ETH zürich

Verifying Vulnerability Fixes in a Rust Verifier

Bachelor's Thesis

Olivia Furrer

September 11, 2023

Advisors: Prof. Dr. Peter Müller, Vytautas Astrauskas, Jonas Fiala

Department of Computer Science, ETH Zürich

Abstract

The Rust programming language guarantees memory safety at compile time by default. However, for some low-level hardware behaviours the rules of ‘safe Rust’ are too restrictive. So-called ‘unsafe Rust’ allows programmers to model these behaviours in sections marked with the `unsafe` keyword. Since the Rust compiler cannot conclude the safety of unsafe sections by itself, the responsibility of ensuring memory safety is temporarily delegated to the programmer in such `unsafe` sections.

Prusti is an automated verifier designed for proving correctness of Rust programs. So far its focus was mainly on *safe* Rust code, now the goal is to expand Prusti to increasingly target *unsafe* Rust code, too. In this project we used Prusti for verifying several examples of Rust code containing unsafe sections, with the aim to evaluate how well Prusti is suited for verifying unsafe Rust.

Over the course of verifying these examples there were some Prusti features and specification patterns that we have found very useful, for example Ghost types, and the ‘unimplemented/trusted’ pattern. Based on what we experienced during our verification processes, we suggested some additional features for Prusti. Most prominently, these suggestions include the idea of Prusti allowing multiple, switchable invariants specified on a single struct, and the concept of specialised specifications that enable Prusti to leverage additional knowledge (if available) in its proof.

Contents

Chapter 1

Introduction

Rust is a systems programming language that guarantees memory safety at compile time by default. This property is based on three core concepts: *ownership* of memory, *borrowing* and *aliasing xor mutability*.

Each value in Rust has an *owner variable*. As soon as the owner variable goes out of scope, the memory used by its value is reclaimed immediately. During the lifetime of the owner variable, Rust allows the *borrowing* of a value, i.e. the creation of a reference to it. These references are not allowed to *outlive* the owner variable, which prevents traditional safety issues like use-after-free or dangling pointers.

Rust's *aliasing xor mutability* property ensures that the two types of borrowing, namely *shared borrowing* for read access and *exclusive mutable borrowing* for write access, are never present at the same time. This makes concurrent reads and writes impossible in Rust and thus prevents conventional race conditions and memory safety bugs like accessing invalid references.

While these safety rules guarantee that no undefined behaviour can ever be caused by *safe* Rust code, they are too restrictive to model some of the low-level hardware behaviours that are required for system software. For this reason Rust introduces the `unsafe` keyword. It is used to temporarily delegate the responsibility of ensuring memory safety in the code to the programmer.

If an API contains unsafe code, its author could choose to directly expose this internal unsafe code to the users of the API. However, it is considered more idiomatic to encapsulate internal unsafe code with a *safe* API. When a programmer declares an API as safe, they assure that the API conforms with Rust's safety rules. This means that the programmer is responsible for ensuring that (1) no matter what input the API is given by a client, no memory safety bug can be triggered, and (2) any internal unsafety hidden in the safe API is properly guarded.

Since the Rust compiler cannot conclude the safety of unsafe sections by itself, it relies on the programmer and assumes that the code is sound and bug-free in order to include it in the program's safety guarantee. However, if this assumption is wrong and there is in fact a bug in unsafe code, this can result in the safety guarantee of the entire Rust program being compromised.

Unfortunately, reasoning about the correctness of unsafe code is very hard for programmers, as one often has to consider actions taken by the Rust compiler that are not visible to the programmer. For example, when reasoning about panic safety, a programmer needs to manually check the consistency of stack variables for every (invisible) unwinding path which is inserted by the compiler automatically [1].

For this reason, tools with varying degrees of automation are built to help programmers to ensure correctness. One strategy is to search for bugs and remove them, another strategy is to prove the correctness of the code given.

One recently developed program which searches for bugs in Rust code is RUDRA [1], written to specifically target `unsafe` sections. RUDRA is programmed to recognise certain patterns in the code that have been found to often appear in connection with memory safety bugs. It scans Rust packages and marks sections where these patterns appear, indicating a *potential* memory safety bug to the programmer.

RUDRA's developers scanned and analysed the entire Rust package registry using RUDRA. They discovered 264 previously unknown memory safety bugs. The 112 RustSec advisories they filed correspond to 51.6% of memory safety bugs reported to RustSec since 2016 (until the time of the publishing of their paper in 2021). Among them were bugs in some of the most often used Rust packages (e.g. in `std`, the Rust standard library, or in `rustc`, the Rust compiler). Some of them had gone undiscovered for years despite these important packages being written and thoroughly reviewed by Rust experts, and despite the Rust community's efforts to manually audit unsafe code in Rust [1].

There is, however, one major drawback of programs like RUDRA that search for bugs: false negatives. Even if we do not find a single bug in a program, that does not mean that there is none. There might still be bugs which go unnoticed, undermining the security of the entire program, or even a whole system.

The problem of false negatives is remedied by a different approach where, instead of searching for bugs, one tries to prove the correctness of a program (and thus the absence of bugs). An automated verifier takes this approach and is designed to prove that a given program conforms with its specifications. An example is Prusti [2], an automated program verifier which is currently being developed at ETH Zürich. It is built upon the Viper [3] verification infrastructure (also developed at ETH) and is targeted at the Rust programming language. Prusti allows the user to provide specifications through various features and checks whether they can be proved for the given piece of code [4].

When Prusti's development began, the focus was on *safe* Rust code. Increasingly now, the goal is to expand Prusti to also target certain patterns of *unsafe* Rust code.

The aim of this project is to evaluate how well Prusti is suited for verifying the absence of vulnerabilities caused by memory safety bugs.

Chapter 2

Methodology

The aim of this project is to assess Prusti's capabilities with regard to verifying *unsafe* Rust code. For this purpose we use Prusti to verify several code examples from the Rust standard library and from the Rust crates directory. Within these code examples, we focus on the vulnerable sections where memory safety bugs have been found in the past, and try to verify them after these bugs have been fixed. Focusing on these sections gives us the possibility of a direct comparison of verifying the fixed code (which should be successful) with verifying the buggy code using the same specifications (which should fail), thus allowing us to assess Prusti's capabilities in verifying unsafe Rust code.

In Sec. 2.1 we describe our procedure for analysing and verifying each of our examples. Furthermore, in Sec. 2.2 we provide information regarding the tools we used for our project and the detailed technical specifications. Sec. 2.3 goes into more detail regarding our process of choosing our examples to verify.

2.1 Verification process

As mentioned above, the selection of our code excerpts is based on previously found memory safety bugs, and our goal for each of our examples is to prove with Prusti that these sections are now memory safe, after they have been fixed. We therefore limit our verification to the parts of the code that are relevant for the bug and its fix, but try to leave out anything else, so as to keep the size of our examples as small as possible.

