ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Towards Verifying Real-World Rust Programs

Bachelor Thesis

Jonas Maier

Wednesday 5th October, 2022

Advisors: Prof. Dr. Peter Müller, Vytautas Astrauskas

Department of Computer Science, ETH Zürich

Abstract

Prusti is a verifier for the Rust programming language. Its goal is to prove that crashes cannot happen in a verified Rust program and that the program adheres to its intended functionality. So far not many big Rust codebases have been verified using Prusti, and the question remains open if Prusti is suitable to verify larger code examples.

In this thesis, we test the suitability of Prusti on real-world Rust code by trying to verify bigger pieces of actual code, that is, by translating parts of the VeriBetrFS codebase to Rust and verifying it. We identified missing features and implemented a subset of them. We added ghost code, termination proofs, and more powerful assertions to Prusti.

Acknowledgements

I would like to thank my thesis supervisor, Vytautas Astrauskas for his support throughout the work of my thesis. He always was quick to respond when I had questions or issues and helped me a great deal to structure the time I worked on this topic into sensible parts.

Further, I would like to thank Léon Othenin-Girard and Julian Steinmann for their proofreading work.

Contents

Chapter 1

Introduction

Rust [3] is a systems programming language that puts emphasis on safety, speed, and concurrency. Its strong type system prevents programming errors of other systems languages, like buffer overruns or use-after-free errors that are common in C. This greatly simplifies reasoning about program behavior, and makes program verification easier, as basic properties are already given, and don't need to be verified first, e.g. ensuring references don't alias.

Sometimes the properties guaranteed by the Rust compiler aren't satisfactory. This is where Prusti [1], a program verifier, comes in – it can be used to verify the absence of exceptions (or panics, in Rust jargon) and correctness of the implementation in regards to specifications. Since Rust's type system already gives a lot of guarantees less effort needs to be spent on proving basic properties, such as references not interfering with each other. Instead, more time can be spent on *functional* verification of programs, i.e. proving that an algorithm or a data structure adheres to its specification. Prusti utilizes the Viper [11] verification infrastructure, which is the common backend of multiple program verifiers.

So far, no larger examples have been verified with Prusti. This thesis tried to change that.

The end goal was to verify a nontrivial data structure, to assess how Prusti compares to other, existing verifiers, like for example Dafny [9]. More specifically, it makes sense to compare Rust to a specific modification of Dafny, namely Dafny with linear types [10]. The linear type system is quite similar to Rust's, so it should be possible to verify similar code with both verifiers, written in the same paradigm, and then compare properties of verification.

Intermediate goals were to implement features in Prusti that were lacking at the moment, but which are needed to verify the chosen code. As the list of still needed features turned out to be longer than anticipated, we only spent time on implementing these, without verifying bigger data structures and comparing Prusti to Dafny.

Chapter 2 concerns itself with the data structure that we tried to implement and the issues that arose. With this implementation, we quickly noticed what features were missing, and implemented a subset of them, which is explained in the remaining chapters.

First, we implemented Ghost Code which we described in chapter 3, then Ghost Variables which are covered by chapter 4. Later we implemented termination proofs which chapter 5 covers. Finally, we implemented a more powerful assertion macro, as will be elaborated upon in chapter 6.

The final conclusion is given in chapter 7.

Chapter 2

Datastructure Implementation in Rust

As mentioned in the introduction, we want to verify an already existing data structure from an existing codebase. We want to compare Prusti to an already established verifier, so we don't compare two verifiers with each other that both rapidly evolve.

We chose the verification-aware programming language Dafny [9] with linear types [10]. Linearly typed Dafny was chosen as it has a similar type system to Rust's, and so code written in one language should be somewhat straightforward to translate to the other. However, since Dafny is a verification-aware language, we expect to encounter features within Dafny that are not present in Prusti, which will need to be addressed.

For the codebase, we chose VeriBetrFS [6]. It is a verified file system implementation based on the design of BetrFS [7]. The overall project does not matter as much, it's more important that there is quite a range of code to choose from, as it has highly separated individual components and many of them.

The data structure we chose from VeriBetrFS was the Linear Mutable Map [8]. We chose this code snippet for a variety of reasons. It does not depend on a lot of other code, which means we only need to verify a comparatively small code snippet, to begin with, although larger than other verified Rust code. It is a comparatively simple implementation of a hashmap. A hashmap is interesting in the sense that we are not limited to only verifying the absence of crashes, but can also try to verify that functions that can be used in manipulating the hashmap adhere to the formal notion of a mathematical definition of a mapping.

2.1 Design of the Datastructure

In this section, we will be describing the Linear Mutable Map, as taken from VeriBetrFS. The key ideas are listed here and we describe how they can or cannot be translated to Rust to be verified with Prusti.

As the name implies, the Linear Mutable Map is a map implementation based on a linear buffer. We maintain a mapping from a set of keys to a set of values. For simplicity, keys are integer-valued (u64).

The data associated with the map is stored in the struct shown in listing 1. The buffer field stores the mapping, and the count variable stores the number of mappings currently stored. A slot in the buffer contains an `Item`. Its definition is given in listing 2. Either it contains an entry with a mapping (`Entry`), or it contains nothing (`Empty`), or there previously was an entry there but is not there anymore (`Tombstone`). Why we make the explicit distinction between empty slots and formerly occupied slots will be explained in section 2.1.2.

```
struct Map<V> {
    buffer: Vector<Item<V>>,
    count: u64,
}
```

Listing 1: Data associated with a map

```
enum Item<V> {
    Empty,
    Entry { key: u64, value: V },
    Tombstone { key: u64 },
}
```

Listing 2: Data associated with an item

2.1.1 Inserting a new mapping

Inserting a new mapping $k \to v$ is rather simple: First, we hash the integer key k to obtain a key k' that should be uniformly distributed over the length of the buffer. Then, we look at the item in slot k' of the buffer. If it is occupied by an `Entry`, we try the following slot, and so on, until we find an `Empty` slot or a `Tombstone` slot that is free.

Once found, we replace that slot with an `Entry` which stores k as key and v as value.

2.1.2 Looking up a mapping

Finding the value to a key k is done the following way: First, we again hash k to obtain k'. Then, we start at position k' in the buffer and search for an `Entry` containing the key k.

If we find one, we can return the associated value. However, if we first find a `Tombstone` to that specific key, we also know that it isn't present in the map. Additionally, if we find an `Empty` item, we know that the value to the key can not be later in the buffer, as we never skip `Empty` slots to insert a value, and `Empty` slots can not be created, as removal places a `Tombstone`.

2.1.3 Removing a mapping

Removing a mapping makes use of the lookup we described in section 2.1.2. First, we look up the position in the buffer of a key k. If there is an `Entry`, we replace it with a `Tombstone` storing k.

2.2 Initial Implementation and Hurdles

Let's take the Dafny function in listing 3 as an example to translate to Rust. Conceptually the `getEmptyWitness` function is rather simple: given a non-full map, it returns an index that is currently empty. A few things can be noticed: It requires that the argument `i` is a valid index in the range `0` to `|self.storage|`, which is annotated on line 4. Further, it requires that every index `j` with `j < i` is non-empty, as written on line 5. Also, it requires that the `count` of occupied slots is less than the storage capacity, which is annotated on line 6.

