ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Automatically Generating Memory Safety Certificates for Rust Programs

Master's Thesis

Pascal Huber

August 21, 2022

Advisors: Prof. Dr. Peter Müller, Vytautas Astrauskas

Department of Computer Science, ETH Zürich

Abstract

The Rust programming language attempts to be both resource-efficient and memory-safe, two properties which hardly any other programming language manages to combine. The borrow checker of the compiler ensures memory safety but is itself a nontrivial program and does not provide any formal proof of correctness. RustBelt tackles this issue by defining a logic to create correctness proofs for Rust programs, verified by the Coq proof assistant, which has a much smaller codebase. However, creating proofs that certify memory safety using RustBelt is a difficult task and has to be done manually for every Rust program. To solve this problem, we implemented the relevant RustBelt rules in the verification language Viper for some of the most important language features of Rust, using the information provided by the experimental borrow checker, Polonius. While further work is necessary to support more real-world applications, we have shown that automatically generating memory safety certificates for Rust programs by modelling the RustBelt rules in Viper is a feasible and sensible approach.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Vytautas Astrauskas, who guided me through this project. His feedback and explanations were invaluable, and I always left our meetings feeling positive, knowing how to tackle the current and upcoming challenges.

I would also like to thank the Programming Methodology Group. The outstanding lectures held by members of this group motivated me to do my thesis in this fascinating field. The advice and feedback I received throughout my thesis were genuinely helpful. In particular, I am indebted to Prof. Peter Müller for allowing me to write my thesis in this group.

Lastly, I would like to thank my mother for supporting me throughout my studies.

Contents

Contents **iii**

1 Introduction **1**
1.1 Contributions . . . 2
1.2 Outline . . . 3
1.3 Architecture Overview . . . 3
1.3.1 Viper . . . 3
1.3.2 Prusti . . . 3
1.3.3 RustBelt . . . 4

2 Ownership **5**
2.1 Rust's Ownership Model . . . 5
2.2 Ownership Encoding . . . 5
2.2.1 RustBelt . . . 6
2.2.2 Viper . . . 6

3 Borrowing **9**
3.1 Borrowing in Rust . . . 9
3.2 Lifetimes . . . 10
3.3 Original Lifetimes . . . 10
3.3.1 RustBelt . . . 11
3.3.2 Viper . . . 11
3.4 Derived Lifetimes . . . 12
3.4.1 RustBelt . . . 12
3.4.2 Viper . . . 13
3.5 Mutable Borrows . . . 15
3.5.1 RustBelt . . . 15
3.5.2 Viper . . . 16
3.6 Immutable Borrows . . . 18
3.6.1 Rustbelt . . . 18

3.6.2 Viper 19

4 Branching and Reborrowing 21
4.1 Branching in Rust 21
4.2 MIR and Lifetimes 22
4.3 Reborrowing 23
4.3.1 RustBelt 23
4.3.2 Viper 23
4.4 Merging Branches 24
4.5 Borrow Shortening 25
4.5.1 RustBelt 25
4.5.2 Viper 25

5 Function calls 27
5.1 Lifetime Elision 27
5.2 Lifetime Encoding 28

6 Loops 31
6.1 Loop invariants 31
6.2 Viper 31

7 Evaluation 33
7.1 Implementation 33
7.2 Features 34
7.2.1 Borrowing Nested Structs 34
7.2.2 Unconstrained Lifetimes in Loops 35
7.2.3 Generics 35
7.3 Coverage 36

8 Conclusion 39
8.1 Future Work 39
8.1.1 Missing Features 39
8.1.2 RustBelt Limitations 40
8.1.3 Fine-grained Tests 40
8.1.4 Coq instead of Z3 41
8.2 Related Work 41

A Appendix 43
A.1 Code Listings 43

Bibliography 47

Chapter 1

Introduction

To this day, C and C++ are still the most widely used systems programming languages. While they are an excellent choice regarding resource usage, their memory accesses are not well defined, making them prone to memory-related bugs. An investigation by Microsoft revealed that approximately 70% of their security vulnerabilities are caused by memory safety issues [25].

Rust is a relatively new programming language which attempts to fix this problem by being both memory-efficient and memory-safe [11]. It does so by introducing an ownership-based type system and rules how values may be referenced and borrowed.

One example of a safety property we have in Rust is the absence of dangling references. The function shown in Listing 1.1 takes a reference `x` to an integer and returns its value. An equivalent function written in C or C++ could cause a program to crash when called with a dangling reference. The Rust compiler, on the other hand, makes sure this function is only ever called with references to allocated and initialised integers which makes this function memory-safe.

```rust
fn deref(x: &i32) -> i32 {
  *x
}
```

Listing 1.1: Rust ensures the absence of dangling references and therefore parameter `x` of the `deref` function is guaranteed to point to an allocated and initialised integer. In other languages such as C or C++ such a function can be called with a reference to an integer which has already been freed from memory.

The borrow checker of the Rust compiler ensures memory safety. It is, however, itself a nontrivial program with a large codebase [28]. From this, we have to conclude that relying on the compiler to verify memory safety is not optimal, especially when working in safety-critical domains. A solution would be to have memory safety certificates which can be checked by a much smaller program. A program logic and a tool to automatically generate and verify proofs written in that logic could be used for this task.

The RustBelt [3] project contains such a program logic. The certificates for programs and the rules of the RustBelt logic themselves can be verified with the Coq proof assistant [1], which has a small codebase [28]. However, creating the proofs still has to be done manually. Integrating the RustBelt program logic into a tool which can automate the process could be a solution.

Viper [15] is an intermediate verification language which comes with tools to automate the process of creating and verifying properties of computer programs. It is already capable of checking different properties of Rust programs through its front-end Prusti [9], including functional properties or the absence of panics.

The main goal of this thesis is to find out if it is feasible to automatically generate memory safety certificates by modelling the RustBelt rules in Viper.

It should be noted that memory safety in Rust programs comes at a price. With its type system, Rust can be overly restrictive at times – in short, it does not allow mutating and aliasing simultaneously. This makes it impossible to, for example, implement a mutex as multiple threads need to be able to write to a shared object. Even implementing a doubly linked list with cyclic references poses a problem. For this reason, Rust provides a second language called *Unsafe Rust* which does not enforce memory safety and can be used inside (safe) Rust [12, 28]. We do not consider unsafe Rust in this thesis.

1.1 Contributions

With this thesis, we make the following contributions.

- Viper Encoding of the RustBelt rules concerning lifetimes for ownership, borrows, branching, function calls and loops. We show how the lifetime rules in RustBelt can be encoded in the intermediate verification language Viper.
- Implementation of the encoding in Prusti. We automate the process of creating the Viper encoding by implementing the rules in the Viper front-end Prusti using the lifetime information from Polonius.
- A description of language features which are difficult to encode. We show Rust programs which are not easy to encode and argue how they could be supported in Prusti.

- An evaluation of the implemented features on real-world applications.
- A discussion of future and related work. We discuss what work can be done to further increase both the number of supported language constructs and the confidence in the results.

1.2 Outline

We give a brief overview of the tools we work with in the remainder of this chapter. Different language features of Rust are introduced in Chapters 2 to 6, along with a description of how RustBelt deals with them and how we can encode the RustBelt rules in question. The Evaluation in chapter 7 describes the implementation in Prusti, provides an overview of supported and unsupported language features covered by the encoding, and shows how the encoding performs for real-world applications. We provide a conclusion in Chapter 8, including a future and related work description.

1.3 Architecture Overview

In this section, we briefly describe how Viper and its Rust front-end Prusti work and interact with the Rust compiler and which parts of RustBelt we require.

1.3.1 Viper

Using the tools surrounding the Viper intermediate verification language [15], we can generate an encoding for programs written in commonly used languages such as Python, Java or Rust and then verify if the encoded specifications hold [30]. In this project, we want to create a Viper program for every function of a given Rust Program to show that the borrows are valid. We can implement this functionality in Prusti, the Viper front-end for Rust.

The Viper code snippets in this document are largely simplified and omit some complexity. Furthermore, to make the examples more readable, we add the Rust type of functions, methods and predicates inside angle brackets. For example, for an `assign` method for `i32`, we write `assign<i32>`.

1.3.2 Prusti

Given a Rust program, Prusti generates and verifies a Viper program for each function. It relies and interacts closely with the Rust compiler by building upon the Mid-level Intermediate Representation (MIR).

The MIR is a Control-Flow Graph (CFG) consisting of basic blocks connected through edges. A basic block contains statements of our program and has the property that if one statement is executed, all must be [19]. As a consequence, a basic block can not contain branches or loops. Every basic block has one entry point, and the last statement determines which basic block, if any, is executed next.

The Rust compiler creates the MIR in multiple steps by rst obtaining an abstract syntax tree (AST) of the program before creating the High-Level intermediate Representation (HIR), which is used for type checking. The HIR is then lowered to the MIR, which is also what the borrow checker of the Rust compiler uses to check memory safety. After those checks, the compiler creates an executable binary from the MIR [32].

To check if memory operations are safe, the Rust compiler introduces so-called lifetimes. Every variable has a lifetime which begins when it is created and ends when it is destroyed [3]. To model memory safety, we require information about those lifetimes, particularly when they begin and end and how they are related to another. While the Rust compiler does not offer this information by default, the prototype borrow checker, Polonius [8], has an API in the form of Datalog facts which can be used to obtain all the required information. With it, we can query the set of lifetimes and their relations for every statement in the MIR.

1.3.3 RustBelt

RustBelt [3] provides a logic to model safe and unsafe Rust programs to check their correctness. It does this by introducing λ_{Rust}, a continuation-passing style language to formally specify various features of the Rust programming language [28]. Modelling all the RustBelt rules would be challenging as the MIR is a CFG and not written in continuation-passing style. Luckily, we are only interested in the RustBelt rules covering the Rust lifetimes, and those rules do not depend on the continuation-passing style.

