type checked code is outputted as a data-structure called a ”crate”, which contains maps and identifiers to organise and access the contents. Furthermore, HIR also removes some of the syntactical sugar by simplifying expression forms, like `for` loop statements being changed into the more generic `loop` statements.

HIR acts as the compiler-friendly version of the AST and is one of two layers in the

compilation process responsible for converting Rust source code into LLVM machine code.

2.2.2 Mid-level Intermediate Representation

MIR is the layer that serves as a bridge between the HIR and LLVM layers. MIR was introduced in Rust 1.11 as an improvement to the compilation process, as the transformation of HIR to LLVM IR was deemed too comprehensive for a single compilation step[9]. As such, some of the work that needed to be done on the HIR output was placed into a completely separate compiler step, MIR, to break this previously comprehensive step into smaller bites in the compiler. By making this separation it is also possible to perform additional optimisations on the code, by further desugaring the HIR output into simpler primitives.

Some of the main benefits from adding MIR is, according to the Rust development team, that it will improve compilation time, improve type-checking, and help reduce redundancy [9].

MIR is constructed by converting syntactical constructs in HIR into a collection of simpler, and more explicit, constructs with equivalent behaviour. These constructs are created from Basic Blocks, which are based on the smaller MIR syntax. Each Basic Block contains a series of statements and a single terminator. The terminator in each Basic Block ensures that the these blocks run sequentially, by pointing to the next Basic Block which must be executed. An example of this structure can be seen in Figure 2.2.

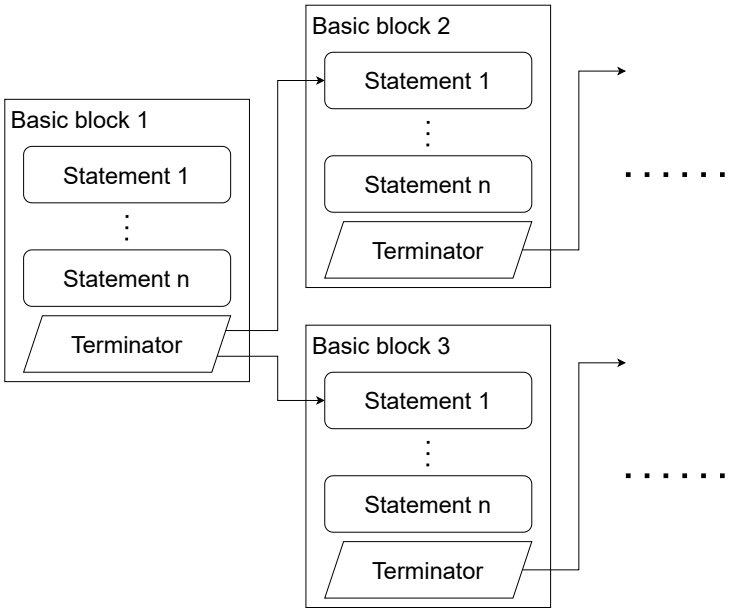

Figure 2.2: Example of the Basic Block Structure.

After converting the HIR output into Basic Block structures, MIR then uses them to construct a CFG which is used by subsequent IR level. This reduction of Rust syntax, results in the remaining code becoming more explicit, which is desirable for the conversion to LLVM [9].

As such, these Basic Blocks serve as the main data structures to perform data-flow analysis on, as the CFG provides the possible execution paths for the program, while retaining the information of the borrows. This additional information is crucial for the borrow checker to work as it is used to calculate non-lexical lifetimes, which are used to denote lifetimes for references and variables. These non-lexical lifetimes will be explained further in Section 2.3.

Borrow Checker

A unique feature of Rust is the borrow checker, which is done during the MIR compilation stage. The borrow checker ensures that each of the borrows in the program are valid. Borrow checking is split into two phases, namely a stage which splits the process into computing borrows, and one for reporting illegal borrows.

Phase one consists of identifying borrows which is done by looking at the in-scope borrows, at each point of the CFG using a fixed-point algorithm. These borrows are represented as a tuple $('a, L_{Type}, L_{Value})$, where $'a$ is the lifetime of the borrow, L_{Type}

is the borrow type, which can be shared, mutable or unique, and L_{Value} is the lvalue that is borrowed. Each assignment's right-hand side, found in MIR, is used to create a borrow tuple, which is given an unique index. Then, for any statement at a Point P in the CFG, a transfer function can be defined to tell us which borrows it brings in or out of the scope. According to Rust RFC 2094, such a transfer function is defined as the following [10]:

- Any borrow whose region does not include P is killed.
- If it is a borrow statement, the corresponding borrow is created.
- If this is an assignment $lv = \langle rvalue \rangle$, then any borrow for some path P of which lv is a prefix is killed.

Phase two consists of reporting the illegal borrows identified during phase one. Each statement is classified as one of two types of actions when traversing the CFG, namely accessing an lvalue or dropping an lvalue. Furthermore, when accessing an lvalue, Rust makes an additional distinctions which are measured on two different axes - shallow/deep and read/write [10].

- **Shallow** means that the immediate field which reaches the lvalue is accessed, but pointers and references are not dereferenced.
- **Deep** means that all data that is reachable through an lvalue might be invalidated or accessed using this action.
- **Read** means that the data is not modifiable and actions may only read the data.
- **Write** means that the data is modifiable and actions can make changes to the data.

The borrow checker iterates through each statement in the CFG to verify whether the action is valid, according to a set of rules that can be found in the NLL RFC [10].

The dropping of an lvalue is treated as a deep write, meaning the same as a `move` operation. However, there are still details missing on precisely how this works, due to much of this still being in active development [10].

2.3 Non-Lexical Lifetimes

An interesting component of the Rust borrow system is its use of non-lexical lifetimes for verifying borrow validity. However, before going into what non-lexical lifetimes are, it is important that the reader first understands what lexical scopes are.

Lexical scopes, also referred to as static scoping, is a scoping ruleset that describes how to resolve name aliasing in variables that are not in the same scope. Lexical scoping allows name aliasing between scope levels, while also allowing variables from parent scope(s) to be accessed by child scopes. An example of this can be seen in Listing 2.3.

```
1  fn main() {
2      // Create variables
3      let x = 10;
4      let y = 3;
5      {
6          // This also creates a variable, but only for this scope
7          let x = 5;
8          println!("{}", x); // Prints 5
9          // Grab the y from parent scope
10         println!("{}", y); // Prints 3
11     }
12     // Print x from current scope
13     println!("{}", x); // Prints 10
14 }
```

Listing 2.3: A brief example of static scopes in Rust.

In Listing 2.3 the innermost print statement found on line 9 prints the value associated with the y variable. This would be the value that is assigned on line 4 in the outer scope, resulting in the program printing the value 3. In contrast, the second print statement found on line 13 would print the value of x declared on line 3. Intuitively one might assume that it would print the x declared on line 7, as it is the most recent declaration. However, due to the fact that the lifetime of the second assignment of x only persists until we exit the scope on line 11, we instead print the initial declaration from line 3.

Contrary to lexical scopes which derive meaning directly from the program code, non-lexical lifetimes instead derive meaning from a CFG. Specifically, non-lexical lifetimes in Rust are derived from the MIR used internally by the compiler, and in turn specify the lifetimes based on execution paths [10].

```
1  fn process_or_default() {
2      let mut map = ...;
3      let key = ...;
4      match map.get_mut(&key) { // -------------+ map lexical
5          Some(value) => process(value),     // | lifetime
6          None => {                          // |
7              map.insert(key, V::default()); // |
8              //  ^^^^^^ STATIC ERROR.           // |
9          }                                  // |
10     } // <-----------------------------------+
11 }
```

Listing 2.4: Snippet of Rust Code that produces error with static scoping but not with NLL. Taken from Rust's NLL RFC [10].

Listing 2.4 shows a code snippet written in Rust. The code attempts to find a value within a map using a unique key. If it finds a value associated with the key, it processes

that value. If no value is found it instead inserts the key into the map with a default value. It is worth noting, that this code snippet would produce an error if one were to employ static scoping. This is due to a borrow conflict of `map` on lines 4 and 7. When we borrow `map` through `get_mut()` on line 4 it is used by the `match` statement in its entirety, and as such extends the borrow until the end of the scope, despite not necessarily being used after line 4. This becomes a problem if the `None` branch tries to borrow the key on line 6, as `map` would already be borrowed mutably. If we instead employ non-lexical scopes, the compiler would realize that the `None` branch does not need to uphold the mutable borrow, and could therefore drop it to allow a new borrow. This is because NLLs try to minimise the set of places in a program wherein a borrow should be considered 'in use', such that more programs are accepted as correct.

To ensure that the borrows in the NLL are correct, an analysis is performed to determine if all borrows are correct. The NLL analysis uses 3 concepts - liveness, subtyping and reborrow constraints - to build the full set of borrowing constraints for the lifetimes in a program. These lifetime constraints are initially created independently, and are then gathered into a final set through a fixed-point iteration.[10]

In relation to non-lexical lifetimes, liveness refers to whether the lifetime of a variable is alive. A lifetime will be considered 'live' at a point if, for some Point in the program, that value will be used in the future. Alternatively, if the value has no uses it will instead be considered dead until a new value is assigned. Once the liveness analysis is done, we can derive a set of liveness constraints from the generated lifetimes.

Subtyping is a type of constraint which ensures that the original variable will outlive other variables that reference it. In MIR, this is extended to be location-aware, meaning that the Point where the copy happens is considered as well.

Reborrow constraints relate to when values are borrowed more than once, such as a value being borrowed and the resulting reference then subsequently being (re-)borrowed. In essence, reborrow constraints ensure that the lifetimes of the original values outlive any borrow or sequence of reborrows. These constraints are often complex, and so the Rust compiler employs 'supporting prefixes' to define said constraints. Supporting prefixes of a value are found by stripping away fields and dereferences, until a point is reached where there is nothing left to strip, or a shared reference is found.

Immutable reborrows may be ignored when no longer relevant, as they cannot be further used to modify the referred value. Mutable borrows, however, create nested lifetime constraints where intermediate borrows cannot be ignored due to them potentially being used to reassign values at a later point.

Chapter 3

Static Analysis

As this body of work focuses on ascertaining any potential benefits that borrow checking in Rust might lend to static analyses, we must first gain an understanding of what a static analysis is.

A static analysis is a type of automated analysis, wherein one reasons about a program without actually executing it. Static analysis is done for many reasons, with examples being for the purposes of optimisation by compilers, verifying correctness via examination of properties, or identifying vulnerabilities in the source code.

Static analysis is an analysis approach that has been widely used for many applications and fields, and as such there are many different ways to approach static analysis.

3.1 Soundness

It is important that we establish a way to reason about the quality of analyses before delving further into static analyses. One way to do this is to measure the 'soundness' of the analysis.

For an analyser to be sound, all reported errors from the analyser must be either true positives, or unknown. Unknown in this context means that the reported error could be a true or false positive. To achieve this, it is common for analyses to over-approximate the program behaviour, to make sure that every case is covered. A trivial analyser that is sound, would be an analyser that reports true for all possible cases, though this is obviously not very useful.

Soundness, in relation to static analysis, will be defined as such:

Definition 3.1.1. A static analyser is sound if the analysis only returns yes or maybe, where all of the program's behaviour is captured.

3.2 Program Approximation

It is not possible to create an analysis that is able to precisely model all program behaviour, as proven by Rice's theorem which states the following:

Definition 3.2.1. Any non-trivial property about a language, which is recognised by a Turing machine is undecidable.

As such, it is necessary to make use of approximations of the program behaviour, either via an over- or under-approximation. These two approximation approaches help to identify possible program executions, and serve as a way to either remove false negatives or false positives, depending on the approach.

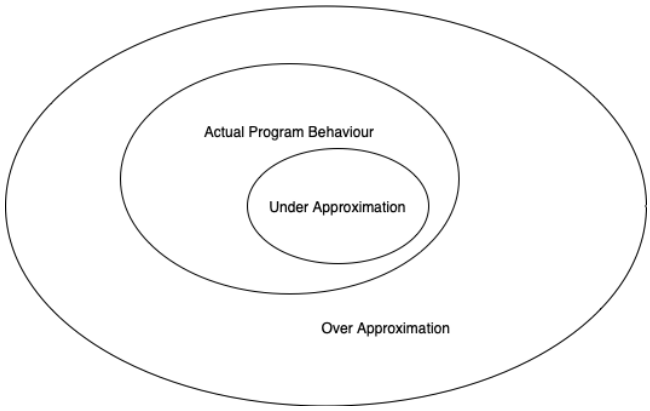

Figure 3.1: Illustation of Approximations

Seen in Figure 3.1, is an illustration how both over- and under-approximations aim to cover program behavior. An over-approximation aims to encompass more than the actual program behaviour, while an under-approximations aims to narrow in and thus potentially only covers a fraction of the actual program behaviour.

As the name implies, an approximation is essentially an estimate, meaning that it can always be improved upon. To date there is no approximation techniques which can be deemed universally applicable, as both have pros and cons depending on the use-case.

While these approximations might not accurately model the right behavior, they still provide some guarantee for the given properties being analysed.

3.3 Partially-Ordered Sets and Lattices

Lattice theory is a subcategory of order theory, which uses partially-ordered sets such that each pair in the set has an upper and lower bound.

One approach to static analysis is via use of these lattices as a way to create an abstract representation of the information being searched for. A simple lattice *Taint* for taint analysis could use 2 elements for 'tainted' and 'untainted', such that 'untainted' $\sqsubseteq$ 'tainted'.

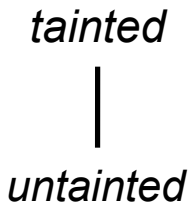

Figure 3.2: An example of a two state lattice.

Figure 3.2 shows one such lattice, only containing two states, "*tainted*" and "*untainted*". While this could also be structured as a binary relationship represented by a binary value in some way, using a lattice ensures that we can expand it in the future should the analysis ever need it.

We will be using the definitions of partial orders and partially-ordered sets, as seen in "Static Program Analysis" by Anders Møller and Micheal I. Schwartzbach [11].

Definition 3.3.1. A partial order is a set S equipped with a binary relation $\sqsubseteq$ where the following properties are satisfied:
Reflectivity: $\forall x \in S : x \sqsubseteq x$
Transitivity: $\forall x, y, z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$
Anti-Symmetry: $\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$

Definition 3.3.2. $y \in S$ is an upper bound to X, written $X \sqsubseteq y$, if $\forall x \in X : x \sqsubseteq y$
$y \in S$ is a lower bound to X, written $y \sqsubseteq X$, if $\forall x \in X : y \sqsubseteq x$

A least upper bound is written $\bigsqcup X$, and a greatest lower bound is written $\bigsqcap X$.

Definition 3.3.3. Least upper bound: $X \sqsubseteq \bigsqcup X \wedge \forall y \in S : X \sqsubseteq y \implies \bigsqcup X \sqsubseteq y$
Greatest lower bound: $\bigsqcap X \sqsubseteq X \wedge \forall y \in S : y \sqsubseteq X \implies y \sqsubseteq \bigsqcap X$

Lattices can be used as an algebraic structure for partially ordered sets, which consist of a set S and two binary operations "Join" and "Meet". These are denoted as $\vee$ for join which is the least upper bound for a pair of elements x and y, and the notation for meet $\wedge$ denotes the greatest lower bound for elements x and y.

A lattice where both the top (greatest) element and the bottom (least) element is unique, is called a bounded lattice. However, it is important to note that not every partially ordered set contains a least upper bound or greatest lower bound.

Definition 3.3.4. A bounded lattice is a lattice where the unique greatest element is denoted $\top$ (top), and the unique smallest element is denoted $\bot$ (bottom)

A lattice where all subsets have a join and meet is called a complete lattice. This also means that all non-empty lattices of finite sizes are complete lattices.

Definition 3.3.5. A complete lattice is a partial order $(S, \sqsubseteq)$ where $\bigsqcap X$ and $\bigsqcup X$ exist for all $X \subseteq S$

3.4 Data-Flow Analysis

Data-flow analysis is a lattice-based technique meant for gathering information about a set of values. Traditionally, data-flow analysis revolves around a CFG and a complete lattice with finite height. The lattice is used to describe abstract information which is inferred for the different CFG nodes. Each node is then assigned a data-flow constraint which relates the value of a variable in the node to those of other nodes. Finally, using a fixed-point algorithm these nodes can then be used to compute the analysis results, as a unique least solution if all constraints for a given program happen to be equations or inequations with monotone right-hand sides [11].

3.4.1 Taint Analysis

Taint analysis is a type of data-flow analysis which aims to detect vulnerabilities in the source code. This could, for example, be via SQL injection attacks that attempt to exploit user input fields [12]. Another example could be usage of a bad random number

generator in use with generating cryptographically secure keys, as the entropy would be much lower, potentially making the keys easier to crack.