So far, everything seems to be possible to translate to Prusti. Then, we have the `decreases` annotation, which annotates termination of the function.

The postconditions are rather straightforward as well: It returns a valid index, and the entry at the index is empty (lines 8–9).

Then, in the body everything can be translated to Prusti, except one thing: On line 11 a lemma function is called. It is merely called to be able to assume its postcondition, and the lemma acts as proof of the postcondition. We could emulate this in Prusti, but the lemma function would also be called during runtime, which is not quite ideal.

We translated the Dafny code from listing 3 to the Rust code in listing 4. The pre- and post-conditions have a slightly different syntax (`[requires(...)]` and `[ensures(...)]`), as it is a different language, but the underlying principle is the same: we need to make sure that the preconditions hold when calling the function, and can assume the postcondition holds after having called the function. Within the function the opposite is true: We can

```
function {:opaque} getEmptyWitness<V>
(self: FixedSizeLinearHashMap<V>, i: uint64) : (res : uint64)
requires FixedSizeInv(self)
requires 0 <= i as int <= |self.storage|
requires forall j | 0 <= j < i :: !self.storage[j].Empty?
requires self.count as int < |self.storage|
decreases |self.storage| - i as int
ensures 0 <= res as int < |self.storage|
ensures self.storage[res].Empty?
{
  allNonEmptyImpliesCountEqStorageSize(self);

  if self.storage[i].Empty? then
    i
  else
    getEmptyWitness(self, i+1)
}

lemma allNonEmptyImpliesCountEqStorageSize<V>
(self: FixedSizeLinearHashMap<V>)
requires FixedSizeInv(self)
ensures (forall j | 0 <= j < |self.storage| ::
        !self.storage[j].Empty?)
    ==> self.count as int == |self.storage|
{ /* body omitted for brevity */ }
```

Listing 3: Example code from the VeriBetrFS project

assume the preconditions hold when entering the function, and must guarantee that the postconditions hold when exiting the function. This forms a basic contract for function calls. Further, Prusti supports quantifiers in the form `quantifier(|var: Type| boolean_expression(var))`. There are the `exists` and `forall` quantifiers, and quantified variables must be copyable types. An example of such a quantifier can be seen on line 14–16 in listing 4. Further, Prusti supports implication arrows, written as `==>`.

A few things are to notice: We can't translate the `decreases` clause to a Prusti equivalent, as Prusti does not yet support termination proofs, hence the comment on line 6. Another thing that is currently not possible in Prusti is to use quantifiers in `assert!` statements, although we used it for illustrative purposes on lines 14–16.

One thing that cannot be noticed by the short example we have given, but which is nevertheless important, is that currently references can only be used in a limited fashion. The `index` function which we defined on the

```rust
#[ensures(self.inv())]
#[requires(i < self.storage.len())]
#[requires(self.count < self.storage.len())]
#[requires(forall(|j: u64|j < i
    ==> !self.storage.index(j).is_empty()))]
//#[decreases(self.storage.len() - i)]
#[ensures(result < self.storage.len())]
#[ensures(self.storage.index(result).is_empty())]
pub fn get_empty_witness(&self, i: u64) -> u64 {
    let entry = self.storage.index(i);
    if entry.is_empty() {
        i
    } else if i + 1 == self.storage.len() {
        assert!(forall(|j: u64|
            j < i ==> !self.storage.index(j).is_empty()
        ));

        self.all_non_empty_implies_count_eq_storage_size();
        unreachable!()
    } else {
        self.get_empty_witness(i + 1)
    }
}

#[pure]
#[trusted]
#[ensures(forall(|j: u64| j < self.storage.len()
    ==> !self.storage.index(j).is_empty())
    ==> self.count == self.storage.len())]
fn all_non_empty_implies_count_eq_storage_size(&self) {}
```

Listing 4: Dafny Code translated to Rust

`Vector` returns a copied value to circumvent that problem. The underlying issue is that chaining pure functions is not fully supported.

We will discuss the observed Dafny features in more detail in the following subsections.

2.2.1 Lemma Functions

VeriBetrFS' implementation is full of lemma functions. These are pure functions that only exist to prove a nontrivial fact, usually with induction. There are two issues with Prusti that make it difficult to emulate that.

First, termination of pure functions is currently not checked. This means we could construct an induction proof that does not have a base case and is therefore not well founded and incorrect.

Second, in Dafny these lemma functions need to be called explicitly. We can already do that in Rust with Prusti, however, it would be highly inefficient. The lemma functions are only called to establish a fact, and not to use the computed result, if there even is a result that is computed. This means that if we were just to naively call the function normally in Rust code, it would execute the proof, which sometimes has exponential runtime, and hurts the runtime performance considerably.

2.2.2 Ghost Code

If we take a quick look at the VeriBetrFS implementation of the linear mutable map, we quickly notice that it heavily makes use of ghost types, ghost code, and built-in mathematical data structures.

Variables of a ghost type merely exist to aid verification but are not included during runtime of the program. This also means that all code that computes ghost values is omitted, which is commonly known as *ghost code*.

As an example, the fixed size map contains a ghost field of a map, which is a known-good model of a mapping, with the correct semantics.

With the use of that field, we can compare our implementation to the known-good field and prove that is equivalent without needing to find complicated pre- and post-conditions that encode the meaning of map operations. Instead, we can just say that the map implementation is identical to the ghost map, and a function on the map corresponds to an operation on the ghost map.

2.2.3 Generalized Assertions

In Dafny, assertions support full first-order logic, as they are omitted during runtime by default. This allows a Dafny programmer to use quantifiers in assertions to test if a property is valid over a range of values. The current Rust assertion macro does not allow us to do this, hence we would like to extend the verifier to support such assertions.

2.3 Selected Missing Features

If we look at the issues found above, we can think of a few features we could implement in Prusti to address them.

2.3.1 Ghost Code

Instead of calling lemma functions at runtime, we would like to have a mechanism to omit some code during runtime. We generalize the idea with *ghost code* in chapter 3.

2.3.2 Ghost Variables

We would like to have built-in types that represent abstract data structures. Additionally, we would like to have auxiliary variables in our code that are only used for verification but omitted at runtime. Both built-in variables and *ghost variables* will be implemented in chapter 4.

2.3.3 Termination

Since lemma functions need to terminate to soundly prove a property, we need a mechanism to prove termination. This will be implemented and explained in chapter 5.

2.3.4 Assertions

Rust does not support quantifiers natively. Hence, we cannot compute booleans that depend on quantifiers, and we cannot use quantifiers within native assertions. This creates the need for a new assertion statement that allows us to use all of Prusti's features. We implement this in chapter 6.

2.3.5 Chaining of Pure Functions

Prusti did not allow chaining of pure functions to occur with references. This can be circumvented by using only copyable types, which is however not practical for many applications. This was an issue in the old version of Prusti, and should not pose an issue in the refactored version. We will not address this issue further in this thesis.

Chapter 3

Ghost Code

One feature that is in use everywhere in betrfs' hashmap [8] is ghost code [4]. Ghost code is conceptually rather simple: It is code that is used solely for verification, and whose existence doesn't influence the rest of the program in any way. Therefore, ghost code acts just as auxiliary help to the verifier, but does not change the semantics of the regular code.

