Prusti in Practice

A case study of using the Prusti auto-active program verifier for Rust

Stef Gijsberts
s1034031

March 22, 2023

Supervisors:
dr. Robbert Krebbers, ir. Ike Mulder

First assessor:
dr. Robbert Krebbers

Second assessor:
prof. dr. Herman Geuvers

Radboud University

Acknowledgement

I thank dr. Robbert Krebbers and ir. Ike Mulder for their time and energy spent helping me in the process of writing this thesis. This thesis would not exist without their guidance. The weekly meetings and feedback were invaluable. I am especially grateful for Robbert's feedback and insights on structuring the text, and Ike's help and dedication in the research process.

I also thank the Prusti developers for their work. Especially Frederico Poli, Jonáš Fiala, and Aurel Bílýthe, for their quick and valuable responses to my questions and bug reports regarding Prusti.

I thank dr. Bernadette Smelik for providing me with the opportunity to use a piece of this report as an assignment for the course Academic Writing, her feedback, and her teaching on academic writing.

Finally, I thank my friends and family. I especially thank my friend and fellow student Marijn van Wezel for his enthusiasm, interest taken, and taking the time to proofread it, providing valuable feedback.

Abstract

The Rust programming language provides many safety guarantees compared to other languages. For example, Rust prevents use-after-free's and double free's. However, Rust programs are not checked for functional correctness or absence of unrecoverable errors such as failing assertions or integer overflows. The auto-active program verifier called Prusti promises to fill this gap. This report describes a case study of using Prusti on a non-trivial Rust program: a key-value store implementation using binary search. This implementation is especially challenging due to its use of so-called borrowing. We present the verification of this store, the end result of which is the Rust program annotated with the created specification. We also present a list of limitations encountered in the verification process.

Contents

1 Introduction **3**

2 Rust fundamentals **7**

2.1 Basic Rust syntax 7
2.2 Copy and move 8
2.3 Clone 11
2.4 Unique reference 12
2.4.1 Lifetimes 14
2.5 Shared reference 15
2.6 Slice 16
2.6.1 Subslice 17
2.7 Panic 17
2.7.1 Integer overflow 17
2.7.2 Integer underflow 18
2.7.3 Out of bounds index 18
2.7.4 Assertion 19

3 Prusti fundamentals **20**

3.1 Recurring example 20
3.2 Pre-condition for panic absence 21
3.3 Post-condition for functional correctness 22
3.4 Purity 26
3.5 Equivalence of the abs-functions 28
3.6 Prusti's logical syntax 29

4 Verification of a key-value store **32**

4.1 Store 33
4.2 Rust store implementation 33

4.2.1 Data structure 33
4.2.2 The ‘contains’ operation 34
4.2.3 The ‘get’ operation 36
4.2.4 The ‘get-mut’ operation 37
4.3 Example and intuitive specification 39
4.3.1 Intuitive specification 40
4.4 Pledges 41
4.4.1 Snd 41
4.4.2 Index 43
4.5 Prusti specification 45
4.5.1 Predicates 45
4.5.2 Specification of `contains` 47
4.5.3 Specification of `get` 48
4.5.4 Specification of `get_mut` 49
4.6 Conclusion 51

5 Prusti limitations **52**
5.1 Borrowing 52
5.1.1 Borrow splitting 52
5.1.2 Structs with reference fields 54
5.1.3 Two-phase borrows 54
5.1.4 Mutable slice indexing 55
5.2 Other limitations 56
5.2.1 usize guarantees 56
5.2.2 Multiple after-expiries 57
5.2.3 Assert that false cannot be proved 58
5.2.4 Trigger documentation 58
5.2.5 Missing termination check on pure functions 59

6 Related work **61**
6.1 Rust verification 61
6.1.1 Dynamic verification tools 61
6.1.2 Static verification tools 62
6.2 Verification for other low-level languages 64
6.2.1 Verifast 64

7 Conclusions and future work **65**

Chapter 1

Introduction

Rust is a general-purpose programming language. Its first stable version was released in 2015 (Rust contributors, 2023c). It quickly gained popularity, and is increasingly used. For example, 9.5% of the Firefox browser is written in Rust. For the last seven years, respondents of the yearly Stack OverFlow survey have indicated they love Rust the most of all programming languages. In 2022, 87% of developers using Rust said they want to continue using it (Stack Overflow, 2022).

The efficiency of the code produced by its compiler is competitive to that of C++, thanks to Rust's static typing, its optimizing compiler, and the absence of a garbage collector. But different from C++, Rust is memory safe, meaning that Rust programs are protected from bugs related to memory access. For example, Rust programs are guaranteed not to be affected by double-free's, use-after-free's (attempting to dereference a pointer after it has been freed), and data races. This memory safety is provided by Rust's type checker, including the so-called borrow checker, the component that enforces strict rules for the use of variables in a program.

Yet, there are some problematic programs that Rust does not prevent the programmer from writing. A well-formed Rust program could still give an output that does not adhere to the program's specification as intended by the programmer or get into an unrecoverable error state called a *panic*.

For example, the following Rust code, accepted by the compiler, does not give the correct output, because of a small mistake: `b` and `a` are mistakenly switched.

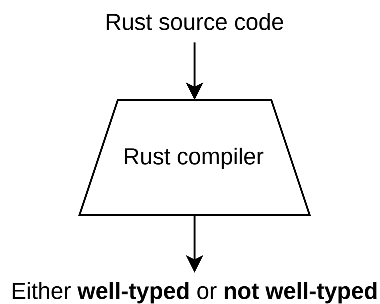
```rust
fn max(a: i32, b: i32) -> i32 {
    if a > b {
        b
    } else {
        a
    }
}
```

And the following Rust code, accepted by the compiler, causes a panic, due to out-of-bounds indexing:

```rust
let a = [0, 1, 2];
let x = a[3];
```

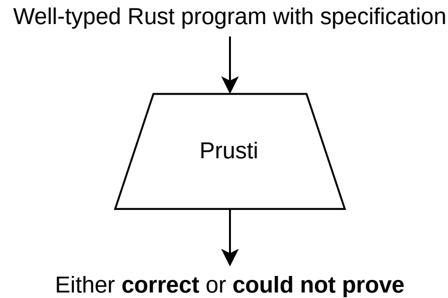
To verify the absence of such problems, program verifiers exist. In this report we focus on one of them: Prusti (Astrauskas et al., 2019). Prusti is specifically made for usage with Rust.

We must make a clear distinction between how the Rust compiler works and how Prusti works. The Rust compiler decides whether a piece of source code is **well-typed**, or **invalid** (the source code does not describe a well-typed Rust program).



Prusti takes in such a *well-typed* Rust program, with a specification, and tries to prove that the program is correct. It either outputs that the program is **correct**, or it outputs that it **could not prove** [1] the correctness of the program. In this process, Prusti does not run the program, but only statically analyses it.

[1] Note that 'Could not prove' is very different from 'proven to be false'! If Prusti cannot prove a program's correctness, the program might still be correct.



Prusti is an *auto-active* program verifier. The term auto-active, coined by Leino and Moskal, 2010, specifies when in the process of verification the user provides guidance. If using an auto-active verifier, the user can give guidance, but only before the verifier is ran. This guidance is often in the form of annotations added to the program. Auto-active verifiers can be thought of to lie in between *automatic* verifiers (no user guidance) and *interactive* (user guidance after the verifier has started).

Prusti combines the information from Rust's type system with user added annotations to automatically verify the correctness of the program "without exposing the underlying formal logic, allowing users to work exclusively at the level of abstraction of the programming language. This enables a new kind of verification tool, with the potential to impact a wide audience and allow the Rust community to benefit from state-of-the-art verification techniques" (Astrauskas et al., 2019).

Although the approach has the potential of impacting a wide audience, we are not aware of widespread usage of Prusti except from usage by Prusti developers.

To gain insight in the practical usage of Prusti, we present in this report a case study on the verification of a non-trivial Rust program, namely a key-value store implementation based on binary search.

This implementation includes functions that return pointers. These pointers can be used to modify the value of the variable it refers to. The verification of such functions is challenging, while this pattern is common in Rust code. Prusti has a mechanism for this.

The verification was an iterative process. Both the code and the specification were changed multiple times based on Prusti errors. Sometimes a change had

to be made because the specification was found to be incorrect. In other cases, the change was necessary because Prusti did not support some feature of Rust, or was not unable to prove a seemingly correct specification.

Contributions. The main two contributions of this work are:

- The verification of a key-value store implementation in Rust, in chapter 4. This includes both the Rust code, which is an implementation of binary search, and the added Prusti annotations, which contain the specification for this store.

  This verification is challenging because one of the functions returns an interior pointer that can be used to mutate a value in the key-value store. Also, the verification requires knowledge of both Rust and Prusti. Prusti's lacking documentation makes obtaining this knowledge difficult.

  This verification shows that Prusti can be used, and how, to verify the correctness of a non-trivial program.

- A list of problems encountered while using Prusti, including possible workarounds and/or (proposed) solutions, in chapter 5. This list is the result of many failed verification attempts. It can be used to improve Prusti and similar program verifiers. It also can be used for first time users of Prusti as a guide for overcoming or avoiding obstacles.

Outline. Chapter 2 introduces the Rust programming language. The reader familiar with Rust may want to skim it. Chapter 3 introduces the basics of the Prusti program verifier. In chapter 4 we discuss the implementation of a key-value store in Rust, and its verification using Prusti. The limitations of Prusti that were encountered in the process are listed in chapter 5. Related work is discussed in chapter 6. Chapter 7 concludes this report, and mentions possible future work.

Chapter 2

Rust fundamentals

In this chapter we discuss the fundamentals of the Rust programming language. The reader familiar with Rust may skip this section.

The diagrams and examples in this chapter are heavily inspired by the diagrams and examples in the chapter 'Understanding Ownership' from the book 'The Rust Programming Language' (Klabnik & Nichols, 2022b).

2.1 Basic Rust syntax

The syntax of Rust is similar to the syntax of other imperative programming languages, like C.

The following piece of Rust code to calculate the 15'th fibonacci number demonstrates some syntactic features of Rust.

```
fn fib(n: u64) -> u64 {
    if n < 2 {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}

fn main() {
    let n = 15;
    println!("{}", fib(n));
}
```

Functions are defined with the keyword `fn`, and types are written behind the name instead of before it. Also, variables must be introduced using the `let` keyword.