Taint analysis examines where tainted values propagate to, and whether it is possible to use them before they are sanitised. Formally, a value must fulfill two conditions to be considered tainted: the value must fall outside of the expected domain of something that uses the value, and the value must be derived from external input to the program. If these are fulfilled, the value is tainted, and its origin is a tainted source [13]. As an example, a text input could easily be a taint source, as a value outside of the expected domain could be submitted. A choice between enumerated options cannot be a taint source, as all potential input would be within a set of expected values. Values that are tainted are not necessarily known to be outside of the expected domain. It is merely not known with any certainty whether they are in the expected domain.

```
let mut line = String::new();
let user_in = std::io::stdin().read_line(&mut line).unwrap();
let sql = format!("SELECT * FROM Users WHERE username = '{}'", user_in);
...
```

Listing 3.1: A faulty Rust program vulnerable to SQL injection attacks.

Listing 3.1 shows the basic idea of a program that is vulnerable to an SQL injection attack. In this snippet, an attacker could give the string "`username'; DROP TABLE Users --`" as input, which, if `sql` is not changed, could result in the Users table being dropped, and any following instructions being commented out. Sanitation of the input may search specifically for SQL injection attacks, and instead of the full input truncate it to just "`username`", or instead terminate the thread.

3.4.2 Fix-Point Algorithms

Fix-point algorithms and monotone functions are a way to perform data-flow analysis, which aims to preserve the order of the flow and guarantee termination. To ensure that the analysis will terminate, all functions in the analysis must be monotone [11].

> **Definition 3.4.1.** A function is monotone, if the order between inputs is preserved, where the inputs belong to some lattice L. This can be written as: $\forall x, y \in L : x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$

To begin analysing a program, an equation system is constructed for it. This equation system denotes a constraint variable, for each variable in the program [11].

Definition 3.4.2. In a complete lattice L, an equation over L is of the form

$$\begin{aligned} x_1 &= f_1(x_1, x_2, ..., x_n) \\ x_2 &= f_2(x_1, x_2, ..., x_n) \\ &\vdots \\ x_n &= f_n(x_1, x_2, ..., x_n) \end{aligned}$$

where $x_1, ..., x_n$ are variables, and $f_1, ..., f_n : L^n \to L$ are functions, which are called constraint functions. A solution to an equation system provides a value from L for each value, such that all equations are satisfied.

This equation system the n functions can then be further reduced into a single statement: $f(x_1, x_2, ..., x_n) = (f_1(x_1, x_2, ...x_n), ..., f_2(x_1, x_2, ..., x_n))$

This can then be further reduced to $x = f(x)$, which is the same as a fixed point for a function.

Definition 3.4.3. A fixed point for a function f is an input x where $x = f(x)$

To get the most precise result, the analysis should select the least fixed point.

Definition 3.4.4. A least fixed point x for a function f, is a fixed point where $x \sqsubseteq y$ for every fixed point y for function f

To find the least fixed point in a singular control-flow structure is trivial, but an increasingly complex control-flow with branching and iterative loops will make it more difficult to find. However, Kleen's theorem tells us that in a complete lattice with a finite height, monotone function will always have an unique least fixed point denoted $lfp(f)$ written as:

Definition 3.4.5. Per Kleen's theorem, a lattice of finite heigh with a monotone function will have a least fixed point, denoted lfp. The formula for this is as follows:

$$lfp(f) = \bigsqcup_{i \geq 0} f^i(\bot)$$

3.5 Interprocedural Analysis

Interprocedural analysis is a technique that examines the interactions between procedures. This is in contrast to intraprocedural analysis which is a technique for examining a single procedure in a vacuum [11].

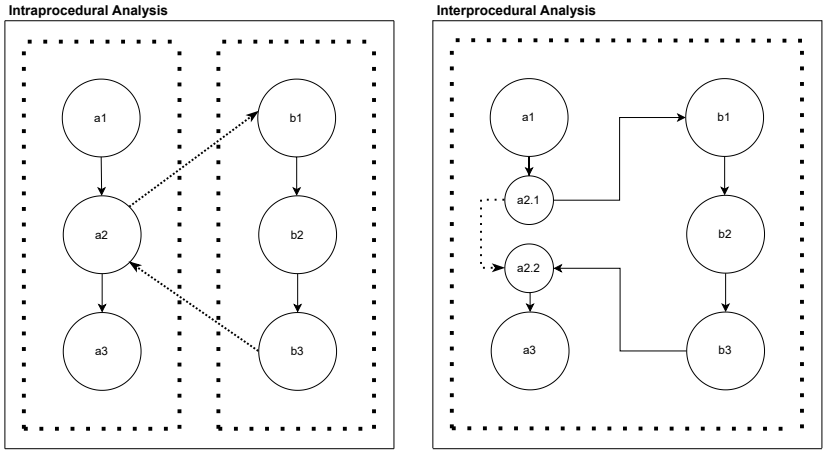

Figure 3.3: Example of intraprocedural and interprocedural analyses.

Figure 3.3 shows a simple example of how both intraprocedural and interprocedural analysis works. The thick dotted lines denote the scope of what is being analysed. In the intraprocedural analysis column we see two procedures `a` and `b` that are analysed on their own only accounting for whatever is found in the function body itself. The dotted arrows from `a2` to `b1` and from `b3` to `a2` denote a call in the source program, from function `a` to `b` which returns something. However, for the purposes of intraprocedural analysis, this is ignored as we look at the individual functions in a vacuum.

The "Static Program Analysis" book simplifies interprocedural analysis to always having the same form for every function call [11]. We extend this for showcasing purposes, resulting in the assumption that all function calls will be in a uniform format. This function call is then further split into a unique entry node and exit node. These can all be seen below:

$$\begin{aligned} \text{function call: } & x = f(...) \\ \text{entry node: } & \square = f(...) \\ \text{exit node: } & x = \square \end{aligned}$$

Now, looking at the interprocedural analysis column in Figure 3.3 again, we instead

see everything being analysed as one cohesive unit. This takes into account a call made in the `a2` node which jumps to the `b` function - a call which was originally ignored by the intraprocedural analysis. In the example, we split `a2` into two smaller nodes to simulate unique entry and exit nodes from the "Static Program Analysis" book.

3.6 Context Sensitivity

A consideration to make when performing interprocedural analysis is whether to have context sensitivity. Context sensitivity is the ability for an analysis to distinguish between call sources to a function, instead of the analysis marking a function once. As an example, context sensitivity and insensitivity could be expressed via the following program:

```
fn sink(input: String) {
    println!("{}", input); // Taint sink
}

fn main() {
    let taint = String::from("Tainted"); // Tainted variable
    let safe = String::from("Safe"); // Safe variable
    sink(taint);
    sink(safe);
}
```

Listing 3.2: A brief example of context sensitivity.

Listing 3.2 shows a function `sink` which takes a string and prints it, acting as an example taint sink. Using the lattice in Figure 3.2 for the analysis, a context insensitive analysis, the `sink` function would be marked as *Tainted* as this would be the least upper bound, between the call with the tainted and safe parameter. This will propagate back to both callers, and in turn mark both of them as tainted. However, with a context sensitive analysis, each caller would be analysed individually. The naive method to do this would be function cloning, where each call to a function would create a new unique and functionally equivalent function, but with the specific parameters given for the call. This, however, quickly increases the program size and could result in infinitely big program sizes if there are infinite mutually recursive functions in the program [11]. There are two main approaches which attempt to circumvent this: call string analysis and the functional approach.

3.6.1 Call String Analysis

The idea of call string analysis is to gain a similar effect to function cloning, but without performing any changes to the CFG.

In a call string analysis, the call nodes in the CFG will be structured into a set. It can

be viewed as the following: $Contexts = Calls^{\leq k}$, where *Calls* are the set of call nodes and k is a positive integer. The notation of $A^{\leq k}$ means the set of tuples of k or fewer elements from the set A, or more formally:

$$A^{\leq k} = \bigcup_{i=0,\ldots,k} A^i$$

Contexts will then contain of all the tuples consisting of calls on the call stack. The length of these tuples have to be bound to make sure that *Contexts* is finite. *Contexts* set also contains the empty call stack denoted by ε as an addition to the program calls.

An example, lets look at Listing 3.2 again, but while employing call sign analysis this time. For this example, let the two function calls to `sink` be denoted as c_1 and c_2, and the value of $k = 1$. The *Contexts* will then contain the two function calls, and the initialising call ε. We can then find the abstract value of `sink` by looking at the function entry at the two calls.

$$\begin{bmatrix} \varepsilon \mapsto unreachable, \\ c_1 \mapsto [taint \mapsto tainted, input \mapsto tainted], \\ c_2 \mapsto [safe \mapsto untainted, input \mapsto untainted] \end{bmatrix}$$

Here, *unreachable* is an extra element on the lattice, such that the lattice would be $unreachable \sqsubseteq S \sqsubseteq T$. The intuition for the element is simply that some states cannot be reached through normal program execution. For example, the initial call to `main` may have no information of any sort, and as such is *unreachable*.

By changing the value of k in $Calls^{\leq k}$, bigger and bigger call stacks can be accommodated in the analysis. Intuitively, a tuple $(c_1, \ldots, c_m) \in Calls^{\leq k}$ will identify exactly m calls on the callstack. If $m < k$, the tuple identifies a callstack up to height m, whereas if $m = k$, the tuple identifies a callstack of at least height m, as further calls are truncated [11].

3.6.2 Functional Approach

The functional approach is a type of analysis which tries to identify calls from the data derived from abstract states instead of looking at the control-flow from the call stack.

First, a lattice is constructed from the CFG, such that a node v is a map $m : States \rightarrow lift(States)$, where *States* is a lattice wherein each is a map that describes the program state for some given point, and the $lift$ function adds a new unique bottom element to the given lattice. Now we can express $m(s)$, where s is a state, as an approximation of the possible states of v, if v was entered from a state that matches s. Consequently, if there is no execution path for such a state, $m(s) = unreachable$. If the node v is a function exit node, the map m becomes a 'summary' of the function [11].

For the example in Listing 3.2, still using the lattice from Figure 3.2, we would get the following lattice element at the exit node of `sink`:

$$[\ S[input \mapsto tainted] \mapsto S[input \mapsto tainted, result \mapsto tainted],$$
$$S[input \mapsto untainted] \mapsto S[input \mapsto untainted, result \mapsto untainted],$$
$$all\ other\ contexts \mapsto unreachable\ ]$$

Chapter 4

Related Work

Analysis of Rust code is a subject which has been explored much in recent years. Larger bodies of work include the MIRI[14] and MIRAI[15] which revolve around analysing the MIR level of the Rust compiler. In particular, we have taken an interest in the ”Static Analysis in Rust” report created by Hilmar Gústafsson and Emil Jørgensen Njor [1]. This project explores static analysis done on the MIR level via implementation of a taint analysis.

In the project, Gústafsson and Njor first define and formalise syntax and semantics for a smaller part of MIR. In particular this covers the most basic assignments, calls, and switches. They utilise this set of semantics as the basis for an implementation of a static analysis tool which runs in Rust. Their implementation is built upon a simplified version of the MIRI interpreter project, which utilises a subset of the functionality that it provides, and as such follows the same project structure.

As mentioned, the focal point of their work is to create a taint analysis. Rather than creating a standalone tool to be used beside a compilation toolchain, they instead hook into the compiler allowing the analysis to run during normal compilation.

We intend to use their work as an inspirational basis for an analysis we create, which will be used to find taint paths for programs in Rust, in addition to serving as the basis for studying any benefits of Rusts borrow system on these types of analyses.

Chapter 5

MIR Semantics

In an effort to investigate whether the borrowing system of Rust lends anything to static analysis, we aim to create a taint analysis, as it is a simple type of static analysis. Gústafsson and Njor employ a similar analysis in ”Static Analysis in Rust”[1], which we draw inspiration from. However, we aim to create a more precise analysis that makes a more precise over-approximation.

To do that, we must first specify what borrowing allows and how we can utilise it. The borrowing definition described in Section 2.1.1 is too broad to be of use, and so we create a more precise clarification. This is done via a set of predicates that describe what the borrowing system is not allowed to do. This is done in natural language, and is based on the original definition from Section 2.1.1 along with relevant error codes pertaining to borrows and closures found in Rust. Furthermore, while functionally sound as a whole, the semantics employed by Gústafsson and Njor is formulated in a very unintuitive fashion. As such, we make an effort to rewrite their semantics via extensive renaming. In addition, we create an instrumentation for our semantics, which we use to create a set of boolean algebra formulas for some of the predicates, to more precisely describe and formalise the predicates and their exact function.

5.1 Borrow Checking Predicates

To properly utilise and analyse the borrow checking system, we must first define what it is, and what it is not allowed to do. Borrow checking in Rust is handled exclusively by the type system, however we deem it out of scope for this project to describe this system via structural operational semantics. Instead, we choose to establish a set of predicates, described in natural language, which limit the broader definition given in Section 2.1.1. For a program to be valid, it must adhere to all of the predicates in the

set.

These predicates are based on a subset of Rust compiler errors relating to the borrow checker and liveness properties. The full set of rules are taken from the 'Rust Compiler Error Index'[16], and can be viewed in Appendix A.

Most of these error codes are not used directly, but have been merged into a smaller and more concise list of predicates. This is in part due to many of the rules dealing with similar subjects. However, it is also due to errors that arise from faulty usage of the type-system, which falls outside the scope of what is being done in this body of work. The list of predicates and and associated errors which inspired them can be seen below in Table 5.1.

#	**Error code(s)**	**Predicates**
1	E0381	A variable must be initialised before use.
2	E0384	A value can only be mutated if it is marked with `mut`.
3	E0503	A value can only be mutated by the *current* owner of the value.
4	E0499	A value can only be owned by exactly one variable at a time.
5	E0597 E0716	A value must be live for at least as long as it is borrowed.
6	E0596	A variable can only be borrowed mutably if it is `mut`
7	E0503	A value cannot be accessed while it is mutably borrowed, except through the `mut` reference.
8	E0502	A value cannot be mutated while it is immutably borrowed.
9	E0502	A reference of type `mut`/`imut` cannot be created if a reference of the variable of the other type exists.
10	E0505	A value cannot be moved while it is borrowed.
11	E0382	A value cannot be moved from the same variable twice.
12	E0507	A static variable cannot be moved.
13	E0508	Individual elements of an non-`Copy` collection cannot be moved.
14	E0500 E0501	A closure must have unique access to variables it uses, until the closure goes out of scope.
15	E0521	A closure acts like a lifetime declaration. As such, borrowed values inside a closure cannot be used outside the closure.

Table 5.1: Table of predicates written in natural language, along with the corresponding Rust error codes that inspired them. Describes what is not allowed when borrowing.

The astute reader will notice that a large portion of errors have been omitted, when cross-referencing the list of predicates in this section with the error list found in [16]. This is due to the fact that many errors, while tangentially related to borrow checking, or being handled by the type system, are deemed outside the scope of this project. They are included in Appendix A to show that we are aware of their existence, but have been

excluded as we choose not to handle them.

To show why the aforementioned predicates are important, and to illustrate what can happen if they are not upheld, we include some counter-examples. For illustration purposes we have included examples that break predicates 2, 4 and 6 respectively. A full list showing examples which break each predicate can be found in Appendix B at the reader's discretion.

Example 1

```
1 fn main(){
2     let mut x = 1;
3     let y = &mut x;
4     x += 1;
5     println!("{}", y);
6     println!("{}", x);
7 }
```

This example tries to mutate a variable `x` while it is not the current owner. On line 3 we temporarily transfer ownership from `x` to the variable `y` instead. This results in errors when we then try to mutate `x` on line 4, as the ownership at that point does not belong to `x`.

Example 2

```
1 fn main(){
2     let x = 1;
3     x += 1;
4     println!("{}", x);
5 }
```

This snippet will emit an error for line 3, as the line is attempting to mutate the immutable `x` variable. This snippet can be fixed by either changing line 3 to `let x = x + 1;` or by declaring `x` with `let mut`. The former would fully re-declare `x`, while the latter would allow `x` to be mutated.

Example 3

```
1 fn main(){
2     let a = 0;
3     let mut x = &mut a;
4     println!("{}", x);
5 }
```

This code snippet shows an attempt to mutably borrow an immutable variable. As a mutable borrow would imply that the variable being referred to is also mutable, this is impossible.