The need for ghost code is illustrated in listing 5. `fib` is the function to compute the n-th number in the Fibonacci sequence. We want to prove that exactly every third number in the sequence is even. This is a simple proof by induction, which is implemented in the function called `lemma`. So we can assume the lemma in any other function, here `foo`, we need to call it with a specific input value. Due to the purity of the function, and the fact that it returns nothing, we could leave the call out during runtime. However, we require the property the lemma function ensures - that every third number in the Fibonacci sequence is even. The lemma function is also rather expensive to run, so we would rather not have that code included during runtime. Hence, we would like to wrap it in a *ghost block* to have the verification *and* runtime benefits. To achieve that, we add a new feature, namely the `ghost!` macro, as seen on line 27 - During runtime it acts as an empty code block, but during verification, it leaves its contents in the code.

Now one might ask: is the computation of the Fibonacci number not expensive as well? In this instance, it is. However, we can also implement a faster algorithm for the Fibonacci sequence, and prove that it is equivalent to the standard definition of the sequence. That fast implementation can be seen in appendix A.1 and is omitted for brevity in this chapter.

Ghost code is currently not supported by Prusti, and in this chapter, we will focus on the necessary changes to implement it.

```rust
#[pure]
#[requires(n >= 0)]
fn fib(n: i64) -> i64 {
    if n < 2 {
        n
    } else {
        fib(n-1) + fib(n-2)
    }
}

#[pure]
#[requires(n >= 0)]
#[ensures((n % 3 == 0) == (fib(n) % 2 == 0))]
fn lemma(n: i64) {
    if n >= 3 {
        fib(n - 1);
        fib(n - 2);
        lemma(n - 1);
        lemma(n - 2);
    }
}

#[requires(x >= 0)]
fn foo(x: i64) {
    let x = 3 * x;
    let f = fib(x);
    ghost! {
        lemma(x);
    }
    assert!(f % 2 == 0);
}
```

Listing 5: Omitting expensive computations using ghost code. This example is simplified, the full version can be found in appendix A.1.

3.1 Methodology

As mentioned before, ghost code does not influence the surrounding code in any way. This is not an incidental property, but a requirement to keep the verification *sound*. An unsound verification of a particular property means that the verifier accepts a wrong proof of it, and hence the property might not hold.

Intuitively, any modification of the semantics or meaning of the program by the inclusion of ghost code potentially introduces unsoundness. If it causes different program semantics, the verifier does not verify the program that is run, and so any proof about the modified program might not apply to the program being run.

There are two ways how incorrect ghost code can change the semantics of the program.

Firstly, we can't let ghost code leak any information to regular code. This specifically means that it may not modify variables that are being used by regular code, and it may not influence control flow beyond its ghost scope.

Secondly, nontermination also changes the semantics of the program. At first glance, it might be unintuitive why that is the case, but since nontermination can change the pieces of code that are reachable, nontermination might make the entire rest of the code unreachable, in which case any assertion trivially verifies.

It would be easy to lower ghost code from Rust to Viper if Viper natively supported ghost code and checked for these soundness requirements itself. However, it does not, and so we must check these properties ourselves. After checking the properties, we can translate the ghost code to regular viper code, as we verified that it does not interfere with the rest of the program.

3.2 Implementation

We implemented `ghost!` as a Rust macro that takes as input a code block and conditionally includes the code during verification. The rest of this section concerns itself with the verification of the required properties of ghost code. Subsection 3.2.1 concerns itself with the detection of ghost blocks before being verified. Then, to guarantee soundness, we will look at the invariance of real control flow, non-interference with real variables, and termination.

3.2.1 Detection of Ghost blocks

The macro on the front end does a great job at conditionally including ghost code depending on if we compile the code or verify it. However, we are not yet done here, because in the verifier we need to find out what pieces of code are ghost code, as certain restrictions apply. On the macro front, the best we can do is insert a marker at the beginning and at the end, to specify at what point we enter ghost code, and at what point we exit ghost code. Sadly, we neither have a hierarchy of nested code blocks, nor a linear list of statements to determine what lies between the beginning and end markers.

The method we resort to is *reachability analysis*.

Reachability Analysis

The code we can work with in the verifier is Rust code compiled down to the Mid-level Intermediate Representation (MIR). MIR represents the code of a method as a set of basic blocks. A basic block is a list of uninterruptible instructions and a terminator. The terminator decides at what basic block we continue execution of the code.

This representation is called the control flow graph (CFG). We can imagine it as a graph of pieces of code, and execution can flow from node to node, a node being a basic block, and edges being determined by the terminator.

The beginning- and end-markers we inserted with the macro show up in the CFG as their own basic blocks. To determine what belongs to a specific ghost block, we need to analyze what code can be reached between the beginning node and the end node.

By the nature of this method, code that is not part of the hierarchical code block inside the ghost macro might be considered to be ghost code. This can occur if we have control flow leaving the ghost macro, for example by breaking a loop outside the ghost block.

It is not possible to detect that happening with this CFG-based representation, as a break out of a loop just looks like any ordinary jump to a different basic block in the actual ghost code.

Thus, we need to detect invalid control flow already earlier, which we'll look at in the next section.

3.2.2 Invariance of Control Flow

The easiest way to find control flow that jumps to code outside the ghost macro invocation is to check for that in the macro expansion itself.

Rust macros can be implemented by using arbitrary Rust code. As a first step, we parse the input of the `ghost!` macro into an abstract syntax tree (AST). The AST representation naturally contains a hierarchy of code blocks, as the tree recursively contains the code blocks and statements. Thus, we can find what loop a statement is contained in, and therefore analyze if the loop we break is within or outside the ghost macro.

Rust only has three explicit control flow statements:
`break`, `continue`, and `return`. Finding these in the AST of the ghost macro is rather easy. Rust's `break` and `continue` statements also support a label to annotate which loop is referred to by this statement. If no label is given, the immediate surrounding loop is taken. Listing 6 and listing 7 illustrate all ways loops can be exited in Rust. `continue` statements are analogous.

```
loop {
    loop { // break this loop
        break;
    }
}
```

Listing 6: Unlabeled break

```
'outer: loop { // break this loop
    loop {
        break 'outer;
    }
}
```

Listing 7: Labeled break

If ghost code is not included, the control flow passes straight through the location the ghost block is at. More precisely, during runtime, the execution of the ghost block is a no-op. Code flow enters the no-op at exactly one location, and exits it at exactly one location. This means that during verification, the same should be the case.

Subsequently, we need to find out if the `break` and `continue` statements refer to loops within the ghost block or loops outside the ghost block. Breaking and continuing loops within the ghost block is fine, but doing the same for loops outside would be a modification of real-code control flow, which is illegal and must be prohibited. In such a case, we issue a verification error as displayed in listing 8. The same applies for using a `continue` statement.

```
fn foo() {
    loop {
        ghost! {
            break;
        }
    }
}
```

```
error: Can't leave the ghost block early
  --> example.rs:4:13
    |
 4  |             break;
    |             ^^^^^
```

Listing 8: Ghost code affecting control flow

The same is the case with labeled `break` and `continue`, as shown in listing 9.

```
fn foo() {
    'outer: loop {
        ghost! {
            loop {
                break 'outer;
            }
        }
    }
}
```

```
error: Can't leave the ghost block early
  --> example.rs:5:17
   |
 5 |                break 'outer;
   |                ^^^^^^^^^^^^
```

Listing 9: Ghost code affecting control flow

`return` statements might not occur at all within ghost blocks, as that also constitutes leaving the ghost code at a different position than the intended one. Thus, we also issue a verification error whenever we find a `return` statement within a ghost block.