Expression-based. For programmers new to Rust, it might be surprising that Rust is expression-based. This means that most syntactic constructs are expressions. For example, the 'block'-expression (notated with curly braces) evaluates to the last value in the block:

```
1 let duration_ms = {
2     let duration_s = 11;
3     duration_s * 1000
4 };
5 assert!(duration_ms == 11_000);
```

And the 'if'-expression evaluates to the value of the evaluated branch:

```
1 let x = if false {
2     a
3 } else {
4     b
5 };
6 assert!(x == b);
```

So using this feature of Rust, the `fib` function can also be written without the return statement, like this:

```
1 fn fib(n: u64) -> u64 {
2     if n < 2 {
3         n
4     } else {
5         fib(n - 1) + fib(n - 2)
6     }
7 }
```

2.2 Copy and move

Consider the following piece of Rust code:

```rust
let a = 5;
let mut b = a;
b += 1;
println!("{}", a);
```

Like you might expect, it prints 5. When executing `let mut b = a`, Rust assigns a copy of `a`'s value (`5`) to `b`. After that, changing `b`'s value does not affect the value of `a`. The following diagram shows the final state.

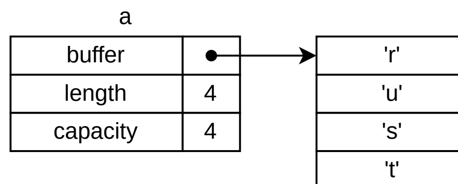
Now consider this similar piece of Rust code:

```rust
let a = String::from("rust");
let mut b = a;
b.push('!');
println!("{}", a);
```

It is similar to the previous example, but with a heap-allocated string instead of an integer. It might seem like a good program at first glance, but the Rust compiler will not accept this. This is because the code contains a serious programming error! Let us go through the code line by line:

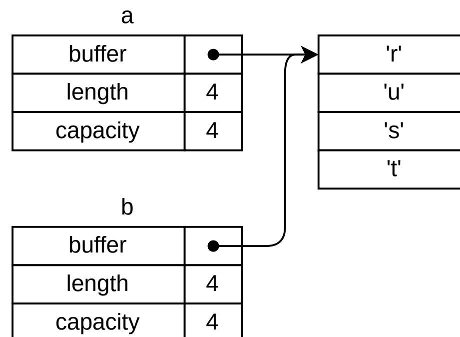
1. `let a = String::from("rust");`

   Executing this line, Rust allocates a buffer on the heap, and puts the string `"rust"` in that buffer. It then stores the pointer to the buffer, the length of the string, and the buffer's capacity in `a`.



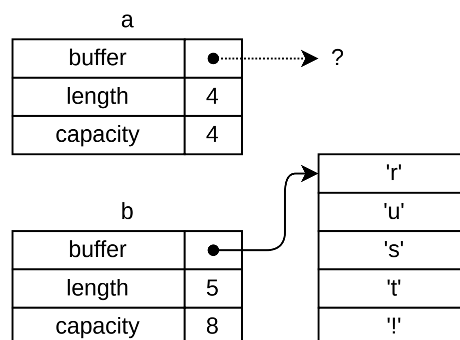
2. `let mut b = a;`

Executing this second line, Rust copies the buffer, length, and capacity of `a` to `b`. Now, both `a` and `b` point to the same buffer.



3. `b.push('!');`

The function `push` is called to add the character `'!'` at the end of `b`. The length is already equal to the capacity, so the buffer is reallocated, which very likely means that the buffer will be put somewhere else in memory, making the old location invalid. Therefore `push` updates `b` to point to the new location.



4. `println!("{}", a);`

Here a problem arises. Printing `a` can cause the program to crash or misbehave in some other way, because `a` points to memory it may not access anymore.

Rust prevents this problem by disallowing the usage of `a` after copying the value of `a` to `b`. Actually, Rust does not call this a copy, but calls it a *move*. Rust will move any value it cannot copy.

But how can we change this program such that Rust does accept it? We have two options: Cloning (explained in section 2.3) or taking a unique reference (explain in section 2.4).

2.3 Clone

Cloning is very similar to copying. It is the explicit version of what C++ does by default. A clone of a heap-allocated resource does not copy the pointer, but it leads to a new heap allocation of an identical resource.

Have a look at the following Rust code:

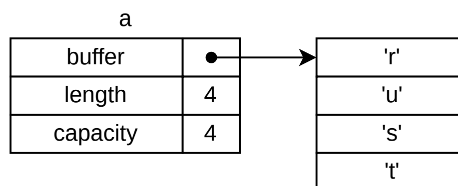
```
let a = String::from("rust");
let mut b = a.clone();
b.push('!');
println!("{}", a);
```

This piece of code is identical to the code in the previous section, but with one difference: the value of `a` is not moved but *cloned*. And while Rust did not accept the previous example, it does accept this one.

We will go through the example line by line.

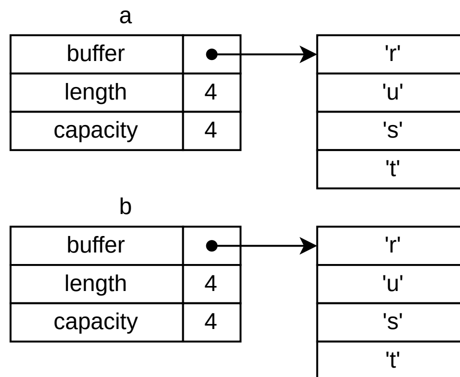
1. `let a = String::from("rust");`

   Again, just like in the example in the previous section, the first line puts the string `"rust"` on the heap.

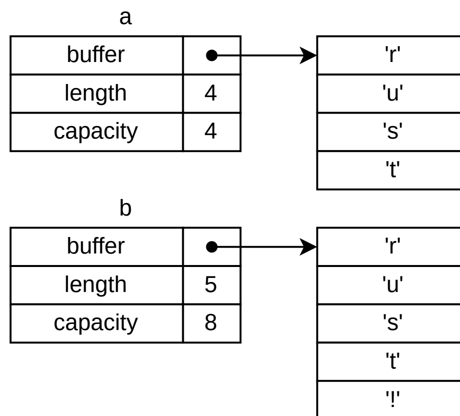
2. `let mut b = a.clone();`

   This second line is where the clone happens. The value of `a` is cloned, and the clone is assigned to `b`.



3.
```
b.push('!');
```

On this line, the character '!' is added to the end of the string value of `b`.



4.
```
println!("{}", a);
```

When this line is executed, the value of `a`, `"rust!"` will be printed.

2.4 Unique reference

Unique references are a way to have full access to a variable without owning the variable, similar to pointers in C++. They are called unique because there may exist at most one unique reference to a particular variable at a time.

Have a look at the following piece of Rust code:

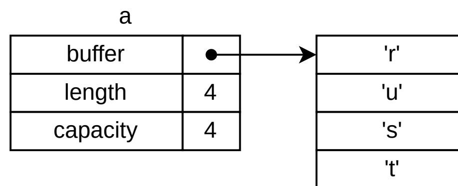
```rust
let mut a = String::from("rust");
let b = &mut a;
b.push('!');
println!("{}", a);
```

In the example above, a unique reference to `a` is assigned to `b`. Then, `a` can be modified through `b`. At the end, `"rust!"` will be printed, because the value of `a` was modified.

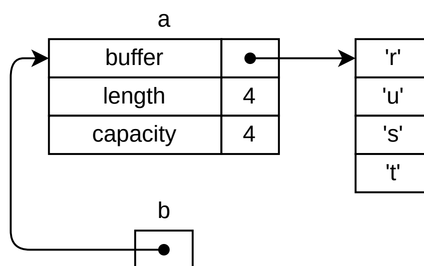
We will walk through the example again line by line:

1. `let mut a = String::from("rust");`

   Exactly like before, the string `"rust"` is placed on the heap.

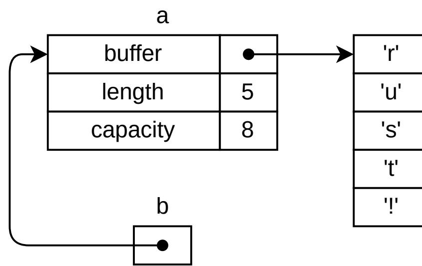


2. `let b = &mut a;`



   When this line is executed, an unique reference to `a` is assigned to `b`.

3. `b.push('!');`



The `push` function is called with the value of `b`, which is a reference to `a`. Inside the `push` function a dereference occurs and the dereferenced value will be changed.

4.
```
println!("{}", a);
```

When this line is executed, the value of `a`, `"rust!"` will be printed.

2.4.1 Lifetimes

The lifetime of a variable in a program is the part of the program where that variable can be used. More specifically, the lifetime begins when the variable is created and it ends when it is destroyed (Rust By Example contributors, 2023). [1]

The lifetimes of a unique reference to a value may not overlap with the lifetime of another reference to that value. So whereas this program is fine:

```
let mut v = vec!['a', 'b'];

let first = &mut v[0]; // Begin lifetime 'first'
*first = 'x';
// End lifetime 'first'

let second = &mut v[1]; // Begin lifetime 'second'
*second = 'y';
// end lifetime 'second'

assert!(v == vec!['x', 'y']);
```

[1] If a variable with a reference is destroyed then that reference is said to *expire*.

The following program is rejected by Rust, because the lifetime of the first unique reference would overlap with the lifetime of the second unique reference:

```
let mut v = vec!['a', 'b'];
let first = &mut v[0];
let second = &mut v[1];
if *first == *second {
    println!("The two characters are equal");
}
```

We can make this example work by using *shared* references, which we discuss in section 2.5.

2.5 Shared reference

Shared references are similar to unique references, but with the difference that there may exist more than one shared reference at a time to the same value. In other words: the lifetimes of shared references to the same variable may overlap.

```
let v = vec!['a', 'b'];
let first = &v[0];
let second = &v[1];
if *first == *second {
    println!("The two characters are equal");
}
```

However, there is a restriction to shared references when compared to unique references: whenever a shared reference to a value exists, that value may not be mutated.

Take for example the following program, which Rust rejects:

```
let mut v = vec!['a', 'b'];
let first = &v[0];
v.push('c');
println!("{}", first);
```

Rust will show the following error:

```
    error[E0502]: cannot borrow `v` as mutable because it is also
    borrowed as immutable

    --> src/main.rs:3:4
     |
   2 |     let first = &v[0];
     |                  - immutable borrow occurs here
   3 |     v.push('c');
     |     ^^^^^^^^^^^ mutable borrow occurs here
   4 |     println!("{}", first);
     |                    ----- immutable borrow later used here
```

It is good that Rust rejects this program, because it does contain a mistake: exactly the same mistake as described earlier in section 2.2.