Chapter 2

Ownership

In this chapter, we introduce the ownership model used by Rust, explain how RustBelt handles ownership and show how we can encode the required RustBelt rules in Viper.

2.1 Rust's Ownership Model

A core difference between Rust and most other programming languages lies in its ownership-based type system. Every resource is owned by exactly one variable, and only the owner can drop it. An ownership can be transferred to another variable or a function, as shown in the following listing, where the ownership of the `String` object is moved from `x` into the function `foo`. The Rust compiler will reject this program because we attempt to use the object after moving it. Allowing this would be dangerous because `foo` could drop the object and free the memory location, making the reference `y` dangling [24]. Note that for types that, unlike `String`, implement the `Copy` trait, the ownership can not be moved because only a copy and not the original object would be transferred. Most non-trivial data types like `String` or `Vec` do not implement the `Copy` trait.

```
let x = String::from("hello");
foo(x);
let y = &x; // error: borrow of moved value: `x`
```

2.2 Ownership Encoding

Creating an object in Rust, for example, with `let x = P{ a: 3 }`, produces multiple statements in the MIR, as shown in the following listing. The `StorageLive` statement will be translated into instructions to allocate the required space on the memory, and the assignment will initialise the object.

At this point, `x` owns the object. There will come a location in the MIR where the object is no longer needed and is dropped from memory, as indicated by the `StorageDead` statement.

```
StorageLive(x)
x = P { a: const 1_i32 }
...
StorageDead(x)
```

2.2.1 RustBelt

In RustBelt, there is a predicate $[\![\tau]\!].own(t, v)$, which represents ownership of an object of type τ. It contains a list of values v and an identifier t of the thread that owns the values. A list of values is necessary as the type may consist of multiple values, such as a struct with multiple fields. When creating a new object with the ghost statement `new`, we get ownership of it in the form of an *own* predicate instance. When deleting the object with the `delete` ghost statement, that predicate instance is removed from the program state [28].

When we are confronted with an assignment which instantiates a new object in the MIR, we want to have an *own* predicate instance for it. Similarly, when we find a `StorageDead` statement for a variable which owns a resource, we want the predicate instance to be removed. Note that because we do not model multi-threading in this thesis, we can ignore the thread identifier.

2.2.2 Viper

We can encode ownership in Viper by creating a predicate `Owned` for every data type. Because Prusti already models memory allocations, we can reuse the `MemoryBlock` predicate and the `Address` domain. Domains in Viper can be used to create additional types, including functions and axioms to define their properties [16]. The `MemoryBlock` predicate represents permission to raw, untyped memory and can be identified by an `Address`. The `Owned<T>` predicate for type `T` only has to hold an address and give us (full) access to the memory locations of the object through the `MemoryBlock<T>` predicate.

An example of an `Owned` predicate for an `i32` integer is shown in the following listing. The Viper construct `acc` allows us to quantify permissions where `acc(x)` denotes full permission on predicate `x`. Full permission gives us the ability to modify the predicate instance. `acc(x, q)` indicates some fraction `q` amount of permission on `x` which gives us read-only access if `none < q` and `q < write`.

```
predicate Owned<i32>(address: Address){
  acc(MemoryBlock<i32>(address))
}
```

We can model ownership for types containing multiple values by encoding a conjunction of `Owned` predicates for the nested values, which gives us access to all the `MemoryBlock` predicate instances. For example, ownership for `struct S` in Listing 2.1 would be encoded as shown in Listing 2.2.

```
struct S {
    a: i32,
    b: i32,
}
```

Listing 2.1: A struct with name `S` containing two integer fields `x` and `y`.

```
predicate Owned<S>(address: Address) {
  acc(Owned<i32>(
    field_address<S>("a", address))) &&
  acc(Owned<i32>(
    field_address<S>("b", address)))
}
```

Listing 2.2: The ownership of the struct can be modelled with a conjunction of ownerships of all its fields. The method `field_address<S>(n, a)` returns the address of the value of the field with name `n` of the struct `S` at address `a`.

To encode the assignment, `let x = P{ a: 3 }`, we translate each of the MIR statements. For the `StorageLive(x)` statement, we first create a new address. Since we do not know what address the variable will have at runtime, we create a fresh, unconstrained address with `var x_addr: Address`. We then use this address to encode the storage allocation by inhaling a memory block with `inhale acc(MemoryBlock<P>(x_addr))`. The Viper statement `inhale x` adds the permissions of `x` to the program state and assumes that the constraints in `x` hold [16]. As there are no constraints in our inhale statement, it will simply add permission to the memory block to the program state.

The next statement in the MIR is the assignment which takes care of initialisation. For our struct example, the assignment `x = P { a: const 3_i32 }` can be encoded in Viper by adding and calling an `assign` method as defined in Listing 2.3 on the following page for the struct `P`. To call the `assign` method, we need full permission for the `MemoryBlock` predicate instance because it has a precondition for it (the `requires` statement). We say the method *consumes* the predicate instance because there is no postcondition (`ensures` statement), which gives us the permission back. However, the method gives us ownership as an `Owned` predicate instance. It also makes sure we have the knowledge of the field value of `P` in the program state, encoded with the `value_of` postcondition. Note that Prusti encodes values differently. The encoding with `value_of`, however, is sufficient for the examples in this thesis as we only need to know that our object is initialised correctly and do not care about the exact values.

At last, there will be a `StorageDead(x)` statement indicating that the object owned by `x` can be removed from memory. We can encode this with `exhale acc(MemoryBlock<P>(x_addr))`, which deletes the predicate instance from the program state. `exhale x` checks if the constraints and permissions in `x` by asserting them, and if the assertion was successful, removes the permissions in `x` [16]. The complete encoding for this example is shown in Listings A.1 and A.2 in the Appendix.

```
method assign<P>(
  address: Address
  value_a: Integer,
) requires acc(MemoryBlock<P>(address)))
  ensures acc(Owned<P>(address))
  ensures value_of<i32>(field_address<S>("a", address))
          == value_a
```

Listing 2.3: `assign<P>` takes a `MemoryBlock<P>` predicate instance and in return gives us ownership in form of an `Owned<P>` predicate instance. Additionally, it has a postconditions to add knowledge about the field value to the program state.

Chapter 3

Borrowing

In this chapter, we first introduce the Rust concepts of borrowing and lifetimes before we explain how different kinds of lifetimes are modelled in RustBelt and in what way we can implement them in Viper. Using the knowledge about ownership and lifetimes, we then show how both mutable and immutable borrows can be encoded.

3.1 Borrowing in Rust

Moving around resources is rather expensive. Instead of passing the objects by value and moving their ownership, we can temporarily borrow them by creating references. So far, this sounds exactly like references as seen in many other programming languages. Rust, however, enforces the correct use of references through its type system. In C and C++, ensuring the safety of the references lies in the responsibility of the programmer. Other languages, such as Java or Go, use a garbage collector for this task. Furthermore, we can only *mutably borrow* (i.e. borrow with write permissions) a resource once at a time. Before we can create a second mutable borrow of an object, the first one must have ended [22, 23].

In the following example, we can see that function `bar` mutably borrows `x`. The borrow ends when function `bar` returns. Because we have not moved the ownership of the `String` object, we can still use it at the end. Also, note that to create a mutable borrow, the variable that owns the object must be declared as mutable.

```rust
let mut x = String::from("hello");
bar(&mut x);
println!("{}", x);
```

3.2 Lifetimes

To check if all borrows are safe, the Rust compiler introduces so-called lifetimes. Every variable has a lifetime which begins when it is created and ends when it is destroyed [6]. The following code snippet shows an example where the borrow checker would complain. We create two mutable borrows `b1` and `b2` of the same object. The lifetime of `b1` begins before the lifetime of `b2` as it is created first. Furthermore, `b1` must live longer than `b2` as it is used on the last line. Its lifetime, therefore, ends after the lifetime of `b2`. Because two mutable borrows point to the same object and their lifetimes overlap, the borrow checker will reject this program.

```
let mut x = 1;
let b1 = &mut x;
let b2 = &mut x;
let _ = *b1;
```

Based on an unpublished project description of this thesis, we distinguish between the following three types of lifetimes.

- *Opaque lifetimes* are lifetimes which are passed to the function. This includes the static lifetime, the lifetime of the function itself and the lifetimes of the arguments the function is called with.
- *Original lifetimes* directly indicate a borrow location. For example, when creating a reference `y` of a value `x` with the statement `y = &'b x`, `'b` is an original lifetime.
- *Derived lifetimes* are lifetimes which are not opaque and do not directly indicate a borrowed location. In the assignment above, `y` will have its own lifetime `'a`, derived from the original lifetime `'b`.

In the following sections, we present how original and derived lifetimes are related to the concepts in Polonius and RustBelt and how we can automatically create lifetime proofs in Viper. The opaque lifetimes will be explained when introducing function calls in Chapter 5.

3.3 Original Lifetimes

Original lifetimes appear when borrowing an object. For example, the statement `let _ = &mut s` introduces a new original lifetime. In the MIR, there will also be a statement which introduces the borrow, and the Polonius facts for that location will contain that lifetime. Eventually, there will come a point in the program where the borrow ends and that lifetime is no longer needed. In that case, the lifetime will also not be in the Polonius facts anymore.

3.3.1 RustBelt

RustBelt uses so-called *lifetime tokens*, which witness a lifetime being alive. In contrast, a *dead lifetime token* is a witness that a lifetime is not alive anymore. One way of creating and ending lifetimes in RustBelt is to use the ghost instructions `newlft` and `endlft` following the RustBelt rules F-NEWLFT and F-ENDLFT. `newlft` adds a new lifetime to the variable context. Consequently, we have access to a lifetime token of this lifetime. `endlft` marks a lifetime as dead and turns the lifetime token into a dead lifetime token [28].