We start our verification of each example by getting an overview of the Rust features that are present in the relevant sections of the code. Some of them might not be supported by Prusti (yet). In such cases, we check whether we can rewrite the code so that these features are avoided, but without changing anything that is relevant to the bug and its fix. Adapting the code in such a way might not be feasible for some examples, meaning we will not be able to verify those examples.

However, if we manage to find a workaround for any features Prusti does not support, then we can proceed by writing Prusti specifications for our code.

If Prusti fails to verify the fixed code with the specifications we wrote, then we have to find the reason for this failure: do we need to refine our specifications? Is Prusti missing a feature that is necessary to verify this code? Is there another bug in the code that has not been noticed yet?

After successfully verifying the code, we check what happens when we run Prusti on the buggy code, with the same specifications. This verification should fail! If it verifies regardless, we have to find out what went wrong. One possible problem is that our Prusti specifications are buggy. We go back to the previous step to correct our specifications, then try to verify the fixed version first again. If our specifications are not the problem, then another possibility is that there is a bug in Prusti itself and we will therefore not be able to verify the example with its current version.

2.2 Tools and technical details

For verifying our examples we used the following version of Prusti:
9e45f825a96c82fb8d95993f1ae5df66c7b77db6 [5].

We used the symbolic execution backend of Viper as the underlying Viper server, specifically an experimental version from the 'meilers_silicarbon_qponly' branch [6], and version 4.8.7 of the Z3 SMT solver.

For our Viper verifications in Chapter 6 we used version 4.2.2 of the Viper IDE for VS Code [7].

A list of all configuration flags we used for our Prusti verifications is provided in Sec. A.1.

2.3 Choosing Code Examples to Verify

As mentioned in the previous chapter, we used the bugs found by the Rudra project [8] as the starting point for choosing our examples to verify. The Rudra team analysed the Rust standard library and the rust crates published on the main Rust package index, crates.io, and published a collection of memory safety bugs in unsafe Rust code [9] [10]. In this chapter we describe our process of choosing suitable examples among these bugs for our project.

Since the Rudra program uses several analysers, the bugs on the list have been categorised according to which analyser found them. For this project we focused on the 83 examples where bugs were found by the so-called 'UnsafeDataflow' analyser, as these were the most suitable ones for assessing Prusti's capabilities regarding unsafe code.

We proceeded by looking at each example in this subset separately. Some of them had to be excluded for various reasons, as they were not suitable for our project after all. For instance, we excluded any examples which have not yet received a fix, as the primary goal for our examples was to verify the fixed code (and only dealing with the buggy code once we have been successful with this task). This restriction reduced our set of examples by half. We further noticed that there was a large subset of bugs that were essentially about the same problem (namely a vector being handed to a reader without being initialised). In most cases, this problem was solved in a way that did not require any `unsafe` sections anymore (by initialising the vector from the start)[1], meaning these examples were not of interest for assessing Prusti on unsafe Rust code. Our set of eligible bugs was again reduced roughly by half.

Once this pre-selection was complete, resulting in a set of 25 examples, we checked for each one of them whether it contained features that Prusti does not support (yet). If so, we tried to find a way to avoid these features by rewriting the code slightly differently. Furthermore, we assessed whether the code excerpt relevant for verification would be of a reasonable size for Prusti with regard to performance. Finally, whenever we chose a new example to verify we tried to avoid examples that were too similar to previous ones. With this approach we hoped to achieve a bigger variety of features and concepts in our examples, which would give us the opportunity to test Prusti's usefulness for verifying unsafe code more thoroughly.

In the following chapters we are going to present the four examples we eventually verified. For each of them we explain the code and the problem that had to be fixed, as well as the process and results of verifying the code with Prusti.

[1]Chapter 3 shows an example where this problem was solved in a different (unsafe) way. We will therefore see more about this group of bugs in that chapter, specifically in Sec. 3.3.

Chapter 3

Example 1: glium::buffer::Content::read

Our first verification example is from a crate called *glium*. In Sec. 3.1 we present the function we are going to verify, before we proceed with verifying it in Sec. 3.2.

3.1 Context

In this section we present the original code of the function we are going to verify, then show what problem the Rudra team found with this function and how this problem has been addressed. Due to the somewhat special way this problem was handled we state a slightly altered goal for verifying the function, before we start the actual process of verifying it in Sec. 3.2.1.

The Content::read function Our first example is about the `read` function in the implementation of the trait `glium::buffer::Content` for type `[T]`, shown in Fig. 3.1 [11][12]. Note that `T` is required to implement the `Copy` trait.

The function first calculates the length `len`, which it uses to create a large enough vector with `Vec::with_capacity(len)`. Then it uses the unsafe `set_len` function to set the length of the vector to `len`, and hands a mutable reference to the vector to the closure `f`. If `f` returns an error, this error is propagated up to the caller of `read`. Otherwise, `read` returns the vector in its state after the call to `f`.

The problem Originally, `read` was declared as a safe function. The problem here is that the vector does not get initialised before being handed to `f`. Since `f` can be user-defined, it is possible that it reads this uninitialised buffer. Reading uninitialised memory is undefined behaviour and is not allowed to happen. This bug was found and reported by the Rudra team [13].

The fix According to this crate's programmers, the `read` function is not meant to be used with just any arbitrary closure `f`. Instead, the closure provided to `read` is

```rust
/// Trait for types of data that can be put inside buffers.
pub unsafe trait Content {
    /// A type that holds a sized version of the content.
    type Owned;

    /// Prepares an output buffer,
    /// then turns this buffer into an `Owned`.
    fn read<F, E>(size: usize, _: F) -> Result<Self::Owned, E>
                  where F: FnOnce(&mut Self) -> Result<(), E>;

    /* ... */
}

unsafe impl<T> Content for [T] where T: Copy {
    type Owned = Vec<T>;

    #[inline]
    fn read<F, E>(size: usize, f: F) -> Result<Vec<T>, E>
                  where F: FnOnce(&mut [T]) -> Result<(), E>
    {
        assert!(size % mem::size_of::<T>() == 0);
        let len = size / mem::size_of::<T>();
        let mut value = Vec::with_capacity(len);
        unsafe { value.set_len(len) };
        f(&mut value)?;
        Ok(value)
    }

    /* ... */
}
```

Figure 3.1: Original code of trait Content and its implementation for [T]

supposed to have certain properties. For this reason, the API of the function was changed to be *unsafe*, and a warning was added in the function's documentation, stating what conditions need to hold so that `read` can be used safely (i.e. so that no undefined behaviour can occur). This note about safety requirements for closure `f` reads as follows [14]:

> User-provided closure `F` must only write to and not read from `&mut Self`.