2.6 Slice

Slices are similar to references, but refer to a contiguous sequence of elements rather than just a single element (Klabnik & Nichols, 2022a).

The following is an example of a slice in Rust:

```
let a = ['a', 'b', 'c', 'd'];
let s = &a[..];
println!("first: {}, length: {}", s[0], s.len());
```

Executing the code will show 'first: a, length: 4'.

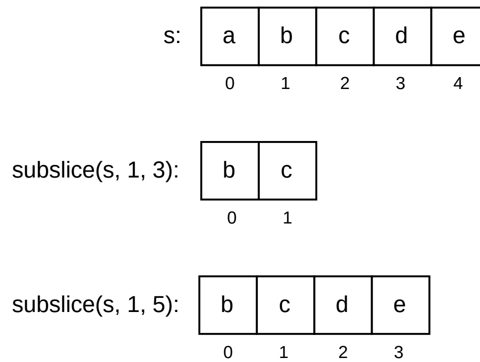
Just like references, slices can be shared or unique. Whereas a shared slice, like the one above, is created with the notation `&s[..]`, a unique slice is created with the notation `&mut s[..]`. Unique slices allow mutation:

```
let mut a2 = ['a', 'b', 'c', 'd'];
let s2 = &mut a2[..];
s2[1] = 'x';
assert!(a2[1] == 'x');
// just like with unique references, s2 may not be used here anymore
```

2.6.1 Subslice

Subslicing is taking a slice and producing a smaller subslice that refers to a part of the original slice.

For example, given the slice `s` from the example above, `&s[1..3]` gives a slice of length 2 that starts at the second element of `s`.



Subslicing can be done on both shared and unique slices. Subslicing a shared slice gives a new shared slice, and subslicing a unique slice gives a new unique slice.

2.7 Panic

While Rust is memory safe and does not have undefined behavior, a Rust program can still get into an unrecoverable error state, called a *panic*.

Important to note is the difference between panics and undefined behavior. Undefined behavior means that anything could happen. In practice, this often results in security issues, crashes, or odd behavior of the program. A panic on the other hand, is clearly defined. The program halts completely, and directly.

There are several ways in Rust to cause such a panic. Here, we will discuss the most common causes of a panic.

2.7.1 Integer overflow

Consider the following example of Rust code:

```rust
fn successor(x: i32) -> i32 {
    x + 1
}
```

Rust accepts this memory-safe program, but it does have a flaw: it results in an integer overflow if `x` is the largest possible value of `i32`. Calling `successor(i32::MAX)` will result in a panic[2].

Another function that might overflow is the following:

```rust
fn negate(x: i32) -> i32 {
    -x
}
```

It will overflow if `x` is the smallest possible value of `i32`, namely `i32::MIN`, because there is no `i32`-value that is the negation of that smallest value.

2.7.2 Integer underflow

The following function has a similar issue:

```rust
fn predecessor(x: i32) -> i32 {
    x - 1
}
```

Calling this function with `i32::MIN` will cause a panic, because `i32::MIN` already is the lowest possible value of `i32`.

2.7.3 Out of bounds index

The following is a valid Rust program:

```rust
fn head(s: &[i32]) -> i32 {
    s[0]
}
```

[2]This is true for programs compiled in Debug mode. If the programs is compiled in Release mode, then an overflow does not cause a panic.

It defines a function `head` that takes a slice of 32-bit signed integers and returns the first of those integers. However, this panics in the case that `s` is empty, because then the first element (at index 0) does not exist.

2.7.4 Assertion

A Rust program might contain an assertion. An assertion always has a boolean condition. If this condition evaluates to `false`, then the program panics.

Programmers can use assertions to add conditions to their program. For example, the following implementation of the fibonacci function is not well defined for negative inputs. Therefore an assertion was added to ensure that the function will not be called with a negative input.

```
fn fib(n: i64) -> i64 {
    assert!(n >= 0);

    if n < 2 {
        n
    }
    else {
        fib(n - 1) + fib(n - 2)
    }
}
```

Chapter 3

Prusti fundamentals

In this chapter we discuss the fundamentals of the Prusti auto-active program verifier.

We first discuss an example Rust program (section 3.1) and use Prusti to prove some properties of this program (section 3.2–3.5). At the end of this chapter we will discuss Prusti's logical syntax 3.6.

The idea to use the `max` function later in this chapter is taken from the 'Basic Usage' section of the Prusti user guide (The Viper Project, 2023a).

3.1 Recurring example

Have a look at the following Rust implementation of the absolute function. It is the recurring example in the following sections.

Note that the absolute function is defined twice, as `abs` (directly) and `abs_composed` (in terms of the `max` function).

```rust
fn max(a: i32, b: i32) -> i32 {
    if a > b {
        a
    } else {
        b
    }
}

fn abs(x: i32) -> i32 {
    if x < 0 {
        -x
    } else {
        x
    }
}

pub fn abs_composed(x: i32) -> i32 {
    max(x, -x)
}
```

We use Prusti to prove the following properties of this program:

1. The functions do not panic as long as some pre-conditions hold. We prove this using Prusti's *pre-conditions* in section 3.2.
2. The result of the `max` function is greater than or equal to both of its parameters. Also, the result of `abs` and of `abs_composed` is always greater than or equal to zero. We prove this using Prusti's *post-conditions* in section 3.3.
3. `abs`, `max` and `abs_composed` are pure: they does not produce side effects and their output is deterministically based on its input. We prove this using Prusti's *pure-annotation* in section 3.4.
4. For any input, `abs_composed` gives the same result as `abs` does for that input. We prove this using a *pure function in a Prusti specification* in section 3.5.

3.2 Pre-condition for panic absence

We want to prove the absence of panics in our example.

Of course, the `abs` function and the `abs_composed` function will panic in one case, namely if `x` is equal to `i32::MIN` (the smallest possible value of `i32`), because that would cause an overflow when negating `x`.

A solution for this is requiring that x may not be `i32::MIN`. That is what we will do. We do this in Prusti by adding a pre-condition to the `abs` and `abs_composed` functions.

Prusti will use pre-conditions in two ways:

1. Prusti verifies that the pre-condition holds whenever the function is called.
2. Prusti may use the pre-condition as an assumption when proving properties of the function.

The syntax of a Prusti pre-condition is very similar to the syntax of a boolean Rust expression. In the expression, the function's parameters may be used. The expression must be written just before the function inside an annotation called `requires`.

To ensure that `x` is not `i32::MIN`, we use the pre-condition `x != i32::MIN`, like this:

```
#[requires(x != i32::MIN)]
fn abs(x: i32) -> i32 {
    if x < 0 {
        -x
    } else {
        x
    }
}

#[requires(x != i32::MIN)]
pub fn abs_composed(x: i32) -> i32 {
    max(x, -x)
}
```

3.3 Post-condition for functional correctness

We want to prove functional correctness of these three functions. In other words, we want that their output always adheres to our specification.

Consider the following specification for `abs`. Firstly, the result of the max function must be greater than or equal to both of its parameters (3.1). Secondly, both the `abs` function and the `abs_composed` function must always give a non-negative result (3.2 and 3.3).

$$\forall x\ y.\ \max(x, y) \geq x \land \max(x, y) \geq y \tag{3.1}$$

$$\forall x.\ \text{abs}(x) \geq 0 \tag{3.2}$$

$$\forall x.\ \text{abs_composed}(x) \geq 0 \tag{3.3}$$

Prusti can verify this specification by itself. We just have to add our specification to the program using a special syntax.

Like with pre-conditions, in Prusti a post-condition is written as a Rust boolean expression. There is one addition: in a post-condition there is a special variable called `result`, which represents the value that the function returns. Also, the name of the post-condition annotation is ‘ensures’.

So in Prusti syntax, our specification looks as follows:

```
#[ensures(result >= x && result >= y)]
fn max(x: i32, y: i32) -> i32 {
    if x > y {
        x
    } else {
        y
    }
}

#[requires(x != i32::MIN)]
#[ensures(result >= 0)]
fn abs(x: i32) -> i32 {
    if x < 0 {
        -x
    } else {
        x
    }
}

#[requires(x != i32::MIN)]
#[ensures(result >= 0)]
```

```
22 pub fn abs_composed(x: i32) -> i32 {
23     max(x, -x)
24 }
```

When proving a specification of a function, Prusti uses:

1. That function's pre-conditions as assumptions,
2. the semantics of the function, and
3. the specifications of other functions.

Prusti proves this automatically. To gain insight in why this works, we now discuss, quite informally, why we should believe that Prusti is right.

Post-condition on the max function

$$\forall x\ y.\ \max(x, y) \geq x \land \max(x, y) \geq y$$

To prove this first part of our specification, we must work with the semantics of the `max` function.

Recall the max function:

```
1 fn max(x: i32, y: i32) -> i32 {
2     if x > y {
3         x
4     } else {
5         y
6     }
7 }
```

For some given `x` and `y`, if `x > y` then in calculating `max(x, y)` the first branch will be taken and `max(x, y) == x`. Trivially, we now know that $x \geq x \land x \geq y$. Substituting using `max(x, y) == x` gives us exactly what we needed to prove: $\max(x, y) \geq x \land \max(x, y) \geq y$.

If it is *not* the case that `x > y` then in calculating `max(x, y)` the second branch will be taken and `max(x, y) == y`. Trivially, we now know that $y \geq x \land y \geq y$. Substituting using `max(x, y) == y` gives us exactly what we needed to prove: $\max(x, y) \geq x \land \max(x, y) \geq y$.

Post-condition on the abs function

$$\forall x.\ \text{abs}(x) \geq 0$$

To prove this first part of our specification, we can use the semantics of `abs` and we can use the precondition of `abs`.

Recall the `abs` function:

```
fn abs(x: i32) -> i32 {
    if x < 0 {
        -x
    } else {
        x
    }
}
```

Looking at this function definition we can see that if `x` is a negative number, then it will be negated, thus giving a positive number. And in the case that `x` already is a positive number, then it remains unchanged.

Therefore we can conclude that indeed `abs(x)` will always be greater or equal to `0`.

Post-condition on the abs-composed function

$$\forall x.\ \text{abs_composed}(x) \geq 0$$

To prove this first part of our specification, we can use the semantics of `abs_composed`, its pre-condition, and the specification of `max`.