5.2 MIR Syntax

Seen in this section is our syntax over MIR, as well as functions we will use for the semantics, which is a modified and expanded version of the syntax proposed in "Static Taint Analysis in Rust"[1]. The syntax presented is functionally equivalent, but most identifiers have changed to increase readability. Furthermore, there have been additions, like the instrumentation tuple, which will be explained later in this section.

$$z \in \mathbb{Z}$$
$$i \in \text{Ident}$$

Where z is a literal integer value, and i is any string that matches the regular expression $[\texttt{a-zA-Z}]^+$.

$$bop \in \text{BinOps} = \{\text{Add}, \text{Sub}, \text{Mul}, \text{Div}, \text{Rem}, \text{BitXor}, \text{BitAnd}, \text{BitOr}, \text{Shl}, \text{Shr}, \text{Eq}, \text{Lt}, \text{Le}, \text{Ne}, \text{Ge}, \text{Gt}\}$$
$$uop \in \text{UnOps} = \{\text{Not}, \text{Neg}\}$$

$$\begin{aligned}
\mathit{Place} \in \text{Places} &::= \{_n \mid n \in \mathbb{N}_0\} \\
\mathit{Opr} \in \text{Operands} &::= \texttt{move}\ \ \mathit{Place} \mid \texttt{copy}\ \ \mathit{Place} \mid \texttt{const}\ z \\
\mathit{Rval} \in \text{Rvalues} &::= \mathit{Opr} \\
&\mid\ \ bop\texttt{(}\mathit{Opr}_1\texttt{,}\ \ \mathit{Opr}_2\texttt{)} \\
&\mid\ \ uop\texttt{(}\mathit{Opr}\texttt{)} \\
&\mid\ \ \texttt{&[mut]}\ \ \mathit{Place} \\
\mathit{Bid} \in \text{BlockID} &::= \{\texttt{bb}n \mid n \in \mathbb{N}_0\}
\end{aligned}$$

In the original semantics Places are denoted with *P*, Operands with *O*, Rvalues with *R*, and BlockID's with *Bid*. We found this notation to be confusing in some cases, so we have decided to increase the verbosity by simply using the natural *Place*, *Opr*, *Rval* and *Bid* for the respective singular selections in the sets. The sets Rval and Bid in the original syntax have also been expanded to more verbose names in ours.

$$\begin{aligned}
\mathit{Stmt} \in \text{Statement} &::= \mathit{Place}\ \texttt{=}\ \mathit{Rval}\texttt{;} \\
\mathit{Target} \in \text{Targets} &::= z\texttt{:}\ \mathit{Bid} \\
\mathit{Tmnt} \in \text{Terminators} &::= \texttt{goto -> }\mathit{Bid}\texttt{;} \\
&\mid\ \ \texttt{switchInt(}\mathit{Opr}\texttt{) -> [}[\mathit{Target},]^*\ \texttt{otherwise:}\ \ \mathit{Bid}\texttt{];} \\
&\mid\ \ \texttt{assert(}\mathit{Opr}\texttt{,}\ z\texttt{) -> }\mathit{Bid}\texttt{;} \\
&\mid\ \ \mathit{Place}\ \texttt{=}\ i\texttt{(}\mathit{Opr}_0,\ldots,\mathit{Opr}_n\texttt{) -> }\mathit{Bid}\texttt{;} \\
&\mid\ \ i\texttt{(}\mathit{Opr}_0,\ldots,\mathit{Opr}_n\texttt{);} \\
&\mid\ \ \texttt{return;}
\end{aligned}$$

For statements, targets, and terminators, we have replaced `S`, `X` and `T` with `Stmt`, `Target` and `Tmnt`. We have also changed the set identifiers for statements and terminators to the full-length version of the words they were previously shorter versions of.

$$
\begin{aligned}
\{Stmt^* Tmnt\} \in \text{Body} &::= \text{Statement}^* \times \text{Terminators} \\
B \in \text{Block} &::= Bid: \quad \{Stmt^* Tmnt\} \\
F \in \text{Fn} &::= \texttt{fn}\ i\texttt{(}Opr^*\texttt{)}\ \texttt{\{}B^+\texttt{\}} \\
M \in \text{Mir} &::= F^+
\end{aligned}
$$

$$
\begin{aligned}
Loc &\in \text{Locations} \\
Env \in \text{Environment}: &\quad \text{Places} \to \text{Locations} \\
Sto \in \text{Store}: &\quad \text{Locations} \to (\mathbb{Z}_\perp^\top + \text{Locations}) \\
prg \in \text{Program}: &\quad \text{Ident} \to \text{Function} \\
fun \in \text{Function}: &\quad \text{BlockID} \to \text{Body} \\
cs \in \text{CallStack} &::= (\text{BlockID} \times \text{Places} \times \text{Environment} \times \text{Function})^* \\
\text{Taint}: &\quad \text{Loc} \to \{untainted, tainted\} \\
\text{BorrowedBy}: &\quad \text{Loc} \to \{Place^*\} \\
\text{BorrowType}: &\quad \text{Place} \to \{not, imut, mut\} \\
\text{LocationType}: &\quad \text{Loc} \to \{imut, mut\} \\
\text{Operation}: &\quad \text{Loc} \to \{read, write, move\} \\
\text{Instrumentation} &::= (\text{Taint} \times \text{BorrowedBy} \times \text{BorrowType} \times \text{LocationType} \times \text{Operation}) \\
\text{Config}_S &::= \text{Statement} \times \text{Store} \\
\text{Config}_B &::= \text{Body} \times \text{CallStack} \times \text{Store} \times \text{Instrumentation} \\
\to_S &\subseteq \text{Config}_S \times \text{Store} \\
\to_B &\subseteq \text{Config}_B \times \text{Config}_B
\end{aligned}
$$

Finally, locations, environments, and store have gotten the new identifiers *Loc*, *Env*, *Sto* rather than ξ, e and ζ in the original semantics.

Additionally, we need the functions *Taint*, *BorrowedBy*, *BorrowType*, *LocationType*, and *Operation*, which we use to create instrumentation for the semantics. These functions will allow us to log information about the program as the semantics are applied, as well as allowing us to create boolean formulas for some of the predicates in Section 5.1. Two of these functions, *LocationType* and *BorrowType*, already existed in a similar form in the semantic rules presented by Gústafsson and Njor, and as such, we remove, rewrite, and repurpose them. *LocationType* keeps track of the mutability of a given location, while *BorrowType* keeps track of whether a Place is currently borrowing, and whether the borrow is mutable or immutable. However, these only enable a

few boolean formulas, which lead us to create, *BorrowedBy* and *Operation*, to allow us to formalise more predicates. *BorrowedBy* keeps track of the set of Places that are currently borrowing a location, while *Operation* keeps track of the operation that is being done to a location. Finally, we create the *Taint* function which will be used to monitor and log data about taint propagation. It is worth noting that the *Taint* function evaluates to lattice elements, such that $untainted \sqsubseteq tainted$, which is the same lattice as described in Section 3.3.

To gather the functions, we create the *Instrumentation* tuple which is comprised of the five aforementioned functions. This tuple serves as a program state log, which is used to examine how MIR behaves during execution.

$$Oeval(Opr, Sto, Env) = \begin{cases} z, & \text{if } Opr = \text{const } z \\ Sto(Env(Place)), & \text{if } Opr = \texttt{move } Place \\ Sto(Env(Place)), & \text{if } Opr = \texttt{copy } Place \end{cases}$$

$$Reval(Rval, Sto, Env) = \begin{cases} Oeval(Opr, Sto, Env), & \text{if } Rval = Opr \\ \texttt{bop}(Opr_1, Opr_2), & \text{if } Rval = bop(Opr_1, Opr_2) \\ \texttt{uop}(Opr), & \text{if } Rval = uop(Opr) \\ Sto(Env(Place)), & \text{if } Rval = \texttt{\&[mut]}\ Place \end{cases}$$

$$EvalOTaint(Opr, Taint) = \begin{cases} untainted, & \text{if } Opr = \text{const } z \\ Taint(Place), & \text{if } Opr = \texttt{move } Place \\ Taint(Place), & \text{if } Opr = \texttt{copy } Place \end{cases}$$

$$EvalRTaint(Rval, Taint) = \begin{cases} EvalOTaint(Opr, Taint), & \text{if } Rval = Opr \vee uop(Opr) \\ EvalOTaint(Opr_1, Taint) \sqcup & \\ EvalOTaint(Opr_2, Taint), & \text{if } Rval = bop(Opr_1, Opr_2) \\ Taint(Place), & \text{if } Rval = \&[mut]Place \end{cases}$$

Finally, the *EvalOTaint* and *EvalRTaint* are also additions to the semantics from the Gústafsson and Njor semantics, which serves as the main way to evaluate the taint propagation in the instrumented semantics. As such they serve as the instrumented versions of the original *Oeval* and *Reval*.

5.3 Predicates as Boolean Formulas

With the description of predicates, and the information gathering from the instrumentation, we can now create boolean formulas for some of the predicates. The amount of predicates we can create formulas for is reliant on the information gathered in the

instrumentation. This information is limited to what we deem most important with respects to the problem domain. As an example, we do not register information about statics, nor do we have a rule for creating MIR statics, and so we cannot create a boolean formula for predicate 12. This does not, however, mean that no formula for this predicate can exist, only that we choose not to describe one.

The predicates we create formulas for are predicates 2, 6 and 8 to 10. These can be seen below:

Predicate #	**Boolean formula**
2	$\forall Loc \in Sto : (Operation(Loc) = write \vee Operation(Loc) = move) \Rightarrow LocationType(Loc) = mut$
6	$\forall Place \in Env, \forall Loc \in Sto : (BorrowedBy(Loc) = Place \wedge BorrowType(Place) = mut) \Rightarrow LocationType(Loc) = mut$
8	$\forall Place \in Env, \forall Loc \in Sto : (Operation(Loc) = write \vee Operation(Loc) = move) \Rightarrow \neg(BorrowedBy(Loc) = Place \wedge BorrowType(Place) = imut)$
9	$\forall Place, Place' \in Env, \forall Loc \in Sto : BorrowedBy(Loc) = \{Place, Place'\} \Rightarrow \neg(BorrowType(Place) = imut \wedge BorrowType(Place') = mut)$
10	$\forall Loc \in Sto : Operation(Loc) = move \Rightarrow BorrowedBy(Loc) = none$

5.4 MIR semantics

This section goes through our semantic rules over MIR semantics which are inspired by the semantics presented in ”Static Taint Analysis in Rust”[1]. The semantic rules described in this section, while functionally identical to the work presented by Gústafsson and Njor, have undergone major changes in how they are written. These rules have undergone major naming changes to improve readability, and said changes will be explained as they are introduced. Many of the premises for each rule have been changed or outright removed due to our instrumentation now handling these instead. Finally, these semantics have been annotated to reflect the predicates presented in Section 5.1.

We want to make a point of noting that all rules should comply with all predicates, and that on a base level, all predicates are of equal import. However, not all predicates are strictly relevant for all rules. As an example, the [*Ass*] rule must adhere to predicate 9 which says that a value cannot be moved while it is borrowed. However, it is impossible for [*Ass*] to break this predicate due to it not having any `Move` premises. As such, we will annotate each rule in a vacuum, only highlighting the rules that are strictly relevant to the individual semantic rule.

Predicate 1 has been excluded from mention in the annotated semantics as it is universally applicable to all rules except [*Goto*]. As such, it should be assumed that it is present for every rule presented in the following section.

The semantic rules have been shaded, lowering visibility for parts that have no direct relevance for predicates, except for the added instrumentation. Furthermore, the remaining parts have received text highlighting borrowing predicates for the corresponding semantic premises. Finally, premises that include `LocationType` and `BorrowType` have been moved to instead be included in a set of instrumentation which, as part of the the full set of semantic rules which can be viewed in Appendix C.

$$[Ass]\ \frac{\begin{array}{c}\textbf{Predicates 2 to 4: } \langle \texttt{Place = Rval}, Sto\rangle \rightarrow_S Sto[Env(Place) \mapsto Reval(Rval)] \\ Sto(Env(Place)) \neq \top \\ \textbf{Predicates 5 and 6: } \begin{array}{c}(Rval = (Opr \vee uop(Opr)) \wedge Opr \neq \texttt{move } Place) \vee (Rval = bop(Opr_1, Opr_2) \\ \wedge Opr_1 \neq \texttt{move } Place' \wedge Opr_2 \neq \texttt{move } Place'')\end{array}\end{array}}{\begin{array}{c} prg \vdash \langle \{\texttt{Place = Rval; Stmt}^*\texttt{Tmnt}\}, (Bid, Place, Env, fun) :: cs, Sto, \\ (Taint, BorrowedBy, BorrowType, LocationType, Operation)\rangle \\ \rightarrow_B \langle \{\texttt{Stmt}^*\texttt{Tmnt}\}, (Bid, Place, Env, fun) :: cs, Sto[Env(Place) \mapsto Reval(Rval)], \\ (Taint[Env(Place) \rightarrow EvalRTaint(Rval, Taint)], BorrowedBy, \\ BorrowType, LocationType, Operation[Env(Place) \rightarrow write])\rangle \end{array}}$$

The [*Ass*] rule assigns a value to a place, where the value is a constant, or the result of a unary or binary operation, and where operands use no move operations.

This rule must uphold predicates 2 to 6 to function as intended. It needs to uphold predicate 2 to ensure that the location we are assigning to is mutable. Furthermore, it needs predicates 3 and 4, as the value in $Sto(Env(Place))$ is being changed. This means that *Place* has to be the unique owner during assignment. Lastly, [*Ass*] must uphold predicates 5 and 6 as we assign an *Rval* which comes from an *Reval*. This *Reval* might include a reference or a mutable borrow, and as such we must ensure that the value is live in case of a borrow, in addition to ascertaining whether the variable is declared mutable in case of a mutable borrow.

$$[Ass_{Move1}]\ \frac{\begin{array}{c}\textbf{Predicates 2 to 4: } \langle \texttt{Place = Rval}, Sto\rangle \rightarrow_S Sto[Env(Place) \mapsto Reval(Rval)] \\ Sto(Env(Place)) \neq \top \\ \textbf{Predicates 5 and 6: } Rval = (Opr \vee uop(Opr)) \\ \textbf{Predicates 10 to 13: } Opr = \texttt{move } Place'\end{array}}{\begin{array}{c} prg \vdash \langle \{\texttt{Place = Rval; Stmt}^*\texttt{Tmnt}\}, (Bid, Place, Env, fun) :: cs, Sto \\ (Taint, BorrowedBy, BorrowType, LocationType, Operation)\rangle \\ \rightarrow_B \langle \{\texttt{Stmt}^*\texttt{Tmnt}\}, (Bid, Place, Env, fun) :: cs, Sto[Env(Place) \mapsto Reval(Rval), \\ Env(Place') \mapsto \top], \\ (Taint[Env(Place) \rightarrow EvalRTaint(Rval, Taint)], BorrowedBy, BorrowType, \\ LocationType, Operation[Env(Place) \rightarrow write, Env(Place') \rightarrow move])\rangle \end{array}}$$

$$[Ass_{Move2}]\ \frac{\begin{array}{c}\textbf{Predicates 2 to 4: } \langle \texttt{Place = Rval}, Sto\rangle \rightarrow_S Sto[Env(Place) \mapsto Reval(Rval)] \\ Sto(Env(P)) \neq \top \\ \textbf{Predicates 5 and 6: } Rval = (bop(Opr_1, Opr_2)) \\ \textbf{Predicates 10 to 13: } Opr_1 = \texttt{move } Place' \\ Opr_2 \neq \texttt{move } Place''\end{array}}{\begin{array}{c} prg \vdash \langle \{\texttt{Place = Rval; Stmt}^*\texttt{Tmnt}\}, (Bid, Place, Env, fun) :: cs, Sto \\ (Taint, BorrowedBy, BorrowType, LocationType, Operation)\rangle \\ \rightarrow_B \langle \{\texttt{Stmt}^*\texttt{Tmnt}\}, (Bid, Place, Env, fun) :: cs, Sto[Env(Place) \mapsto Reval(Rval), \\ Env(Place') \mapsto \top], \\ (Taint[Env(Place) \rightarrow EvalRTaint(Rval, Taint)], BorrowedBy, BorrowType, \\ LocationType, Operation[Env(Place) \rightarrow write, Env(Place') \rightarrow move])\rangle\end{array}}$$

$$[Ass_{Move3}]\ \frac{\begin{array}{c}\textbf{Predicates 2 to 4: } \langle \texttt{Place = Rval}, Sto\rangle \rightarrow_S Sto[Env(Place) \mapsto Reval(Rval)] \\ Sto(Env(P)) \neq \top \\ \textbf{Predicates 5 and 6: } Rval = (bop(Opr_1, Opr_2)) \\ Opr_1 \neq \texttt{move } Place' \\ \textbf{Predicates 10 to 13: } Opr_2 = \texttt{move } Place''\end{array}}{\begin{array}{c} prg \vdash \langle \{\texttt{Place = Rval; Stmt}^*\texttt{Tmnt}\}, (Bid, Place, Env, fun) :: cs, Sto, \\ (Taint, BorrowedBy, BorrowType, LocationType, Operation)\rangle \\ \rightarrow_B \langle \{\texttt{Stmt}^*\texttt{Tmnt}\}, (Bid, Place, Env, fun) :: cs, Sto[Env(Place) \mapsto Reval(Rval), \\ Env(Place'') \mapsto \top], \\ (Taint[Env(Place) \rightarrow EvalRTaint(Rval, Taint)], BorrowedBy, BorrowType, \\ LocationType, Operation[Env(Place) \rightarrow write, Env(Place') \rightarrow move])\rangle\end{array}}$$

$$[Ass_{Move4}]\ \frac{\begin{array}{c}\textbf{Predicates 2 to 4: } \langle \texttt{Place = Rval}, Sto\rangle \rightarrow_S Sto[Env(Place) \mapsto Reval(Rval)] \\ Sto(Env(Place)) \neq \top \\ \textbf{Predicates 5 and 6: } Rval = (bop(Opr_1, Opr_2)) \\ \textbf{Predicates 10 to 13: } Opr_1 = \texttt{move } Place' \\ Opr_2 = \texttt{move } Place''\end{array}}{\begin{array}{c} prg \vdash \langle \{\texttt{Place = Rval; Stmt}^*\texttt{Tmnt}\}, (Bid, Place, Env, fun) :: cs, Sto, \\ (Taint, BorrowedBy, BorrowType, LocationType, Operation)\rangle \\ \rightarrow_B \langle \{\texttt{Stmt}^*\texttt{Tmnt}\}, (Bid, Place, Env, fun) :: cs, Sto[Env(Place) \mapsto Reval(Rval), \\ Env(Place') \mapsto \top, Env(Place'') \mapsto \top], \\ (Taint[Env(Place) \rightarrow EvalRTaint(Rval, Taint)], \\ BorrowedBy, BorrowType, LocationType, \\ Operation[Env(Place) \rightarrow write, Env(Place') \rightarrow move, Env(Place'') \rightarrow move])\rangle\end{array}}$$

The [Ass_{Move}] rules all make an assignment of a value to a place where an operand makes use of a `move` operation. The reason for having four rules is to specify all possible combinations of `move` and non-`move` operands. [Ass_{Move1}] is for cases where there is only one operand alone or via a unary operation, and that operand uses a `move`

operation. [Ass_{Move2}] is for a binary operation where two operands are present and the first operand uses a `move`. [Ass_{Move3}] is the same scenario as [Ass_{Move2}] but instead of the first operand, it is the second operand that uses a `move` operation. The final [Ass_{Move4}] is where the both operands uses a `move`.