3.2.3 Modification of Real Variables

Ghost code may not influence the execution of the program. This implies that by executing ghost code, no values needed by the program at runtime may be modified. In other words, the set of values that may be modified by ghost code and the set of values that may be read by regular code may not intersect.

Exceptions to this rule are zero-sized types. The most prominent example of a zero-sized type would be Rust's unit type, written as `()`. A zero-sized type has, as its name implies, a size of zero. During runtime this means that the type can only have one possible value, and therefore carries no information whatsoever. The explicit implementation of this exception is of significance as many of Rust's syntactic code blocks implicitly return the unit type, which is translated into the intermediate language. This is also the case for the ghost blocks.

This can be implemented by a simple analysis of the function body. We iterate over all instructions in the function, and a set of read-from variables in regular code is collected. Then, we search for all modifications of variables in ghost code. If a variable that is being modified is in the set of read-from variables and is not zero-sized, this is an error that is being reported.

Apart from modifying local variables, ghost code may also not influence any global state. Calling any impure function may modify state. Thus, we may only call pure functions from ghost code.

3.2.4 Termination of Ghost Code

A subtle constraint of ghost code is that it must terminate. This goes in a similar direction as the non-interference of control flow but is far more difficult to detect.

Listing 10 illustrates the general problem with nontermination in ghost blocks. While the ghost code is included during verification, the `unreachable!` statement is obviously unreachable, and no verification error is returned. However, during runtime, the ghost block is removed, and so the `unreachable!` statement is reachable, causing a panic. Here it never terminates, but also partial termination causes issues.

Listing 11 shows ghost code that terminates in certain cases. This also introduces unsoundness. In a more general pattern, using ghost blocks in that way could be used to introduce certain assumptions of the variables. However, this is not the goal of ghost blocks, and a bit unintuitive, so it should be prevented.

The only way to prevent this is to prove all ghost blocks to terminate.

As a first step, we require all function calls from within ghost code to terminate. By the requirement of non-interference with global state, calls are already restricted to pure functions. As pure functions already are required to terminate, no further action is needed here.

Proving that code terminates in a general case is highly nontrivial, as the Halting problem is algorithmically unsolvable. However, certain proof patterns can be used to verify termination. Chapter 5 shows the implementation of the termination verification, which will not be explained here.

```rust
fn foo() {
    ghost! {
        loop {}
    }
    unreachable!();
}
```

Listing 10: Nontermination within ghost code

```rust
fn fun(x: u32) {
    ghost! {
        while x != 0 {}
    }
    assert!(x == 0);
}
```

Listing 11: Nontermination within ghost code leading to unsound results

3.3 Evaluation

After implementing this, we have a basic ghost block that can be used to calculate auxiliary values helping the verification, without actually needing to execute that code during runtime. Depending on how computationally expensive the ghost code is, this is can be a major improvement for runtime. However, ghost code in its current form is quite weak. We can only prove properties about the current variables right outside the ghost code block.

One feature that is desired would be the ability to reason about changing state between different ghost blocks. To do so, we would like to store ghost state across different ghost blocks. This will be explained in chapter 4.

3.3.1 Performance

We use the code snippet in listing 12 to test the performance impact of ghost blocks. We tested the snippet with ghost annotation and also without. The example with ghost annotation, as displayed in the listing, ran on average 30% slower. It is statistically significantly slower, although I'd say acceptable for the runtime benefit it provides.

This is a rather simple example, and other ghost code might contain more than just a simple list of statements. In the case of loops, we must prove termination, which adds some overhead. We will come to that verification overhead in chapter 5 which discusses termination.

```rust
fn main() {
    ghost!{
        let mut i = 0;
        i += 1;
        i += 1;
        i += 1;
    };
}
```

Listing 12: Simple Ghost Block

Any details to the method of testing performance of pieces of code can be found in appendix B.

Chapter 4

Ghost Types

4.1 Motivation

Ghost blocks help a bit by introducing small scopes to compute auxiliary values to prove statements about non-ghost code. However, much of the power of ghost code comes from keeping state in the ghost blocks and keeping track of that state alongside real state. As with ghost code, ghost state is not present in the compiled code but only helps in verification. We model state that is only present during verification using *ghost types*, which will be explained in the following sections.

Additionally, we would like to use variable-size data structures from within pure functions. Unbounded data structures cannot be created or modified in a pure function, as those need to be heap-allocated, and pure functions are prohibited from heap-allocations as they are not referentially transparent due to a dependence on global heap state. This motivates us to add verifier-only built-in unbounded data structures that can be created and used within pure code. By the necessity of being only present during verification, these data structures are ghost types as well.

Due to this twofold need for ghost types, we split the following sections into built-in ghost types, and custom ghost types.

4.2 Methodology

4.2.1 Custom Ghost Types

To keep ghost state, we need a mechanism to transfer data from one ghost block to another.

To achieve that, we introduce *ghost types*. Variables are of a ghost type we will from now on refer to as *ghost variables*.

Ghost types are types, which are only really present during verification, and removed from compilation. Of course, the type cannot be removed, but instead, it will be replaced by a trivial type storing no information, i.e. a zero-sized type like Rusts unit type `()`. This way, state can be kept during verification, while it does not need to be computed or stored during runtime.

4.2.2 Builtin Ghost Types

Additionally, we can add special types representing abstract concepts like unbounded *sequences*, *maps*, or *integers*. These types still are zero-sized during runtime, which means that they adhere to Rusts requirements for *Copy* types. Rusts Copy is a marker that is reserved for types that can be copied bit-by-bit. In particular, this means that the type contains no heap references, and all fields of the type have the Copy marker as well. This makes it possible for these types to be used as arguments and return types of pure functions. As explained earlier, pure functions cannot do heap allocations.

4.3 Implementation

4.3.1 Custom Ghost Types

We add a new type to the Prusti prelude, namely `Ghost<T>`. The implementation of the ghost type differs during runtime and verification.

During runtime, it is simply a zero-sized type that carries no information whatsoever, as seen in listing 13. The one field `_phantom` is solely there to tell the Rust compiler that we actually need the generic parameter in the type, even though we have no real variables that use the parameter. Otherwise, the compiler would throw an error reminding us of using the parameter. The `PhantomData` type itself is zero-sized though and carries no information.

During verification, the ghost type has a different form, and that is displayed in listing 14. During runtime, it simply is isomorphic to the type parameter `T`.

```rust
pub struct Ghost<T> {
    _phantom: PhantomData<T>,
}
```

Listing 13: Ghost Type Implementation for use during Runtime

As the ghost type is zero-sized during runtime, it is safe to assign values to ghost variables at any time, in real or ghost code. Since the ghost block safety checks look for assignments of non-zero-sized variables from within ghost blocks, we need to add a special case here.

```rust
pub struct Ghost<T> {
    data: T,
}
```

Listing 14: Ghost Type Implementation for the Verifier

During verification, the ghost type is an actual struct containing data instead of being a type alias to the parameter, as we need to track what variables are of a ghost type, and type aliases would be invisible at the MIR level where we receive the to-be-verified program.