Aim of verification Under these circumstances, the objective of our verification is slightly different than in the examples that will follow: instead of verifying the code itself as it is, we verify the code assuming that all the safety requirements stated in the documentation hold (but nothing more). If the verification is infeasible under these conditions, then the safety requirements are possibly insufficient for guaranteeing safe use of this function.

3.2 Verification

In this section we show our approach to verifying the `read` function under the conditions stated above. In Sec. 3.2.1 we show our Prusti specifications for the code and explain the changes we need to make in the code in order to be able to use Prusti for this verification. In Sec. 3.2.2 we use the model and specifications we created in Sec. 3.2.1 and test whether the code is verifiable under the assumption that the safety conditions from the documentation hold.

3.2.1 Specification process

The function we want to verify in this section is the `read` function in `glium::buffer ::Content for [T]`, where `T` is a generic type parameter for a type implementing the `Copy` trait. Since slices and traits are both not (completely) supported in Prusti we rewrite this function as an independent function `read`, and replace the slice type with a vector. Therefore, the types `Self::Owned` and `Self`, which in the original code correspond to `Vec<T>` and `[T]`, respectively, both become vectors in our version of the code.

The `read` function takes a closure `f` as an argument, with the type of the closure defined by the type parameter `F`. To avoid closures, which are not supported by Prusti, we remove this argument from the function and write `f` as a separate function instead. For simplicity, we also omit the type parameter `E` that is used for the errors and instead use a very simple error type `SomeError` whenever an error appears in the code.

After these changes, the signatures of our functions `read` and `f` look as follows:

```
pub unsafe fn read (size: usize) -> Result<Vec, SomeError>

fn f (v: &mut Vec) -> Result<(), SomeError>
```

Vec and its Type Invariant The Rust vector type `Vec<T>` takes a generic type parameter `T` that defines the type of the vector's elements. It has a length and a capacity, where the length is the number of elements that are currently in the vector, and the capacity is the total amount of space allocated for elements of the vector [15]. The vector's elements must be valid values of type `T`, i.e. the vector has to be initialised up to its length. The rest of the allocated space does not have to be initialised. These 'rules' make up the type invariant of `Vec<T>`.

The goal is now to model this type and its invariant for our verification. We create a struct `Vec` with three fields: a raw pointer of type `*const T` named `ptr` (meant to indicate the location of the vector in memory), as well as `len` and `cap`, both of type `usize` and representing the vector's length and capacity, respectively. For the element type we use a type alias named `T` instead of a generic type parameter. This way we can avoid problems caused by non-linear arithmetic in our verification, which could otherwise occur due to the type size of generic type `T` not being known.

Now that we have defined the `Vec` struct, we use Prusti to describe its type invariant, as shown in Fig. 3.2: the size of the allocation is limited according to Rust rules (line 2), the pointer field must not be null (line 3), and the length can never be greater than the capacity[1] (line 4). On lines 5-34 the permissions of access to the vector are defined, first for the case of non-zero-sized types in the `if`-clause, then for zero-sized types (ZSTs) in the `else`-clause.

On lines 12-16 and 28-32,

```
raw_range(self.ptr, std::mem::size_of::<T>(),
          self.len, self.cap)
```

means that the `Vec` has access to a range of raw memory blocks of the type size of `T`, starting at an offset of `self.len` from `self.ptr`, and ranging until the end of the vector's allocation (at offset `self.cap` from `self.ptr`). Such raw memory blocks can only be written to but not read from.

The macro

```
own_range!(self.ptr, 0, self.len)
```

on the other hand states that the vector does not only own the raw memory blocks for the range of the first `self.len` blocks, but that this range consists of valid, initialised values of the vector's element type, i.e. reading and using these values is allowed. Contrary to `raw_range`, which has to receive the size of the blocks as an extra argument, the block size for `own_range` is given by the vector's element type. Finally, the `dealloc!` macro gives `Vec` the right to deallocate the corresponding memory.

This invariant that we defined in Prusti corresponds to the type invariant of `Vec` that we described before. We defined it using Prusti's `#[structural_invariant]`, which is used to specify type invariants that are used to prove memory safety.

Prusti's design for verifying memory safety Prusti is designed to verify memory safety separately from general correctness and the absence of panics. Keywords and instructions containing the prefix (or infix)'`structural`' are used for proving memory safety, whereas their 'normal' counterparts are used for proving correctness. For example, for proving memory safety we need to use

- `#[structural_requires(...)]` instead of `#[requires(...)]` for preconditions,

[1] If a Rust vector's length exceeds its capacity, the whole vector has to be reallocated [15]. Therefore, a vector will never be in a state where its length is greater than its capacity.

```
#[structural_invariant(
    std::mem::size_of::<T>() * self.cap <= (isize::MAX as usize)
    && !self.ptr.is_null()
    && self.len <= self.cap
    && (
        if std::mem::size_of::<T>() != 0 {
            (self.cap != 0 ==> (
                own_range!(
                    self.ptr,
                    0,
                    self.len)
                && raw_range(
                    self.ptr,
                    td::mem::size_of::<T>(),
                    self.len,
                    self.cap)
                && raw_dealloc!(
                    *self.ptr,
                    std::mem::size_of::<T>() * self.cap,
                    std::mem::align_of::<T>())
            ))
        } else {
            self.cap == usize::MAX
            && own_range!(
                self.ptr,
                0,
                self.len)
            && raw_range(
                self.ptr,
                std::mem::size_of::<T>(),
                self.len,
                self.cap)
        }
    )
)]
pub struct Vec {
    ptr: *const T,
    len: usize,
    cap: usize,
}
```

Figure 3.2: Vec and its invariant

- `#[structural_ensures(...)]` instead of `#[ensures(...)]` for postconditions,
- `#[structural_invariant(...)]` instead of `#[invariant(...)]` for type invariants, and
- `prusti_structural_assert!` instead of `prusti_assert!` for assertions.

This isolation of the memory safety proof from the correctness proof helps Prusti prevent us from writing some nonsensical specifications. For example, Prusti does not allow us to specify memory safety requirements as preconditions for safe functions (i.e., using `#[structural_requires(...)]` on a safe function is not allowed). By definition, a safe function must be safe regardless of what context it is called in, or what values it gets passed as arguments. If there are preconditions that need to be fulfilled in order for some function to be memory safe, then that means that the function is `unsafe`.

As for structural *post*conditions: since they are used for proving memory safety, the properties specified inside structural postconditions must hold independently of whether the function exits regularly or with a panic. As a consequence, Prusti prohibits the use of the `result` keyword in structural postconditions, because a result is only available if the function does not panic, i.e. only in one of the two cases where a structural postcondition must hold.