Those two ghost statements are precisely what we need for the original lifetimes. When we reach a statement in the MIR where the Polonius facts contain an original lifetime that was not present in the previous statement, we can create it with `newlft`. Similarly, when confronted with the first statement in the MIR for which the Polonius facts do not contain the lifetime anymore, we can use `endlft` to mark it as dead.

3.3.2 Viper

To model the lifetimes in Viper, we can use a custom domain `Lifetime` such that we can create variables of type `Lifetime`. The two abstract (bodyless) predicates `LifetimeToken` and `DeadLifetimeToken`, each holding a lifetime, are used to distinguish between alive lifetimes and dead lifetimes. The resulting Viper code will look as follows.

```
domain Lifetime {}
predicate LifetimeToken(lft: Lifetime)
predicate DeadLifetimeToken(lft: Lifetime)
```

We can implement the two ghost instructions `newlft` and `endlft` as abstract methods where `newlft` gives us full access to `LifetimeToken(lft)` of a new lifetime `lft`. The `endlft` method consumes that `LifetimeToken(lft)` and, in return, gives us full access to `DeadLifetimeToken(lft)`. The methods as shown in the following listing.

```
method newlft() returns (lft: Lifetime)
  ensures acc(LifetimeToken(lft))

method endlft(lft: Lifetime)
  requires acc(LifetimeToken(lft))
  ensures acc(DeadLifetimeToken(lft))
```

For a borrow statement such as `let _ = &mut s` where `s` is some variable owning a resource, we can see in the Polonius facts that a new original lifetime appears at that location. Let us say that lifetime has the name `bw0`. Before we encode the borrow assignment itself, we create that lifetime by first adding a new variable with `var bw0: Lifetime` and then initialising it with `bw0 := newlft()`, which gives us full permission on the `LifetimeToken` of the new lifetime. Once we reach a location in the MIR where the borrow is no longer needed, the lifetime will also not be present in the Polonius facts anymore. Given that we have not lost any permission amount of the lifetime token, we can end the lifetime by calling `endlft(bw0)`.

3.4 Derived Lifetimes

As with original lifetimes, we can use Polonius to find out where we need to introduce new derived lifetimes and where they end. For example, the Rust assignment `let x = &mut s` will have a corresponding assignment in the MIR, which introduces a new derived lifetime. The new relation can be found in the Polonius facts of that location. When the derived lifetime ends, it will also be removed from the Polonius facts. Note that it is also possible for a lifetime to be derived from multiple original lifetimes and those relations can even change from statement to statement.

3.4.1 RustBelt

RustBelt introduces the concepts of *lifetime inclusion* and *lifetime intersection*, which are essential to understanding how derived lifetimes are handled.

Lifetime κ being included in lifetime κ', written $\kappa \sqsubseteq \kappa'$, means that κ' outlives κ. Consequently, if we are given a lifetime token for κ, we can also obtain a lifetime token for κ'. Moreover, if we have a dead lifetime token for κ', we should be able to obtain a dead lifetime token for κ [28].

The intersection $\kappa \sqcap \kappa'$ of two lifetimes is a lifetime which ends when either κ or κ' ends [28]. It follows that the intersection of two lifetimes is included in either one of the lifetimes, written $(\kappa \sqcap \kappa') \sqsubseteq \kappa$ and $(\kappa \sqcap \kappa') \sqsubseteq \kappa'$ [28].

When a new derived lifetime becomes alive, we can not simply use `newlft` to create it as this would not model the constraint that the new lifetime must be included in some other lifetime. Instead, RustBelt has a set of rules, including Lftl-Tok-Inter and Lftl-Begin, which define how we can use lifetime inclusion to create new derived lifetimes. The rules describe that creating the new lifetime takes some permission amount away from the lifetime token of the lifetime it is derived from. Only when the newly derived lifetime ends, is the permission returned [28].

Lifetimes can also be derived from multiple original lifetimes. We can create a new lifetime κ which is derived from κ' and κ''. In this case we get the relations $\kappa \sqsubseteq \kappa'$ and $\kappa \sqsubseteq \kappa''$ which can be simplified to $\kappa \sqsubseteq (\kappa' \sqcap \kappa'')$.

Whenever a new derived lifetime is introduced or removed by Polonius, we can use those rules and concepts to encode them. When an already existing lifetime changes, we can first remove it and add it again with the updated relation.

3.4.2 Viper

For the encoding of the concepts of lifetime intersection and lifetime inclusion, we can extend our lifetime domain by adding the abstract domain functions `included` and `intersect`. Axioms can be used to define their properties and how they are connected to each other as shown in Listing 3.1 where two axioms are used to define the semantics of lifetime inclusion and intersection.

```
domain Lifetime {

  // returns true iff lft_a is included in lft_b
  function included(lft_a: Lifetime, lft_b: Lifetime): Bool

  // returns the intersection of lft_a and lft_b
  function intersect(
    lft_a: Lifetime,
    lft_b: Lifetime
  ): Lifetime

  // Every lifetime is included in itself
  axiom included_in_itself {
    (forall lft: Lifetime :: included(lft, lft))
  }

  // The intersection of lft_a and lft_b is included in
  // both lft_a and lft_b
  axiom included_intersect {
    (forall lft_a: Lifetime, lft_b: Lifetime ::
      included(intersect(lft_a, lft_b), lft_a) &&
      included(intersect(lft_a, lft_b), lft_b))
  }
}
```

Listing 3.1: Using axioms and domain functions, we can encode the semantics of lifetime inclusion and intersection.

We can define the method `lft_tok_sep_take` as shown in Listing 3.2 on the following page to create a new lifetime which is derived from other lifetimes. The method takes the two lifetimes `lft_a` and `lft_b` as arguments and consumes some non-zero permission amount q on their lifetime tokens. In return, it gives us a new lifetime `lft` and q amount of permission on its lifetime token. In this case, we have defined the method to derive from two other lifetimes; but we can define similar methods for any number of lifetimes. Because the method consumes permission of the lifetime tokens, we can not end the lifetimes `lft_a` or `lft_b` (using the `endlft` method) before ending the newly created derived lifetime. The last postcondition of the method ensures that the newly created lifetime is included in both `lft_a` and `lft_b`.

```
method lft_tok_sep_take(
  lft_a: Lifetime,
  lft_b: Lifetime,
  q: Perm
) returns (lft: Lifetime)
  requires none < q
  requires acc(LifetimeToken(lft_a), q)
  requires acc(LifetimeToken(lft_b), q)
  ensures acc(LifetimeToken(lft), q)
  ensures lft == intersect(lft_a, lft_b)
```

Listing 3.2: The `lft_tok_sep_take` can be used to create a new derived lifetime.

We also encode the inverse method `lft_tok_sep_return` to end a derived lifetime. It does the opposite to `lft_tok_sep_take` by consuming the lifetime token of the derived lifetime and giving us back `q` amount of permission on the lifetime tokens `lft_a` and `lft_b`. Note that when an original lifetime is ending, we have to call `lft_tok_sep_return` for all lifetimes derived from it before we can call `endlft`. Otherwise, we would not have full permission on its lifetime token. The encoding of the method is shown in Listing A.3 in the Appendix.

`lft_tok_sep_return` does not give us a dead lifetime token for the derived lifetime. Just as defined in RustBelt, if lifetime `a` is included in lifetime `b` for which we have a dead lifetime token, we can also get a dead lifetime token for `a`. We encode this with the method `dead_inclusion` as shown in Listing 3.3 on the next page. After calling the method, we have dead lifetime tokens for both the original and derived lifetime.

We can put everything together and create all the lifetimes needed to encode a borrow `let x = &mut s` as shown in Listing 3.4 on the facing page. First, we create the original lifetime `bw0` and then derive the lifetime `lft_x` from it. `bb0[n]` is the first location in the MIR, where the Polonius facts do not contain `bw0` or `lft_x`. Before we can end `bw0` by calling `endlft`, we need to get back `1/3` amount of permission to its lifetime token using `lft_tok_sep_return`. We then have a dead lifetime token for `bw0` which allows us to also obtain a dead lifetime token for `lft_x` with `dead_inclusion`.

```
method dead_inclusion(
  lft_derived: Lifetime,
  lft_original: Lifetime
) requires acc(DeadLifetimeToken(lft_original))
  requires included(lft_derived, lft_original)
  ensures acc(DeadLifetimeToken(lft_original))
  ensures acc(DeadLifetimeToken(lft_derived))
```

Listing 3.3: Using method `dead_inclusion`, we can acquire a dead lifetime token for a derived lifetime.

We have not yet discussed how we derive the permission amount q. In the previous example, a fraction of `1/3` is small enough that we do not lose all permission on the lifetime token of `bw0`. We could therefore derive more lifetimes from it. When creating the encoding, we can specify a globally used variable holding the permission amount q. A fraction $\frac{1}{n}$ where n is the total number of lifetimes in the program gives us a small enough permission amount that we never run out of permission when deriving lifetimes.

```
// bb0[4] x = &'bw0 mut s
// lft_x ⊑ bw0
var bw0: Lifetime
bw0 := newlft()
lft_x := lft_tok_sep_take(bw0, 1/3)
// borrow encoding
// ...

// bb0[n]
lft_tok_sep_return(lft_x, bw0, 1/3)
endlft(bw0)
dead_inclusion(lft_1, bw0)
```

Listing 3.4: Creating and ending an original lifetime `bw0` and a derived lifetime `lft_x`. At the end of this snippet, we have dead lifetime tokens for both lifetimes.

3.5 Mutable Borrows

Now that we can encode ownership and lifetimes, we have all the means to model borrowing.

3.5.1 RustBelt

The RustBelt rule C-Borrow specifies that to mutably borrow an object, we require ownership of it and, in return, we get a mutable reference with a lifetime. It also guarantees that we can regain ownership of the object when that lifetime has ended [28].