4.3.2 Builtin Ghost Types

The user-facing implementation of the ghost types is rather simple and shown in listing 15. It contains no information whatsoever.

```rust
pub struct Seq<T> {
    _phantom: PhantomData<T>,
}

pub struct Map<K, V> {
    _key_phantom: PhantomData<K>,
    _val_phantom: PhantomData<V>,
}

pub struct Int(());
```

Listing 15: Definitions of Builtin Ghost Types

During verification, the types have the same definition, hence we are omitting those. The real semantics come by adding functions adding meaning to the types, which only exist during verification. Since the types have no real data associated with them, we do not return a meaningful value.

The function implementations for the sequence type are given in listing 16. For brevity, we omitted similar definitions for the map and integer type. The functions should never be called at runtime, hence the panic statements inside.

Meaning to these dummy functions is given within the verifier. Inside the verifier, any variable that makes use of these dummy types will be replaced by a variable with the same name, using a verifier-internal built-in type. Any function call to these functions will be replaced by a call to an internal function.

In the end, when encoding to viper, vipers internal sequences, maps, and unbounded integers will be used. Also, any built-in functions will be low-

```
impl<T: Copy> Seq<T> {
    pub fn empty() -> Self {
        panic!()
    }
    pub fn single(_: T) -> Self {
        panic!()
    }
    pub fn concat(self, _: Self) -> Self {
        panic!()
    }
    pub fn lookup(self, _index: usize) -> T {
        panic!()
    }
    pub fn len(self) -> Int {
        panic!()
    }
}
```

Listing 16: User-facing implementation of the `Seq` type

ered to the corresponding operation on these primitives. An example of this can be seen in listing 17. The `Seq::single` function gets lowered to a Sequence literal containing one entry. The `Seq::concat` function gets translated to Vipers concatenation operator (`++`).

```
let s1 = Seq::single(1);
let s2 = Seq::single(2);
let s3 = s1.concat(s2);
```

```
var s1: Seq[Int] := Seq(1)
var s2: Seq[Int] := Seq(2)
var s3: Seq[Int] := s1 ++ s2
```

Listing 17: Prusti's `Seq` translated to Viper

Simplifications using Macros

As writing code like in listing 18 is rather cumbersome, we would like to have a way to write it similar to a new vector in Rust.

To achieve that, we write a macro that takes off this workload for us. The macro in listing 19 takes a comma-separated list of values and writes us an expression similar to the one in listing 18.

The sequence construction from listing 18 can be rewritten to `seq![1, 2, 3, 4]`.

```
        Seq::single(1)
.concat(Seq::single(2))
.concat(Seq::single(3))
.concat(Seq::single(4))
```

Listing 18: Creating a new sequence of values

```
#[macro_export]
macro_rules! seq {
    ($val:expr) => {
        $crate::Seq::single($val)
    };
    ($($val:expr),*) => {
        $crate::Seq::empty()
        $(
            .concat(seq![$val])
        )*
    };
}
```

Listing 19: `seq!` macro to simplify sequence construction

4.4 Evaluation

With this, we can finally store state that is only shared between ghost code. However, the ghost variables are still quite cumbersome to use. As an example, each time we assign to a ghost field outside the ghost block, we need to construct a new ghost variable by using `Ghost::new`. This is shown in listing 20. It might seem contradictory to initialize the ghost value in regular instead of ghost code, but this poses no issue: regular code may influence ghost code, it's that ghost code may not influence regular code. Thus, assigning values to ghost variables from regular code is fine, as long as we don't read them. We can only read the contents of the ghost variable from within ghost blocks, as the implementation to dereference a ghost value to its contents (`*x`) is only present during verification. We could also read it during verification outside the ghost block, but during compilation, this would fail, and hence, it poses no soundness issue. We could improve this by also checking for reading from ghost values in regular code during verification, such that we don't get surprises during compilation.

```rust
let mut x: Ghost<i64> = Ghost::new(0);
ghost! {
    x = Ghost::new(1);
}
// cannot read the value of x here during runtime
ghost! {
    prusti_assert!(*x == 1);
}
```

Listing 20: Ghost Variables in use

4.4.1 Performance

We wrote the code in listing 21 to assess the performance impact of using ghost variables. We verify the upper code to see how long it takes in comparison to the code below.

The variant with the ghost variables ran statistically slower, with a mean slowdown of 8%.

```rust
let mut x: Ghost<i32> = Ghost::new(42);
x = Ghost::new(43);
x = Ghost::new(44);
```

```rust
let mut x: i32 = 42;
x = 43;
x = 44;
```

Listing 21: Assigning Ghost Variables Performance Test

Any details to the method of testing performance of pieces of code can be found in appendix B.

Chapter 5

Termination

This chapter concerns itself with adding features to prove termination of Rust code. It is split into two parts, first proving termination of pure functions, and then proving termination of general Rust code.

5.1 Pure Functions

At the moment, pure functions in Prusti are required to terminate. However, this property is currently not checked anywhere.

The decision that pure functions must terminate is not just an arbitrary decision, but an important criterion to keep verification sound. Pure functions in themselves are not unsound if they don't terminate, but pure functions can also be used from within specifications. Nonterminating specifications are indeed unsound.

Consider the code in listing 22 When calling the `poison` function, we can be sure that all following code is unreachable. This is fine in potentially nonterminating code but introduces a soundness issue when used in a specification.

Nowhere in the precondition does the function `foo` specify that its argument must be 42, but the code verifies as the precondition `poison()` ensures that `false == true`, i.e. any assertion verifies as there is no difference between a correct and an incorrect statement anymore.

```
#[pure]
#[ensures(false == true)]
fn poison() -> bool {
    poison()
}

#[requires(poison() == true)]
fn foo(i: i32) {
    assert!(i == 42);
}
```

Listing 22: Nonterminating pure function

The issue lies with the fact that `poison` does not terminate. If we call `poison` *within* the function, this is valid, as no assertion after the call needs to be verified as the code is unreachable. However, because the precondition is not present during runtime, the assertion is reachable and will fail.

5.1.1 Methodology

We will be implementing *termination measures* on pure functions to prove termination. Conceptually they are not that complicated: We assign an integer expression dependent on function arguments to each pure function. We call this expression the termination measure. From a pure function, we may not call any pure function with arguments that have negative termination measures. All called functions also must have a strictly lower termination measure than the caller function. This ensures that there is only bounded recursion from any pure function, which implies termination.

5.1.2 Implementation

We add a `terminates` attribute to Prusti, that can be used to annotate the termination measure. Pure functions are explicitly required to have the `terminates` attribute. An example of such a termination measure can be seen in listing 23. How it is encoded in the verifier can be seen in listing 24. At the begin of the function body on line 5, we set the variable `termination_measure` to the expression annotated in the `terminates(..)` macro. Then, at each call site within the function, we calculate the termination measures of the calls and assert that those measures are lower than the termination measure of this call and non-negative. This can be seen in listing 24 on lines 10&11 and 15&16 respectively.