If the postcondition about the result is nevertheless needed for the proof of memory safety, we can annotate the function with `#[no_panic_ensures_postcondition]` in order for Prusti to be able to assume the result in the non-panic-situation at least. Without such an annotations, 'normal' postconditions are simply ignored for the memory safety proof, so this annotation is like a notification for Prusti to include this postcondition (in the case of no panic) when it verifies memory safety. We are going to see this in use several times throughout our examples.

Contrary to structural preconditions, `#[structural_ensures(...)]` is allowed on safe functions. It is convenient for properties that are ensured by the function both in the panic and no-panic cases. Otherwise, duplicate postcondition specifications would be necessary for properties like these, one with `#[ensures(...)]` for the case without a panic, and one with `#[panic_ensures(...)]`, for the panicking case.

The Vec::with_capacity function In the `read` function of our example, a new vector is created using `with_capacity`. Fig. 3.3 shows the Prusti specifications of the `Vec::with_capacity` function we use for the verification. The function's body is omitted and replaced with the `unimplemented!` macro, which is why the function has to be annotated with `#[trusted]` in order to verify.

In its precondition it says that allocation must never fail and that the capacity given as an input must not be too big, so that Rust's size limit for allocations is not

```
 1 impl Vec {
 2     #[trusted]
 3     #[no_panic_ensures_postcondition]
 4     #[requires(allocation_never_fails())]
 5     #[requires(
 6         std::mem::size_of::<T>() * capacity
 7         <= (isize::MAX as usize)
 8     )]
 9     #[ensures(result.capacity() == capacity)]
10     #[ensures(result.len() == 0)]
11     pub fn with_capacity(capacity: usize) -> Vec {
12         unimplemented!();
13     }
14 }
```

Figure 3.3: Vec::with_capacity

exceeded. The postconditions ensure that the `Vec` returned by the function will actually have the capacity that was given as the argument, and that it will be empty (i.e. `len` is zero). The methods `capacity` and `len` used on lines 9 and 10 here are both pure getter methods for the vector's capacity and length fields, respectively. Their exact code and Prusti specifications can be seen in Sec. A.2, Fig. A.1.

Note that we cannot use *#[structural_ensures]* for the postconditions here, since they use the keyword `result` to describe the vector returned by the function. As explained before, Prusti verifies functionality and memory safety separately and a 'normal' postcondition like this one is simply ignored by the memory safety proof. We therefore need to add the annotation *#[no_panic_ensures_postcondition]*, which allows Prusti to consider the postconditions for its memory safety proof. If there is no panic, Prusti can assume the postconditions to hold at the call site of the function after it returns.

The unsafe set_len method Once the vector is created, `read` uses the unsafe method `Vec::set_len` on line 24 of Fig. 3.1 to set its length to `len`, the length it calculated previously (line 22) and which it already used as the capacity argument in `with_capacity`. The `set_len` method forces the length of a vector to the given argument `new_len`. However, in doing so, it does not maintain the invariant of the vector [16], because it completely ignores the initialisation of the vector elements. No excess elements are dropped when the length of the vector gets decreased by `set_len`, and no new elements are initialised when the length is increased.

The fact that the type invariant of `Vec` might not hold anymore after a use of `set_len` is exactly the reason why this method is declared as an `unsafe` function. It can only be used safely if additional precautions are taken. The documentation on `set_len` states two safety conditions which need to hold for a safe use of `set_len` [16]:

(1) `new_len` must be less than or equal to the capacity

(2) all elements at `old_len..new_len` must be initialised

In the case of our example, the function is as follows on line 24, just after the creation of the vector `value`:

```
unsafe { value.set_len(len) };
```

The new length `len` is equal to the capacity we used for creating the vector, therefore, the first safety condition holds. However, no elements have been initialised so far, thus the second condition is violated in our example, meaning that after the call to `set_len` the invariant of `Vec` will be broken.

But is it possible that the overall code is memory safe nonetheless? Regarding the safety of the function, remember that the documentation of `glium::buffer::Content::read` only states: "User-provided closure `F` must only write to and not read from `&mut Self`." According to the programmers of this functions it should therefore be safe to use `read` as long as the closure fulfils these conditions.

The vector with the broken invariant We therefore want to find out whether it is possible to verify `read` despite the invariant of `Vec` being broken, but assuming that the conditions stated in the documentation hold. In order to do this, we need a way to model the vector in the state where its invariant is broken from the call to `set_len`. However, Prusti does not allow us to define several alternative invariants on the same struct, so we create a new, slightly different struct for the vector type, with a weaker invariant. We call this struct `BrokenVec`, and show it in Fig. 3.4.

BrokenVec has an additional field `init_pt` that allows us to define the initialisation separately from the `len` field: instead of `len`, we now use `init_pt` in the definition of the permission ranges, to indicate the index up to which the vector is initialised. Both `len` and `init_pt` still have to be within the capacity, but we do not say anything about the relation between `len` and `init_pt` in the invariant, i.e. we allow `len` to be greater than `init_pt`, so that this version of the vector can contain uninitialised elements within its length.

Ghost code Note that we use a different type for index `init_pt` than for `len` and `cap`: `Int` instead of `usize`. `Int` is a mathematical (i.e. unbounded) integer type used in Prusti, and it is a Ghost type. So-called Ghost code refers to data and computations that are only inserted into code for the purpose of its verification [17]. It is only taken into consideration during verification, but not when the code is run normally. Therefore, Ghost code must not affect the regular code in a way that changes the program's behaviour, it may only be deployed to facilitate the program's verification.

```
#[structural_invariant(
    std::mem::size_of::<T>() * self.cap <= (isize::MAX as usize)
    && !self.ptr.is_null()
    && self.len <= self.cap
    && Int::new(0) <= self.init_pt
    && (self.init_pt).to_usize() <= self.cap
    && (
        if std::mem::size_of::<T>() != 0 {
            (self.cap != 0 ==> (
                own_range!(
                    self.ptr,
                    0,
                    (self.init_pt).to_usize())
                && raw_range(
                    self.ptr,
                    std::mem::size_of::<T>(),
                    (self.init_pt).to_usize(),
                    self.cap)
                && raw_dealloc!(
                    *self.ptr,
                    std::mem::size_of::<T>() * self.cap,
                    std::mem::align_of::<T>())
            ))
        } else {
            self.cap == usize::MAX
            && own_range!(
                self.ptr,
                0,
                (self.init_pt).to_usize())
            && raw_range(self.ptr,
                std::mem::size_of::<T>(),
                (self.init_pt).to_usize(),
                self.cap)
        }
    )
)]
struct BrokenVec {
    ptr: *const T,
    len: usize,
    cap: usize,
    init_pt: Int,
}
```

Figure 3.4: BrokenVec and its invariant

Since `init_pt` is only used for our Prusti specification and does not appear in any of the actual computations of the program, it makes sense to use a Ghost type here to make it clear that this field is separate from the rest of the code and does not affect its behaviour, it only serves to aid our verification. Because `init_pt` is of type `Int`, we need a conversion function from `Int` to `to_usize` if we want to compare it with `usize` values, and we also need to add a condition in the invariant stating that `init_pt` is non-negative.