The inherent similarity in all [Ass_{Move}] rules, mean that they all must uphold the same predicates and so will be explained simultaneously. As such, all [Ass_{Move}] rules must account for predicates 2 to 6 and 10 to 13. Predicates 2 to 6 are included for the reasons as in the [*Assign*] rule. All [Ass_{Move}] rules relate to using `move` operations, and as such it is especially important that we take extra care to comply with predicates 10 to 13 which pertain to `move` operations. In particular, we must ensure that a value cannot be moved while borrowed, that the value cannot be moved from the same variable twice, or that moves are not made from inside a reference. Furthermore, we must also ensure that we are not moving static variables or elements from non-`Copy` collections.

$$[Ass_{Ref:mut}]\ \frac{\begin{array}{c}\textbf{Predicates 2 to 4: } \langle \texttt{Place = Rval}, Sto\rangle \rightarrow_S Sto[Env(Place) \mapsto Env(Place')] \\ Sto(Env(Place)) \neq \top \\ \textbf{Predicates 5, 6 and 9: } Rval = \&mut\ Place'\end{array}}{\begin{array}{c} prg \vdash \langle \{\texttt{Place = Rval; Stmt}^*\texttt{Tmnt}\}, (Bid, Place, Env, fun) :: cs, Sto, \\ (Taint, BorrowedBy, BorrowType, LocationType, Operation)\rangle \\ \rightarrow_B \langle \{\texttt{Stmt}^*\texttt{Tmnt}\}, (Bid, Place, Env, fun) :: cs, Sto[Env(Place) \mapsto Env(Place')], \\ (Taint[Env(Place) \rightarrow EvalRTaint(Rval, Taint)], \\ BorrowedBy[Env(Place') \rightarrow Place], BorrowType[Place \rightarrow mut], \\ LocationType, Operation[Env(Place) \rightarrow write, Env(Place') \rightarrow move])\rangle \end{array}}$$

The [$Ass_{Ref:mut}$] rule assigns a mutable reference to a place. Predicates 2, 4 to 7 and 9 must hold for this rule to work. Predicate 2 must hold as the place we are assigning to must be mutable. Predicate 4 must hold as we must not be allowed to create a mutable reference, if the place we are referencing is already mutably borrowed by something else. Predicate 5 must hold because we are creating a borrow, and as such we must ensure that the value we borrow persists until we are done with it. Predicate 6 must hold as we cannot mutably reference a value if the value is not declared mutable. Finally, predicates 7 and 9 must hold to ensure that we only assign the value in the reference via the reference, while also ensuring that there are no other references before or during use of the reference we are using.

$$[Ass_{Ref:imut}]\ \frac{\begin{array}{c}\textbf{Predicates 2 to 4: } \langle \texttt{Place = Rval}, Sto\rangle \rightarrow_S Sto[Env(Place) \mapsto Env(Place')] \\ Sto(Env(Place)) \neq \top \\ \textbf{Predicates 5, 6 and 9: } Rval = \&Place'\end{array}}{\begin{array}{c} prg \vdash \langle \{\texttt{Place = Rval; Stmt}^*\texttt{Tmnt}\}, (Bid, Place, Env, fun) :: cs, Sto, \\ (Taint, BorrowedBy, BorrowType, LocationType, Operation)\rangle \\ \rightarrow_B \langle \{\texttt{Stmt}^*\texttt{Tmnt}\}, (Bid, Place, Env, fun) :: cs, Sto[Env(Place) \mapsto Env(Place')], \\ (Taint[Env(Place) \rightarrow EvalRTaint(Rval, Taint)], \\ BorrowedBy[Env(Place') \rightarrow Place], BorrowType[Place \rightarrow imut], \\ LocationType, Operation[Env(Place) \rightarrow write, Env(Place') \rightarrow read])\rangle\end{array}}$$

The $[Ass_{Ref:imut}]$ rule assigns an immutable reference to a place. This rule relies on predicates 2 to 5, 8 and 9. Predicate 2 must hold to ensure that we are allowed to assign the immutable reference, and predicates 3 and 4 must be upheld to ensure that the value is mutable and being modified by the current owner exclusively. Like with $[Ass_{Ref:mut}]$, predicate 5 must hold as we need to ensure that whatever value we are reading from survives until our borrow is finished. Predicate 8 must hold to ensure that the contents of the immutable reference are not mutated while we use it. Finally, predicate 9 must hold to ensure that there are no mutable references being created, before or during use of the reference we are using.

$$[Goto]\ \frac{}{\begin{array}{c} prg \vdash \langle \{\texttt{goto -> } Bid\texttt{;}\}, (Bid', Place', env', fun') :: cs, Sto, \\ (Taint, BorrowedBy, BorrowType, LocationType, Operation)\rangle \\ \rightarrow_B \langle fun'(Bid), (Bid', Place', Env', fun') :: cs, Sto, \\ (Taint, BorrowedBy, BorrowType, LocationType, Operation)\rangle\end{array}}$$

The [*Goto*] rule simply sets the next basic block to be executed. As such there are no relevant predicates to apply here, nor any changes in the instrumentation.

$$[SwitchInt_1]\ \frac{\begin{array}{c}\textbf{Predicates 7 and 10 to 13: } z_i = Oeval(Opr, Sto, Env'') \\ i \in \{1, ..., n\}\end{array}}{\begin{array}{c} prg \vdash \langle \{\texttt{SwitchInt(}Opr\texttt{) -> [}z_1 : Bid_1, ..., z_n : Bid_n\texttt{, otherwise: } Bid'\texttt{];}\}, \\ (Bid'', Place'', Env'', fun'') :: cs, Sto, \\ (Taint, BorrowedBy, BorrowType, LocationType, Operation)\rangle \\ \rightarrow_B \langle fun''(Bid_i), (Bid'', Place'', Env'', fun'') :: cs, Sto, \\ (Taint, BorrowedBy, BorrowType, LocationType, Operation)\rangle\end{array}}$$

$$[SwitchInt_2]\ \frac{\textbf{Predicates 7 and 10 to 13: } \forall z \in \{z_1,...,z_n\} : z \neq Oeval(Opr,Sto,Env'')}{\begin{array}{c} prg \vdash \langle \{\texttt{SwitchInt(}Opr\texttt{) -> [}z_1 : Bid_1,...,z_n : Bid_n\texttt{, otherwise: }Bid'\texttt{];}\}, \\ (Bid'',Place'',Env'',fun'') :: cs, Sto, \\ (Taint,BorrowedBy,BorrowType,LocationType,Operation)\rangle \\ \rightarrow_B \langle fun''(Bid'),(Bid'',Place'',Env'',fun'') :: cs, Sto, \\ (Taint,BorrowedBy,BorrowType,LocationType,Operation)\rangle \end{array}}$$

$$[Assert]\ \frac{\textbf{Predicates 7 and 10 to 13: } z = Oeval(Opr,Sto,Env')}{\begin{array}{c} prg \vdash \langle \{\texttt{Assert(}Opr\texttt{, }z\texttt{) -> }Bid\texttt{;}\},(Bid',Place',Env',fun') :: cs, Sto \\ (Taint,BorrowedBy,BorrowType,LocationType,Operation)\rangle \\ \rightarrow_B \langle fun'(Bid),(Bid',Place',env',fun') :: cs, Sto, \\ (Taint,BorrowedBy,BorrowType,LocationType,Operation)\rangle \end{array}}$$

The [*SwitchInt*$_1$] and [*SwitchInt*$_2$] rules pertain to `Switch` cases wherein we iterate through a set of cases in an attempt to find a match. The [*SwitchInt*$_1$] rule chooses an execution path between n choices, if one of the n expressions are equal to a given constant. [*SwitchInt*$_2$] on the other hand chooses an execution path if none of n expressions are equal to a given constant - also known as the 'default' case in programming languages such as C, C++, or Java.

Furthermore, the [*Assert*] rule - which is a rule that asserts that some variable is equal to some constant value - requires the same predicates as the [*SwitchInt*] rules, these being predicates 7 and 10 to 13.

Predicate 7 must hold because the operand *Opr* we are using can be a possibly mutable reference. This is also the reason why predicates 10 to 13 must hold, as the operand can be a `move` operation.

$$[Call_{Normal}]\ \frac{\begin{array}{c} Loc_i \texttt{ fresh} \\ Sto(Env(Place)) \neq \top \\ Env' = [\_j \mapsto Loc_j,...] \\ Sto' = Sto[Loc_j \mapsto Oeval(Opr_j,Sto,Env'), \\ Env'(Place_j) \mapsto \begin{cases} \top, & \textbf{Predicates 10 to 13: } \text{if } Opr_j = \texttt{move } Place_j \\ Sto(Env'(Place_j)), & \text{otherwise} \end{cases}, ...], \\ j \in \{0,...,n\} \end{array}}{\begin{array}{c} prg \vdash \langle \{\texttt{Place = }i\texttt{(}Opr_0,...,Opr_n\texttt{) -> }Bid\texttt{;}\},(Bid',Place',Env',fun') :: cs, Sto, \\ (Taint,BorrowedBy,BorrowType,LocationType,Operation)\rangle \\ \rightarrow_B \langle prg(i)(\texttt{bb0}),(Bid,Place,Env,prg(i)) :: (Bid',Place',Env',fun') :: cs, Sto', \\ (Taint[Env'(Place_j) \rightarrow Taint(Env(Place_j))],BorrowedBy,BorrowType,LocationType, \\ Operation[Env'(Place_j) \rightarrow write, Env(Place_j) \rightarrow move \text{ if } Opr_j = \texttt{move } Place_j])\rangle \end{array}}$$

$$[Call_{Panic}]\ \frac{\begin{array}{c} Loc_i \ \texttt{fresh} \\ Env' = [\_j \mapsto Loc_j, ...] \\ Sto' = Sto[Loc_j \mapsto Oeval(Opr_j, Sto, Env'), \\ Env'(Place_j) \mapsto \begin{cases} \top, & \textbf{Predicates 10 to 13: if } Opr_j = \texttt{move } Place_j \\ Sto(Env'(Place_j)), & \text{otherwise} \end{cases}, ...], \\ j \in \{0, ..., n\} \end{array}}{\begin{array}{c} prg \vdash \langle \{i(Opr_o, ..., O_n);\}, (Bid', Place', Env', fun') :: cs, Sto, \\ (Taint, BorrowedBy, BorrowType, LocationType, Operation) \rangle \\ \rightarrow_B \langle prg(i)(\texttt{bb0}), (\_, \_, Env, prg(i)) :: (Bid', Place', Env', fun') :: cs, Sto', \\ (Taint[Env'(Place_j) \rightarrow Taint(Place_j)], BorrowedBy, BorrowType, LocationType, \\ Operation[Env'(Place_j) \rightarrow write, Env(Place_j) \rightarrow move \text{ if } Opr_j = \texttt{move } Place_j]) \rangle \end{array}}$$

The [$Call_{Normal}$] rule calls a normal function, and assigns the functions return value to a place. The [$Call_{Panic}$] rule is functionally similar, but instead calls a function when the program fails, ensuring that the program shuts down gracefully. As such, it is imperative that predicates 10 to 13 must hold for them to function.

Predicates 10 to 13 relate to the storage update line $\top$,if $Opr_j = \texttt{move } Place_j$, and must hold to ensure that the `move` operation is done correctly.

$$[Return]\ \frac{\textbf{Predicates 2 to 4: } Sto' = Sto[Env'(Place) \mapsto Env(\_0)]}{\begin{array}{c} prg \vdash \langle \{\texttt{Return};\}, (Bid, Place, Env, fun) :: (Bid', Place', Env', fun') :: cs, Sto, \\ (Taint, BorrowedBy, BorrowType, LocationType, Operation) \rangle \\ \rightarrow_B \langle fun'(Bid), (Bid', Place', Env', fun') :: cs, Sto', \\ (Taint[Place \rightarrow Taint(\_0)], BorrowedBy, BorrowType, \\ LocationType, Operation[Env(\_0) \rightarrow write, Env'(Place) \rightarrow move]) \rangle \end{array}}$$

The [*Return*] rule returns a value from a given basic block. Predicates 2 to 4 must hold for this rule to function. This is because of the fact that while [*Call*] rules technically contain assignments, they only declare that an assignment should happen, whereas the actual assignment happens via the [*Return*] rule. This means that the place we are returning to must be mutable, and that we must hold ownership of the value we assign to.

5.5 Instrumented Semantics

To show how the instrumentation can be used to verify whether the boolean forms of the predicates hold, we construct a small example program. We knowingly create a program with an error, and then apply our semantics to it.

```
1 fn main() {
2     let mut test = 8;
```

```
3       let mut_borrow = &mut tst;
4       let imut_borrow = &tst;
5   }
```

Listing 5.1: Instrumented Semantics example 1

Listing 5.1 shows the faulty program. We create a value, and then try to borrow it with conflicting borrow types, which causes an error at compile time, as the conflicting borrows break predicate 9. However to further show this, we reduce Listing 5.1 to MIR and apply the instrumented semantic rules to show a more detailed overview of the program state during execution. This can be seen in Listing 5.2 below.

```
1   bb0: {
2           _1 = const 8_i32;
3           _2 = &mut _1;
4           _3 = &_1;
5           return;
6       }
```

Listing 5.2: MIR output of Listing 5.1

Following the semantic rules, the instrumentation tuple will contain the program state information during execution. The changes in this information can be seen in the table below:

Line	Instrumentation	Rule	PS
Start	{}		OK
2	{Taint: Env(_1) → {untainted}, Operation: Env(_1) → {write}}	Ass	OK
3	{Taint: Env(_1) → {untainted}, Env(_2) → {untainted}, BorrowedBy: Env(_1) → _2, BorrowType: _2 → {mut}, Operation: Env(_1) → {read}, Env(_2) → {write}}	AssRefMut	OK
3	{Taint: Env(_1) → {untainted}, Env(_2) → {untainted}, Env(_3) → {untainted}, BorrowedBy: Env(_1) → _2, Env(_1) → _3, BorrowType: _2 → {mut}, _3 → {imut}, Operation: Env(_1) → {read}, Env(_2) → {move}, Env(_3) → {write}}	AssRefImut	Error

Table 5.2: A program state for the code shown in Listing 5.2.

In Table 5.2 we can follow the execution of the MIR code shown in Listing 5.2, and see the application of the instrumented assignment rules. Line refers to the original line of code in Listing 5.2, Instrumentation shows the instrumentation being applied, Rule denotes which rule is being applied, and PS refers to the current program state. During the execution on line 3 of the program, it can be seen that both _2 and _3 borrows the same place _1 but with conflicting types, which breaks predicate 9.