While C-Borrow describes the creation and destruction of borrows, it does not let us access the resource behind it and creating a mutable borrow usually comes with dereferencing and modifying the resource at some point. The rule LftL-bor-acc allows opening a mutable borrow to obtain access to the borrowed object. The rule states that we can temporarily give up a borrow and its lifetime token to gain access to the resource. A so-called update can later be used to restore the borrow and regain the lifetime token. If this update is not used, the lifetime token and the borrow will be lost [28].

3.5.2 Viper

We can encode borrowing in Viper using a method `borrow` as shown in Listing 3.5 on the next page for our `struct S` from the previous chapter. The method consumes ownership of our struct instance and the memory block for the borrow itself. In return it gives us ownership of a mutable borrow through the predicate `Owned<&mut S>`. Ownership of a mutable borrow consists of a memory block for the reference in addition to the abstract predicate `MutRef<S>` which we will use later to access the resource. It also ensures that we have some fractional permission for the lifetime token of the (original) lifetime of the borrow. The last postcondition is a magic wand that encodes the guarantee that when the borrow has ended, we can regain ownership of the resource. This magic wand can be applied when we have a dead lifetime token for the lifetime of the borrow. Note that magic wands themselves can consume the predicates on the left-hand side. In order not to lose the dead lifetime token, it is also present on the right-hand side.

We can now encode the borrow `let x = &mut s` where `s` owns an instance of the struct. The Viper encoding is shown in Listing 3.6 on page 18. The encoding of the first two MIR statements `bb0[0]` and `bb0[1]` give us ownership of the resource. `bb0[2]` then allocates the reference before we create the lifetimes and perform the borrow in `bb0[3]`. In `bb0[n]` the borrow ends and we therefore end the lifetimes and exhale the memory block. Lastly, the magic wand can be applied to regain the ownership of the struct.

It remains to show how we model the behaviour of the LftL-bor-acc rule to access the resource of a mutable borrow. We can encode this rule with the two methods `open_mut_ref` and `close_mut_ref` to temporarily gain ownership of the value behind the reference and to return the ownership again. This may sound like a violation as we have established that a mutable borrow should not give us ownership abilities. However, as every `open_mut_ref` comes with a `close_mut_ref` which requires ownership to be called, this is not an issue. Dropping the object would take away the `MemoryBlock` predicate of the object, and we would lose the `Owned` predicate

```
method borrow<S>(
  target_address: Address,
  operand_address: Address,
  operand_lifetime: Lifetime,
  q: Perm
) requires none < q && q < write
  requires acc(MemoryBlock<&mut S>(target_address))
  requires acc(Owned<S>(operand_address))
  requires acc(LifetimeToken(operand_lifetime), q)
  ensures operand_address == points_to_addr(target_address)
  ensures acc(LifetimeToken(operand_lifetime), q)
  ensures acc(Owned<&mut S>(
              operand_lifetime,
              target_address,
              operand_address))
  ensures acc(DeadLifetimeToken(operand_lifetime))
          --* acc(Owned<S>(operand_address)) &&
              acc(DeadLifetimeToken(operand_lifetime))

predicate Owned<&mut S>(
  address: Address,
  lifetime: Lifetime
) {
  acc(MemoryBlock<&mut S>(address)) &&
  acc(MutRef<S>(lifetime, address))
}
```

Listing 3.5: The method `borrow` consumes ownership of the resource and, in return, gives us ownership of the borrow.

instance. Consequently, the encoding would not verify because the preconditions of `close_mut_ref` are not met.

The `open_mut_ref` method consumes both the `MutRef` predicate instance, which the `borrow` method call added to the program state, and some fractional permission of the lifetime token of the borrow. In return, it gives us access to an `Owned` predicate instance of the resource behind the reference together with a `CloseMutRef` predicate instance. This predicate holds all the information about the consumed `MutRef` predicate instance. Most importantly, it contains the address of the reference, the lifetime of the borrow and the permission amount a as those values are not in the `Owned` predicate. The `close_mut_ref` method requires this information to restore the `MutRef` predicate instance and give us back the predicate instances which `open_mut_ref` consumed. The two methods are shown in Listing A.4 in the Appendix.

```
// bb0[0] StorageLive(s)
var s_address: Address
inhale acc(MemoryBlock<S>(s_address)))

// bb0[1] s = S { x: const 1_i32, y: const 2_i32 }
assign<S>(s_address, Integer(1), Integer(2))

// bb0[2] StorageLive(x)
var x_address: Address
inhale acc(MemoryBlock<&mut S>(x_address))

// bb0[3] x = &'bw0 mut s
bw0 := newlft()
lft_x := lft_tok_sep_take(bw0, q)
borrow<S>(x_address, s_address, bw0, q)

// ...

// bb0[n]: StorageDead(x)
lft_tok_sep_return(lft_x, bw0, q)
endlft(bw0)
dead_inclusion(lft_x, bw0)
exhale acc(MemoryBlock<&mut S>(x_address))
apply acc(DeadLifetimeToken(bw0))
      --* acc(Owned<S>(s_address)) &&
          acc(DeadLifetimeToken(bw0)) &&
```

Listing 3.6: Viper encoding of a mutable borrow.

3.6 Immutable Borrows

Mutable borrows are quite restrictive as it is a common pattern in programming to have multiple references to the same object. To allow this, Rust introduces *immutable borrows*, which, as the name suggests, allow reading but not modifying the borrowed objects. In contrast to mutable borrows, it possible to immutably borrow an object multiple times. In other words, the lifetimes of immutable borrows of the same resource may overlap [22, 23].

3.6.1 Rustbelt

In RustBelt, immutable borrows are produced by creating a mutable borrow and then turning it into a *fractured borrow* as described by the rule LftL-bor-fracture. The rule LftL-fract-acc then can be used to gain some fractional permission on the resource. The key idea is that we can always split a fractional permission into two smaller fractional permissions, which allows us to have multiple fractured borrows of the same object, all with some non-zero amount of permission on the resource [28]

3.6.2 Viper

In Viper, we can follow the same idea. We create fractured borrows by introducing the method `bor_fracture` which takes a mutable borrow in form of a `MutRef` predicate instance and turns it into a `FracRef` predicate instance with the same attributes as shown in Listing 3.7. The method is called right after the creation of the mutable borrow, i.e. right after the call of the `borrow` method.

```
method bor_fracture<T>(
  lifetime: Lifetime,
  reference_address: Address,
  value_address: Address,
  q: Perm
) requires rd > none
  requires acc(LifetimeToken(lifetime), q)
  requires acc(MutRef<T>(
                 lifetime,
                 reference_address,
                 value_address))
  ensures acc(LifetimeToken(lifetime), q)
  ensures acc(FracRef<T>(
                lifetime,
                reference_address,
                value_address))
```

Listing 3.7: The method `bor_fracture` turns a mutable borrow into an immutable one.

This allows us to distinguish between mutable and immutable references and we can implement the rules for immutable borrows by using the `FracRef` predicate. If we want to access an object behind an immutable borrow, we can use the method `frac_bor_atomic_acc` which consumes a non-zero permission amount of the lifetime token and the `FracRef` instance. In return, it gives us some fractional permission on the ownership of the object, which allows us to read but not change it. When calling the method, we are also given a guarantee in the form of a magic wand which allows us to return the fractional permission on the ownership and reobtain the consumed permission of the lifetime token.

So far, this allows us to have exactly one immutable borrow as our `FracRef` instance is consumed by `frac_bor_atomic_acc`. We can implement the splitting of permissions in RustBelt by allowing `FracRef` instances to be duplicated by adding a method `duplicate_frac_ref`. It requires one `FracRef` predicate instance to be called and has two postconditions which each give us an identical `FracRef` predicate instance. The encoding of this method can be found in Listing A.5 in the Appendix.

```
method frac_bor_atomic_acc<T>(
  lifetime: Lifetime,
  reference_address: Address
  value_address: Address
  q: Perm
) returns (frac_q: Perm)
  requires q > none
  requires acc(LifetimeToken(lifetime), q)
  requires acc(FracRef<T>(
                 lifetime,
                 reference_address,
                 value_address))
  ensures none < frac_q && frac_q < write
  ensures acc(Owned<T>(object), frac_q)
  ensures acc(Owned<T>(object), frac_q)
          --* acc(LifetimeToken(lifetime), q)
```

Listing 3.8: The method `frac_bor_atomic_acc` gives us some fractional permission of the ownership. The resource can be returned to regain the consumed permission of the lifetime token.

Chapter 4

Branching and Reborrowing

In this chapter, we explain the challenges which arise with branching and how they are dealt with in RustBelt and Viper. The example we introduce for this will also show us the concept of reborrowing which comes with an additional challenge for the lifetime encoding.