Specifications for BrokenVec's set_len method We can now define `set_len` as a method of `BrokenVec` (see Fig. 3.5). As a precondition we require the first of the two safety conditions from `set_len`'s documentation to hold. The second safety condition (that `new_len` must be smaller or equal to `init_pt`) is left out to allow the situation occurring in the `read` function, where this condition does not hold. The postcondition says that after the call to `set_len`, the length of the vector will have the value of `new_len`, while the capacity and `init_pt` will remain unchanged.

Inside the body of the method, we unpack the invariant using the Prusti macro `unpack!` (line 9), which makes the conditions and knowledge held by the invariant available inside the function. Then, the length of the vector is set to its new value. In the end, the invariant has to be packed again (line 13). If the `len` field was changed in a way that violates the conditions of the invariant, then calling `pack!` will fail. But if `BrokenVec`'s invariant still holds, it can be packed again without a problem.

Since the vector is behind reference `&mut self` in this method, this reference needs to be opened first, using `open_mut_ref!` (line 8), before the invariant can be unpacked. The macro `take_lifetime!` (line 7) is needed to give a name (`lft_self`) to `self`'s lifetime, which is used by `open_mut_ref!` as an argument. At the end of the function, when the invariant has been packed, the reference needs to be closed again, using `close_mut_ref!` (line 14).

Function f The `read` function calls closure `f` after `set_len` was used for forcing the length of the vector to its capacity without initialising any elements, i.e. at a time when the invariant of the vector is broken. Therefore, we change the input type of our function `f` from `&mut Vec` to `&mut BrokenVec`.

Due to no elements being initialised at this point, `init_pt` will be zero. The permissions that `f` gets are therefore only raw permissions for the whole range of the vector:

```
raw_range(self.ptr, std::mem::size_of::<T>(), 0, self.cap)
```

Meanwhile, the range of `own_range` is `[0..0]`. Therefore, `f` does not hold any permissions to read the elements of the vector, it can only write to them. This situation corresponds exactly to the safety requirements stated in the documentation of `read`, i.e. we are able to test if verification is possible under exactly the

```rust
impl BrokenVec {
    #[structural_requires(new_len <= self.cap)]
    #[structural_ensures(self.len == new_len)]
    #[structural_ensures(self.init_pt == old(self.init_pt))]
    #[structural_ensures(self.cap == old(self.cap))]
    unsafe fn set_len(&mut self, new_len: usize) {
        take_lifetime!(self, lft_self);
        open_mut_ref!(lft_self, *self, self_witness);
        unpack!(*self);

        self.len = new_len;

        pack!(*self);
        close_mut_ref!(*self, self_witness);
    }
}
```

Figure 3.5: BrokenVec::set_len

conditions that are supposedly sufficient to make the `read` function safe, which is the goal we set for this example at the beginning of this section.

Due to `f` being a closure given to `read` as an argument in the original code, we do not know anything about its implementation. Since we model it as a separate function now and not as an argument anymore, we need a different way to model this 'black box' behaviour. We can leave the body unknown by using the macro `unimplemented!`. For our verification we then have to annotate `f` as a trusted function with `#[trusted]`. This tells Prusti to just assume that this function verifies, regardless of its implementation. The specifications of `f` now look as follows:

```rust
#[trusted]
fn f(v: &mut BrokenVec) -> Result<(), SomeError> {
    unimplemented!();
}
```

Implementing the read function Fig. 3.6 shows how we implement the `read` function and what Prusti specifications we use for it. The two preconditions at the top are needed to make sure the preconditions of `Vec::with_capacity` hold and the vector can be created smoothly. The third precondition makes sure that the `assert!` statement at the very beginning holds. Note that putting such an assertion in the code is essentially just a way of declaring (and enforcing!) a precondition in plain Rust: it only lets the rest of the function body execute if it holds.

This precondition not only makes sure that `size` is a multiple of the type size of `T`, it also prevents this function from being executed when `T` is a zero-sized type, since a calculation modulo zero is not allowed and therefore the precondition never

```
#[requires(allocation_never_fails())]
#[requires(size <= (isize::MAX as usize))]
#[requires(size % std::mem::size_of::<T>() == 0)]
pub unsafe fn read (size: usize) -> Result<Vec, SomeError> {
    assert!(size % std::mem::size_of::<T>() == 0);
    let len = size / std::mem::size_of::<T>();

    let mut value: Vec = Vec::with_capacity(len);

    let mut broken_value = break_vec(value);

    unsafe { broken_value.set_len(len) };
    match f(&mut broken_value) {
        Ok(_) => {
            Ok(unsafe { repair_vec(broken_value)} )
        },
        Err(e) => Err(e),
    }
}
```

Figure 3.6: Function read with vector transformations

holds. If we set `T` to be a zero-sized type, our verification of `read` will consequently always be successful, as the function would never even be entered.

Note also that the original Rust implementation used the `?`-operator to directly return errors returned by `f`. However, there is no support for the `?`-operator in Prusti yet, so we replaced it with a `match`-expression instead.

In the remainder of this subsection we show our implementation of the main part of the function and how we put our two structs `Vec` and `BrokenVec` to use. When the `read` function creates the vector using `Vec::with_capacity`, the usual type invariant of `std::vec::Vec` still holds. It is only afterwards, when `set_len` is called, that the invariant becomes broken, just before the call to `f`. However, the 'normal' invariant needs to hold again at the time when `read` returns the vector, at the very latest. Otherwise, the broken vector would be exposed to code outside of `read`, which relies on the invariant of the vector to be intact.

Therefore, we use `BrokenVec` only temporarily: after the vector is created as a normal `Vec` through `Vec::with_capacity`, we build a `BrokenVec` out of it before `set_len` is called, use this for the rest of the execution, but turn it back into a normal `Vec` before the function returns. For these transformations from `Vec` to `BrokenVec` and back we introduced two extra functions, `break_vec` and `repair_vec`.