This instrumentation will be used to create a more logical formulation of the predicates, as well as to identify whether the analysis made by Gústafsson and Njor is correct.

Chapter 6

Taint Analysis

This chapter describes how we create a simple taint analysis, to show how the instrumentation and predicates depicted in Chapter 5 can be used to create more precise approximations. Furthermore, this chapter also goes through how the predicates and instrumentation could help make our implementation of the taint analysis more precise than existing ones, like what is depicted in Gústafsson and Njor work.

6.1 Static Taint Analysis

We establish a taint analysis, which will largely be based on the instrumented semantics presented in Chapter 5. Additionally, the analysis itself draws inspiration from the monotone framework created by Gústafsson and Njor in ”Static Analysis in Rust”[1]. We use the same lattice as the *Taint* element in the instrumentation, as seen Section 5.5. This is done partly to prepare for potential expansion, like what was described in Section 3.3, but also to make direct comparisons to the instrumentation easier.

Finally, we want to make a note of saying that this analysis assumes that Rust's borrow checker has already been run on the program being analysed.

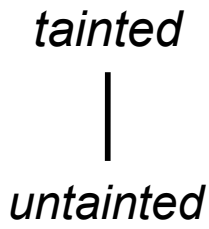

Figure 6.1: The taint lattice used for the static taint analysis

Figure 6.1 shows the lattice *Taint*, which we will use as the target for mapping Places to. Like Gústafsson and Njor, we will prepend function names to Places, to differentiate between Places with the same name. For example, the function `foo` will return whatever is in Place `_0` to the caller at the end of the function. This Place would thus be renamed `foo:_0`. These Places are gathered in the set $Places_{Analysis}$. We create the lattice $State$ to map $Places_{Analysis}$ to $Taint$.

$$State = Places_{Analysis} \rightarrow Taint$$

This lattice records the state of a single CFG node from MIR. We create a further mapping from CFG labels, in the set $Labels$, to each $State$.

$$States = Labels \rightarrow State$$

Finally, we extend the $States$ to use the functional approach for context sensitivity.

$$States = Labels \rightarrow State_{\perp} \rightarrow State_{\perp}$$

In this, The $Labels$ can be thought of as function calls, with the first $State_{\perp}$ being the state of the inputs and the second being the state of the output after the function has completed. These lattices simply have a unique bottom element appended, but are otherwise identical to the $State$ lattice. Thus, the same state of inputs to a function will give the same state of outputs.

To evaluate the taint of a value, we use the same functions $EvalOTaint$ and $EvalRTaint$ as in the semantics, but use $States$ instead of $Taint$ as the parameter. The functions are thus as follows:

$$EvalOTaint(Opr, States) = \begin{cases} untainted, & \text{if } Opr = \text{const } z \\ States(Place), & \text{if } Opr = \texttt{move } Place \\ States(Place), & \text{if } Opr = \texttt{copy } Place \end{cases}$$

$$EvalRTaint(Rval, States) = \begin{cases} EvalOTaint(Opr, States), & \text{if } Rval = Opr \vee uop(Opr) \\ EvalOTaint(Opr_1, States) \sqcup & \\ EvalOTaint(Opr_2, States), & \text{if } Rval = bop(Opr_1, Opr_2) \\ States(Place), & \text{if } Rval = \&[mut]Place \end{cases}$$

With the *States* lattice and our evaluation functions, we are now equipped to make the transfer functions for the taint analysis.

We begin by making transfer functions for the assignment rules. All assignments are of the form `Place = Rval`. For these, we create the function $transfer_{assignment}$.

$$transfer_{Assignment}(State) = State[Place \mapsto EvalRTaint(Rval, State)]$$

Because of predicates 3 and 4, we can perform a strong update on *State*. A strong update in static analysis, is an update where an abstract value completely overwrites an existing one [11, p. 130]. Because aliasing is disallowed, all assignments to a Place can only be made from a single source at a time, and we can thus always be sure that an assignment will cause a Place to be either tainted or untainted, and can update the value accordingly.

As Gústafsson and Njor remarks, conditionals could be considered taint sinks, but they consider it out of scope to model this. We will however include this in our analysis, as we believe that using tainted values in control-flow decisions should be considered erroneous behavior. As, $[SwitchInt_1]$ and $[SwitchInt_2]$ will not make any changes in the state, the transfer function can simply be the identity function for both constructs.

$$transfer_{SwitchInt}(State) = State$$

However, if a tainted value is used in the *Opr* of the condition, regardless of whether the given case holds or not, the analysis should report this as a flaw.

$[Goto]$ and $[Assert]$ do not make any changes in the state either. These will both use the identity transformation.

$$transfer_{Goto}(State) = State$$
$$transfer_{Assert}(State) = State$$

This leaves only the $[Call_Normal]$, $[Call_Panic]$, and $[Return]$ rules. We identify the `input`, `sanitise`, and `output` functions as special, as these will introduce taint to the system, remove taint from the system, and output something from the system respectively.

We assume the `input` and `sanitise` functions only take a single *Place* as parameter at a time, such that multiple inputs or sanitations result from multiple function calls. The `input` function will take an input and assign it to the *Place* given. The `sanitise` function will take the given *Place* and sanitise it in some way, to make it untainted. As such, the transfer functions are quite simple.

$$transfer_{Call:input}(State) = State[Place \mapsto tainted]$$
$$transfer_{Call:sanitise}(State) = State[Place \mapsto untainted]$$

The `output` function does not change anything in the program, and thus will also use the identity function. However, if this function is given a tainted *Place* as a parameter, the analysis should report an error.

$$transfer_{Call:output}(State) = State$$

Finally, the $[Return]$ rule does an assignment from the callee's *Place* `_0`, to the caller's provided *Place*, and thus simply transfers *Place* `_0`'s taintedness directly. For a given caller function *caller* that calls a given function *callee*, we have the following transition function:

$$transfer_{Call:callee:return}(State) = State[caller:Place \mapsto callee:\_0]$$

Another benefit of using Rust's borrow system, is that there are no need to perform any kind of alias analysis. As a value can only be owned by a single variable at a time, there cannot exist two variables that can make changes to the same variable at the same time.

6.1.1 Justification for Strong Updates

In languages where aliasing is allowed, it is possible for variables to be changed indirectly. This makes it harder to approximate the behavior of the program, meaning that an analysis must grow in complexity if aliasing should be accounted for. For Rust we can instead always perform strong updates on the abstract value of a variable for any given state. This is due to the illegality of aliasing in Rust.

```c
#include <stdio.h>
int input();
void change_from_other_thread(int* var);

int main() {
    int x = input();
    int* y = &x;
    change_from_other_thread(&x);
    printf("%d", *y);
    return 0;
}

int input() {
    return 3;
}

void change_from_other_thread(int* var) {
    if (*var == 0) {
        *var += 5;
    }
}
```

Listing 6.1: Snippet of C code with a pretend-thread that would change a variable.

As an example, Listing 6.1 shows a code snippet which would compile in C. The value of `x` could be changed from a different thread in a multi-threaded environment, which would be valid, as aliasing is allowed. Most importantly, in more complicated programs, this could happen in much more indirect ways, and at varying times, depending on the specific timings of thread executions. This would make an error due to aliasing much harder to track.

```rust
fn main() {
    let mut x = input();
    let y = &mut x;
    change_from_other_thread(&mut x);
    println!("{}", y)
}

fn input() -> i32 {
    3
}

fn change_from_other_thread(var_in: &mut i32) {
    if *var_in == 0 {
        *var_in += 5;
    }
}
```

Listing 6.2: Snippet of Rust code with a pretend-thread that would change a variable. This snippet will not compile.

Listing 6.2 shows an equivalent code snippet in Rust. This snippet is functionally the same as the one shown in Listing 6.1, with the only difference being that it will not compile in Rust. This is because of the fact that aliasing is disallowed, meaning we cannot create a new reference on line 4, as `x` is already being borrowed mutably by `y`.

Were we to create a comprehensive analysis for C, we would have to account for aliasing in assignments. This would increase the complexity of the analysis, if we wish to keep the same accuracy. In Rust, on the other hand, we could rely on the type system to find these aliasing situations, and having them reported as errors. We believe that, more often than not, any analysis that analyses pointers and/or heap memory may benefit from the borrowing system, as is the case with, for example, points-to analysis, escape analysis, and shape analysis.

This means that, for our analysis, we can discern possible execution paths wherein a value that is marked as tainted is used in a sink. The intuition behind this, is that whenever two edges in the program CFG go to the same node, we perform a least upper bound operation on the state from each incoming node. As such, given the taint lattice seen in Figure 3.2, there are only two possibilities of the outcome: A value is tainted, or a value is untainted. This can be thought of as a logical AND operation, where untainted would be equivalent to logical true.

With this in mind, we can generally be sure that any program that does not loop infinitely will have the most accurate taint information retained until the program's exit points. For any given exit point of the program, there will be either 0 or any number of branches and merges in the CFG. Branches have no immediate impact on information accuracy, while merges only have the limited effect discussed before. Because of this, we can be sure that any use of taint in a sink will at the very least have a possible execution path where the given value is tainted.

6.1.2 Comparison of Static Taint Analysis and Instrumented Semantics

To test the theoretical basis of our analysis, we aim to verify whether the predictions pertaining to taint in the static analysis are accurate to the actual reports of the instrumented semantics. We will compare the results of the static taint analysis we have designed, with the results of using the instrumented semantics on the possible execution paths of a program.

To begin, we will use the code example established by Gústafsson and Njor in their body of work as a baseline for comparison.

```
1  fn main() {
2      bb1{
3          _1 = input() -> bb2; //l1
4      }
5      bb2{
6          _2 = Sub(copy _1, const 3); //l2
7          _3 = Gt(const 0, copy _2); //l3
8          switchInt(copy _3) -> [0: bb3, otherwise: bb4]; //l4
9      }
10     bb3{
11         _4 = const 2; //l5
12         goto -> bb5; //l6
13     }
14     bb4{
15         _4 = input() -> bb5; //l7
16     }
17     bb5{
18         _5 = output(_4) -> bb6; //l8
19     }
20     bb6{
21         return; //l9
22     }
23 }
```

Listing 6.3: An example of MIR code, with comments at the end of lines acting as labels. Taken from "Static Analysis in Rust"[1].

Listing 6.3 shows a simple code snippet written in MIR syntax. The example shows that we accept some input, as seen in `bb1` on line 3. This input is then used in `bb2` in a `SwitchInt` statement on line 7. Depending on the conditional behavior, we assign `_4` to either an input or a constant value. A CFG of the program can be see in Figure 6.2 below.

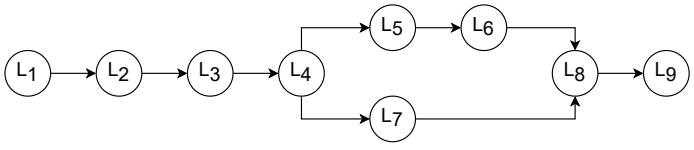

Figure 6.2: CFG representation of the program in Listing 6.3

If we apply our static taint analysis to the example, we create the steps shown in Table 6.1 below.

Label	Transfer function	Input	Output
l_1	$transfer_{Call:input}$	$\forall Place \in Places : Place \mapsto untainted$	$States(l_1)[\_1 \mapsto tainted]$
l_2	$transfer_{Assignment}$	$output(l_1)$	$States(l_2)[\_2 \mapsto tainted]$
l_3	$transfer_{Assignment}$	$output(l_2)$	$States(l_3)[\_3 \mapsto tainted]$
l_4	$transfer_{SwitchInt}$	$output(l_3)$	$States(l_4)$
l_5	$transfer_{Assignment}$	$output(l_4)$	$States(l_5)$
l_6	$transfer_{Goto}$	$output(l_5)$	$States(l_6)$
l_7	$transfer_{Call:input}$	$output(l_4)$	$States(l_7)[\_4 \mapsto tainted]$
l_8	$transfer_{Call:output}$	$output(l_6) \sqcup output(l_7)$	$States(l_8)$
l_9	$transfer_{Call:main:return}$	$output(l_8)$	$States(l_9)$

Table 6.1: The steps taken in the static taint analysis, as well as the inputs and outputs for each given line.

Seen in Table 6.1, we have a log consisting of snapshots of the input and output of each line in Listing 6.3. Of note, are the labels l_1, l_2, and l_3 which results in `_1`, `_2`, and `_3` being tainted. We can see in l_7 that the `SwitchInt` results in `_4` becoming tainted as well.

Furthermore, we would report an error on account of l_4 and l_8. This is due to the fact that we consider it unintended behavior for taint to have an influence on control flow, and that the use of a possibly tainted value in the `output` function should give an error.

Seen in Table 6.2 and Table 6.3 are logs of which rules are applied for the two possible execution paths for the program. Table 6.2 show the execution path wherein the `SwitchInt` is evaluated to the `0` case. Table 6.3 shows the opposite, being the `otherwise` case of the `SwitchInt`.

The headers for these particular tables are self-explanatory with the exception of the 'Taint' column. Taint refers to the *Taint* function of the instrumentation tuple. The contents show what values the input evaluates to for each rule applied.

Label	Rule applied	Taint
l_1	$[Call_{normal}]$	$\{Env(\_1) \rightarrow tainted\}$
l_2	$[Ass]$	$\{Env(\_1) \rightarrow tainted, Env(\_2) \rightarrow tainted\}$
l_3	$[Ass]$	$\{Env(\_1) \rightarrow tainted, Env(\_2) \rightarrow tainted, Env(\_3) \rightarrow tainted\}$
l_4	$[SwitchInt_1]$	$\{Env(\_1) \rightarrow tainted, Env(\_2) \rightarrow tainted, Env(\_3) \rightarrow tainted\}$
l_5	$[Ass]$	$\{Env(\_1) \rightarrow tainted, Env(\_2) \rightarrow tainted, Env(\_3) \rightarrow tainted, Env(\_4) \rightarrow untainted\}$
l_6	$[Goto]$	$\{Env(\_1) \rightarrow tainted, Env(\_2) \rightarrow tainted, Env(\_3) \rightarrow tainted, Env(\_4) \rightarrow untainted\}$
l_8	$[Call_{normal}]$	$\{Env(\_1) \rightarrow tainted, Env(\_2) \rightarrow tainted, Env(\_3) \rightarrow tainted, Env(\_4) \rightarrow untainted\}$
l_9	$[Return]$	$\{Env(\_1) \rightarrow tainted, Env(\_2) \rightarrow tainted, Env(\_3) \rightarrow tainted, Env(\_4) \rightarrow untainted\}$

Table 6.2: The status of the Taint element of the instrumentation, with the SwitchInt taking the path through the `0` case.

Label	Rule applied	Taint
l_1	$[Call_{normal}]$	$\{Env(\_1) \rightarrow tainted\}$
l_2	$[Ass]$	$\{Env(\_1) \rightarrow tainted, Env(\_2) \rightarrow tainted\}$
l_3	$[Ass]$	$\{Env(\_1) \rightarrow tainted, Env(\_2) \rightarrow tainted, Env(\_3) \rightarrow tainted\}$
l_4	$[SwitchInt_2]$	$\{Env(\_1) \rightarrow tainted, Env(\_2) \rightarrow tainted, Env(\_3) \rightarrow tainted\}$
l_7	$[Call_{normal}]$	$\{Env(\_1) \rightarrow tainted, Env(\_2) \rightarrow tainted, Env(\_3) \rightarrow tainted, Env(\_4) \rightarrow tainted\}$
l_8	$[Call_{normal}]$	$\{Env(\_1) \rightarrow tainted, Env(\_2) \rightarrow tainted, Env(\_3) \rightarrow tainted, Env(\_4) \rightarrow tainted\}$
l_9	$[Return]$	$\{Env(\_1) \rightarrow tainted, Env(\_2) \rightarrow tainted, Env(\_3) \rightarrow tainted, Env(\_4) \rightarrow tainted\}$

Table 6.3: The status of the Taint element of the instrumentation, with the SwitchInt taking the path through the `otherwise` case.

Both execution paths are identical from l_1 to l_3, but then diverge at label l_4 with one case applying the $SwitchInt_1$ rule, and the other applying $SwitchInt_2$, corresponding to either the `0` or `otherwise` cases. From there, Table 6.2 goes through labels l_5, l_6, l_8 and l_9 wherein no tainted input is used. Table 6.3 on the other hand, goes through labels l_7, l_8 and l_9 which utilises tainted input.

From this information, we deem it accurate for the taint analysis to report possible errors on l_4 and l_8. The conditional on l_4 will always use a tainted value, as the taint

propagates from _1 through _2 to _3. The output function in l_8 can use either a tainted or untainted _4, meaning that this too, should report a possible error.

Chapter 7

Reachability Analysis

Another promising prospect we found was based in the potential benefits of Rusts non-lexical lifetime concept and how it could potentially be used to keep track of program states.