Recall the `abs_composed` function:

```
fn abs_composed(x: i32) -> i32 {
    max(x, -x)
}
```

We may use the previously proven specification of the `max` function here, namely:

$$\forall x\ y.\ \max(x, y) \geq x \land \max(x, y) \geq y$$

Using this specification for this specific call gives:

$$\max(x, -x) \geq x \land \max(x, -x) \geq -x$$

Combining this with the fact that either $-x \geq 0$ or $x \geq 0$ lets us conclude that $\max(x, -x) \geq 0$, and therefore $\text{abs_composed}(x) \geq 0$.

3.4 Purity

Definition of purity. A pure function must have these two properties:

1. It does not produce side-effects.
2. Its output is deterministically based on the inputs.

A function that is not pure is called 'impure'.

In Prusti, pure functions can be used inside of pre-conditions and post-conditions. This makes them specifically useful.

Prusti will try to verify a function's purity if the 'pure'-attribute is added to that function. It can prove purity of functions with the following two properties:

1. All functions it calls are pure.
2. All parameter types must be copyable.

Examples of pure functions. `max`, `abs` and `abs_composed` are all pure, as they do not produce side effects and their outputs are deterministically based on their inputs.

After adding the `pure` annotation to them, Prusti can prove their purity.

```
#[pure]
#[ensures(result >= x && result >= y)]
fn max(x: i32, y: i32) -> i32 {
    if x > y {
        x
    } else {
        y
    }
}
```

```
#[pure]
#[requires(x != i32::MIN)]
#[ensures(result >= 0)]
fn abs(x: i32) -> i32 {
    if x < 0 {
        -x
    } else {
        x
    }
}
```

```
#[pure]
#[requires(x != i32::MIN)]
#[ensures(result >= 0)]
pub fn abs_composed(x: i32) -> i32 {
    max(x, -x)
}
```

Examples of impure functions.

- ```
  fn greet() {
      println!("Hello, world!");
  }
  ```

  This function is not pure because it has side-effects: it prints 'Hello, world!'.

  Prusti will reject the purity of this function because it uses an impure function, namely the print function.

- ```
  fn update(x: &mut i32) {
      *x += 1;
  }
  ```

  This function is not pure because it has side-effects: it mutates the value of a variable it does not own[1].

[1] note the *does not own* part. Local mutable variables *are* considered pure by Prusti.

Prusti rejects the purity of this function because the mutable reference does not implement copy.

- ```
  fn roll_die() -> u8 {
      random_byte() % 6 + 1
  }
  ```

  This function is not pure because its result is non-deterministic: calling the function multiple times gives different results.

  Prusti will reject the purity of this function because it uses an impure function, namely the `random_byte` function.

3.5 Equivalence of the abs-functions

We want to prove that `abs_composed` gives the same result as `abs`, for every input. In other words:

$$\forall x\ y.\ \text{abs\_composed}(x) = \text{abs}(x)$$

Prusti can prove this, but only if the abs function and the `abs_composed` functions are pure, because only pure functions may be used in specifications.

The annotated program is as follows:

```
#[pure]
#[requires(x > i32::MIN)]
fn abs(x: i32) -> i32 {
    if x < 0 {
        -x
    } else {
        x
    }
}

#[pure]
fn max(a: i32, b: i32) -> i32 {
    if a > b {
        a
```
```

```
    } else {
        b
    }
}

#[pure]
#[requires(x > i32::MIN)]
#[ensures(result == abs(x))]
pub fn abs_composed(x: i32) -> i32 {
    max(x, -x)
}
```

3.6 Prusti's logical syntax

The following is the same `max` function as described earlier, but with a more complete specification:

```
#[ensures(result == a || result == b)]
#[ensures(result >= a && result >= b)]
fn max(a: i32, b: i32) -> i32 {
    if a > b {
        a
    } else {
        b
    }
}
```

Its specification, described in natural language, is as follows:

1. The result is equal to at least one of the two input values.
2. The result is greater than or equal to both of the input values.

We now define a max-function that does not just work on two values, but on any number of values:

```
fn slice_max(s: &[i32]) -> i32 {
    let n = s.len();

    if n == 1 {
        s[0]
```

```
    } else {
        max(s[0], slice_max(subslice(s, 1, n)))
    }
}
```

We would like to give this function a specification similar to the specification of the two-parameter max-function:

1. The result is equal to at least one of the input values.
2. The result is greater than or equal to all of the input values.

For this we use the `exists` and `forall` quantifiers, and an implication (`==>`):

```
#[requires(s.len() > 0)]
#[ensures(exists(|i: usize| i < s.len() && s[i] == result))]
#[ensures(forall(|i: usize| i < s.len() ==> s[i] <= result))]
fn slice_max(s: &[i32]) -> i32 {
    let n = s.len();

    if n == 1 {
        s[0]
    } else {
        max(s[0], slice_max(subslice(s, 1, n)))
    }
}
```

The following table, directly taken from the Prusti user guide (The Viper Project, 2023b) lists all parts of the syntax. The parts we will use are printed in a bold font:

Syntax	Meaning
`old(...)`	**Value of expression in a previous state**
`... ==> ...`	**Implication**
`... <== ...`	Implication
`... <==> ...`	Biconditional
`... === ...`	**Snapshot equality**
`... !== ...`	Snapshot inequality
`forall(...)`	**Universal quantifier**
`exists(...)`	**Existential quantifier**
`... |= ...`	Specification entailment

Chapter 4

Verification of a key-value store

In this chapter, we take a Rust program and prove its correctness using Prusti. We do this to gain knowledge of the practical use of Prusti to verify the correctness of a non-trivial program.

The Rust program we look at is an implementation of a key-value store.

The product of our work is a specification for this store. This specification must:

- Be automatically verifiable by Prusti;
- Connect to our intuitive idea of how a key-value store should behave;
- Be usable in the verification of code that uses this store.

This chapter is subdivided into six sections:

- *Section 4.1* describes the concept of a key-value store.
- *Section 4.2* describes the Rust implementation: the data structure and the implemented variant of binary search.
- *Section 4.3* introduces an example usage of the implementation, as well as an intuitive description the implementation's specification.
- In *section 4.4* we take a step back and cover the concept of 'pledges'. We take two examples of Rust functions that return a mutable reference,

and learn how ‘pledges’ help in proving their correctness. This concept is used in the remainder of this chapter.

- *Section 4.5* describes the actual specification in Prusti syntax, which we add to our store implementation.

4.1 Store

The Rust program which we will look at is an implementation of a key-value store. A key-value store is a mapping from keys to values, for example:

$$\boxed{\begin{array}{c} a \mapsto x \\ b \mapsto y \\ c \mapsto z \end{array}}$$

The following operations are commonly defined on a store:

- An operation to get for a specific key the corresponding value. For example: ‘what does the key a point to?’ gives x.
- An operation to check whether a key exists in the store. For example: ‘Does the key d exist in the store?’ gives ‘no’.
- An operation to update a value corresponding to a key. For example: ‘update the value of the key b to q’ gives the following store:

$$\boxed{\begin{array}{c} a \mapsto x \\ b \mapsto q \\ c \mapsto z \end{array}}$$

- An operation to add a new key-value pair to the store. For example: ‘add the pair $d \mapsto x$’.

For this case study, we will only implement and verify the first three operations.

4.2 Rust store implementation

4.2.1 Data structure

We encode the key-value store in Rust as a slice of tuples, where each tuple is a key-value pair. The first element of the tuple is the key, and the second

element of the tuple is the value.

This slice of pairs must be sorted based on the key, in increasing order. This is important because we apply binary search on these pairs.

As an example, the following store:

$1 \mapsto$ `'a'`
$3 \mapsto$ `'x'`
$5 \mapsto$ `'y'`

Is encoded in Rust as follows:

```
[(1, 'a'), (3, 'x'), (5, 'y')]
```

Note that we choose to give the keys the type `i32` (a 32-bit signed integer), and the values the type `char` (a character, for example `'a'` or `':'`). There is no particular reason for choosing these types, except for the fact that it is important that the keys can be compared with each other. This is the case for `i32`.

4.2.2 The 'contains' operation

The first operation we implement is the operation to decide if the store contains a certain key. The implementation is as follows:

```rust
pub fn contains(
    pairs: &[(i32, char)],
    key: i32
) -> bool {
    let n = pairs.len();
    if n == 0 {
        return false;
    }

    let mid = n / 2;

    if key < pairs[mid].0 {
        contains(subslice(pairs, 0, mid), key)
    } else if key == pairs[mid].0 {
        true
    } else {
        contains(subslice(pairs, mid + 1, n), key)
    }
}
```

In the base case, `pairs` is the empty slice (`pairs.len()` = 0). The empty slice of course does not contain any key, so we can return `false` .

If `pairs` is not empty, then we first determine the middle of the slice. After that, we compare the value of the key we are looking for with the value of the key that is at the middle.

If the key we are looking for has a smaller value than the one in the middle, then we restrict searching to just the left half of `pairs` , by first creating a slice for the left half, and subsequently calling the same `contains` function on this left half.

If the value of the key we search for matches the value of the key in the middle, then we are done and we can return true, because we know that the store contains the key.

And in the third case, if the value of the key we are looking for is greater than de value of the key in the middle, then we restrict the search to the right half of `pairs` .

For the reader familiar with the Rust programming language, it might stand out that the `subslice` function used here is not a standard Rust function, and

that `pairs[mid..]` could be written instead of `subslice(pairs, 0, mid)`, as well as `pairs[mid+1..]` instead of `subslice(pairs, mid+1, n)`. However, at the moment of writing, the built in Prusti specification for this notation is not strong enough for us to use later in this chapter to prove the specification we want to prove. That is why we use this `subslice` wrapper function, which we will give our own, trusted (meaning unverified by Prusti), specification.

`subslice` is defined as follows:

```
#[trusted]
#[requires(start <= end && end <= s.len())]
#[ensures(result.len() == end - start)]
#[ensures(result.len() <= s.len())]
#[ensures(forall(|i: usize|
    i < result.len() ==> result[i] === s[start + i]))]
#[ensures(forall(|i: usize|
    start <= i && i < end ==> s[i] === result[i - start]))]
fn subslice<T>(s: &[T], start: usize, end: usize) -> &[T] {
    &s[start..end]
}
```

4.2.3 The 'get' operation

The second operation we implement is the operation to get, for a certain key, the corresponding value.

This function too uses the just described `subslice` function.