4.1 Branching in Rust

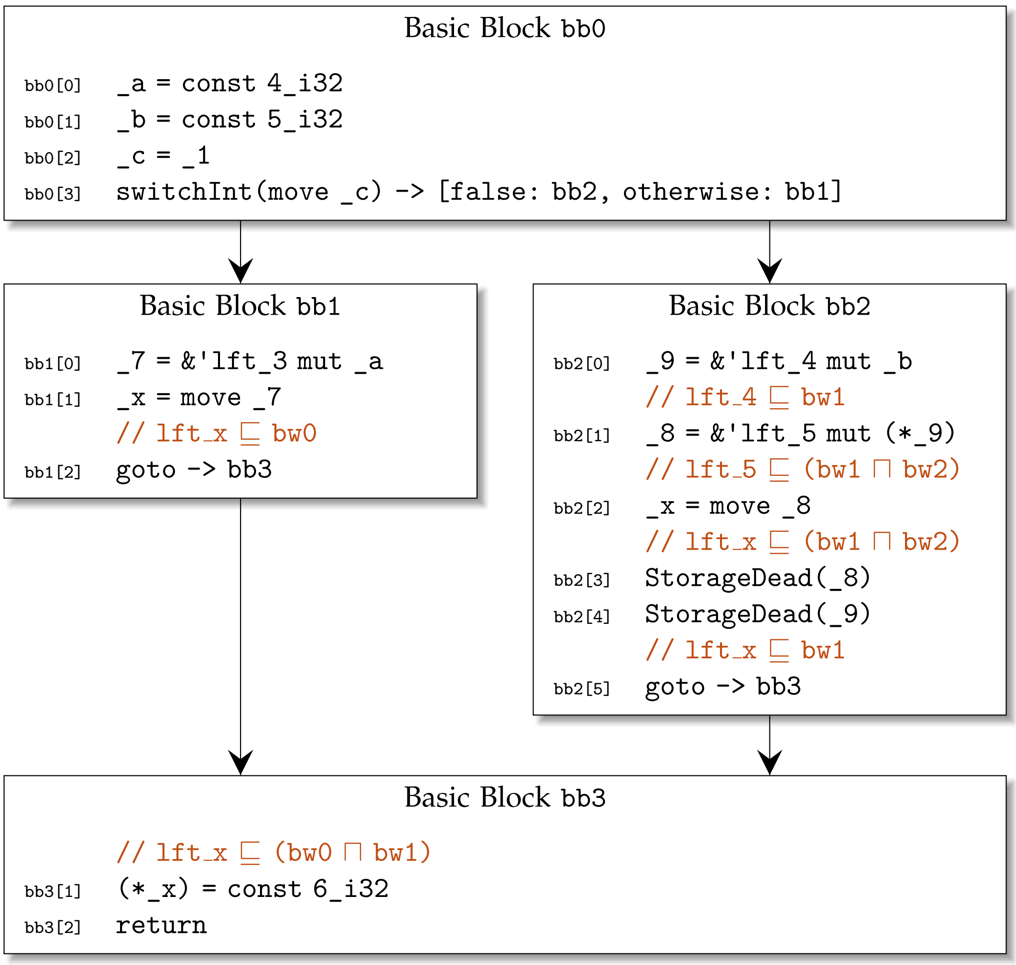
We consider the function `foo` defined in Listing 4.1. Depending on parameter `c`, the variable `x` mutably borrows either `a` or `b`. The assignment after the branching on line 10 ensures that `x` and the object it points to can not be dropped inside the branch bodies. This will bring forth a synchronization issue regarding the lifetimes and we will have to make sure in our encoding that we have a consistent and well-defined state after the if-else statement regardless of which branch was taken.

```
1  fn foo(c: bool){
2    let mut a: i32 = 4;
3    let mut b: i32 = 5;
4    let x;
5    if c {
6      x = &mut a;
7    } else {
8      x = &mut b;
9    }
10   *x = 6;
11 }
```

Listing 4.1: Depending on the value of the boolean `c`, `x` mutably borrows either `a` or `b`.

4.2 MIR and Lifetimes

To get a better understanding of the problem we will first inspect the MIR of this program shown in Figure 4.1. It contains four basic blocks: `bb0` is the block which initialises our integers and then checks the condition of the if-statement, `bb1` and `bb2` are the bodies of the two branches and `bb3` is the basic block after the if-else-statement. Also note that we have removed most of the `StorageLive` and `StorageDead` statements as they do not play an important role here. The lifetime relations from Polonius are denoted in the comments for some of the statements.



Basic Block bb0

```
bb0[0]   _a = const 4_i32
bb0[1]   _b = const 5_i32
bb0[2]   _c = _1
bb0[3]   switchInt(move _c) -> [false: bb2, otherwise: bb1]
```

Basic Block bb1

```
bb1[0]   _7 = &'lft_3 mut _a
bb1[1]   _x = move _7
         // lft_x ⊑ bw0
bb1[2]   goto -> bb3
```

Basic Block bb2

```
bb2[0]   _9 = &'lft_4 mut _b
         // lft_4 ⊑ bw1
bb2[1]   _8 = &'lft_5 mut (*_9)
         // lft_5 ⊑ (bw1 ⊓ bw2)
bb2[2]   _x = move _8
         // lft_x ⊑ (bw1 ⊓ bw2)
bb2[3]   StorageDead(_8)
bb2[4]   StorageDead(_9)
         // lft_x ⊑ bw1
bb2[5]   goto -> bb3
```

Basic Block bb3

```
         // lft_x ⊑ (bw0 ⊓ bw1)
bb3[1]   (*_x) = const 6_i32
bb3[2]   return
```

Figure 4.1: The MIR of Listing 4.1 on the previous page, consisting of four basic blocks. Some of the statements are annotated with their lifetime relations from Polonius.

4.3 Reborrowing

One might naïvely assume that the two basic blocks `bb1` and `bb2` have the same number and kinds of statements, but looking at the MIR, we can see that `bb2` contains an additional assignment. Instead of performing the borrow and moving it to the correct location as in `bb1`, it dereferences the borrow only to reference it again and store the result in a different variable before moving it to `_x`. We call this a *reborrow*, and in fact, the Rust compiler does it all the time. Every time it sees an assignment to a reference, it adds a reborrow statement. Because we did not defined the type of `x` when we declared it, the first branch will define it, and as a consequence, we do not have a reborrow in `bb1`. In `bb2`, the compiler knows that `x` is a reference and therefore adds the reborrow.

This becomes interesting when inspecting the lifetime relations of the variables given by Polonius as shown in the comments between the lines. We can see that `lft_5` and as a consequence `lft_x` are derived not only from `bw1` but also from `bw2`, a new original lifetime only used for the reborrow. And after the `StorageDead(_9)` statement, it is again derived from `bw1` only. In other words, the lifetime is shortened for the reborrow and extended again after the reborrow has ended.

4.3.1 RustBelt

RustBelt provides a rule C-Reborrow which, given a borrow, creates a new borrow with a new lifetime which is included in the lifetime of the original borrow [28].

To extend the lifetime after the reborrow has ended, the RustBelt rule F-Equalize could be used [26]. However, this rule was not added to RustBelt for reborrowing but to support programs which the Rust developers are intending to support in the future. Currently, those programs are still rejected by the compiler. While F-Equalize is sound, it is incomptable with other projects such as GhostCell [34]. For this reason, we decided to completely ignore the lifetimes introduced for the reborrow only.

4.3.2 Viper

We can perform a reborrow with a slightly modified version of the `borrow` method. The main difference is that it does not need ownership to be called but instead requires a `MutRef` predicate instance. An additional parameter to specify the lifetime of the original borrow is necessary and a precondition ensures that the reborrow lifetime (`operand_lft`) is included in the lifetime of the original borrow. An example method to reborrow an integer value is shown in Listing 4.2 on the following page.

```
method reborrow<&mut i32>(
  target_addr: Address,
  operand_addr: Address,
  operand_lft: Lifetime,
  borrow_lft: Lifetime,
  q: Perm
) requires none < q && q < write
  requires acc(MemoryBlock<i32>(target_addr))
  requires acc(MutRef<i32>(borrow_lft, operand_addr))
  requires acc(LifetimeToken(operand_lft), q)
  requires acc(LifetimeToken(borrow_lft), q)
  requires included(operand_lft, borrow_lft)
  ensures operand_address == points_to_addr(target_addr)
  ensures acc(LifetimeToken(borrow_lft), q)
  ensures acc(LifetimeToken(operand_lft), q)
  ensures acc(Owned<&mut i32>(target_addr, operand_lft))
  ensures acc(DeadLifetimeToken(operand_lft))
          --* acc(MutRef<i32>(borrow_lft, operand_addr)) &&
              acc(DeadLifetimeToken(operand_lft))
```

Listing 4.2: The `reborrow` method creates a new borrow of a resource and guarantees that we can restore the original borrow when the lifetime of the reborrow has ended.

4.4 Merging Branches

The reborrowing does not affect our original problem of merging branches since the lifetime extension and shortening happens within the basic block of the else-branch. What we require for `bb3` is a well-defined state of the lifetimes. We will first look at the state at the end of `bb1` and `bb2`. In particular, we are interested in the lifetime `lft_x` of the borrow held by `_x`. We know from Polonius that at the beginning of the last basic block, the two original lifetimes, `bw0` and `bw1`, must exist. However, in `bb1`, the lifetime `bw1` does not exist and `lft_x` is only derived from `bw0`.

To fix this issue, we have to create the missing lifetimes at the end of the block, make sure `lft_x` is derived from both lifetimes `bw0` and `bw1` and also ensure the borrow `_x` contains the new version of `lft_x`. Similarly, we have to create `bw0` in `bb2` to shorten the borrow.

4.5 Borrow Shortening

4.5.1 RustBelt

RustBelt has a rule Lftl-Bor-Shorten which tells us that when lifetime κ' is included in κ, and we have a mutable reference with lifetime κ, we can shorten the mutable reference to lifetime κ' [28].

4.5.2 Viper

In order to shorten the borrows, we need to also create the missing lifetimes using the `newlft` method. The shortening of the borrow can be done by creating and using a method `bor_shorten` as defined in Listing 4.3. The method takes the old lifetime `lft_old` and the new (shorter) lifetime `lft_new` and consumes the existing `MutRef` predicate instance and, in return, gives us a new `MutRef` instance with the new lifetime.

```
method bor_shorten<&mut i32>(
  lft_new: Lifetime,
  lft_old: Lifetime,
  address: Address,
  q: Perm,
) requires none < q && q < write
  requires included(lft_new, lft_old)
  requires acc(LifetimeToken(lft_new), q)
  requires acc(LifetimeToken(lft_old), q)
  requires acc(MutRef<i32>(lft_old, address))
  ensures acc(LifetimeToken(lft_new), q)
  ensures acc(LifetimeToken(lft_old), q)
  ensures acc(MutRef<i32>(lft_new, address))
```

Listing 4.3: `bor_shorten` exchanges the lifetime of a mutable borrow with a shorter one.