The proof for termination is rather easy here, as the only argument always decreases, and is required to be non-negative.

```rust
#[pure]
#[terminates(x)]
#[requires(x >= 0)]
fn fibonacci(x: i64) -> i64 {
    if x < 2 {
        x
    } else {
        let x1 = x - 1;
        let f1 = fibonacci(x1);

        let x2 = x - 2;
        let f2 = fibonacci(x2);

        f1 + f2
    }
}
```

Listing 23: Pure, terminating function

```rust
#[pure]
#[terminates(x)]
#[requires(x >= 0)]
fn fibonacci(x: i64) -> i64 {
    let termination_measure = x;
    if x < 2 {
        x
    } else {
        let x1 = x - 1;
        assert!(x1 < termination_measure);
        assert!(x1 >= 0);
        let f1 = fibonacci(x1);

        let x2 = x - 2;
        assert!(x2 < termination_measure);
        assert!(x2 >= 0);
        let f2 = fibonacci(x2);

        f1 + f2
    }
}
```

Listing 24: Pure, terminating function, desugared

We can also leave out the termination measure, for an implied termination measure of 1, as seen in listing 25. This simply means that the function does not call any other function and terminates that way.

```
#[pure]
#[terminates]
fn zero() -> i64 {
    0
}
```

Listing 25: Pure function with implicit termination measure

5.1.3 Evaluation

```
#[pure]
#[terminates(1)]
fn foo0() -> i32 {
    0
}
#[pure]
#[terminates(2)]
fn foo1() -> i32 {
    foo0()
}
#[pure]
#[terminates(3)]
fn foo2() -> i32 {
    foo1()
}
```

Listing 26: Nonrecursive terminating functions, annotated with termination measures

Termination measures are an effective way to prove termination of pure functions. However, it's quite cumbersome to use.

Regard the scenario from listing 26. We need to annotate every function with a termination measure that equals the recursion depth that is created by calling the function. The termination measure is highly coupled to the implementation of the function and may need to change every time the implementation is modified. Worse even, changing the termination of a function requires changing the termination measure in all callers of that function, and so on.

We tried preventing that by only requiring lower termination measures when the called function *may* calls the caller again. However, trait functions complicate that, as a specific implementation could always introduce mutual recursion. This is highly problematic as traits play a central role in Rust's built-in functions that can be used as unary or binary operations. Additionally, the verifier does not have much information about external functions, and so callgraph analysis returns imprecise, unusable results.

This left us with the only option to require lower termination measures for each call, even if it may seem trivial to see that the functions don't introduce mutual recursion.

Performance

We test the code from listing 27 to assess performance hit with actual function calls when we want to prove termination of functions. It is significantly slower with the termination proof, with a mean slowdown of 6 percent.

```rust
#[terminates(Int::new(x) + Int::new(1))]
#[requires(x >= 0)]
fn foo(x: i64) {
    bar(x);
    if x > 0 {
        foo(x - 1);
    }
}

#[terminates(Int::new(x))]
#[requires(x >= 0)]
fn bar(x: i64) {
    if x > 0 {
        bar(x - 1);
    }
}
```

Listing 27: Benchmarked recursive functions

Any details to the method of testing performance of pieces of code can be found in appendix B.

5.2 Impure Functions

This section concerns itself with the verification of terminating *impure* code. Since impure code is a superset of pure code, there are a few additional constructs that need to be proven to terminate. The main thing that may introduce nontermination in impure functions is control flow - we need to prove that loops terminate.

Not only does this introduce additional functionality to the verifier, but it also guarantees soundness of ghost code. Ghost code is quite similar to functional specifications in that regard, and also needs to terminate.

The issue is exemplified in listing 28. During runtime, the value of x is always seventeen. However, due to the nontermination of the ghost code, during verification, the assertion below seems unreachable and therefore verifies. This leads to an assertion failure during runtime, as the value of x clearly is not 42.

```
let x = 17;

ghost! {
    while x != 42 {}
}

assert!(x == 42);
```

Listing 28: Misbehaving ghost code

5.2.1 Methodology

Analogous to termination measures on functions, we can use a similar concept on loops. We want an integer expression that strictly decreases each loop iteration, and has some minimal, limiting value. By ensuring that the expression always decreases and never reaches some value, we can use the value of the expression as an upper bound for remaining loop iterations and can be sure that the loop terminates. We call this integer expression the *loop variant*.

5.2.2 Implementation

To loops, we add an optional `body_variant!` macro, that can be used to annotate loop variants to a loop body. It can occur at the same place as loop body invariants and applies to the immediately surrounding loop. An example of a loop variant in use is shown in listing 29 and how it is encoded on a high level is shown in listing 30.

Loop encoding is a bit more complex than function encoding. The loop is completely rewritten and flattened such that the resulting code contains no loops. First, we encode the `while` loop to an `if` statement. At the beginning of the `if` body, we assert that the loop invariants hold in the first iteration, which can be seen in listing 30 on lines 3–5. Next, we delete all information

```
let mut x = 10;
while x > 0 {
    body_invariant!(x > 0);
    body_invariant!(x <= 10);
    body_variant!(x);
    x -= 1;
}
```

Listing 29: Simple Loop Variant

we have to the mutated variable `x` by havocking it on line 10. Then, we assume the loop invariants on lines 12–14. Together with the havocking, this simulates that we are in an arbitrary loop iteration just before the loop body begins. Further, we save the value of the loop variant expression in the variable `variant`, on line 17. Finally, we just insert the normal loop body, shown on lines 19–21. The loop invariants only need to hold if we enter the loop again, hence we simulate that again by checking for the loop condition again on line 23. Then, we assert that the loop invariants still hold upon re-entering the loop on lines 24–26. Now to the important part: we check that the loop variant expression decreased, i.e. is smaller than the value we stored in `variant` earlier, as seen on line 29. Also, we check that the variant is still non-negative, as seen on line 30. Finally, we assume the loop was not re-entered, with an `assume false` on line 32, to simulate that the loop exited at some point.

This encoding enforces that the chosen expression decreases each loop iteration, and is non-negative at all times.

5.2.3 Evaluation

Loop Variants are an easy and effective way to prove termination for loops. It gives us a tool to prove termination of impure functions, or just parts of impure functions that need to terminate, such as ghost blocks.

However, there are still cases where one cannot prove termination of a loop because the decreasing object might not be an integer expression. Consider the code snippet from listing 31. The Node stores either a Leaf or an Element which contains an integer value in addition to an owned next Node. By the memory rules of safe Rust, it is clear that this implementation of a linked list contains no loops, as no two owned box values can point to the same Node. The loop variant would need to be the length of the remaining list, but since that is unknown, we cannot give an upper bound on the remaining loop iterations. We merely know that the list must be *finite* and thus by traversing it, we must reach the end eventually. Hence, we can not provide a concrete loop variant, and cannot prove termination of this loop.

```
let mut x = 10;
if x > 0 {
    // loop invariant assertions
    assert x > 0;
    assert x <= 10;

    // loop variant assertions
    assert x >= 0;

    havoc x;

    // loop invariant assumptions
    assume x > 0;
    assume x <= 10;

    // loop variant assumptions
    assume variant = x;

    // begin body
    x -= 1;
    // end body

    if x > 0 {
        // loop invariant assertions
        assert x > 0;
        assert x <= 10;

        // loop variant assertions
        assert x < variant;
        assert x >= 0;

        assume false;
    }
}
```

Listing 30: Simple Loop Variant, desugared

Performance

We test the performance of the termination proof with the code snippet in listing 32. It ran significantly slower, as expected, with a mean increase in runtime of 30 percent.