```
pub fn get(
    pairs: &[(i32, char)],
    key: i32
) -> &char {
    let mid = pairs.len() / 2;

    if key < pairs[mid].0 {
        get(subslice(pairs, 0, mid), key)
    } else if key == pairs[mid].0 {
        &pairs[mid].1
    } else {
        get(subslice(pairs, mid + 1, pairs.len()), key)
    }
}
```

Important to notice here is that, unlike the `contains` function, `get` does not check if the slice is empty. This is because `get` assumes that the store contains the key. We will later in this chapter convert that assumption to a pre-condition.

When the key is found, a reference will be returned to the corresponding value. The notation `t.1`, where `t` is a tuple, means: take the value in the tuple on index 1, 0-indexed, so the second value.

4.2.4 The 'get-mut' operation

The third operation, get-mut, is very similar to the 'get' operation, with the difference that 'get-mut' returns a *mutable* reference, meaning that the user of 'get-mut' can change the value of the variable it refers to, effectively updating the store.

'get-mut' is implemented as follows:

```
pub fn get_mut(
    pairs: &mut [(i32, char)],
    key: i32
) -> &mut char {
    let n = pairs.len();
    let mid = n / 2;

    if key < pairs[mid].0 {
        get_mut(subslice_mut(pairs, 0, mid), key)
    } else if key == pairs[mid].0 {
        snd_mut(index_mut(pairs, mid))
    } else {
        get_mut(subslice_mut(pairs, mid + 1, n), key)
    }
}
```

In the same manner as with the `subslice` function, where we wrapped a standard Rust functionality, we do that here with the `subslice_mut` function (the mutable counterpart of `subslice`), `index_mut`, and `snd_mut`. Again, we do this to be able to add an extra (sometimes trusted) specification to these functions, because at the time of writing Prusti does not give us strong enough of a specification.

These three functions are implemented as follows:

```
#[trusted]
#[requires(index < s.len())]
#[ensures(*result === old(&s)[index])]
#[after_expiry(
        s.len() == old(&s).len()
    &&
        forall(|i: usize|
                i < s.len() && index != i
            ==>
                s[i] === old(&s)[i])
    &&
        s[index] === *before_expiry(result)
)]
fn index_mut<T>(s: &mut [T], index: usize) -> &mut T {
    &mut s[index]
}
```

```
#[ensures(*result === old(&tuple).1)]
#[after_expiry(
        tuple.0 === old(&tuple).0
    &&
        tuple.1 === *before_expiry(result)
)]
fn snd_mut<T, U>(tuple: &mut (T, U)) -> &mut U {
    &mut tuple.1
}
```

```rust
#[trusted]
#[requires(start <= end && end <= s.len())]
#[ensures(result.len() == end - start)]
#[ensures(forall(|i: usize|
    i < result.len() ==> result[i] === old(&s)[start + i]))]
#[ensures(forall(|i: usize|
    start <= i && i < end ==> old(&s)[i] === result[i - start]))]
#[after_expiry(
        s.len() == old(&s).len()
    &&
        // The values outside the start..end range don't change
        forall(|i: usize|
                (i < start || (end <= i && i < s.len()))
            ==>
                s[i] === old(&s)[i])
    &&
        // The values inside the start..end range are equal to
        // the full result range
        forall(|i: usize|
                start <= i && i < end
            ==>
                s[i] === before_expiry(result)[i - start])
)]
fn subslice_mut<T>(
    s: &mut [T],
    start: usize,
    end: usize
) -> &mut [T] {
    &mut s[start..end]
}
```

4.3 Example and intuitive specification

We now look at an example of code using our key-value store. We do this for two reasons:

1. To gain insight in how the store is used and how it should behave.
2. To be able to test the end product, the specification of the store implementation, against this example. Prusti must be able to verify this example. This will give us more certainty that the specification is sensible and that

other usages of the store implementation could be verified as well.

In our example, we use each operation at least one. We add multiple assertions with our ideas of how this store should work.

Note that this example is arbitrary, made such that it meets these requirements. On its own, it is not an interesting program.

This is the example:

```
let mut pairs_array = [(0, 'a'), (2, 'c'), (10, 'd')];
let pairs = as_mut_slice(&mut pairs_array);

assert!(contains(pairs, 2));
assert!(!contains(pairs, 3));

let r0 = get(pairs, 0);
assert!(*r0 == 'a');

let r1 = get_mut(pairs, 2);
assert!(*r1 == 'c');
*r1 = 'h';

let r2 = get_mut(pairs, 0);
assert!(*r2 == 'a');

let r3 = get(pairs, 2);
assert!(*r3 == 'h');
```

4.3.1 Intuitive specification

Based on knowledge of the implementation, expectations of a key-value store, and the example usage, we now draw up an intuitive specification.

Contains

For the contains function to work properly, it requires that the pairs it is given are incrementally sorted by key.

We expect the contains function to return true if and only if there exists an element in the slice that has a key that matches the key value given to the contains function.

Get

For the get function to work properly, the pairs must be sorted by key, incrementally. Also, the key value that is searched for must exist in the pairs. Finally, the keys must be unique.

We expect the value of the result to be the value that corresponds to the key in the store.

Get-mut

For the get-mut function to work properly, the pairs must be sorted by key, incrementally. Also the key value that is searched for must be in the pairs. Finally, the keys must be unique.

We expect the value of the result to be the value that corresponds to the key, in the pairs.

We also expect the get-mut function to not change any key in the store.

Finally, we expect that the return value is a mutable reference to the value that corresponds to the key. So if the returned mutable reference is used to mutate its referent, and update it to some value x, we then expect that the pair in the store with a matching key now has the value x.

4.4 Pledges

We take a step back from our key-value store to cover the concept of 'pledges', because we need this concept later in the chapter.

The challenge here is the mutability of references to a piece of a larger reference. We must include in the specification how changes in the returned mutable reference influence the original, input, larger reference.

Pledges allow us to draw up such a specification.

4.4.1 Snd

Take for example the wrapper-function `snd_mut`, which takes a mutable reference to a tuple and returns a mutable reference to the second element of that tuple:

```rust
fn snd_mut<T, U>(tuple: &mut (T, U)) -> &mut U {
    &mut tuple.1
}
```

Here is an example of using the `snd_mut` function:

```rust
let mut tuple = ('a', 2);

let r = snd_mut(&mut tuple);
*r += 1;

assert!(tuple.0 == 'a');
assert!(tuple.1 == 3);
```

But which specification must we give to `snd_mut` to verify this example?

The first important thing to include in the specification is that the returned value matches the value in the tuple. Pledges are not needed for this. The following post-condition suffices:

```rust
#[ensures(*result === old(tuple).1)]
```

Note the usage of the function `old`, which is part of the Prusti notation. Here, it is needed because we want to refer to the value of the tuple as it was given to the function, not the value as it is now.

But now how do we prove that the two assertions succeed? For that we must include in our specification how modifying the value the result references, influences the value of the full tuple. We do this with a pledge:

```rust
#[after_expiry(
        tuple.0 === old(&tuple).0
    &&
        tuple.1 === *before_expiry(result)
)]
```

This says that *after the expiry of the returned value*, the first value of the tuple is unchanged, and the second value of the tuple is equal to the value that the returned reference referred to *just before it expired*.

This is sufficient to prove the example usage of `snd_mut`.

4.4.2 Index

Another function in which we use pledges is `index_mut`. This function takes a mutable slice and an index in that slice, and then gives back a mutable reference to the element on that index.

The implementation is as follows:

```
fn index_mut<T>(s: &mut [T], index: usize) -> &mut T {
    &mut s[index]
}
```

It is simply a wrapper around the standard Rust indexing. We create this wrapper because Prusti currently does not seem to support such a mutable indexing function. We give this wrapper a trusted specification.

This `index_mut` can be used to read or update an element from a slice. This is an example of that:

```
let mut array = [15, 16, 18];
let s = as_mut_slice(&mut array);

let r = index_mut(s, 1);
assert!(*r == 16);
*r += 1;

assert!(s[0] == 15);
assert!(s[1] == 17);
assert!(s[2] == 18);
```

Which specification must be added to `index_mut` to be able to verify this example?

Firstly, the precondition `index < s.len()` must be added. This pre-condition is needed to prevent a panic due to out-of-bounds indexing:

```
#[requires(index < s.len())]
```

Secondly, for the `assert!(*r == 16)` to succeed, a post-condition must be added that states that the value of the returned reference matches the value on the index location:

```
#[ensures(*result === old(&s)[index])]
```

Finally, Prusti's concept of pledges is needed to prove that the last three assertions succeed. To reiterate: with an `after_expiry` we give a specification that says something about the mutable reference arguments directly after the returned mutable reference expires (also see section 2.4.1 on lifetimes).

Without an `after_expiry`, Prusti cannot say anything about `s` after the returned borrow expires. The only information it then has, is that it has the type 'slice of T'. In other words: after expiry, the original slice `s` can be anything.

So the following must be specified with regard to `s`:

1. The length is the same as before `index_mut` was called;
2. Each element on an index other than `index` has the value it had before `index_mut` was called;
3. On the index `index` is now the value the returned reference referred to just before it expired.

Translated to Prusti syntax:

```
#[after_expiry(
        s.len() == old(&s).len()
    &&
        forall(|i: usize|
                i < s.len() && index != i
            ==>
                s[i] === old(&s)[i])
    &&
        s[index] === *before_expiry(result)
)]
```

This specification is enough for Prusti to prove the example, and will later be a sufficient specification when verifying the complete specification of the store.

4.5 Prusti specification

In this section we discuss the specification in Prusti syntax, which we add to our store implementation. We first introduce predicates (section 4.5.1) which will help in specification for `contains` (section 4.5.2), `get` (section 4.5.3) and `get_mut` (section 4.5.4).

4.5.1 Predicates

In this section we take the intuitive specification as we described earlier (section 4.3) and convert it into pre-conditions, post-conditions and pledges, notated in Prusti syntax.

First we define three predicates. These will make the remainder of the specification more readable, because their usage will reduce unnecessary repetition. The usage of predicates will also make it more clear how the intuitive specification matches the specification in Prusti syntax.

Increasing

The first predicate is 'increasing'. Simply said: a slice of pairs is increasing if it is sorted in increasing order of keys. Or more formally: A sequence of key-value pairs is increasing if and only if every pair x occurring before (to the left) of some pair y, it holds that the key of x is smaller than the key of y.

Note that for every sequence of increasing pairs, it also holds that the keys are unique. Because if there would be two of the same keys then of course it would not hold that one of these keys is smaller than the other.

In Prusti, the predicate is defined as follows:

```
predicate! {
    fn increasing(
        pairs: &[(i32, char)],
    ) -> bool {
        forall(|i: usize, j: usize|
            (i < j && j < pairs.len() ==> pairs[i].0 < pairs[j].0))
    }
}
```

The attentive reader might notice that this definition deviates from the usual

definition, where every two adjacent elements are compared with each other.

That definition is logically equivalent to the first definition. The reason for using the first definition is a practical one. Prusti *seems* to more easily verify specifications with this definition.

Has key

The second predicate is called `has_key`. A sequence of pairs *has a certain key* if it contains a pair with that key. More formally: A sequence `pairs` has a certain key if and only if there exists an index smaller than the length of `pairs` for which on that index there exists a pair with a matching key.

In Prusti, the predicate is defined as follows:

```
predicate! {
    fn has_key(
        pairs: &[(i32, char)],
        key: i32
    ) -> bool {
        exists(|i: usize|
            i < pairs.len() && pairs[i].0 === key,
            triggers=[(pairs[i],)])
    }
}
```

The added 'triggers' help Prusti in verification and are not logically adding anything to the specification. We are not entirely clear on what triggers do, because they are not mentioned in Prusti's documentation. But they are certainly essential here. In essence they seem to help in finding a particular value for the `exists`.

Maps

The third and last predicate is called `maps`. We say that a slice of pairs *maps* the key k to the value v if every pair in the slice with a key that matches k, has a value that matches v. Or, more formally said: a slice of pairs maps the key k to the value v if and only if for every index `i` smaller than the amount of pairs, it holds that if the key of the pair on index `i` matches the key `k`, then the value of the pair on index `i` matches `v`.

This definition might seem to weak, because it does not require a matching key to be in the pairs at all. Also, it does not require key uniqueness. However, this predicate will be combined with the predicates `increasing` (providing uniqueness) and `has_key`. Having these requirements in 'maps' as well would be redundant.

In Prusti, the predicate is defined as follows:

```
predicate! {
    fn maps(
        pairs: &[(i32, char)],
        key: i32,
        value: char
    ) -> bool {
        forall(|i: usize|
                i < pairs.len() && pairs[i].0 === key
            ==>
                pairs[i].1 === value)
    }
}
```

4.5.2 Specification of `contains`

Recall our intuitive specification of contains as described in section 4.3.

We stated that for the function to work properly, it is essential that the pairs are sorted, increasingly [1], by key. We add this requirement in the form of a pre-condition.

```
#[requires(increasing(pairs))]
```

We also stated that the contains function must return true if and only if there is a pair for which the key matches. We add this requirement as a post-condition.

```
#[ensures(result == has_key(pairs, key))]
```

[1] Note that requiring the pairs are increasing is stronger than necessary here. Nondecreasingness (sorted, but may contain duplicates) would suffice. We still use `increasing`, because a well-formed store should not contain any duplicates anyway.

This concludes the specification of `contains`.

```rust
#[requires(increasing(pairs))]
#[ensures(result == has_key(pairs, key))]
pub fn contains(
    pairs: &[(i32, char)],
    key: i32
) -> bool {
    let n = pairs.len();
    if n == 0 {
        return false;
    }

    let mid = n / 2;

    if key < pairs[mid].0 {
        contains(subslice(pairs, 0, mid), key)
    } else if key == pairs[mid].0 {
        true
    } else {
        contains(subslice(pairs, mid + 1, n), key)
    }
}
```

4.5.3 Specification of `get`

Recall from the intuitive specification that for `get` to work properly, the pairs must be increasingly sorted, by key. Also, the keys must be unique. Futhermore, the key searched for must be in the pairs. We convert those requirements to the following Prusti pre-condition:

```rust
#[requires(increasing(pairs))]
#[requires(has_key(pairs, key))]
```

We also stated that the result should be a reference to a value that belongs to the key. This can be converted into the following post-condition:

```rust
#[ensures(maps(pairs, key, *result))]
```

This is the complete specification of `get`:

```rust
#[requires(increasing(pairs))]
#[requires(has_key(pairs, key))]
#[ensures(maps(pairs, key, *result))]
pub fn get(
    pairs: &[(i32, char)],
    key: i32
) -> &char {
    let mid = pairs.len() / 2;

    if key < pairs[mid].0 {
        get(subslice(pairs, 0, mid), key)
    } else if key == pairs[mid].0 {
        &pairs[mid].1
    } else {
        get(subslice(pairs, mid + 1, pairs.len()), key)
    }
}
```

4.5.4 Specification of `get_mut`

Recall from the intuitive specification that the specification of `get_mut` is very similar to that of `get`, but with some extra requirements, regarding mutability.

We first copy the specification of `get`:

```rust
#[requires(increasing(pairs))]
#[requires(has_key(pairs, key))]
#[ensures(maps(old(pairs), key, *result))]
```

The specification above does not exactly match the specification of `get`: it contains the expression `old(pairs)` where the specification of `get` contains just `pairs`. This is because pairs is now a mutable reference, and its referent could be changed in the meantime.

Now, we have yet to extend the specification to say something about the value of `pairs` directly after the expiry of the returned mutable reference. For this we need a pledge in Prusti.

We can say about `pairs` the following at the moment the returned borrow expires:

1. The length of `pairs` is the same as before `get_mut` was called;
2. All keys are the same as before `get_mut` was called;
3. All values for which the corresponding key does not match are the same as before `get_mut` was called;
4. The pair for which the key matches now has a value that is equal to the value of the returned reference just before it expired.

These four properties can be written in Prusti as follows:

```
#[after_expiry(
        pairs.len() == old(pairs.len())
    &&
        // Keys don't change
        forall(|i: usize|
                i < pairs.len()
            ==>
                pairs[i].0 === old(&pairs)[i].0)
    &&
        // Pairs with non-matching keys don't change
        forall(|i: usize|
                i < pairs.len() && pairs[i].0 != key
            ==>
                pairs[i] === old(&pairs)[i])
    &&
        maps(pairs, key, *before_expiry(result))
)]
```

And with that we complete the specification of `get_mut`:

```
#[requires(increasing(pairs))]
#[requires(has_key(pairs, key))]
#[ensures(maps(old(pairs), key, *result))]
#[after_expiry(
        pairs.len() == old(pairs.len())
    &&
        // Keys don't change
        forall(|i: usize|
                i < pairs.len()
            ==>
                pairs[i].0 === old(&pairs)[i].0)
```

```
12      &&
13          // Pairs with non-matching keys don't change
14          forall(|i: usize|
15                  i < pairs.len() && pairs[i].0 != key
16              ==>
17                  pairs[i] === old(&pairs)[i])
18      &&
19          maps(pairs, key, *before_expiry(result))
20  )]
21  pub fn get_mut(
22      pairs: &mut [(i32, char)],
23      key: i32
24  ) -> &mut char {
25      let n = pairs.len();
26      let mid = n / 2;
27
28      if key < pairs[mid].0 {
29          get_mut(subslice_mut(pairs, 0, mid), key)
30      } else if key == pairs[mid].0 {
31          snd_mut(index_mut(pairs, mid))
32      } else {
33          get_mut(subslice_mut(pairs, mid + 1, n), key)
34      }
35  }
```

4.6 Conclusion

Returning to the implementation's usage example (section 4.3), we see that Prusti proves its correctness, using the specification of the key-value store.

The specification we created connects to our intuition, and is easily readable thanks to the use of predicates.

The proof rests on several trusted specifications that had to be added because, at the time of writing, Prusti does not provide them. So if we trust Prusti to convincingly prove the correctness of the program, just these assumptions have to be manually checked with great care.

Chapter 5

Prusti limitations

In this chapter we discuss a list of limitations encountered while creating the case study. For each of the limitations, we discuss both the problem and possible workarounds or solutions.

The limitations are split into two categories. The first is all limitations related to borrowing (section 5.1), the second is all other limitations (section 5.2).

5.1 Borrowing

5.1.1 Borrow splitting

Recall from chapter 4 the `snd_mut` function:

```
#[ensures(*result === old(tuple).1)]
#[after_expiry(
        tuple.0 === old(&tuple).0
    &&
        tuple.1 === *before_expiry(result)
)]
fn snd_mut<T, U>(tuple: &mut (T, U)) -> &mut U {
    &mut tuple.1
}
```

`snd_mut` takes a unique reference to a tuple of values and returns a unique reference to the second value. This worked in chapter 4 for the specific use case. But it would be much nicer to use a more general and reusable function

instead. Namely a function that takes a unique reference to a tuple of values and returns a tuple of two unique references, one for each value.

This is possible in Rust:

```rust
fn split_mut<T, U>(tuple: &mut (T, U)) -> (&mut T, &mut U) {
    (&mut tuple.0, &mut tuple.1)
}
```

Unfortunately, Prusti does not support Rust's borrow splitting. Borrow splitting is taking a reference and splitting it into separate references such that they do not overlap in the reachable memory.

Upon trying to verify this example, Prusti gives the following error: '[Prusti: unsupported feature] the encoding of pledges does not support this kind of reborrowing'.

Even giving the function a trusted specification does not help. The following example fails with a Prusti error as well: 'Details: Type Tuple([&mut char, &mut i32]) can not be dereferenced'.

```rust
#[trusted]
#[ensures(
        *result.0 === old(&tuple).0
    &&
        *result.1 === old(&tuple).1
)]
#[after_expiry(
        tuple.0 === *before_expiry(result.0)
    &&
        tuple.1 === *before_expiry(result.1)
)]
fn split_mut<T, U>(tuple: &mut (T, U)) -> (&mut T, &mut U) {
    (&mut tuple.0, &mut tuple.1)
}

fn test_it() {
    let mut pair = ('a', 2);
    let (l, r) = split_mut(&mut pair);
    *l = 'b';
    assert!(pair.0 == 'b');
}
```

The workaround for this problem is to change the Rust program. A possibility is to change it such that just one sub-reference is taken, and then another, instead of splitting into two references. That is what we did in chapter 4. Another form of this is to have a 'getter' and a 'setter' function to update the values instead of splitting.

5.1.2 Structs with reference fields

Prusti also does not support structs with reference fields. The most common example of that is a reference wrapped in an `Option` type. Take for example the following function on which Prusti fails, although it is valid and idiomatic Rust code:

```
pub fn first(s: &[i32]) -> Option<&i32> {
    if s.len() == 0 {
        None
    } else {
        Some(&s[0])
    }
}
```

A workaround for this particular problem, where optionally a reference is returned, is to change the Rust function to always return the reference. A pre-condition can then be added to guard against the case where a `None` type would have been returned:

```
use prusti_contracts::*;

#[requires(s.len() > 0)]
pub fn first(s: &[i32]) -> &i32 {
    &s[0]
}
```

5.1.3 Two-phase borrows

Prusti does not support two-phase borrows. Take for example the following valid piece of Rust code:

```rust
subslice_mut(pairs, 0, pairs.len());
```

Two-phase borrows are a ‘more permissive version of mutable borrows that allow nested method calls. Such borrows first act as shared borrows in a “reservation” phase and can later be "activated" into a full mutable borrow’ (Rust contributors, 2018).

To understand two-phase borrows, it helps to first look at an expanded form of the function calls in the above example:

```rust
subslice_mut(&mut pairs, slice::len(&pairs));
```

Two references are taken of `pairs`, of which one is a unique reference. This expanded form is therefore rejected by Rust’s borrow checker.

But in this specific case, the usage is safe, because of Rust’s evaluation order (Rust contributors, 2023a): the argument, and therefore the `slice::len` function is evaluated *before* the `subslice_mut` function is called. To handle this, Rust internally uses two-phase borrows.

Prusti will say about the first example that ‘Two-phase borrows are not supported’. The workaround for this problem is to bind variables to a name, like this:

```rust
let mut v = vec![];
let n = v.len();
v.push(n);
```

5.1.4 Mutable slice indexing

Prusti does not support mutable slice indexing. For example, Prusti can not verify the following program:

```rust
#[requires(s.len() > 0)]
pub fn head(s: &mut [i32]) -> &mut i32 {
    &mut s[0]
}
```

The workaround is to wrap the mutable slice index function into a trusted function and give it a specification. The trusted function can then be used instead.