Transforming Vec to BrokenVec Fig. 3.7 shows the function transforming a 'normal' `Vec` into a `BrokenVec`. First of all, we unpack `Vec`'s invariant to make its

```
#[no_panic_ensures_postcondition]
#[ensures(
    result.ptr == old(vec.ptr)
    && result.len == old(vec.len)
    && result.init_pt == old(Int::new_usize(vec.len))
    && result.cap == old(vec.cap)
)]
fn break_vec(mut vec: Vec) -> BrokenVec {
    unpack!(vec);
    let broken = BrokenVec {
        ptr: vec.ptr,
        len: vec.len,
        cap: vec.cap,
        init_pt: Int::new_usize(vec.len)
    };

    vec.ptr = new_ptr();
    vec.len = 0;
    vec.cap = 0;
    pack!(vec);
    broken
}
```

Figure 3.7: Transformation from Vec to BrokenVec

permissions and properties available to the function. Then we create a `BrokenVec` instance, giving its fields the values from `vec`'s fields. `init_pt` gets its value from `vec.len` since at this point initialisation and length are still the same. Our new vector also gets the permissions stated in its invariant from `vec`.

Right now, both `vec` and the new `broken` refer to the same allocation in memory (and share the same permissions to access it). Before we can return the `BrokenVec` to `read`, we need to make `vec` forget about this allocation and its old field values. Otherwise, the allocation would get deallocated when `vec` goes out scope and is dropped at the end of `break_vec`. Simply overwriting the `vec` variable itself with a new `Vec` would not do the trick, as the old `Vec` would in this case be dropped when it gets overwritten, thus we would not be able to prevent the allocation from being deallocated this way.

Instead, we have to overwrite each of the fields inside `vec` with new values. The integers can be overwritten with zero, but to obtain a new pointer we create a function `new_ptr` with return type `*const T`. We leave it unimplemented and annotate it as a trusted function, and in its postcondition we ensure that the returned pointer is non-null.

Note that the behaviour described above only occurs as described if `Vec` actually gets dropped, i.e. if `Vec` implements the `Drop` trait. Otherwise, nothing would

```
#[no_panic_ensures_postcondition]
#[structural_requires(
    broken_vec.len == (broken_vec.init_pt).to_usize()
)]
#[ensures(
    result.ptr == old(broken_vec.ptr)
    && result.len == old(broken_vec.len)
    && result.cap == old(broken_vec.cap)
)]
unsafe fn repair_vec(mut broken_vec: BrokenVec) -> Vec {
    unpack!(broken_vec);
    let repaired = Vec {
        ptr: broken_vec.ptr,
        len: broken_vec.len,
        cap: broken_vec.cap
    };

    broken_vec.ptr = new_ptr();
    broken_vec.len = 0;
    broken_vec.cap = 0;
    broken_vec.init_pt = Int::new(0);

    pack!(broken_vec);
    repaired
}
```

Figure 3.8: Transformation from BrokenVec back to Vec

happen when `vec` goes out of scope and the function could incorrectly be verified even if `vec`'s fields are not overwritten. To ensure that `vec`'s behaviour upon going out of scope is realistically modelled, we added a `Drop` implementation for `Vec` when we verified this example. (We did the same for `BrokenVec` as well, for the reverse transformation that we will see later on).

Once the fields of `vec` have been overwritten, we can pack `vec`'s invariant again and then return our new vector, `broken`. In the postcondition of `break_vec`, we have to use `old` expressions to refer to the values of `vec`'s fields at the beginning of the function, before they were overwritten.

Transforming BrokenVec back to Vec For the reverse transformation, we do essentially the same in the other direction (see Fig. 3.8): we unpack the invariant of `broken_vec`, create a `Vec`, `repaired`, which takes the field values from `broken_vec`, then overwrite the fields of `broken_vec` before we pack its invariant again and return the `Vec` we created. The postcondition also works in the same way as the one of `break_vec`.

The only big difference is that we have a structural precondition for this function: `init_pt` must be equal to `len`, otherwise we cannot build a `Vec` out of the `BrokenVec`. This precondition means that the 'normal' vector invariant must not *actually* be broken (with `init_pt` $\neq$ `len`) at this point anymore if we want to make a 'normal' `Vec` out of the `BrokenVec` again. As explained before, functions with a structural precondition must be unsafe. We therefore have to turn `repair_vec` into an *unsafe* function.

3.2.2 Final verification specifications and results

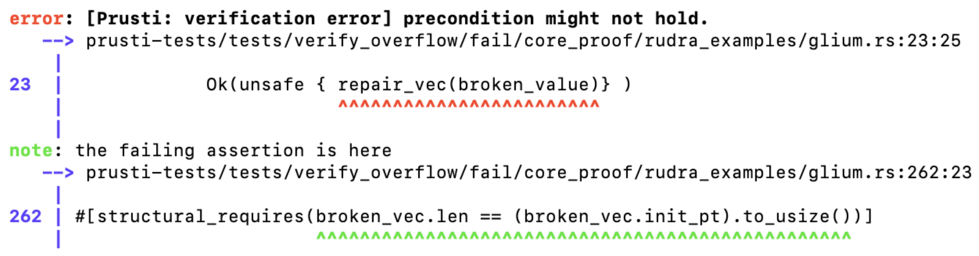
Let us summarise briefly what we prepared for the verification in the last subsection. First of all, we moved closure `f` outside of `read`'s arguments to be an independent function. We left it unimplemented and marked it as a trusted function. Then we defined two versions of the vector struct: `Vec`, the 'normal' one with the usual type invariant that is also described in the documentation, and a second one, `BrokenVec`, which contains an additional field, the Ghost field `init_pt`, and has a slightly weaker invariant, allowing `init_pt` and `len` to be unequal.

In our main `read` function we use the 'normal' `Vec` at the beginning, for creating the vector through its associated function `with_capacity`, as well as at the end, when a vector is returned. For the computation in between, however, we use `BrokenVec`, and its method `set_len`. We use the two functions `break_vec` and `repair_vec` for changing the vector from `Vec` to `BrokenVec` and back. Among these two functions, `repair_vec` is an *unsafe* function due to its precondition that `init_pt` and `len` must be equal once more before the `BrokenVec` can be transformed back to a 'normal' `Vec`.

Recall that the documentation of `Content::read` states that for `read` to be safe, the closure `f` provided to it must only write to the vector and not read from it. Remember also that our Prusti specifications correspond to the assumption that this condition holds. When we run our verification with these specifications, Prusti gives us the output shown in Fig. 3.9. It is not able to prove that the precondition to `repair_vec` holds, i.e. it cannot prove that `broken_vec` is initialised exactly up to its length after it returns from `f`, meaning that `read` is actually not safe under this assumption.

This makes sense, since the assumption only said that `f` may only write to but not read from the vector, it does not say that it *has to* write to it or that it has to initialise exactly the first `len` elements of the vector. Therefore, this safety condition does not prevent the vector from still being only partly initialised, or not at all.