This chapter goes into depth about this idea by describing a potential reachability analysis which functions much like program slicing. Additionally, this chapter describes in-depth how the borrow-checking system could potentially reduce the state space via use of the lifetimes it employs.

7.1 Reachability Analysis

An idea of how to use Rust's memory management system to a greater extent is to use the lifetime constraints to analyse taint propagation. It should be possible to track the propagation of a tainted variable through the program and see if it possible for a source to reach a sink. This is done by looking at the constraints the borrow checker creates for lifetimes, and over-approximating all constraints to be related to assignments. Then, if both the sources and sinks are specified in the program, it should be possible check if a source is able to reach a sink by searching for a path leading from the source to the sink through overlapping points in the lifetime constraint sets.

To show how this works, we construct a simple program in Rust.

```
1 fn main() {
2     let a = input(4); // Taint source
3     let c = &a + 3;
4     output(c); // Taint sink
5 }
```

Listing 7.1: Rust snippet where a taint source will reach a taint sink.

In Listing 7.1, a variable is assigned through the source `input` which then further in the program reaches the sink `output`. The purpose of this is to see if it is possible to use the lifetime constraints to identify whether the source reaches the sink.

To calculate the lifetime constraints for our sample program, we must first reduce it into MIR code. This allows us to more easily calculate the lifetime constraints manually, due to MIR being the CFG representation of a Rust program.

```
1  bb0: {
2      _1 = input() -> bb1;              // scope 0 | A/0
3  }
4  bb1: {
5      _2 = &_1;                         // scope 1 | B/0
6      _4 = _2;                          // scope 2 | B/1
7      _3 = <&i32 as Add<i32>>::add(move _4,
8               const 3_i32) -> bb2;     // scope 2 | B/2
9  }
10 bb2: {
11     _6 = _3;                          // scope 3 | C/0
12     _5 = output(move _6) -> bb3;      // scope 3 | C/1
13 }
14 bb3: {
15     return;                           // scope 0 | D/0
16 }
```

Listing 7.2: The MIR representation of Listing 7.1.

Listing 7.2 show the MIR representation of our original sample program. Of note is the fact that each basic block is categorised into regions based on its location in the CFG. Here, each basic block gets assigned a letter as an identifier, and a number serving as an index for Points in the basic block. These indexes serve as the main way to keep track of liveness for lifetimes , and as such are also the main way of notating them.

Looking at the variable `a` on line 2 in Listing 7.1, we can track its lifetime `'a` based on where it is declared and when it is no longer in scope. In Listing 7.2 it is declared on line 2 as `_1 = input()` and is therefore live from the next Point in the CFG, Point `B/0`. From here on, it is possible to further extend `'a`, by identifying where in the program it is necessary for `a` to be live, these being Points `B/0, B/1, B/2, C/0` and `C/1`. Following these principles, the next step is to calculate the lifetime constraints for each of the lifetimes in the program. In this case `'a` will cover the entire program until, and including, the output call in BB2. Continuing with `c`, it is live at Point `C/0` and dies at the same Point as `a`. The lifetime constraints for `'a` and `'c` can then be written as `(Lifetime: Point) @ Point`, meaning we would end with the following lifetime constraints:

```
('a: B/0) @ B/0
('a: B/1) @ B/1
('a: B/2) @ B/2
('a: C/0) @ C/0
('a: C/1) @ C/1
('c: C/0) @ C/0
```

```
('c: C/1) @ C/1
```

The next step is to make sure that the lifetime of a source, outlives the lifetime of references that borrows it. In Listing 7.1 this means that the lifetime of `'a`, has to outlive the lifetime `'c`. To make sure of this, the Rust compiler uses its subtyping rules to identify the places which contains a borrow expression, and produce a subtyping constraint from the Point where the borrow is valid. In the case of our running example it would be at Point `C/0`, as `c` is not valid at Point `B/2` due to it being the Point it is assigned and therefore not yet alive. As such the following subtyping constraint, would be added to our existing lifetime constraints:

```
('c: 'a) @ C/0
```

This should be read as `'c` being a subtype of `'a`, which is valid from Point `C/0`, meaning that lifetime `'a` should outlive lifetime `'c`.

Once all the constraints have been identified, the Rust compiler then solves them to make sure that all lifetimes are valid [10]. This means identifying all subtype constraints in the constraint set, and using an inference algorithm to iterate through the set. Using a fixed-point iteration, an empty set is created for each lifetime variable. The lifetime constraints are used as the set is iterated over, to grow the set until all constraints are satisfied. This is done by looking for subtyping type constraints in the form of `('a:'b) @ Point P`, and making sure that all Points in the lifetime `'a` that can be reached from the Point P, are contained in the lifetime `'b`. This algorithm uses a depth-first search algorithm which adds each Point found to the constraint set of `'a`, which stops when we reach the end of lifetime `'b`. However, in the running example, both of the constraints end up with no real changes as `'c` is fully contained in `'a`. As such, the solved lifetime constraints will look as follows:

```
'a = {B/0, B/1, B/2, C/0, C/1}
'c = {C/0, C/1}
```

We theorise that it is possible to create an algorithm that uses these constraints to analyse whether there is a path from a source to a sink. One possibility would be a type of fixed-point algorithm that would grow a candidate set, based on overlap in constraint sets. We base this on the knowledge that there exists a fixed-point for such an algorithm for all programs, as all programs are of finite size. As such, there must also exist a finite amount of lifetimes with finite places they must be alive for. This means that we should be able to start at a sink Point and go through our set of lifetimes to ascertain whether there is any path through the lifetime constraints, that lead from a sink to a source.

To support this idea, let us create another example wherein the source does not reach the sink. If our theory holds, we should be able to see this when looking at the lifetime constraints of the program.

```
1 fn main() {
2     let a = input(4); //Taint source
```

```
3       let b = &a;
4       let c = 5 + 3;
5       output(c); //Taint sink
6   }
```

Listing 7.3: Rust snippet where a taint source can never reach a taint sink.

In Listing 7.3 there is no way for the tainted source `a` to reach the sink `output` on line 5. As such, it should also be possible to see this reflected in the lifetime constraints.

Following the same steps as in Listing 7.1, we start by reducing the Rust source code to MIR code.

```
1  bb0: {
2      _1 = input() -> bb1;              // scope 0 | A/0
3  }
4  bb1: {
5      _2 = &_1;                         // scope 1 | B/0
6      _3 = const 8_i32;                 // scope 2 | B/1
7      _5 = const 8_i32;                 // scope 3 | B/2
8      _4 = output(move _5) -> bb2;      // scope 3 | B/3
9  bb2: {
10     return;                           // scope 0 | C/0
11 }
```

Listing 7.4: The MIR representation of Listing 7.3.

From the MIR code shown in Listing 7.4, we can identify which Points each lifetime is valid at. It is worth noting that there will be no valid lifetime for `b`, as its assignment becomes irrelevant immediately after assignment. This automatically divests it and frees the memory it points to. In the example there are no subtypings either, meaning the solved constraint set will not contain any changes when compared to the initial lifetime constraints. The solved constraints are therefore as follows:

$$
\begin{aligned}
\text{'a} &= \{B/0,\ B/1\} \\
\text{'c} &= \{B/2,\ B/3,\ C/0\}
\end{aligned}
$$

In this solved constraint set, we can see that the lifetimes of `'a` and `'c` do not overlap. It is therefore safe to assume that any taint sources that are valid during `'a`'s lifetime cannot reach the taint sink in `'c`'s lifetime. Therefore, we would be able to remove those sources from consideration.

However only considering whether a source reaches a sink, should not make a meaningful reduction of the state space. An error in logic, like the one shown in Listing 7.3, should only be a result of bad coding practises, hopefully resulting in it being a rare occurrence.

It might be possible for us to remove certain lifetimes from the collection of lifetimes, if we could annotate a lifetime as a sanitiser in the same way we annotate it as a source. Doing this should hopefully leave us with no overlapping lifetimes around the sink.

```
fn main() {
    let a = input(4); //Taint source
    let b = &a;
    let c = sani(b); //Sanitiser
    let d = c + 3;
    output(d); //Taint sink
}
```

Listing 7.5: Rust snippet where a taint source is sanitised before reaching a sink.

In Listing 7.5 there is a sanitiser function `sani`, which takes a tainted value and returns a untainted value. As such, when considering overlaps in the constraint sets, `c`'s set could be ignored.

```
bb0: {
    _1 = input() -> bb1;                    // scope 0 | A/0
}
bb1: {
    _2 = &_1;                               // scope 1 | B/0
    _4 = _2;                                // scope 2 | B/1
    _3 = sani(move _4) -> bb2;              // scope 2 | B/2
}
bb2: {
    _6 = _3;                                // scope 3 | C/0
    _7 = CheckedAdd(_6, const 3_i32); // scope 3 | C/1
    assert(!move (_7.1: bool), "", move _6, const 3_i32) -> bb3;
    ↪                                           // scope 3 | C/2
}
bb3: {
    _5 = move (_7.0: i32);                  // scope 3 | D/0
    _9 = _5;                                // scope 4 | D/1
    _8 = output(move _9) -> bb4;            // scope 4 | D/2
}
bb4: {
    return;                                 // scope 0 | E/0
}
```

Listing 7.6: The MIR representation of Listing 7.5.

Listing 7.6 show the MIR representation of Listing 7.5 wherein we can follow the lifetime of `c` which is denoted as `_6` in MIR. As such solving the lifetime constraints for this example, results in the following constraints:

```
'a: {B/0, B/1, B/2, C/0, D/0}   //Tainted lifetime
'b: {B/1, B/2, C/0, D/0}        //Tainted lifetime
'c: {C/0, C/1, D/0, D/1, D/2}   //Sanitised lifetime
'd: {D/0, D/1, D/2}             //Sanitised lifetime
```

With `'a` as a tainted lifetime, we have a subtyping constraint for Listing 7.6 `'b: 'a @ B/1`. As such, we can be sure that `'b` is also a tainted lifetime. Likewise, we know that `'c` is a sanitised lifetime, and we have a subtyping constraint `'d: 'c @ D/0`, meaning we can know that `'d` is also a sanitised lifetime.

As sanitised lifetimes are safe for use, we can now remove the lifetimes that are affected by the sanitiser. The remaining lifetime constraints, being `'a` and `'b` can then be used to identify remaining paths from source to sink, of which there are none.

7.2 Theory for Reachability Analysis

As mentioned in Section 7.1, we know that all Rust programs must be of some finite size, meaning that there is a finite number of lifetimes covering a finite set of Points in the CFG of the program. Furthermore, we can also trivially increase the granularity of the CFG from basic block level to Point level. This is possible because all basic blocks have a single Point that is the entry, and a single Point that is the terminator for the block. Each block is simply a single line of nodes, with each node representing a discrete Point in the program.

For any given program, we should be able to ascertain which lifetimes originate from taint sources, and which Points in the program are sinks. With that we should be able to grow a taint propagation set of Points, based on the lifetimes in the program.

To illustrate this idea, we have included a naive fix-point pseudo-algorithm. This algorithm runs source to sink, and find the potential Points where the lifetime propagation sets for sources can overlap with the sink Points. The idea is to begin with the set of constraints for the sources, and expand this with the constraint sets of lifetimes, which have at least one Point in common and is not a sanitiser or sanitised lifetime. This continues until a fix-point is reached. This algorithm can be seen below in Algorithm 1

Algorithm 1 Constraint-set fix-point-based reachability

```
function ConstraintReach(SourceLifetime, SinkPoint, SanitisedLifetimes, AllLifetimes)
    let ConSet ← ConstraintsOf(SourceLifetime)
    let NonSanitisers ← AllLifetimes \ SanitisedLifetimes
    repeat
        let PrevConSet ← ConSet
        for each Var ∈ NonSanitisers do
            if ∃Point ∈ ConstraintsOf(Var) : Point ∈ ConSet then
                let ConSet ← ConSet ∪ ConstraintsOf(Var)
            end if
        end for
    until PrevConSet = ConSet
    if SinkPoint ∈ ConSet then
        return TRUE
    else
        return FALSE
    end if
end function
```

This specific specification of the algorithm would have to be performed on a per-source basis, but could be expanded to consider all sources at once.

For iteration P_0, we would start with just the constraint set of the source lifetime. Iteratively, we add all the Points from other lifetimes that have at least 1 Point in P_0 already, unless the lifetime originates from a sanitiser, and thus makes P_1. We can continue this until we reach a fix-point where P_n is the same as P_{n-1}. Finally, we simply check whether the sink Point is contained in P_n, and thus answer whether a given taint source can possibly reach the sink.

If we consider the CFG, sources, sinks, and sanitisers, we can generalise to 3 cases:

- The CFG is in a single line with a source node, then a sanitiser node, and finally sink node.
- The Source and sanitiser nodes are separate, but both lead to the sink node.
- The Source node leads to both the sanitiser and sink nodes, which are separate.

These cases can be seen depicted in Figure 7.1 below in that order. It is worth noting that all cases could be seen as a sub-graph in a larger CFG. Furthermore, we also note that it is only possible to remove the source from consideration in the first case. This is due to the second and third cases having a path from source to sink that goes around a sanitiser, thus not enabling us to ensure that the taint has been sanitised.

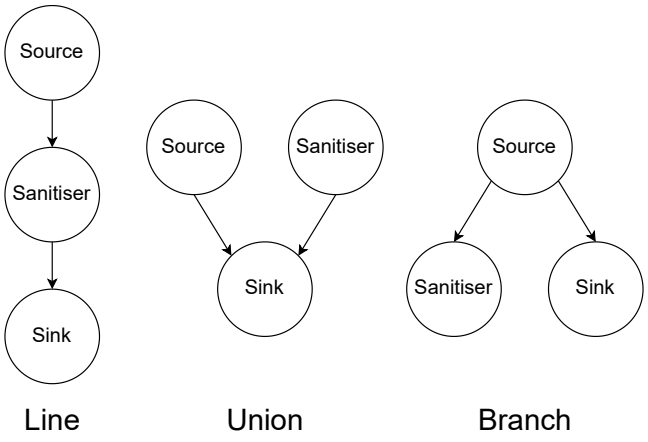

Figure 7.1: 3 generalised node constructions.

If we instead look at the last non-trivial step before the simplifications shown in Figure 7.1 we would, for each of the general cases, get a diamond-shaped CFG. These can be seen below in Figure 7.2

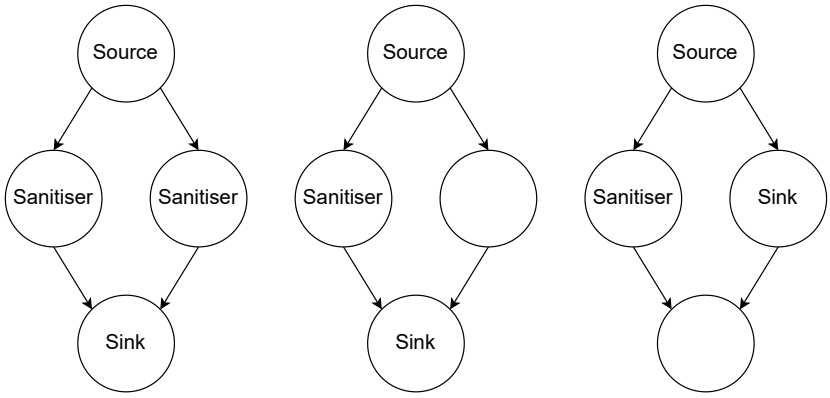

Figure 7.2: Examples depicting the last non-trivial step before simplification for the Line, Union, and Branch cases.

The leftmost graph in Figure 7.2 can remove either sanitiser to arrive at the line general case. The middle can remove the blank node (and edge between sanitiser and sink) to arrive at the union case. Finally, the rightmost can remove the blank node to arrive at the branch case. These simplification operations should be trivial to perform, as well as general enough to abstract to larger CFGs.

Chapter 8

Discussion

This chapter reflects on the choices made during development, and contains considerations and arguments relating to the semantics, instrumentation, and analyses described throughout this body of work.

8.1 Semantics

One of the primary goals of this report, defined in Chapter 1 was to create a taint analysis. While we spend time throughout the report describing various static analyses, we ultimately ended up deciding against an actual implementation. This was due to various limitations imposed by both Rust but also the potential costs of creating an analysis that would be functionally identical to existing ones like the one made by Gústafsson and Njor. We instead chose to focus on a theoretical approach, wherein we describe all the novel features that we have worked with, which we then supplemented by testing the parts of our analysis that were functionally identical to the analysis created by Gústafsson and Njor.

To reiterate, our semantics are heavily inspired by the semantics presented in "Static Analysis in Rust" by Gústafsson and Njor [1]. We make many substantial changes, but argue that none impact the actual base functionality of the semantics. This allowed us to utilise the taint analysis they created, which we forked from their repository. This was necessary as their analysis no longer works. This was primarily due to the functionality previously found in the `rustc` being split into multiple other crates. We fixed these errors in our fork, and restored the functionality of the taint analysis. The updated code can be found in our repository at `https://gitlab.com/Rawrior/P9-code` under the folder `Taint-Analysis`. The code should appear in a functioning state, and can be run at the readers leisure.