In order to call this function, we need both the old and new lifetime. We thus need to create a backup copy of our lifetime before we shorten it. We also return the old derived lifetime before we call `lft_tok_sep_take` such that we do not loose any permission of `bw0`. At the end of `bb1`, we will have to add the code shown in following listing.

```
bw1 := newlft()
var lft_old: Lifetime := lft_x
lft_tok_sep_return(lft_x, bw0, q)
lft_x = lft_tok_sep_take(bw0, bw1)
bor_shorten(lft_x, lft_old, x_address, q)
```

Chapter 5

Function calls

In this chapter, we first show how function calls and lifetime relations can be encoded in Rust before we describe what has to be done in the Viper encoding for both the caller and callee.

5.1 Lifetime Elision

So far we have assumed that all lifetimes are given to us by the compiler, and the programmer does not have to specify them. In some cases the compiler can not figure out what the lifetimes of the variables are and the developer has to explicitly list them. Such an example can be found in Listing 5.1, where the Rust compiler requires the programmer to provide lifetime names of the returned reference and consequently also the lifetimes of the parameters.

```rust
fn first<'a, 'b>(x: &'a i32, y: &'b i32) -> &'a i32 {
    x
}
```

Listing 5.1: Function `first` takes two integer references and also returns an integer reference. We can also see that the two borrows in the parameters have distinct lifetimes with the names `'a` and `'b` and the returned reference has the same lifetime as parameter `x`. This is an example of a function which the Rust compiler would reject had we not explicitly specified the lifetimes.

We have introduced RustBelt's lifetime inclusion in Section 3.4 on page 12. Such lifetime relations can also be specified directly in Rust. In Listing 5.2 on the following page, we can see that function `foo` has two lifetimes `'a` and `'b` and in order to call this function we require `'b` to outlive `'a`, written `'b : 'a`. This means that `'a` may not start before `'b` and must end together with `'b` at the latest. In RustBelt, we would write `'a` $\sqsubseteq$ `'b`. This knowledge allows us to perform the assignment `x = y`, which is only allowed because we know that the value of `y` will not be destroyed before lifetime `'a` ends.

```
fn foo<'a, 'b: 'a>(mut x: &'a i32, y: &'b i32){
    x = y;
}
```

Listing 5.2: Function `foo` can only be called if lifetime `'b` outlives lifetime `'a`.

5.2 Lifetime Encoding

The lifetime specifications of a function are given by its opaque lifetimes, which exist on entry of the function. Polonius provides a list of opaque lifetimes for every function. Because every Rust function is encoded in its own Viper method, we can add some fractional permission to the lifetime tokens as preconditions of the called method. To avoid losing the permissions on the caller side, we also add them as postconditions.

Polonius not only gives us the list of opaque lifetimes but also specifies their relations using lifetime inclusion and intersection. To ensure those relations hold, we also add preconditions for them using the `included` and `intersect` functions. For function `foo` in Listing 5.2, the encoded method would look as shown in Listing 5.3 on the facing page.

Because Polonius maintains different sets of lifetimes for the two functions, the remaining challenge is to find the matching lifetimes on the caller side. Every function has at least two opaque lifetimes, the static lifetime and a lifetime of the function itself. The static lifetime is trivial to find as it is the lifetime which outlives all other lifetimes. As a consequence, the static lifetime of the called function is equivalent to the static lifetime of the caller function, and it can easily be identified using the information from Polonius. It is the lifetime which is not included in any other lifetime. The lifetime of the called function does not exist on the caller side, but it can be constructed as it is derived from all other opaque lifetimes. It is also the shortest opaque lifetime and all parameters outlive the function call.

The remaining challenge is to find the lifetimes of the parameters. For the function in Listing 5.2, we need to find the lifetimes on the caller side corresponding to `lft_a` and `lft_b`. From the MIR statement for the function call, `foo::<'lft_7, 'lft_8>(...)`, we know that `lft_7` and `lft_8` are the lifetimes we are looking for. However, we do not know which lifetime corresponds to `lft_a` and which to `lft_b`. The only option we have is to look at the subset relations of Polonius. Indeed, we can see there that `lft_8` outlives `lft_7`, and thus `lft_a` must correspond to `lft_7` and `lft_b` to `lft_8`. This is only possible because we have only two lifetimes in this example. If we had four parameter lifetimes with the relations `'b: 'a` and `'d: 'c`, there would be no easy way to find out which lifetime corresponds to which.

```
method foo(
  lft_foo: Lifetime,
  lft_a: Lifetime,
  lft_b: Lifetime,
  lft_static: Lifetime,
  ...
)
  // opaque lifetime tokens
  requires acc(LifetimeToken(lft_foo), q)
  requires acc(LifetimeToken(lft_a), q)
  requires acc(LifetimeToken(lft_b), q)
  requires acc(LifetimeToken(lft_static), q)
  // opaque lifetime relations
  requires included(lft_foo, intersect(lft_a, lft_b, lft_static)
  requires included(lft_a, intersect(lft_b, lft_static))
  requires included(lft_b, lft_static)
  // return opaque lifetime tokens
  ensures acc(LifetimeToken(lft_foo), q)
  ensures acc(LifetimeToken(lft_a), q)
  ensures acc(LifetimeToken(lft_b), q)
  ensures acc(LifetimeToken(lft_static), q)
{
  // encoding of function foo
}
```

Listing 5.3: The Rust function `foo` is called from another function. Permissions for the lifetime tokens for the opaque lifetimes are added as preconditions and postconditions of method `foo`. The relations between the lifetimes are also added as preconditions to ensure they hold.

As a workaround for this issue, we manually encode the preconditions and postconditions with `inhale`, `exhale`, `assert` and `assume` statements. In Viper, `assert` and `assume` are similar to `exhale` and `inhale`, but do not remove or add permissions [16]. Manually encoding the conditions gives us more flexibility, mainly because we can remove the lifetimes from the parameters. At the beginning of the method `foo`, we create variables for the lifetimes, inhale access to their tokens and assume the relations between them. At the end of the method, we exhale access to the lifetime tokens again to ensure they have not been ended inside the function.

On the caller side, we can do the opposite. We exhale the lifetime tokens before the function call and inhale them again when the function returns. Furthermore, we can easily assert the relations for the static and function lifetimes as we can identify or construct them on the caller side. We need to do something different to assert the relations of the parameter lifetimes. We can assert all relations we can find which contain the parameter lifetimes. If the information provided by Polonius is correct and complete, this will contain the required relations. Consequently, we put more trust in Polonius and do not assert *precisely* what we assume in the function encoding, which is not the idea of preconditions.

Chapter 6

Loops

This chapter introduces verifications of loops in Viper and describes how we add support for lifetimes.

6.1 Loop invariants

Loops are a particularly challenging language construct in program verification because it is not possible to statically determine the number of iterations. To verify the specifications for a program with a loop, we need an induction-based approach where we introduce an induction hypothesis which must hold before the loop begins and is preserved in every execution of the loop body. We can add the hypothesis as a *loop invariant* in Viper. It can then be verified that the specifications of the invariant hold before and after every iteration.

For example, consider the Viper method `id` shown in Listing 6.1, where we want to prove that the result `res` is equal to the (positive) input parameter `n`. Without a loop invariant, the only thing we know after the loop is `res >= n` because the loop condition does not hold anymore. However, with the loop invariant `res <= n`, Viper knows that both `res >= n` and `res <= n` hold and therefore, the postcondition `res == n` holds. It is also easy for Viper to verify that if the invariant holds before one execution of the loop body, it must also hold after the body as we know that `res < n` holds at the beginning of the body and we only increase `res` by 1.

6.2 Viper

Recall that we use the MIR as a basis for the Viper encoding, and the MIR does not contain loop statements. There are, however, cycles of basic blocks in the CFG and the loop invariants are also modelled in Prusti.

```
method id(n:Int)
  returns(res:Int)
  requires n >= 0
  ensures res == n
{
  res := 0
  while(res < n)
    invariant res <= n
  {
    res := res + 1
  }
}
```

Listing 6.1: The method `id` contains a loop with a loop invariant without which the postcondition could not be verified.

Because we use the Polonius information to encode the lifetimes for every statement in every basic block and also prepare the lifetimes for subsequent blocks at the end of every basic block, the task of adding lifetime specifications is not particularly challenging. We can add the lifetime knowledge consisting of the relations between the lifetimes to the loop invariant to check if they actually hold. For example, the loop invariant of the program in the following listing is encoded at the beginning of basic block `bb5`, where Polonius gives us the four derived lifetimes $\texttt{lft\_0} \sqsubseteq \texttt{bw5}$, $\texttt{lft\_1} \sqsubseteq (\texttt{bw5} \sqcap \texttt{bw6})$, $\texttt{lft\_21} \sqsubseteq (\texttt{bw0} \sqcap \texttt{bw2})$ and $\texttt{lft\_22} \sqsubseteq \texttt{bw0}$. We therefore add assertions for those four relations to verify that they hold before we inhale them again for the next iteration. Note that the `IteratorWrapper` has to be implemented such that the `next` function, which the loop requires, can be marked as trusted so that the iterator itself does not get verified. The implementation can be found in Listing A.6 in the Appendix.

```
let mut ve = Vec::new();
let mut v: IteratorWrapper<i32> =
  IteratorWrapper::new(&mut ve);
for x in &mut v {}
```

Chapter 7

Evaluation

In this chapter, we first give a brief overview of how the lifetime encoding was added to Prusti before describing which features are supported and the remaining problems. Finally, we show how the implementation performs for real-world Rust programs.

7.1 Implementation

The features described in chapters 2 to 6 have been implemented in Prusti. The functionality was added to a new version of the *core proof*, which is still under active development. The core proof verifies the properties checked by the Rust compiler [18]. Some features (not only concerning the lifetime encoding) are still missing and Prusti aborts when confronted with one of them.