Verification with an additional postcondition for f Let us now make an experiment where we assume that we know that `f` does not read from the vector, and that it writes exactly `len` elements to the vector, then returns. To assume this, we add a postcondition to our function `f` as shown in Fig. 3.10. With this additional



```
error: [Prusti: verification error] precondition might not hold.
   --> prusti-tests/tests/verify_overflow/fail/core_proof/rudra_examples/glium.rs:23:25
    |
23  |             Ok(unsafe { repair_vec(broken_value)} )
    |                         ^^^^^^^^^^^^^^^^^^^^^^^^
    |
note: the failing assertion is here
   --> prusti-tests/tests/verify_overflow/fail/core_proof/rudra_examples/glium.rs:262:23
    |
262 | #[structural_requires(broken_vec.len == (broken_vec.init_pt).to_usize())]
    |                       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Figure 3.9: Caption

```rust
#[trusted]
// additional condition that is NOT stated in the documentation!!
#[structural_ensures((v.init_pt).to_usize() == v.len)]
fn f(v: &mut BrokenVec) -> Result<(), SomeError> {
    unimplemented!();
}
```

Figure 3.10: Adding a postcondition to f

postcondition, Prusti is able to verify the program successfully. Therefore, this additional assumption about `f` is what it takes for `read` to actually be safe.

3.3 Similar examples

During our analysis of the list of bugs that were found with the Rudra tool we came across many examples with a similar problem to this one. Oftentimes a vector was created using `Vec::with_capacity` (or a similar method), then given to a user implemented function (often some read function from the `Read` trait) without being initialised first.

The programmers expected these read functions to fill the buffer exactly up to its length, since this is what read functions of the `Read` trait are used for conventionally. However, the `Read` trait does in fact not promise this behaviour of its read functions, and since these functions are often user-implemented there is no guarantee that they actually behave in this way, i.e. that they fill the whole buffer and do not read its contents before it is initialised. It is problematic if such a function does read the buffer's contents before it has been initialised, because reading uninitialised memory is undefined behaviour and must never occur in a safe function.

While in the example we discussed in this chapter the problem was solved by turning `Content::read` into an *unsafe* function, most of the other examples with this problem were solved by simply initialising the vector[2] before handing it to the

[2] e.g. by creating it directly with the `vec!` macro, or by using `Vec::resize` instead of `Vec::set_len` to change its length

user-implemented function. This initialisation is of course an overhead (especially if in most of the use cases, the initialised values are just overwritten again right away), but it makes the function safe again. Note that initialising the vector from the beginning is completely safe and no `unsafe` blocks are needed for this, which is why after the fix these examples were not eligible for our project anymore.

3.4 Suggestion for Prusti: switchable invariants

In the verification of the `read` function that we showed in Sec. 3.2.1, we modelled the `Vec` type both in its usual state, when its type invariant holds, as well as in a state where this type invariant was broken temporarily. We were only able to do this by defining two different structs, each holding a different invariant, and transforming our vector back and forth between these two implementations.

However, verifications like this would be more practical if Prusti allowed us to define multiple invariants on one type, and to switch from one invariant to the other depending on the state our struct is in. Being able to provide multiple invariants for one struct (e.g. a stronger and a weaker one) would allow us to model programs where the invariant of a struct gets temporarily broken for some part of the computation, but holds again in the end.

And while it could of course be considered ironic to allow an invariant to have multiple versions, or ‘variants’, it *would* facilitate the use of Prusti for verifying *unsafe* Rust, where not all properties of a type can be maintained at all times during a computation.

Chapter 4

Example 2: bam::bgzip::Block::load

This example is about the `load` function in `bam::bgzip::Block`. We start Sec. 4.1 by presenting the original (fixed) code relevant for this chapter, and explaining how it makes use of special type properties in order to work the way it does. We will then show the bug that Rudra found in the previous version of this code. Sec. 4.2 is about the verification of this example, showing how we simplified and adapted the fixed code, what Prusti specifications we wrote for it, and what we got as a result when we ran Prusti first on the fixed, then on the buggy version of the code.

Sec. 4.3 discusses a special aspect of this example and how the verification of similar examples could follow the pattern we used for our Prusti specifications here. Based on our experience of verifying this example, Sec. 4.4 suggests the addition of a new Prusti feature to better accommodate the verification pattern described in Sec. 4.3.

4.1 Context, features, and concepts

The struct `bam::bgzip::Block` is shown in Fig. 4.1. It has four fields, and among them are `uncompressed` and `compressed`, both of type `Vec<u8>`. They each have a maximum size, stated in the comments. The definitions of the constants used are shown in Fig. 4.2. As we will see later on, the two remaining fields of `Block` (`buffer` and `offset`) are irrelevant in this discussion of the example, which is why we do not go into further detail about them here.

Fig. 4.3 shows the `Block` struct's `load` method, which is where Rudra found a bug. It makes sure both `uncompressed` and `compressed` are empty, then performs various computations (which we left out here) before ultimately calculating a variable `block_size` via the function `analyze_extra_fields`.

This `block_size` is used for calculating the `compressed` vector's new length that is used by the `set_len` method on lines 37-38 of Fig. 4.3.

```rust
/// A bgzip block, that can contain compressed,
/// uncompressed data, or both.
/// ...
#[derive(Clone)]
pub struct Block {
    // Uncompressed contents,
    // max size = [MAX_BLOCK_SIZE](constant.MAX_BLOCK_SIZE.html).
    uncompressed: Vec<u8>,
    // Compressed contents + footer (empty if uncompressed),
    // max size = `MAX_COMPRESSED_SIZE + FOOTER_SIZE`.
    compressed: Vec<u8>,

    // Buffer used to read the header.
    buffer: Vec<u8>,
    offset: Option<u64>,
}
```

Figure 4.1: bam::bgzip::Block original code

```rust
/// Biggest possible size of the compressed and uncompressed block
/// (`= 65536`).
pub const MAX_BLOCK_SIZE: usize = 65536;

const HEADER_SIZE: usize = 12;
const MIN_EXTRA_SIZE: usize = 6;
const FOOTER_SIZE: usize = 8;

/// Biggest possible length of the compressed data
/// (excluding header + footer).
/// Equal to [MAX_BLOCK_SIZE](constant.MAX_BLOCK_SIZE.html)
/// `- 26 = 65510`.
pub const MAX_COMPRESSED_SIZE: usize =
    MAX_BLOCK_SIZE - HEADER_SIZE - MIN_EXTRA_SIZE - FOOTER_SIZE;
```

Figure 4.2: Definitions of the constants

Before this new length is set, there is a safety check aiming to make sure that the new length (left-hand side below) will not exceed `compressed`'s maximum capacity (right-hand side below; see comments in Fig. 4.1), i.e. that

```
block_size - HEADER_SIZE - MIN_EXTRA_SIZE
<= MAX_COMPRESSED_SIZE + FOOTER_SIZE
```

By rearranging the inequality above, we get