```
#[trusted]
#[requires(index < s.len())]
#[ensures(*result == old(s[index]))]
#[after_expiry(
        s.len() == old(s.len())
    &&
        forall(|i: usize|
            (i < s.len() && index != i) ==> s[i] == old(s[i]))
    &&
        s[index] == before_expiry(*result)
)]
pub fn index_mut(s: &mut [i32], index: usize) -> &mut i32 {
    &mut s[index]
}
```

A more permanent solution would be to include this specification in Prusti itself.

5.2 Other limitations

5.2.1 usize guarantees

In Rust, the size of the `usize` integer type is 'how many bytes it takes to reference any location in memory' (Rust contributors, 2023b). Therefore, there will never be an integer overflow in the following code:

```
enum List<T> {
    Nil,
    Cons(T, Box<List<T>>),
}

impl<T> List<T> {
    fn len(&self) -> usize {
        match self {
            List::Nil => 0,
            List::Cons(_, next) => {
```

```
                // overflow will never happen here, but
                // how to convince prusti of that?
                next.len() + 1
            }
        }
    }
}
```

But Prusti is not able to prove this.

This problem is not specific to Prusti. Extensive formal semantics of C and Rust in proof assistants, like CompCert (Leroy & Blazy, 2008), RustBelt (Jung et al., 2018a), and the semantics of C by Krebbers (Krebbers, 2015), assume an unbounded memory model, i.e., an infinite address space. This means that these semantics allow for lists that are bigger than `usize::MAX` so one cannot prove that overflow does not happen in this example.

5.2.2 Multiple after-expiries

Prusti fails with an 'unexpected error' when trying to verify a program with multiple `after_expiry` annotations on the same function, like this one:

```
#[after_expiry(true)]
#[after_expiry(true)]
fn id(x: &mut i32) -> &mut i32 {
    x
}
```

The above should not fail, or at least a descriptive error message should be shown.

The fix is to write just one `after_expiry`, possible with multiple statements separated by a conjunction (`&&`).

A GitHub issue was created for this problem [1].

[1] https://github.com/viperproject/prusti-dev/issues/1190

5.2.3 Assert that false cannot be proved

While writing a trusted specification, it is very useful to temporarily insert an `assert!(false)` into the program to check if the verification with Prusti now fails. If it does not fail, then we know that the specification contained an error: for example, the precondition might be contradictory.

An ergonomic improvement would be to have a statement that achieves this that does not have to be removed.

5.2.4 Trigger documentation

A first time Prusti user might find out that to prove some statements in Prusti, it seems to be necessary to add a `prusti_assert` that exactly repeats a known fact. For example, Prusti fails to verify the following example:

```
1  #[requires(pairs.len() > 0)]
2  #[ensures(result ==> exists(|i: usize| pairs[i].0 == key))]
3  pub fn head_match_first(pairs: &[(i32, char)], key: i32) -> bool {
4      if pairs[0].0 == key {
5          true
6      } else {
7          false
8      }
9  }
```

But, perhaps surprisingly, just one minor change fixes that:

```
1  #[requires(pairs.len() > 0)]
2  #[ensures(result ==> exists(|i: usize| pairs[i].0 == key))]
3  pub fn head_match_first(pairs: &[(i32, char)], key: i32) -> bool {
4      if pairs[0].0 == key {
5          prusti_assert!(pairs[0].0 == key);
6          true
7      } else {
8          false
9      }
10 }
```

It turns out that this is due to Prusti automatically choosing a *trigger*. This choice was influenced by the `prusti_assert`.

The fix is to manually set a trigger, such that the `exists` line looks like this:

```
#[ensures(
        result
    ==>
        exists(|i: usize| pairs[i].0 == key, triggers=[(pairs[i],)])
)]
```

However, this is not documented. A GitHub issue exists for this [2].

5.2.5 Missing termination check on pure functions

We can write a pure function in Prusti that does not terminate. Take for example this function, which Prusti accepts as a good pure function:

```
#[pure]
pub fn nonterminating_pure() -> i32 {
    nonterminating_pure()
}
```

We can now add an `ensures(..)` about the result:

```
#[pure]
#[ensures(result == 10)]
pub fn nonterminating_pure() -> i32 {
    nonterminating_pure()
}
```

This is fine, because the `ensures` is about partial correctness, and the function never terminates.

Going further, we can `ensure` *any* partial correctness property of this function:

```
#[pure]
#[ensures(result == 10)]
#[ensures(result == 8)]
#[ensures(false)]
```

[2]https://github.com/viperproject/prusti-dev/issues/1285

```
pub fn nonterminating_pure() -> i32 {
    nonterminating_pure()
}
```

We can then use this `nonterminating_pure` in another proof (here for `foo`), and prove anything there as well:

```
#[pure]
#[ensures(result == 10)]
#[ensures(result == 8)]
#[ensures(false)]
pub fn nonterminating_pure() -> i32 {
    nonterminating_pure()
}