To generate a Viper program, the MIR is first translated into the Viper Intermediate Representation (VIR), containing internal statements holding all the information required to create the Viper program. This happens in multiple stages where the result of the previous stage is processed block by block and statement by statement. To add the required function calls and methods, we first query the lifetime relations using the Polonius facts. With this information, statements can be added to the encoding. In particular, we prepend Prusti statements for every MIR statement to create lifetime backups (required for `bor_shorten`), extend derived lifetimes (`lft_tok_sep_return`), end lifetimes (`endlft`, `dead_inclusion`), create lifetimes (`lft_tok_sep_take`, `newlft`) and shorten the borrows (`bor_shorten`). Similarly, the statements required for function calls, loop invariants and assignments (e.g. `borrow`, `bor_fracture`, `open_mut_ref`) can be added to this representation.

Finally, the encoding for all the methods, functions, domains, and predicates is added such that Prusti can generate the Viper program.

7.2 Features

We show an overview of the implementation status of the different language features in Table 7.1.

Supported	Partially Supported	Not Supported
Branching	Borrowing	Slices
Function calls	Dereferencing	Generics
Reborrowing	Loops	

Table 7.1: Overview of Rust language features for which lifetime encoding is supported.

While branching, function calls, and reborrowing are fully operational, some features require more work to be fully supported. Due to time constraints, they were not implemented in this thesis. In particular, dereferencing is not yet implemented for all types and does not yet work for struct fields or enums, for example. Additionally, the missing dereferencing features make it hard to test the features with more complex data types. Slices are not covered yet but adding support for them should not be a big problem as they are not much different to borrows of arrays in terms of the lifetime encoding.

The following subsections describe three remaining issues: a problem concerning borrowing objects with nested lifetimes, loops with more complex data types, and the yet unsupported generics.

7.2.1 Borrowing Nested Structs

Listing 7.1 shows a Rust program where we use the same lifetime `'a` for both fields in the two nested structs `OuterStruct` and `NestedStruct`.

```rust
struct NestedStruct<'a> {
    x: &'a mut i32,
}
struct OuterStruct<'a> {
    x: &'a mut NestedStruct<'a>,
}
fn main () {
    let mut n = 4;
    let mut i = NestedStruct { x: &mut n };
    let mut o = OuterStruct { x: &mut i };
}
```

Listing 7.1: Two nested structs which use the same lifetime for their reference type fields.

The MIR of the main function contains the statements shown in the following listing. The interesting lifetime relations are shown as comments. The first two assignments are a normal borrow and reborrow, and we ignore the newly introduced lifetime bw3 to avoid the shortening and extension of lft_7 as described in Section 4.3 on page 23. It becomes interesting when we compare the two lifetimes of the fields in the two structs. In Rust, we only have one lifetime, 'a. Polonius gives us two lifetimes: lft_16 for the lifetime of the field in NestedStruct and lft_8 for the field in OuterStruct. We can see that lft_8 is shorter than lft_16 as it is derived from both bw0 and bw2. While this should not be problematic as bw2 is the lifetime of the original borrow, this example will require some changes in Prusti. Currently, it does not verify as it tries to use the Owned predicate with the unshortened lifetime.

```
// lft_6 ⊑ bw2
borrow_i = &'lft_6 mut i

// lft_16 ⊑ bw0
reborrow_i = &'lft_7 mut (*borrow_i)

// lft_8 ⊑ (bw0 ⊓ bw2 ⊓ bw3)
x = OuterStruct::<'lft_8> { x: move reborrow_i }
```

While further investigation is necessary for this issue, we strongly believe it is fixable. Also note that this shortening does not happen when we do not use the same lifetime in the nested struct, i.e., if we define the outer struct as OuterStruct<'a, 'b>{ x: &'a mut NestedStruct<'b>}.

7.2.2 Unconstrained Lifetimes in Loops

Iterating over more complex data types turns out to be more complicated. In the example shown in Listing 7.2 on the next page, we iterate over a vector of structs where our struct X contains a field with lifetime 'a. This example is more challenging than the loop with the integers because there exist lifetimes for which Polonius gives us no subset relation but are required for the encoding. No other language feature we have encoded so far uses such lifetimes. A solution would be to replace the problematic lifetimes with the static lifetime. Due to time constraints, only a prototype was created to check that the solution works.

7.2.3 Generics

Generics make the lifetime encoding more complex, in particular for function calls. The function fn f<T>(p: T){}, for example, takes an argument of the generic type T, which may contain an arbitrary number of lifetimes. In

```rust
struct X<'a>{
  a: &'a i32,
}
let mut ve = Vec::new();
let mut v: IteratorWrapper<X> =
  IteratorWrapper::new(&mut ve);
for x in &mut v {}
```

Listing 7.2: Rust program which iterates over an empty vector of structs.

order to prove the correctness of this function, we would have to show that it is correct for any type `T`. Unfortunately, we can not quantify over predicates in Viper as we can in RustBelt. Prusti currently uses abstract predicates to verify functions with generics. The idea is that if the function verifies with an abstract predicate, it will also verify for any concrete predicate on the caller side. However, if we add lifetimes to the generic, it becomes more challenging. An example of such a function would be `fn g<T: 'a>`, which states that all lifetimes of `T` must outlive lifetime `'a` [31]. Prusti does not yet support this.

7.3 Coverage

Test functions for the different features have been implemented to verify that the encoding works. However, we are most interested in how many real-world applications we cover with our lifetime encoding. Crater [2] is a tool with which we can test this by running Prusti on the 500 most used crates. In a first crater run, it was found that 7.2% of all crates were successfully encoded and verified. This is exactly the same percentage as the current version of the core proof without the RustBelt encoding achieved. It should be noted that the sets of crates, which were successfully verified of the two versions, are not identical. There are crates that verify with the current version but not with the new core proof and vice versa.

The current version of Prusti has a flag `PRUSTI_SKIP_SUPPORTED_FEATURES` that, if enabled, ignores unsupported features. The new version of the core proof does not have this flag and will always panic for programs with unsupported features. To have more comparable results, this flag was disabled. When the flag is enabled, the current Prusti version can handle more than three times as many crates without crashing.

The Prusti and Viper versions and the relevant settings are shown in Table 7.2. The list of the crates can be found in the Prusti GitHub repository [10].

Prusti Commit Hash	`a70ce38939eefde053e1070fb0fdb6e08a5f4f86`
Viper Version	`v-2022-07-01-0736`

Environment Variables	`PRUSTI_CHECK_PANICS=false`
	`PRUSTI_CHECK_OVERFLOWS=false`
	`PRUSTI_ASSERT_TIMEOUT=60000`
	`PRUSTI_SKIP_UNSUPPORTED_FEATURES=false`

Table 7.2: Crater Evaluation setup

Chapter 8

Conclusion

In this thesis, we attempted to investigate the feasibility of creating memory safety certificates for Rust programs by modelling the RustBelt rules in Viper.

We have shown how we can encode Rust's ownership model and borrowing rules in Viper. Furthermore, we have described how support for branching, function calls and loops can be added to the encoding.

To automate the process of creating the proofs, we have extended the Viper front-end Prusti to create the necessary encoding and verify that given Rust programs are indeed memory-safe. Polonius, Rust's nightly borrow-checker, provides the necessary information about the lifetimes needed to perform this task. Some language features came with surprising challenges, such as the lifetime shortening when performing a reborrow or finding the matching lifetimes for function calls.

Even though encoding different language features is not always easy and still requires more work to support a larger portion of real-world applications, we are confident that creating memory safety certificates by modelling the RustBelt rules in Viper is a feasible and sensible approach.

8.1 Future Work

8.1.1 Missing Features

To cover more real-world applications, the yet unsupported features have to be implemented. This includes missing features of the new core proof and features of the lifetime encoding as described in Section 7.2 on page 34. There also exist language constructs which RustBelt does not yet support. Trying to implement them would also be an exciting project.

8.1.2 RustBelt Limitations

An advantage of Prusti is that implementing and testing new rules may be easier than extending RustBelt. Yet unsupported features could be implemented in Prusti and added to RustBelt if they work.

Two-Phase Borrows

One of the known limitations of RustBelt are *Two-Phase Borrows* which require both a mutable and immutable reference to the same object with overlapping lifetimes [26]. Rust only allows this in some special cases, and RustBelt does not support them. One example is shown in Listing 8.1, where the vector `v` is needed to both read its length and insert the value. Listing 8.2 shows how the compiler expands the program. We can see that `temp1` is a special kind of borrow of `v` which allows using `v` even though the borrow has not yet ended.

```rust
fn main(){
    let mut v = Vec::new();
    v.push(v.len());
}
```

Listing 8.1: Two-phase Borrow

```rust
fn main(){
    let mut v = Vec::new();
    let temp1 = &two_phase v;
    let temp2 = v.len();
    Vec::push(temp1, temp2);
}
```

Listing 8.2: Expanded Two-Phase Borrow

Overlapping Shared References

Another limitation of RustBelt are overlapping shared references. An example is given in Listing 8.3, where `t1` immutably borrows only one of the fields of the struct `t` and `t2` immutably borrows the entire object. Note that because both variables are used in the `println!` function call, their lifetimes overlap.

8.1.3 Fine-grained Tests

Once the new core proof supports the missing features and does not throw as many `unimplemented` errors, a more fine-grained evaluation can be per-

```rust
fn main (){
  let t = T { x: 1, y: 2};
  let t1 = &t.x;
  let t2 = &t;
  println!("{} {}" t1, t2);
}
```

Listing 8.3: Overlapping Shared References

formed to see how the encoding works for real-world applications. Crater can be used for this.

8.1.4 Coq instead of Z3

In this thesis, we encoded RustBelt rules in Viper. It is also thinkable to generate the proofs for the Coq proof assistant [1]. An advantage of Coq is that it has an exceptionally small codebase compared to Z3 [17], which Viper uses. Furthermore, a bug in the Viper encoding, such as a forgotten precondition, could cause an unsafe program to verify successfully. Coq would complain if a rule is not sound as the RustBelt rules themselves are verified too.