Any details to the method of testing performance of pieces of code can be found in appendix B.

```rust
enum Node {
    Leaf,
    Element(i64, Box<Node>),
}

#[terminates]
fn length(node: mut Node) -> usize {
    let mut length = 0;
    while let Node::Element(_, next) = node {
        body_variant!(...);
        node = next;
        length += 1;
    }
    length
}
```

Listing 31: Currently unprovable loop termination

```rust
#[terminates]
fn main() {
    let mut i = 0;
    while i < 100 {
        body_variant!(Int::new(100) - Int::new(i));
        i += 1;
    }
}
```

Listing 32: Benchmarked code

Chapter 6

prusti_assert!

Prusti adds a bit of its own syntax on top of Rust's syntax to allow users to express full first-order logic expressions. Among other things, it adds universal and existential quantifiers and implication arrows. This Prusti-specific syntax can only be used in pre- & post-conditions, and loop invariants. It cannot yet be used within assertions. To circumvent this, one had to define auxiliary functions requiring an expression as precondition and then calling that function from where one would have the assertion, as seen in listing 33.

```rust
#[pure]
fn identity(x: u64) -> u64 {
    x
}
fn foo() {
    // some code
    aux();
    // some more code
}
#[requires(forall(|x: u64| x == identity(x)))]
fn aux() {}
```

Listing 33: Workaround to use quantifier-based assertions in the middle of some code

As that introduces quite some writing and also comprehension overhead, we would like to introduce a way to extend assertions such that we don't need auxiliary functions, and therefore write assertions inline in the function directly. This can be observed in listing 34.

Additionally to an assertion macro, a thing that can be implemented analogously is an *assumption* macro. As opposed to asserting that a fact holds at the position, it assumes that the fact holds from that point on.

```
#[pure]
fn identity(x: u64) -> u64 {
    x
}
fn foo() {
    prusti_assert!(forall(|x: u64| x == identity(x)));
}
```

Listing 34: New Assertion using quantifiers

6.1 Methodology

To begin tackling this problem, it is important to note that we cannot extend the syntax of the `assert!` macro easily. Since the syntax we extend it with contains quantifiers that cannot be implemented in Rust code, we can only use this extended syntax during verification, and not runtime. This means that any kind of assertion that *might* contain quantifiers must be omitted during runtime. Hence, replacing Rust's existing `assert!` macro is unfavorable, as then assertions cannot fail during runtime anymore. One could argue that if the assertions are verified it doesn't matter anyway. However, there might be cases where we leave out verification of a specific function, and where we still would like to have runtime assertions.

Thus, we implement a new kind of assertion, called `prusti_assert!`, that can support the whole range of Prusti syntax. Analogously, we implement a macro `prusti_assume!`, which we will not explicitly name during the implementation, as it is extremely similar.

6.2 Implementation

The first thing to notice is that there is already a macro with similar semantics to our intended `prusti_assert!`. This macro is `body_invariant!`. It supports the full Prusti syntax and can be used within regular Rust code. However, it is limited to the annotation of loops.

So, we encode the `prusti_assert!` macro in a quite similar manner as the `body_invariant!` macro, except clearly marking it that it is an assertion and not an invariant. Then, in the lowered code, we extract the encoded assertion and insert appropriate statements in the Prusti intermediate representation.

To be more precise, we insert dead code in the Rust source containing a closure that has specification attributes, as seen in listing 35. We insert it in an `if false` such that the code surely is never executed. Additionally, we annotate the closure that it is *only* used for specification, and serves no further use, using `prusti::spec_only`. We also add an annotation that it is an assertion, using `prusti::prusti_assertion`. For each specification, we also add a unique identifier, the `spec_id`, which we shortened in this example for illustrative purposes. Lastly, the expression within the assertion is encoded in the closure. This is done because closures are one of the only constructs that can occur in a function body, can be annotated, and can explicitly be provided with a type.

```
fn main() {
    prusti_assert!(42 < 43);
}
```

```
fn main() {
    #[prusti::specs_version = "0.1.0"]
    if false {
        #[prusti::spec_only]
        #[prusti::prusti_assertion]
        #[prusti::spec_id = "4a549f135e"]
        || -> bool { 42 < 43 };
    };
}
```

Listing 35: source and encoded `prusti_assert!` macro

Later, when compiled down to MIR code, we search for all specification items and will find the closure again. We read the information that the closure provides us with and insert an appropriate assertion statement in the Prusti intermediate representation.

6.3 Evaluation

After this change, we can now use the full Prusti syntax in assertions and assumptions. There are more advantages than just the additional usable syntax, however.

Since `prusti_assert!` is encoded as a single statement in the intermediate representation, it is a lot simpler than an `assert!` macro in standard Rust. The standard assertion might include an arbitrary amount of impure code, which might be slower than if we were to encode the assertions as pure values.

6.3.1 Performance

To test if `prusti_assert!` is any faster than the standard `assert!` macro, we ran the code in listing 36 twenty times and compared it to an equivalent snippet using the `assert!` macro. The `prusti_assert!` variant ran significantly faster, with a mean speed-up of 37 percent.

```rust
fn test1() {
    let mut a = [1; 100];
    a[1] = 2;
    prusti_assert!(a[1] == 2);
    prusti_assert!(a[0] == 1);
}
```

Listing 36: Benchmark setup to compare `assert` and `prusti_assert`.

Any details to the method of testing performance of pieces of code can be found in appendix B.

Chapter 7

Conclusion

Initially, we wanted to verify the hashmap data structure from VeriBetrFS in Rust, and test the suitability of Prusti to verify that data structure. In particular, we wanted to verify the absence of panics and integer overflows. Additionally, we wanted to add built-in mathematical types to Prusti, such as a map, to verify that the hashmap has the intended behavior of a map. In the end, we wanted to compare the verification in Prusti with the one already verified in Dafny with the original VeriBetrFS implementation.

A lot of the planned goals didn't work out, there were more roadblocks than anticipated. Thus, we spent time implementing necessary (but probably not sufficient) features in Prusti that inch closer to the goal of verifying larger code examples. In particular, we added support for ghost code to Prusti, added mechanisms to verify termination of code, and implemented built-in mathematical types.

In combination, these features can be used to verify a property using induction by calling a pure lemma function inside a ghost block. An example is given in appendix A.1.

7.1 Related Work

This thesis bases a lot of decisions on previous work.

The initial data structure which we tried to verify in Rust using Prusti was the linear mutable map of the VeriBetrFS project [6]. It uses a wide range of features of Dafny [9] that were not present in Prusti, such as lemma functions, ghost types, and built-in mathematical types.

The design decisions for the ghost code are heavily influenced by the spirit of ghost code [4]. It simplified the work of thinking about the precise semantics for ghost code in Prusti.

We base our termination proofs on the work of Floyd's *Assigning meanings to programs* [5]. It goes into detail about what can be used as a termination proof, and even uses a more general notion of termination measures.

Similar work has been done by Sarek Høverstad Skotåm on CreuSAT [12]. He used the Rust verifier Creusot [2] to verify a series of SAT solvers. Although his goal was to write a formally verified SAT solver, and not to verify a large codebase to test the verifier, he did so as anyways and wrote one of the largest, if not the largest formally verified Rust codebase in the process. Work could be spent comparing Prusti and Creusot to see if there are fundamental differences.