#[ensures(false)]
#[ensures(nonterminating_pure() == 7)]
pub fn foo() {
}
```

The example above is problematic. No assumptions were explicitly introduced. Yet, we can add `ensures(false)` to `foo`.

This could be prevented by requiring that `pure` functions must be proven to terminate, but Prusti currently has no such check.

Chapter 6

Related work

In this chapter, we discuss related work. The focus lies on verification tools similar to Prusti. This means that we will mostly discuss verifiers for Rust programs, and specifically static analysis tools.

This chapter is partly based on an online list of Rust verification tools ("Rust verification tools", 2021).

6.1 Rust verification

6.1.1 Dynamic verification tools

Dynamic verification tools run the code under inspection in the verification process.

MIRI is an interpreter for Rust's intermediate representation (MIRI contributors, 2023). It checks for undefined behavior while running. MIRI can, for example, detect out-of-bound memory accesses, use-after-free, and invalid usages of uninitialized data.

Loom (loom contributors, 2023) and *Shuttle* (shuttle contributors, 2023) are both dynamic concurrency checkers. They can be used to find bugs related to concurrency. They work by controlling the scheduling of each thread. Loom exhaustively checks all thread scheduling permutations, while trying to reduce the search space. Shuttle, on the other hand, checks randomly selected permutations guided by heuristics, trading soundness for scalability.

Proptest is a property testing framework (Proptest contributors, 2023). It can

be used to test, for arbitrary inputs, that certain user-defined properties hold. When it finds an input for which the program crashes or the property does not hold, it automatically finds the minimal test case to reproduce the failure.

6.1.2 Static verification tools

Static verification tools, like Prusti, do not run the code under inspection in the verification process, but rather statically analyze it. They generally provide more assurances than dynamic verification tools, but may require more expertise to use.

A common architecture for program verifiers is to first generate a set of verification conditions, and then process them (Leino & Moskal, 2010).

This notion helps us categorize program verifiers based on nature of user-interaction. We distinguish three categories (Leino & Moskal, 2010):

1. Automatic: the user does not provide input to guide the program verifier;
2. Auto-active: the user provides guiding input before the generation of the verification conditions.
3. Interactive: the user provides guiding input after the generation of the verification conditions;

Automatic

Automatic program verifiers just take the source code as input, possibly with an added specification. The tool then attempts to verify the code's correctness without any guidance from the user.

Clippy (Clippy contributors, 2023c) is a linter for Rust that finds known, common mistakes in the source code (Clippy contributors, 2023a). It is a very popular tool, and it already includes more than 550 lints (Clippy contributors, 2023b).

KANI is a model checker, 'particularly useful for verifying unsafe code in Rust' (*The Kani Rust Verifier*, 2023). It verifies memory safety of unsafe code, absence of unexpected behaviour like integer overflows, and the absence of panics (including failing user-specified statements). The user must write a 'proof harness', similar to a test, but with the difference that it can be used to

check a function's correctness for *all* valid inputs instead of some particular input.

Crux-mir is a tool for static symbolic execution of Rust code (Galois, Inc., 2022). Just like KANI it also tries to prove that a test passes for all valid inputs, and not just for some particular inputs. It seems to have less of a focus on unsafe code than KANI does.

Crust can check unsafe code. It 'combines exhaustive test generation and bounded model checking' (Toman et al., 2015). The Crust project is no longer maintained (Toman et al., 2022).

MIRAI is 'an abstract interpreter for the Rust compiler's mid-level intermediate representation (MIR)' (Facebook, Inc., 2023). It can both check for panics, and it can check for functional correctness properties, added as annotations to the Rust program.

Auto-active

Generally, auto-active program verifiers do not just allow the user to add a specification to the source code, but also allow the user to add hints to the code to guide the tool.

Right now there are two big auto-active program verifiers for Rust: Prusti, which is the subject of this report, and Creusot.

Creusot (Denis et al., 2022) is an auto-active verification tool. Like Prusti, it allows the user to annotate the Rust program with a specification, including for example loop invariants. Creusot takes such an annotated program and translates it to the Why3 language. *CreuSAT* (Skotåm, 2022) is a verified SAT solver. It is a big example of Creusot used in practice.

Interactive

Interactive program verifiers are guided by the user in the process of verification. This guidance is often in the form of the user choosing tactics.

Electrolysis translates Rust programs to definitions in the Lean theorem prover (de Moura et al., 2015; Ullrich, 2016, 2017a, 2017b).

Aeneas (Ho & Protzenko, 2022) translates Rust programs into a pure lambda calculus. This functionally equivalent program can then be proved in a theo-

rem prover of the users choice. Right now only an F* backend is implemented, and a Coq backend is in the works. Aeneas explicitly does not focus on proving unsafe-blocks, it “shines on non-unsafe Rust programs, and can be complemented by more advanced tools such as RustBelt for unsafe parts.” Aeneas also does not tackle interior mutability, leaving that to future work. In Aeneas, no annotations are added to the Rust code. Instead, it expects the user to guide the theorem prover if needed after translation.

Rustbelt (Jung et al., 2018b) provides a ‘formal (and machine-checked) safety proof for a language representing a realistic subset of Rust.’ The proof can be used to verify the correctness of Rust libraries using unsafe features.

RustHornBelt is ‘the first machine-checked proof of soundness for RustHorn-style verification’ (Matsushita et al., 2022). It uses and extends RustBelt’s model to reason about both safety and semantics.

6.2 Verification for other low-level languages

Rust provides a lot of safety guarantees by default. For this reason, Rust program verification can omit the checks for some program properties and even use them as assumptions. Therefore, the verification of programs is more involved for other low-level languages that do not have these safety guarantees that Rust provides by default, like C.

6.2.1 Verifast

Verifast (Jacobs & Piessens, 2008) is an auto-active verification tool for C and Java. It is based on separation logic (Reynolds, 2002), and forward symbolic execution. To use it, users must add annotations to the source program. There are many examples of its usage (Jacobs, 2011), including the verification of a Java Chat Server (Cuypers et al., 2009), and the verification of cryptographic protocol implementations in C (Vanspauwen & Jacobs, 2015).

Chapter 7

Conclusions and future work

In this report, we discussed the case study of verifying a key-value store using Prusti. This shows how Prusti can be used to verify a non-trivial Rust program.

We also presented a list of limitations encountered in the process of verifying the key-value store. This can help users of Prusti, and it can aid in the development of Prusti and similar program verifiers.

Further research directions include comparing Creusot to Prusti by verifying the same key-value store using Creusot. Other future research could be the verification of a similar, bigger program, for example an implementation of a hash table, and discuss what challenges arise. Finally, one could take one or more of the limitations from section 5 and fix them.

Bibliography

Astrauskas, V., Müller, P., Poli, F., & Summers, A. J. (2019). Leveraging rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, *3*(OOPSLA), 1–30.

Clippy contributors. (2023a). *Clippy documentation*. Retrieved February 22, 2023, from https://doc.rust-lang.org/clippy/index.html

Clippy contributors. (2023b). *Clippy lints*. Retrieved February 22, 2023, from https://rust-lang.github.io/rust-clippy/master/index.html

Clippy contributors. (2023c). *Rust-lang/rust-clippy: A bunch of lints to catch common mistakes and improve your rust code*. Retrieved February 22, 2023, from https://github.com/rust-lang/rust-clippy

Cuypers, C., Jacobs, B., & Piessens, F. (2009). Verification of data-race-freedom of a java chat server with verifast. *CW reports*.

de Moura, L., Kong, S., Avigad, J., Van Doorn, F., & von Raumer, J. (2015). The lean theorem prover (system description). *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, 378–388.

Denis, X., Jourdan, J., & Marché, C. (2022). Creusot: A foundry for the deductive verification of rust programs. In A. Riesco & M. Zhang (Eds.), *Formal methods and software engineering - 23rd international conference on formal engineering methods, ICFEM 2022, madrid, spain, october 24-27, 2022, proceedings* (pp. 90–105). Springer. https://doi.org/10.1007/978-3-031-17244-1_6

Facebook, Inc. (2023, February 7). *MIRAI: Rust mid-level IR abstract interpreter*. Retrieved March 6, 2023, from https://github.com/facebookexperimental/MIRAI

Galois, Inc. (2022, April 26). *Crux-mir*. Retrieved March 6, 2023, from https://github.com/GaloisInc/crucible/blob/master/crux-mir/README.md

Ho, S., & Protzenko, J. (2022). Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.*, *6*(ICFP), 711–741. https://doi.org/10.1145/3547647

Jacobs, B. (2011, January 13). *VeriFast examples*. Retrieved February 22, 2023, from https://people.cs.kuleuven.be/~bart.jacobs/verifast/examples/

Jacobs, B., & Piessens, F. (2008). *The verifast program verifier* (tech. rep.). Technical Report CW-520, Department of Computer Science, Katholieke …

Jung, R., Jourdan, J., Krebbers, R., & Dreyer, D. (2018a). Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, *2*(POPL), 66:1–66:34. https://doi.org/10.1145/3158154

Jung, R., Jourdan, J., Krebbers, R., & Dreyer, D. (2018b). RustBelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, *2*(POPL), 66:1–66:34. https://doi.org/10.1145/3158154

The Kani Rust verifier. (2023). Retrieved February 22, 2023, from https://model-checking.github.io/kani/

Klabnik, S., & Nichols, C. (2022a). The slice type. *The Rust programming language*. No Starch Press. Retrieved March 6, 2023, from https://doc.rust-lang.org/book/ch04-03-slices.html

Klabnik, S., & Nichols, C. (2022b). What is ownership? *The Rust programming language*. No Starch Press. Retrieved March 6, 2023, from https://doc.rust-lang.org/stable/book/ch04-01-what-is-ownership.html

Krebbers, R. (2015). *The c standard formalized in coq* (Doctoral dissertation). Radboud University.

Leino, K. R. M., & Moskal, M. (2010). Usable auto-active verification. *Usable Verification Workshop*.

Leroy, X., & Blazy, S. (2008). Formal verification of a c-like memory model and its uses for verifying program transformations. *J. Autom. Reason.*, *41*(1), 1–31. https://doi.org/10.1007/s10817-008-9099-0

loom contributors. (2023). *Tokio/loom: Concurrency permutation testing tool for rust*. Retrieved February 22, 2023, from https://github.com/tokio-rs/loom

Matsushita, Y., Denis, X., Jourdan, J., & Dreyer, D. (2022). Rusthornbelt: A semantic foundation for functional verification of rust programs with unsafe code. In R. Jhala & I. Dillig (Eds.), *PLDI ’22: 43rd ACM SIGPLAN international conference on programming language design and implementation, san diego, ca, usa, june 13 - 17, 2022* (pp. 841–856). ACM. https://doi.org/10.1145/3519939.3523704

MIRI contributors. (2023). *Rust-lang/miri: An interpreter for rust's mid-level intermediate representation* [Internet Archive: https://web.archive.org/web/20230107104009/https://github.com/rust-lang/miri/]. Retrieved February 22, 2023, from https://github.com/rust-lang/miri/#miri

Proptest contributors. (2023). *Proptest*. Retrieved February 22, 2023, from https://altsysrq.github.io/proptest-book/intro.html

Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Rust By Example contributors. (2023). Rust by example. Retrieved March 9, 2023, from https://doc.rust-lang.org/rust-by-example/scope/lifetime.html

Rust contributors. (2018). Two-phase borrows. *Rust Compiler Development Guide (rustc-dev-guide)*. Retrieved March 6, 2023, from https://rustc-dev-guide.rust-lang.org/borrow_check/two_phase_borrows.html

Rust contributors. (2023a). Evaluation order of operands. *The Rust reference*. Retrieved March 8, 2023, from https://doc.rust-lang.org/reference/expressions.html#evaluation-order-of-operands

Rust contributors. (2023b). Primitive type usize. *The rust standard library documentation*. Retrieved March 9, 2023, from https://doc.rust-lang.org/std/primitive.usize.html

Rust contributors. (2023c, March 5). *Rust releases* [Internet Archive: https://web.archive.org/web/20230306145928/https://github.com/rust-lang/rust/blob/master/RELEASES.md]. Retrieved March 6, 2023, from https://github.com/rust-lang/rust/blob/master/RELEASES.md

Rust verification tools. (2021). Retrieved March 10, 2023, from https://rust-formal-methods.github.io/tools.html#rust-verification-tools-2021

shuttle contributors. (2023). *Awslabs/shuttle: Shuttle is a library for testing concurrent rust code*. Retrieved February 22, 2023, from https://github.com/awslabs/shuttle

Skotåm, S. H. (2022). *Creusat - using rust and creusot to create the world's fastest deductively verified sat solver* (Master's thesis). University of Oslo. https://www.duo.uio.no/bitstream/handle/10852/96757/11/SarekSkota-m_thesis.pdf

Stack Overflow. (2022, June 22). *Stack Overflow developer survey 2022*. Retrieved March 6, 2023, from https://survey.stackoverflow.co/2022/

The Viper Project. (2023a). Basic usage. https://viperproject.github.io/prusti-dev/user-guide/basic.html

The Viper Project. (2023b). Specification syntax. Retrieved March 8, 2023, from https://viperproject.github.io/prusti-dev/user-guide/syntax.html

Toman, J., Pernsteiner, S., & Torlak, E. (2015). Crust: A bounded verifier for rust (N). In M. B. Cohen, L. Grunske, & M. Whalen (Eds.), *30th IEEE/ACM international conference on automated software engineering, ASE 2015, lincoln, ne, usa, november 9-13, 2015* (pp. 75–80). IEEE Computer Society. https://doi.org/10.1109/ASE.2015.77

Toman, J., Pernsteiner, S., & Torlak, E. (2022, September 24). *Crust: A compiler from rust to c, and a checker for unsafe code*. Retrieved March 6, 2023, from https://github.com/uwplse/crust

Ullrich, S. (2016). *Simple verification of rust programs via functional purification* (Master's thesis). Karlsruhe Institute of Technology. https://pp.ipd.kit.edu/uploads/publikationen/ullrich16masterarbeit.pdf

Ullrich, S. (2017a). *Electrolysis reference*. Retrieved February 22, 2023, from http://kha.github.io/electrolysis/

Ullrich, S. (2017b). *Kha/electrolysis: Simple verification of rust programs via functional purification in lean 2(!)* Retrieved February 22, 2023, from https://github.com/Kha/electrolysis

Vanspauwen, G., & Jacobs, B. (2015). Verifying protocol implementations by augmenting existing cryptographic libraries with specifications. *Software Engineering and Formal Methods: 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*, 53–68. https://people.cs.kuleuven.be/~bart.jacobs/sefm2015.pdf