8.2 Related Work

There is a large variety of tools for different programming languages to check correctness properties and identify bugs. VeriFast, for example, can prove properties of annotated C and Java programs [14]. Different Viper front-ends exist for programs written in Python, Java, and OpenCL [15]. Infer [4] is a static analyzer which can detect potential bugs in Java, C, C++ and Objective-C, including null pointer dereferences and memory leaks [5].

Even though Rust is a comparably new language, there are some tools and projects which formalize and verify different aspects of Rust programs. MIRAI is a static analyzer working on the MIR which can find panics and verify other correctness properties or Rust programs [7]. As we have seen in this thesis, the RustBelt project [3] provides an incredibly expressive program logic with which we can create correctness and memory safety proofs, but it lacks automation. Stacked Borrows [13] defines rules to describe aliasing, which can be used to detect undefined behaviour. In contrast to this thesis, it performs a dynamic analysis and works in both safe and unsafe Rust. [27].

Rust is different from most other languages because its type system is built in a way that the compiler can check memory safety. RustHorn [29] and Creusot [21] both make use of this type system and can be used to prove functional specifications of Rust programs. They do not, however, prove memory safety and rely on the borrow checker for this task. To our knowledge, there currently is no other verifier which automatically proves memory safety for Rust programs.

Related in a different way is Voila, a proof outline checker for the TaDA [20] logic. Just as we have determined that Viper is an excellent fit to encode parts of the RustBelt logic, it was found that the TaDA proofs can be encoded in Viper [33].

Appendix A

Appendix

A.1 Code Listings

```
struct P {
  a: i32
}
let x = P{
  a: 1,
};
```

Listing A.1: Declaration and initialisation of the struct P

```
// bb0[0] StorageLive(x)
var x_address: Address
inhale acc(MemoryBlock<P>(x_address))

// bb0[1] x = P { a: const 3_i32 }
assign<P>(x_address, Integer(1))

// bb0[n] StorageDead(x)
exhale acc(MemoryBlock<P>(x_address))
```

Listing A.2: The encoding of the relevant statements MIR of Listing A.1 to create, initialise and delete the struct owned by x.

```
method lft_tok_sep_return(
  lft_derived: Lifetime,
  lft_orig_a: Lifetime,
  lft_orig_b: Lifetime,
  q: Perm
) requires none < q
  requires acc(LifetimeToken(lft_derived), q)
  requires lft == intersect(lft_orig_a, lft_orig_b)
  ensures acc(LifetimeToken(lft_orig_a), q)
  ensures acc(LifetimeToken(lft_orig_b), q)
```

Listing A.3: The method `lft_tok-sep_return` consumes a derived lifetime token and in return gives back access to the (original) lifetime tokens it was derived from.

```
method open_mut_ref<&mut S>(
  ref_address: Address,
  val_address: Address,
  lifetime: Lifetime,
  q: Perm
) requires none < q
  requires acc(LifetimeToken(lifetime), q)
  requires acc(MutRef<&mut S>(
    lifetime,
    ref_address,
    val_address))
  ensures acc(Owned<S>(val_address))
  ensures acc(CloseMutRef<&mut S>(
    ref_address,
    val_address,
    lifetime,
    q))

method close_mut_ref<&mut S>(
  ref_address: Address,
  val_address: Address,
  lifetime: Lifetime,
  q: Perm
) requires none < q
  requires acc(CloseMutRef<&mut S>(
    ref_address,
    val_address,
    lifetime,
    q))
  requires acc(Owned<S>(val_address))
  ensures acc(LifetimeToken(lifetime), lifetime_perm)
  ensures acc(MutRef<&mut S>(
    lifetime,
    ref_address,
    val_address))
```

Listing A.4: `open_mut_ref` consumes some fractional permission on the lifetime token of the borrow and gives us both ownership in form of the `Owned` predicate and an instance of `CloseMutRef`. `close_mut_ref` restores the borrow by doing the opposite. It consumes `CloseMutRef` and the ownership and in return gives us back the `MutRef` predicate and the permission on the lifetime token.

```
method duplicate_frac_ref<T>(
  lifetime: Lifetime,
  reference_address: Address
  value_address: Address
) requires acc(FracRef<T>(
               lifetime,
               reference_address,
               value_address))
  ensures acc(FracRef<T>(
              lifetime,
              reference_address,
              value_address))
  ensures acc(FracRef<T>(
              lifetime,
              reference_address,
              value_address))
```

Listing A.5: `duplicate_frac_ref` consumes one `FracRef` predicate instance and in return provides two identical predicate instances.

```
extern crate prusti_contracts;
use prusti_contracts::*;

#[trusted]
struct IteratorWrapper<'a, T>{
    iter_mut: std::slice::IterMut<'a, T>,
}
impl<'a, T> IteratorWrapper<'a, T> {
    #[trusted]
    fn new(x: &'a mut Vec<T>) -> Self {
        IteratorWrapper {
            iter_mut: x.iter_mut(),
        }
    }
}
impl<'a, T> Iterator for IteratorWrapper<'a, T> {
    type Item = &'a mut T;
    #[trusted]
    fn next(&mut self) -> Option<Self::Item> {
        self.iter_mut.next()
    }
}

fn main() {
    let mut ve = Vec::new();
    let mut v: IteratorWrapper<i32> =
      IteratorWrapper::new(&mut ve);
    for x in &mut v {}
}
```

Listing A.6: To test the encoding of loops, we have to create a wrapper around the iterator such that we can mark the required functions as trusted.

Bibliography

[1] The coq proof assistant. https://coq.inria.fr/. Accessed: 2022-08-09.

[2] Crater: a tool to run experiments across parts of the Rust ecosystem. https://github.com/rust-lang/crater. Accessed: 2022-06-14.

[3] ERC Project "RustBelt". https://plv.mpi-sws.org/rustbelt. Accessed: 2022-02-24.

[4] Infer: A tool to detect bugs in java and c/c++/objective-c code before it ships. https://fbinfer.com/docs/about-Infer. Accessed: 2022-08-07.

[5] Infer: About infer. https://fbinfer.com/docs/about-Infer. Accessed: 2022-08-17.

[6] Lifetimes - rust by example. https://doc.rust-lang.org/rust-by-example/scope/lifetime.html. Accessed: 2022-08-18.

[7] Mirai: an abstract interpreter for the rust compiler's mid-level intermediate representation. https://github.com/facebookexperimental/MIRAI. Accessed: 2022-08-17.

[8] Polonius. https://github.com/rust-lang/polonius. Accessed: 2022-08-10.

[9] Prusti. https://www.pm.inf.ethz.ch/research/prusti.html. Accessed: 2022-08-11.

[10] Prusti - github. https://github.com/viperproject/prusti-dev. Accessed: 2022-08-15.

[11] Rust Programming Language. https://www.rust-lang.org/. Accessed: 2022-03-03.

[12] The rust programming language - unsafe rust. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html. Accessed: 2022-08-10.

[13] Stacked borrows. https://plv.mpi-sws.org/rustbelt/stacked-borrows/. Accessed: 2022-08-17.

[14] Verifast. https://github.com/verifast/verifast. Accessed: 2022-08-17.

[15] Viper. https://www.pm.inf.ethz.ch/research/viper.html. Accessed: 2022-08-10.

[16] Viper tutorial. https://viper.ethz.ch/tutorial. Accessed: 2022-08-12.

[17] The z3 theorem prover. https://github.com/Z3Prover/z3. Accessed: 2022-08-09.

[18] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers. The prusti project: Formal verification for rust (invited). In *NASA Formal Methods (14th International Symposium)*, pages 88–108. Springer, 2022.

[19] John Cocke. Global common subexpression elimination. *SIGPLAN Not.*, 5(7):2024, jul 1970.

[20] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A Logic for Time and Data Abstraction. In Richard E. Jones, editor, *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP'14)*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer, July 2014.

[21] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: a Foundry for the Deductive Verication of Rust Programs. In *International Conference on Formal Engineering Methods - ICFEM*, Lecture Notes in Computer Science, Madrid, Spain, October 2022. Springer Verlag.

[22] Rust By Example. Borrowing. https://doc.rust-lang.org/rust-by-example/scope/borrow.html. Accessed: 2022-08-18.

[23] Rust By Example. Mutablity. https://doc.rust-lang.org/rust-by-example/scope/borrow/mut.html. Accessed: 2022-08-18.

[24] Rust By Example. Ownership and moves. https://doc.rust-lang.org/rust-by-example/scope/move.html. Accessed: 2022-08-18.

[25] Sebastian Fernandez. We need a safer systems programming language. https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/. Accessed: 2022-03-03.

[26] Ralf Jung. Understanding and evolving the rust programming language, 2020.

[27] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: An aliasing model for rust. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[28] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):134, 2018.

[29] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. Rusthorn: Chc-based verification for rust programs. *ACM Trans. Program. Lang. Syst.*, 43(4), oct 2021.

[30] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[31] The Rust Reference. Trait and lifetime bounds. https://doc.rust-lang.org/reference/trait-bounds.html. Accessed: 2022-08-09.

[32] Guide to Rustc Development. Overview of the compiler. https://rustc-dev-guide.rust-lang.org/overview.html. Accessed: 2022-08-10.

[33] F. A. Wolf, M. Schwerhoff, and P. Müller. Concise outlines for a complex logic: A proof outline checker for TaDA. In M. Huisman, C. S. Păsăreanu, and N. Zhan, editors, *Formal Methods (FM)*, volume 13047 of *LNCS*, pages 407–426. Springer, 2021.

[34] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. Ghostcell: Separating permissions from data in rust. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Automatically Generating Memory Safety Certificates for Rust Programs

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Huber	Pascal

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Dübendorf, 21.08.2022	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.