As none of our changes to the semantics have any impact on the fundamental functionality of the rules as they are used in their work, we argue that it is possible for us to use their analysis implementation as proof that our work functions. Given the fact that all of the tests for the implementation pass, combined with covering a satisfactory variety of situations, we deem them to be enough to help prove the correctness of the semantics.

8.2 Increasing Static Taint Analysis Accuracy

While we deem the static taint analysis described in Chapter 6 adequate, we argue that it could be made more accurate. With the current taint lattice, we can only report 'untainted' or 'tainted'. As such, any reports of taint include all reports that are either possible or definite, with no way to differentiate the two. If we were to use a different lattice, we believe it would be possible to differentiate the scenarios. This new lattice would have 3 layers: A bottom element $\perp$, a middle layer with elements *tainted* and *untainted*, and a top element *maybe*. This can be seen in Figure 8.1, below.

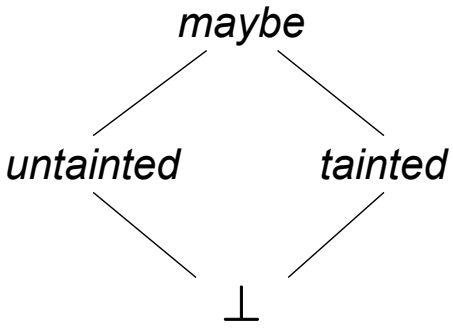

Figure 8.1: An alternative lattice

If a value was tainted in one execution path, but untainted in another, the least upper bound between them would be the *maybe* element. With this, it should be possible to differentiate between errors and warnings, and possibly even deny compilation if an error is present.

We can generally be sure that any program that does not loop infinitely will have the most accurate taint information retained until the program's exit points. For any given program exit point, there will be either 0 or any number of branches and merges in the CFG. Branches have no immediate impact on information accuracy, while merges only have the limited effect discussed before. Because of this, we can be sure that any use of taint in a sink will at the very least have a possible execution path where the given value is tainted.

8.3 Alternative Reachability Algorithm

As an alternative to the fixed-point algorithm described in Section 7.2, we argue that it should be possible to create a graph-based algorithm instead. We construct the graph with each lifetime represented by a node in the graph, excluding those lifetimes which originate from sanitisers. For all pairs of lifetimes (nodes), if there are any Points in common between their constraint sets, we create an edge between the nodes. With this, we can see which lifetimes overlap with each other, and can do a number of operations on the graph. As examples, we could calculate whether the graph is connected, such that from any node, we can reach any other node, or we could use a breadth-first search and check if the sink can be reached from the source.

Crucially, we would choose an operation that would tell us whether a source can reach any sink in the graph. If there is, it would tell us that there may be an execution path where tainted output could be used. If there isn't, it is possible that we can ignore the source for the purpose of taint analysis.

Chapter 9

Conclusion

The goal of this project was to verify whether the borrowing system employed by Rust could provide any benefits to static analysis. Our solution to this consists of a set of formalised semantic rules and associated instrumentation, a definition of what borrowing is, a set of predicates which define exceptions to the definition, and a theoretical basis as to how to employ these semantics on an analysis.

As examples, we chose two analyses wherein we saw potential for improvement via utilisation of borrowing. One approach was static taint analysis, wherein we applied our instrumentation to an analysis created by Gústafsson and Njor, as a test-case to test whether our instrumentation was sound when applied to an actual implementation. The other approach is purely theoretical, and relates to something akin to data-slicing. We formulated a hypothesis, which revolves around Rusts non-lexical lifetime to possibly reduce the state space of a taint analysis.

We argue that our presented theory is sound, and back this up via the presented intuitive, generalised arguments detailed in Chapter 8. While these are not formal mathematical proofs, we argue that these arguments should be enough to convince the reader, that the theory we have presented is correct. In the case of the data-slicing via lifetimes, we were unable to implement it fully in code, due to us not being able to get the necessary access in the `rustc` crate. The majority of content in those crates was marked private, and thus would have to be used within the crate itself. We would have had to change the source code of the compiler itself to leverage these features, which we concluded that we did not have the expertise, nor the resources, to do. We tried to reach out to the Rust community at large through both the online messaging board Reddit, in addition to the Rust community chat on the Matrix platform. However, both of these yielded no results, as we received no answers to any of our questions

This, of course, does not mean that it is impossible, only that it is outside of our current capabilities. A more refined version of the data-slicing we have created may be able to

leverage the accessible features, or use some other way to create the desired graph.

Though we were not able to, we are mostly confident that our presented theory should be able to be implemented fully, either as designed, or with minor modifications. We are also confident that the theory described in this report would provide a benefit in security for a given user-interfacing program. As there would be taint analysis in place at compile time, warnings and errors could be given earlier, making it easier for developers to create more secure applications, if they employed this feature.

9.1 Future Works

There are unexplored ideas, or aspects of explored ideas, that were either not directly considered for implementation or cut from development due to resource restraints, but whose addition could nonetheless prove beneficial. This section presents these features, and discusses some advantages they could give.

9.1.1 Accessing `rustc` Crates

One of the major obstacles during the course of this body of work was the fact that we only had limited access to the features of certain `rustc` crates. Specifically, we never managed to find a feasible way to access the private features pertaining to lifetime constraints. That is to say, we argue that it should be possible to change or replicate the source code of the compiler itself to leverage these feature, but that this was outside the scope of this project.

If given more time, it would be interesting to delve into whether it could be possible to access these functions - either via changing the source code, replicating the source code, or some other approach.

9.1.2 Further Use of Instrumentation

An added benefit of our instrumentation is the fact that it should be possible to use it as the basis for a dynamic taint analysis. As taint is tracked throughout the rules, it should be possible to define when we would consider something a taint sink, and report it when the execution causes a tainted value to be used therein. As well, the boolean formulas for the borrow checking are based on the instrumentation. Should more information be added to the instrumentation, we would be able to formalise more formulas.

9.1.3 Implementation of Reachability Analysis

While we believe that the theory behind the reachability analysis is solid, we argue that a practical implementation of the theory would aid in either confirming of denying our hypothesis.

While this problem would have to also overcome the problems presented in Section 9.1.1, it could potentially serve as a powerful data-slicing tool, were we to successfully accomplish this feat.

9.1.4 Expanding Semantics

Our semantics cover only a subset of syntax constructions in MIR, though these are sufficient for our current taint analysis. Expanding the semantics to cover all of the missing constructions could lead to more general use for the formalisation.

Some of the missing syntax constructions include, but are not limited to, variable declarations via `let` statements, type annotations, structs, and indexing in collection variables such as arrays or tuples.

9.1.5 Creating Semantic Type System for MIR

A large part of this body of work has been to formalise part of the inner workings of Rust with the MIR layer of the Rust compiler. As such, it might prove beneficial to extend the formalisation of Rust to also include the type system for MIR.

By formalising the type system in semantics, we could ease the burden of using Rust, by allowing readers to gain an intrinsic understanding of the capabilities of the types available in Rust.

9.1.6 Define Semantic Rules for Borrowing

Semantic rules for borrowing were never formalised in this body of work, despite the primary focus being ascertaining potential benefits of the borrowing system in Rust. The reason for this is based in the fact that such an attempt would quickly become complex, and require competences that fall outside of the skill-set we have.

Regardless, we do deem it beneficial to extend the semantics by fully formalising the rules for borrowing, as this could lead to formal proofs about the safety of borrowing, or potential flaws therein.

Bibliography

[1] Emil Jørgensen Njor and Hilmar Gústafsson.
"Static Taint Analysis in Rust".
MA thesis. Department of Computer Science, Aalborg University, 2021.
51 pp.
URL: https://projekter.aau.dk/projekter/files/421583418/Static_Taint_Analysis_in_Rust.pdf.

[2] Mitre Corporation.
CWE - Common Weakness Enumeration.
URL: https://cwe.mitre.org/index.html (visited on 02/22/2022).

[3] Mitre Corporation.
CWE - Common Weakness Enumeration.
URL: https://cwe.mitre.org/about/index.html (visited on 02/22/2022).

[4] CWE Top 25 Team.
CWE - 2019 CWE Top 25 Most Dangerous Software Errors.
URL: https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html (visited on 10/26/2021).

[5] CWE Top 25 Team.
CWE - 2020 CWE Top 25 Most Dangerous Software Weaknesses.
URL: https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html (visited on 10/26/2021).

[6] CWE Top 25 Team.
CWE - 2021 CWE Top 25 Most Dangerous Software Weaknesses.
URL: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html (visited on 10/26/2021).

[7] CWE Top 25 Team.
CWE - 2021 CWE Top 25 Most Dangerous Software Weaknesses - Methodology.
URL: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html#methodology (visited on 02/23/2022).

[8] Rust Development Team.
What Is Ownership?
URL: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html (visited on 06/16/2022).

[9] Rust Development Team.
Introducing MIR.
URL: https://blog.rust-lang.org/2016/04/19/MIR.html (visited on 02/18/2022).

[10] Rust Language Foundation.
2094-nll - The Rust RFC Book.
URL: https://rust-lang.github.io/rfcs/2094-nll.html (visited on 03/01/2022).

[11] Anders Møller and Michael I. Schwartzbach.
Static Program Analysis.
Department of Computer Science, Aarhus University, 2021.
176 pp.

[12] Microsoft.
SQL Injection - SQL Server.
URL: https://docs.microsoft.com/en-us/sql/relational-databases/security/sql-injection (visited on 02/28/2022).

[13] Confluence.
Taint Analysis - SEI CERT C Coding Standard.
URL: https://wiki.sei.cmu.edu/confluence/display/c/Taint+Analysis (visited on 11/03/2021).

[14] *MIRI*.
URL: https://github.com/rust-lang/miri (visited on 03/22/2022).

[15] *MIRAI*.
URL: https://github.com/facebookexperimental/MIRAI (visited on 03/22/2022).

[16] *Rust Compiler Error Index*.
URL: https://doc.rust-lang.org/stable/error-index.html (visited on 04/06/2022).

Appendix A

Rust Error Codes

Error Code	Description
E0312	Reference's lifetime of borrowed content does not match the expected lifetime.
E0381	It is not allowed to use or capture uninitialized variables.
E0382	A variable was used after its contents have been moved elsewhere.
E0384	An immutable variable was reassigned.
E0492	A borrow of a constant containing interior mutability was attempted.
E0499	A variable was borrowed as mutable more than once.
E0500	A borrowed variable was used by a closure.
E0501	A mutable variable is used but it is already captured by a closure.
E0502	Borrowed data escapes outside of closure.
E0503	A value was used after it was mutably borrowed.
E0505	A value was moved out while it was borrowed.
E0506	An attempt was made to assign to a borrowed value.
E0507	A borrowed value was moved out.
E0508	A value was moved out of a non-copy fixed-size array.
E0521	Borrowed data escapes outside of closure.
E0596	Tried to mutably borrow a non-mutable variable.
E0597	A value was dropped while it was still borrowed.
E0621	A mismatch between the lifetimes appearing in the function signature and the data-flow found in the function body.
E0626	A borrow in a generator persists across a yield point.
E0712	A borrow of a thread-local variable was made inside a function which outlived the lifetime of the function.
E0713	An attempt is made to borrow states past the end of the lifetime of a type that implements the *Drop* trait.
E0716	A temporary value is being dropped while a borrow is still in active use.
E0764	A mutable reference was used in a constant.

These are the Rust error codes that are used as a basis to formulate the predicates used to explore the capabilities of the borrow checker. These error codes have been pulled directly from the 'Rust Compiler Error Index'[16].

We have elected to exclude errors E0312, E0492, E0621, E0626, E0712, E0713, and E0764 as we deem them based more in typing errors, or misuse of types or keywords.

Appendix B

Predicate Examples

These are small Rust snippets to be used as examples to show the results of errors, when subjected to the borrow checker. Each example is created to show one predicate not being upheld, with the example indexes corresponding to the predicates in Section 5.1, i.e. example 1 in this appendix breaking predicate 1 and so on. All of the examples shown below have been formatted so that readers can compile them directly using the Rust compiler or an online compilation tool, such as the Rust Playground (`https://play.rust-lang.org/`) and view the corresponding error messages.

Example 4

```
1 fn main(){
2     let x: i32;
3     println!("{}", x);
4 }
```

This example shows a straight-forward code snippet wherein a printing statement found on line 3 tries to print `x`, which has not been initialised with a value. The specific error reports `x` as a ”possibly-uninitialised variable”.

Example 5

```
1 fn main(){
2     let mut x = 1;
3     let y = &mut x;
4     x += 1;
5     println!("{}", y);
6     println!("{}", x);
7 }
```

This example tries to mutate a variable `x` while it is not the current owner. On line 3 we temporarily transfer ownership from `x` to the variable `y` instead. This results in errors

when we then try to mutate `x` on line 4, as the ownership at that point does not belong to `x`.

Example 6

```
1 fn main(){
2     let mut x = 5;
3     let y = &mut x;
4     let z = &mut x;
5     println!("{}", y);
6     println!("{}", z);
7 }
```

This example would result in an error due to mutable borrows found on lines 4 and 5. While these lines of code trigger no errors on their own, an error will occur as soon as both of them are used simultaneously, as is done via the printing statements on lines 6 and 7.

Example 7

```
1 fn main(){
2     let x = 1;
3     x += 1;
4     println!("{}", x);
5 }
```

This snippet will emit an error for line 3, as the line is attempting to mutate the immutable `x` variable. This snippet can be fixed by either changing line 3 to `let x = x + 1;` or by declaring `x` with `let mut`. The former would fully re-declare `x`, while the latter would allow `x` to be mutated.

Example 8

```
1 fn main(){
2     let x: &i32;
3     {
4         let y: i32 = 2;
5         x = &y;
6     }
7     println!("{}", *x)
8 }
```

This code snippet shows an example of a borrow not living long enough. As `y` only exists on the scope it is declared in, the assignment to `x` is erroneous, as `x` attempts to use the value later.

Example 9

```
1 fn main(){
2     let a = 0;
3     let mut x = &mut a;
4     println!("{}", x);
5 }
```

This code snippet shows an attempt to mutably borrow an immutable variable. As a mutable borrow would imply that the variable being referred to is also mutable, this is impossible.

Example 10

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

This example shows a code snippet where we attempt to use `x` in the assignment to `z` on line 4, while it is mutably borrowed by `y`. This is disallowed, as `y` is the current owner of the value behind `x`, and the assignment would thus have to go through `y` to be valid.

Example 11

```
1 fn main(){
2     let mut x = 1;
3     let y = &x;
4     x += 1;
5     println!("{}", y);
6     println!("{}", x);
7 }
```

In this example we attempt to mutate a variable while it is immutably borrowed. We initialise the mutable variable `x` on line two. We then immutably borrow the contents of `x` to the variable `y` on line 3. This results in an error when we try to mutate `x` on line 4, as the borrow is still in effect until after line 5.

Example 12

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

This example shows a code snippet wherein we attempt to create an immutable borrow of a value that is already mutably borrowed. On line 3 we mutably borrow the variable `x`, and then subsequently make an immutable borrow on line 4. This causes an error as Rust does not support creating immutable references to a value while that value is mutably borrowed by another variable.

Example 13

```
1 fn main(){
2     let x = vec![1,2,3];
3     let y = &x;
4     let z = x;
5     println!("{}", y[0]);
6     println!("{}", z[0]);
7 }
```

This example shows a code snippet wherein we attempt to move the contents of a variable `x` to another variable `z` on line 4 after already borrowing `x` to the variable `y` on line 3. This results in an error as you cannot change ownership while the variable is being borrowed.

Example 14

```
1 fn main(){
2     let x = vec![1,2,3];
3     let y = x;
4     let z = x;
5     println!("{}, {}", y[0], z[0]);
6 }
```

This example shows that values cannot be moved from the same variable twice. All of Rust's primitive types (with the exception of `str`) implement the `Copy` trait, meaning assignments create a copy of the value. However, some types, like vectors, do not inherit this trait, and instead have to employ `Move` operations instead. In the example above, we initialise a variable `x` which is assigned a vector on line 2. This vector is then assigned to the variables `y` and `z` on lines 3 and 4. This is not allowed, as we move the vector from `x` twice in a row resulting in an error.

Example 15

```
1 use lazy_static::lazy_static;
2 lazy_static!{static ref X: Vec<i32> = vec![1,2,3];}
3
4 fn main(){
5     let y = X;
6     println!("{}", y[0]);
7 }
```

In this example we try to move a static variable in the form of the vector `X` into another variable `y`. Of note, is the import of the `lazy_static` macro from the crate of the same name. We include this macro, as you currently (for Rust stable 1.59.0) cannot use functions in static declarations if said functions are not constant. This includes the `vec!` macro, but this can be bypassed via the `lazy_static` macro seen on line 3. Once we try to move this static reference `X` on line 5 we get an error, as we cannot move a static variable.