7.2 Future Work

7.2.1 Termination

As already mentioned in the section on termination, the current proof for recursive termination is not ideal. Every function call needs to be justified to have a lower termination measure, even if it cannot reach back to the caller. The Rust compiler provides a `mir_callgraph_reachable` interface to check callgraph reachability. However, this can not be used for external functions, i.e. functions defined in a different crate. One would need to implement complete callgraph analysis on their own while knowing external function definitions.

Additionally, if one were to implement callgraph analysis, one could extend that to do specification graph analysis, i.e. which pure functions depend on other pure functions in their specifications. For soundness, this must be acyclic. An unsound, currently verifying example is given in listing 37. No part of the code implies nontermination, or `false`, however, that is the postcondition of both functions. It is purely introduced by a pair of specifications and calls which together introduce a cycle. Thus, cyclic specification and/or cyclic specification and recursion pairs must be prohibited as well.

```rust
#[pure]
#[requires(bar() == true)]
#[ensures(false)]
fn foo() -> bool {
    true
}

#[pure]
#[ensures(false)]
fn bar() -> bool {
    foo()
}
```

Listing 37: Unsound mutual specifications

7.2.2 User Defined Mathematical Types

The built-in types like `Seq`, `Map`, and `Int` seem to work fine. However, extensibility is limited as any additional mathematical type would need to be manually added to the special cases of type translations. It would be favorable if one could define semantics of other mathematical types within Rust itself, and then encode the type as a domain in Viper.

Then, the currently special cases of the above-mentioned mathematical types could be removed from the verifier and defined completely by the user-facing side of the verifier.

One downside of exposing domains to the user would be that there is the possibility of introducing unsoundness by creating paradoxical domain axioms.

7.2.3 Better Quantifier Support

Finally, it could be of advantage to have the possibility of assigning arbitrary boolean expressions to a `bool` variable from within ghost code. In particular, we mean quantified expressions.

This way, one could remove the special case of the `prusti_assert!` macro for quantifiers.

Appendix A

Code Snippets

A.1 Complete Implementation of Fibonacci

```
#[pure]
#[terminates(Int::new(n))]
#[requires(n >= 0)]
fn fib(n: i64) -> i64 {
    if n < 2 {
        n
    } else {
        fib(n-1) + fib(n-2)
    }
}

#[pure]
#[terminates(Int::new(n) - Int::new(i))]
#[requires(i >= 0 && i <= n)]
#[requires(f1 == fib(i) && f2 == fib(i + 1))]
#[ensures(result == fib(n))]
fn fast_aux(f1: i64, f2: i64, i: i64, n: i64) -> i64 {
    if i == n {
        f1
    } else {
        assert!(i < n);
        assert!(n - (i + 1) >= 0);
        assert!(n - (i + 1) < n - i);
        fast_aux(f2, f1 + f2, i + 1, n)
    }
}

#[pure]
```

```
#[terminates(Int::new(n))]
#[requires(n >= 0)]
#[ensures(result == fib(n))]
fn fast_fib(n: i64) -> i64 {
    fast_aux(0, 1, 0, n)
}

#[pure]
#[terminates(Int::new(n))]
#[requires(n >= 0)]
#[ensures((n % 3 == 0) == (fib(n) % 2 == 0))]
fn lemma(n: i64) {
    if n >= 3 {
        fib(n - 1);
        fib(n - 2);
        lemma(n - 1);
        lemma(n - 2);
    }
}
```

Appendix B

Benchmarking Setup

We used a benchmarking script that starts a Prusti server and warms it up by verifying a simple program 20 times. Each example was verified 20 times unless specified otherwise.

We ran the benchmark on a computer featuring an Intel Core i7-8565U running at 4.6 GHz. 16 Gigabytes of memory are present on the system, but never fully used. The only built-in permanent storage device was an SSD, all executables and swap space reside there.

The OS in use was Arch Linux with kernel version 5.19.9. OpenJDK 18.0.2 was used as Java installation. The Z3 version in use was 4.8.6. The specific Prusti version in use can be found at the following url:
https://github.com/JM4ier/prusti-dev/tree/4e322057.

B.1 Significance of Results

We used the code below to test the statistical significance of timing differences between related code snippets. It reports the mean change in runtime and the likelihood that such a difference would occur if the runtimes had the same distribution in the form of the p-value. If we write anywhere that a result is *statistically significant* we refer to a significance level of 1 percent unless specified otherwise.

```python
#! /usr/bin/env python3

import scipy
import os
import json

def mean(list):
    return sum(list) / len(list)
```

```python

bench_out = 'benchmark-output'
json_files = [
    os.path.join(bench_out, f)
    for f in os.listdir(bench_out)
    if f.endswith('json')
]
json_file = max(json_files, key = os.path.getctime)

print(f'Analyzing data from {json_file}')
print()

with open(json_file, 'r') as f:
    data = json.load(f)

for key in data:
    if key.endswith('ref.rs'):
        continue

    test_name = key.split('/')[-1]
    ref = key.replace('.rs', '_ref.rs')

    new_data = data[key]
    ref_data = data[ref]

    new_mean = mean(new_data)
    ref_mean = mean(ref_data)

    change = (new_mean - ref_mean) / ref_mean
    change_100 = change * 100

    test = scipy.stats.ttest_ind(
        new_data,
        ref_data,
        equal_var=False,
        alternative='two-sided'
    )

    print(f'analyzing {test_name}')
    print(f'  {change_100:+05.1f}% in runtime')
    print(f'  {test.pvalue:.4f} p-value')
    print()
```

Bibliography

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 3, pages 147:1–147:30. ACM, 2019.

[2] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: a Foundry for the Deductive Verication of Rust Programs. In *ICFEM 2022 - 23th International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science, Madrid, Spain, October 2022. Springer Verlag.

[3] The Rust Project Developers. Rust programming language. `https://www.rust-lang.org/`, 2022. [Online; accessed 27-September-2022].

[4] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3):152–174, Jun 2016.

[5] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[6] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that Way!). In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 99–115. USENIX Association, November 2020.

[7] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: A Right-Optimized Write-Optimized file system. In *13th USENIX Conference on File and Storage*

Technologies (FAST 15), pages 301–315, Santa Clara, CA, February 2015. USENIX Association.

[8] Andrea Lattuada, Travis Hance, Rob Johnson, Jialin Li, Chris Hawblitzel, Gerd Zellweger, and Jon Howell. Linear mutable map. `https://github.com/vmware-labs/verified-betrfs/blob/master/lib/DataStructures/LinearMutableMap.i.dfy`. [Online; accessed 25-February-2022].

[9] K Rustan M Leino. Accessible software verification with dafny. *IEEE Software*, 34(6):94–97, 2017.

[10] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. Linear types for large-scale systems verification. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022.

[11] 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.

[12] Sarek Høverstad Skotåm. Creusat. `https://sarsko.github.io/_pages/SarekSkot%C3%A5m_thesis.pdf`, 2022. [Online; accessed 02-October-2022].

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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):

TOWARDS VERIFYING REAL-WORLD RUST PROGRAMS

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

Name(s):	**First name(s):**
Maier	Jonas

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)**
Zürich, 5.10.2022	Jonas Maier

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