Example 16

```
1 fn main(){
```

```
2       let x: [String; 2] = [String::from("Hello"), String::from("world!")];
3       let y = x[0];
4       println!("{}", y);
5   }
```

In this example, we create an array containing the `String` type. `String` does not implement the `Copy` trait, making this array a non-`Copy` collection. In general, we say that any collection containing a uniform type, where the type does not implement the `Copy` trait, is a non-`Copy` collection. As the elements in a non-`Copy` collection do not copy on assignment, the assignment on line 3 would try to move the value out of the vector, which is not allowed.

Example 17

```
1   fn main(){
2       let x = vec![1,2,3];
3       let y = &x;
4       let z = *y;
5       println!("{}", z[0]);
6   }
```

This code snippet shows an example wherein we attempt to move the value housed in a variable `x`. We create the immutable reference `y` line 3, then attempt to `move` the vector to the variable `z` on line 4. This results in an error, as Rust does not support moves made from inside a reference.

Example 18

```
1   fn main(){
2       let mut x = 5;
3       let y = &mut x;
4       add_10_to_ref(y);
5   }
6
7   fn add_10_to_ref(num_in: &mut i32) {
8       let x = &num_in;
9       || { *num_in += 10; };
10      println!("{}", x);
11  }
```

This code snippet shows an example where the principle of unique access in closures is violated. Closures, or anonymous/lambda functions in Rust, require unique access to all variables that are used in them, for the entirety of the lifetime of the closure.

Example 19

```
1   fn main(){
2       let a = 0;
3       let mut x: &i32 = &a;
4
5       let mut convoluted_assign = |n: &i32|{ x = n; };
6       convoluted_assign(&123);
7   }
```

This code snippet shows how a borrow would escape from a closure. The closure takes a reference as argument and assigns it to `x` from the surrounding scope. However, as the closure borrows the reference, the reference is only valid within the closure, and as such cannot exist outside of the closure.

Appendix C

Gústafsson and Njor Semantics

C.1 MIR Syntax

$$z \in \mathbb{Z}$$
$$i \in \text{Ident}$$

Where z is a literal integer value, and i is any string that matches the regular expression $[\texttt{a-zA-Z}]^+$.

$bop \in \text{BinOps} = \{\text{Add}, \text{Sub}, \text{Mul}, \text{Div}, \text{Rem}, \text{BitXor}, \text{BitAnd}, \text{BitOr}, \text{Shl}, \text{Shr}, \text{Eq}, \text{Lt}, \text{Le}, \text{Ne}, \text{Ge}, \text{Gt}\}$
$uop \in \text{UnOps} = \{\text{Not}, \text{Neg}\}$

$$\begin{aligned}
P \in \text{Places} &::= \{\texttt{_}n \mid n \in \mathbb{N}_0\} \\
O \in \text{Operands} &::= \texttt{move}\ P \mid \texttt{copy}\ P \mid \texttt{const}\ z \\
R \in \text{Rvals} &::= O \\
&\mid\ bop\texttt{(}O_1\texttt{,}\ O_2\texttt{)} \\
&\mid\ uop\texttt{(}O\texttt{)} \\
&\mid\ \texttt{\&[mut]}P \\
\beta \in \text{Bid} &::= \{\texttt{bb}n \mid n \in \mathbb{N}_0\}
\end{aligned}$$

$$
\begin{aligned}
S \in \text{Stmt} &::= P \texttt{ = } R\texttt{;} \\
X \in \text{Targets} &::= z\texttt{: } \beta \\
T \in \text{Tmnt} &::= \texttt{goto -> } \beta\texttt{;} \\
&\mid \texttt{switchInt(}O\texttt{) -> [}[X,]^*\texttt{ otherwise: } \beta\texttt{];} \\
&\mid \texttt{assert(}O\texttt{, }z\texttt{) -> } \beta\texttt{;} \\
&\mid P\texttt{ = }i\texttt{(}O_0,...,O_n\texttt{) -> } \beta\texttt{;} \\
&\mid i\texttt{(}O_0,...,O_n\texttt{);} \\
&\mid \texttt{return;}
\end{aligned}
$$

$$
\begin{aligned}
\{S^*T\} \in \text{Body} &::= \text{Stmt}^* \times \text{Tmnt} \\
B \in \text{Block} &::= \beta\texttt{: } \{S^*T\} \\
F \in \text{Fn} &::= \texttt{fn } i\texttt{(}O^*\texttt{) \{}B^+\texttt{\}} \\
M \in \text{Mir} &= F^+
\end{aligned}
$$

$$
\begin{aligned}
\xi &\in \text{Locations} \\
e \in \text{Environment} &= \text{Places} \to \text{Locations} \\
\zeta \in \text{Store} &= \text{Locations} \to (\mathbb{Z}_\perp^\top + \text{Locations}) \\
\mathit{LocationType} &= \text{Locations} \to \{\mathit{mut}, \mathit{imut}\} \\
\mathit{prg} \in \text{Program} &= \text{Ident} \to \text{Function} \\
\mathit{fun} \in \text{Function} &= \text{Bid} \to \text{Body} \\
\mathit{cs} \in \text{CallStack} &= (\text{Bid} \times \text{Places} \times \text{Environment} \times \text{Function})^* \\
\text{Conf}_S &= \text{Stmt} \times \text{Store} \\
\text{Conf}_B &= \text{Body} \times \text{CallStack} \times \text{Store} \\
\to_S &\subseteq \text{Conf}_S \times \text{Store} \\
\to_B &\subseteq \text{Conf}_B \times \text{Conf}_B
\end{aligned}
$$

$$
\mathit{Oeval}(O,\zeta,e) = \begin{cases} z, & \text{if } O = \texttt{const}z \\ \zeta \circ e(P), & \text{if } O = \texttt{move}P \\ \zeta \circ e(P), & \text{if } O = \texttt{copy}P \end{cases}
$$

$$
\mathit{Reval}(R,\zeta,e) = \begin{cases} \mathit{Oeval}(O,\zeta,e), & \text{if } R = O \\ \texttt{bop}(O_1,O_2), & \text{if } R = \mathit{bop}(O_1,O_2) \\ \texttt{uop}(O), & \text{if } R = \mathit{uop}(O) \\ \zeta \circ e(P), & \text{if } R = \&[\mathit{mut}]P \end{cases}
$$

$$
\mathit{BorrowType} = P \to \{\mathit{borrowed}_{\mathit{not}}, \mathit{borrowed}_{\mathit{imut}}, \mathit{borrowed}_{\mathit{mut}}\}
$$

C.2 MIR semantics

$$[Ass]\ \frac{\begin{array}{c}\langle \texttt{P = R}, \zeta\rangle \to_S \zeta[e(P) \mapsto Reval(R)] \\ (BindingType(e(P)) = mut \vee \zeta \circ e(P) = \bot) \qquad \zeta \circ e(P) \neq \top \\ (R = (O \vee uop(O)) \wedge O \neq \texttt{move}P) \vee (R = bop(O_1, O_2) \wedge O_1 \neq \texttt{move}P' \wedge O_2 \neq \texttt{move}P'')\end{array}}{\begin{array}{c}prg \vdash \langle \{\texttt{P = R; S}^*\texttt{T}\}, (\beta, P, e, fun) :: cs, \zeta\rangle \\ \to_B \langle \{\texttt{S}^*\texttt{T}\}, (\beta, P, e, fun) :: cs, \zeta[e(P) \mapsto Reval(R)]\rangle\end{array}}$$

$$[Ass_{Move1}]\ \frac{\begin{array}{c}\langle \texttt{P = R}, \zeta\rangle \to_S \zeta[e(P) \mapsto Reval(R)] \qquad (BindingType(e(P)) = mut \vee \zeta \circ e(P) = \bot) \\ \zeta \circ e(P) \neq \top \qquad O = \texttt{move}P' \qquad R = (O \vee uop(O))\end{array}}{\begin{array}{c}prg \vdash \langle \{\texttt{P = R; S}^*\texttt{T}\}, (\beta, P, e, fun) :: cs, \zeta\rangle \\ \to_B \langle \{\texttt{S}^*\texttt{T}\}, (\beta, P, e, fun) :: cs, \zeta[e(P) \mapsto Reval(R), e(P') \mapsto \top]\rangle\end{array}}$$

$$[Ass_{Move2}]\ \frac{\begin{array}{c}\langle \texttt{P = R}, \zeta\rangle \to_S \zeta[e(P) \mapsto Reval(R)] \qquad (BindingType(e(P)) = mut \vee \zeta \circ e(P) = \bot) \\ \zeta \circ e(P) \neq \top \qquad O_1 = \texttt{move}P' \qquad O_2 \neq \texttt{move}P'' \qquad R = (bop(O_1, O_2))\end{array}}{\begin{array}{c}prg \vdash \langle \{\texttt{P = R; S}^*\texttt{T}\}, (\beta, P, e, fun) :: cs, \zeta\rangle \\ \to_B \langle \{\texttt{S}^*\texttt{T}\}, (\beta, P, e, fun) :: cs, \zeta[e(P) \mapsto Reval(R), e(P') \mapsto \top]\rangle\end{array}}$$

$$[Ass_{Move3}]\ \frac{\begin{array}{c}\langle \texttt{P = R}, \zeta\rangle \to_S \zeta[e(P) \mapsto Reval(R)] \qquad (BindingType(e(P)) = mut \vee \zeta \circ e(P) = \bot) \\ \zeta \circ e(P) \neq \top \qquad O_1 \neq \texttt{move}P' \qquad O_2 = \texttt{move}P'' \qquad R = (bop(O_1, O_2))\end{array}}{\begin{array}{c}prg \vdash \langle \{\texttt{P = R; S}^*\texttt{T}\}, (\beta, P, e, fun) :: cs, \zeta\rangle \\ \to_B \langle \{\texttt{S}^*\texttt{T}\}, (\beta, P, e, fun) :: cs, \zeta[e(P) \mapsto Reval(R), e(P'') \mapsto \top]\rangle\end{array}}$$

$$[Ass_{Move4}]\ \frac{\begin{array}{c}\langle \texttt{P = R}, \zeta\rangle \to_S \zeta[e(P) \mapsto Reval(R)] \qquad (BindingType(e(P)) = mut \vee \zeta \circ e(P) = \bot) \\ \zeta \circ e(P) \neq \top \qquad O_1 = \texttt{move}P' \qquad O_2 = \texttt{move}P'' \qquad R = (bop(O_1, O_2))\end{array}}{\begin{array}{c}prg \vdash \langle \{\texttt{P = R; S}^*\texttt{T}\}, (\beta, P, e, fun) :: cs, \zeta\rangle \\ \to_B \langle \{\texttt{S}^*\texttt{T}\}, (\beta, P, e, fun) :: cs, \zeta[e(P) \mapsto Reval(R), e(P') \mapsto \top, e(P'') \mapsto \top]\rangle\end{array}}$$

$$[Ass_{Ref:mut}]\ \frac{\begin{array}{c}\langle \texttt{P = R}, \zeta\rangle \to_S \zeta[e(P) \mapsto e(P')] \qquad (BindingType(e(P)) = mut \vee \zeta \circ e(P) = \bot) \\ \zeta \circ e(P) \neq \top \qquad R = \&mut P' \qquad BorrowType(P') = borrowed_{not}\end{array}}{\begin{array}{c}prg \vdash \langle \{\texttt{P = R; S}^*\texttt{T}\}, (\beta, P, e, fun) :: cs, \zeta\rangle \\ \to_B \langle \{\texttt{S}^*\texttt{T}\}, (\beta, P, e, fun) :: cs, \zeta[e(P) \mapsto e(P')]\rangle\end{array}}$$

$$[Ass_{Ref:imut}]\ \frac{\begin{array}{c}\langle \texttt{P = R}, \zeta\rangle \to_S \zeta[e(P) \mapsto e(P')] \qquad (BindingType(e(P)) = mut \vee \zeta \circ e(P) = \bot) \\ \zeta \circ e(P) \neq \top \qquad R = \&P' \qquad BorrowType(P') \neq borrowed_{mut}\end{array}}{\begin{array}{c}prg \vdash \langle \{\texttt{P = R; S}^*\texttt{T}\}, (\beta, P, e, fun) :: cs, \zeta\rangle \\ \to_B \langle \{\texttt{S}^*\texttt{T}\}, (\beta, P, e, fun) :: cs, \zeta[e(P) \mapsto e(P')]\rangle\end{array}}$$

$$[Goto]\ \frac{}{prg \vdash \langle \{\texttt{goto -> } \beta\texttt{;}\}, (\beta', P', e', fun') :: cs, \zeta\rangle \to_B \langle fun'(\beta), (\beta', P', e', fun') :: cs, \zeta\rangle}$$

$$[SwitchInt_1]\ \frac{z_i = Oeval(O,\zeta,e'') \qquad i \in \{i,...,n\}}{\begin{array}{c} prg \vdash \langle \{\texttt{SwitchInt(}O\texttt{) -> [}z_1 : \beta_1,...,z_n : \beta_n\texttt{, otherwise: }\beta'\texttt{];}\}, \\ (\texttt{bb}n'',P'',e'',fun'') :: cs,\zeta\rangle \rightarrow_B \langle fun''(\beta_i),(\beta'',P'',e'',fun'') :: cs,\zeta\rangle \end{array}}$$

$$[SwitchInt_2]\ \frac{\forall z | z \in \{z_1,...,z_n\} : z \neq Oeval(O,\zeta,e'')}{\begin{array}{c} prg \vdash \langle \{\texttt{SwitchInt(}O\texttt{) -> [}z_1 : \beta_1,...,z_n : \beta_n\texttt{, otherwise: }\beta'\texttt{];}\}, \\ (\texttt{bb}n'',P'',e'',fun'') :: cs,\zeta\rangle \rightarrow_B \langle fun''(\beta'),(\beta'',P'',e'',fun'') :: cs,\zeta\rangle \end{array}}$$

$$[Assert]\ \frac{z = Oeval(O,\zeta,e')}{\begin{array}{c} prg \vdash \langle \{\texttt{Assert(}O\texttt{, }z\texttt{) -> }\beta\texttt{;}\}, \\ (\beta',P',e',fun') :: cs,\zeta\rangle \rightarrow_B \langle fun'(\beta),(\beta',P',e',fun') :: cs,\zeta\rangle \end{array}}$$

$$[Call_{Normal}]\ \frac{\begin{array}{c} \xi_i\texttt{fresh} \qquad\qquad e' = [\_j \mapsto \xi_j,...] \\ \zeta' = \zeta[\xi_j \mapsto Oeval(O_j,\zeta,e'),e'(P_j) \mapsto \begin{cases} \top, & \text{if } O_j = \texttt{move}P_j \\ \zeta \circ e'(P_j), & \text{otherwise} \end{cases},...] \\ j \in \{0,...,n\} \qquad (LocationType(e(P)) = mut \vee \zeta \circ e(P) = \bot \end{array}}{\begin{array}{c} prg \vdash \langle \{\texttt{P = }i\texttt{(}O_0,...,O_n\texttt{) -> }\beta\texttt{;}\},(\beta',P',e',fun') :: cs,\zeta\rangle \rightarrow_B \\ \langle prg(i)(\texttt{bb0}),(\beta,P,e,prg(i)) :: (\beta',P',e',fun') :: cs,\zeta'\rangle \end{array}}$$

$$[Call_{Panic}]\ \frac{\begin{array}{c} \xi_i\texttt{fresh} \qquad\qquad e' = [\_j \mapsto \xi_j,...] \\ \zeta' = \zeta[\xi_j \mapsto Oeval(O_j,\zeta,e'),e'(P_j) \mapsto \begin{cases} \top, & \text{if } O_j = \texttt{move}P_j \\ \zeta \circ e'(P_j), & \text{otherwise} \end{cases},...] \\ j \in \{0,...,n\} \qquad (LocationType(e(P)) = mut \vee \zeta \circ e(P) = \bot \qquad \zeta \circ e(P) \neq \top \end{array}}{\begin{array}{c} prg \vdash \langle \{i(O_o,...,O_n)\texttt{;}\},(\beta',P',e',fun') :: cs,\zeta\rangle \\ \rightarrow_B \langle prg(i)(\texttt{bb0}),(\_,\_,e,prg(i)) :: (\beta',P',e',fun') :: cs,\zeta'\rangle \end{array}}$$

$$[Return]\ \frac{\zeta' = \zeta[e'(P) \mapsto e(\_0)]}{\begin{array}{c} prg \vdash \langle \{\texttt{Return;}\},(\beta,P,e,fun) :: (\beta',P',e',fun') :: cs,\zeta\rangle \\ \rightarrow_B \langle fun'(\beta),(\beta',P',e',fun') :: cs,\zeta'\rangle \end{array}}$$