```
block_size
<= MAX_COMPRESSED_SIZE + FOOTER_SIZE + HEADER_SIZE + MIN_EXTRA_SIZE
```

The sum on the right hand side is equal to `MAX_BLOCK_SIZE`, hence the safety check in the code, on line 25:

```
block_size <= MAX_BLOCK_SIZE.
```

If this inequality does not hold, an error is returned.

The second part of the safety check, on line 26, makes sure that the subtraction on line 38 does not overflow. It does so by checking that `block_size` is not smaller than what is subtracted from it.

Once the new length of the vector is set, the vector is given to the `read_exact` function of `stream` (one of the arguments of `load`). `stream` is of a type that implements the `Read` trait, i.e. it is a so-called 'reader'.

Remember the discussion about reader implementations in Sec. 3.3. Even though the programmers of this crate probably expected the `read_exact` function to read exactly `compressed.len` many elements into `compressed` and that it does not read any of the contents of `compressed` while doing so. However, the second part of this assumption is not guaranteed by the `Read` trait [18][19]. The `load` function could be called with a user-implemented reader, where nothing is known about what `stream.read_exact` actually does. Therefore, in order to be sure that nothing bad happens in `read_exact` (or after), the vector's type invariant must hold when `read_exact` is called.

How can it therefore be alright for `Block::load` to take an empty vector, forcibly increase its length, and then hand it to an unknown `read_exact` function? In general, it is indeed not safe to do this. However, in this specific case the vector's invariant actually holds when `read_exact` is called because of a special properties of the vector's element type `u8`.

The Copy trait In Rust, 'move semantics' are the default, meaning that after an assignment `y = x;`, the value in `x` has 'moved' into `y` and `x` cannot be used anymore. However, the vector element type `u8` implements the trait `std::marker::Copy`, meaning it has 'copy semantics' instead, and `x` is still in scope after an assignment `y = x;`. A value of a `Copy` type is defined purely by its bit pattern and can therefore be duplicated by copying its bits. `Copy` types do not have a drop handler, therefore

```rust
impl Block {

    /* ... */

    /// Reads the compressed contents from `stream`.
    /// Panics if the block is non-empty
    /// (consider using [reset](#method.reset)).
    pub fn load<R: Read>(
        &mut self,
        offset: Option<u64>,
        stream: &mut R)
        -> Result<(), BlockError> {

        assert!(
            self.compressed.is_empty()
            && self.uncompressed.is_empty(),
            "Cannot load into a non-empty block");

        /* ... */

        let block_size =
            analyze_extra_fields(&self.buffer[HEADER_SIZE..])?
            as usize + 1;

        if block_size > MAX_BLOCK_SIZE
        || block_size < HEADER_SIZE + MIN_EXTRA_SIZE {
            return Err(BlockError::Corrupted(
                format!(
                    "Block size {} > {} or < {}",
                    block_size, MAX_BLOCK_SIZE,
                    HEADER_SIZE + MIN_EXTRA_SIZE)));
        }

        unsafe {
            // Include footer in self.compressed
            // to read footer in one go.
            self.compressed.set_len(
                block_size - HEADER_SIZE - MIN_EXTRA_SIZE);
        }
        stream.read_exact(&mut self.compressed)?;
        Ok(())
    }

    /* ... */

}
```

Figure 4.3: Original code of Block::load

```rust
impl Block {
    /// Creates an empty block.
    pub fn new() -> Self {
        // Initialize vectors so that we do not have problems
        // with uninitialized memory.
        let mut uncompressed = vec![0; MAX_BLOCK_SIZE];
        uncompressed.clear();
        let mut compressed
            = vec![0; MAX_COMPRESSED_SIZE + FOOTER_SIZE];
        compressed.clear();

        Self {
            uncompressed,
            compressed,
            buffer: Vec::new(),
            offset: None,
        }
    }

    /* ... */
}
```

Figure 4.4: Original code of Block::new

calling drop on such a value is a no-op and does therefore not affect the value's bit pattern, meaning the value remains valid despite the drop.

How the Copy trait allows the read_exact call in load to be safe New instances of `Block` are created through the constructor `Block::new` (shown in Fig. 4.4). Both `uncompressed` and `compressed` have `u8` as their fixed element type, and both of them are first zero-initialised up to their maximum capacity, then cleared using `Vec::clear`.

However, being of type `u8`, the elements of the vectors are not affected by the drop in `Vec::clear` and therefore the whole memory area of the vectors still consists of valid values of type `u8` even after the two vectors are cleared in `Block::new`.

So despite the length of the vectors being zero at the beginning of `Block::load`, their allocated memory still holds valid values up to their capacity, and when `set_len` increases the length of `compressed`, the elements in this area will already be initialised `u8`-values.

The range `[0..new_len]` of the vector is therefore initialised, meaning the type invariant of `Vec` (as shown in Chapter 3) holds, and handing the vector to a potentially user-defined reader does not pose a problem here.

```rust
pub struct Block {
    // Uncompressed contents,
    // max size = [MAX_BLOCK_SIZE](constant.MAX_BLOCK_SIZE.html).
    uncompressed: Vecu8, // Vec<u8>,
    // Compressed contents + footer (empty if uncompressed),
    // max size = `MAX_COMPRESSED_SIZE + FOOTER_SIZE`.
    compressed: Vecu8, // Vec<u8>,
}
```

Figure 4.5: Block struct
(without Prusti specifications)

The bug and its fix At the time when Rudra analysed this crate, the overflow check on line 26 was missing. If `block_size` was too small, it was therefore possible that the `set_len` function set the length to a value greater than `compressed`'s capacity on lines 37-38. This problem was solved by the additional condition in the safety check.

4.2 Verification

This section shows our Prusti verification of the example described above. In Sec. 4.2.1 we discuss the process of simplifying the code and writing suitable specifications for verification. Sec. 4.2.2 provide a brief summary of the final specifications resulted from this process, then state the outcome of the verification of the fixed code. Sec. 4.2.3 shows what happens when we run Prusti with the same specifications on the *buggy* version of the code.

4.2.1 Specification process

We start this section by modelling the Block and vector structs for our verification.

The `offset` field of `Block` does not appear in the relevant code sections that we want to verify (see Fig. 4.3). We are going to see later in this section that we will not need the `buffer` field either, so we can omit both of these fields from our definition of the `Block` struct (Fig. 4.5).

The remaining fields, `uncompressed` and `compressed`, are both vectors with fixed element type `u8`. In Sec. 4.1 we explained that this fixed element type is important because of the way the code leverages `u8`'s special properties. Therefore, we create a specialised version of the vector type, which we call `Vecu8`.

Vecu8 and its invariant We model `Vecu8` similarly to `Vec` back in Chapter 3, with three fields: `ptr` (the pointer to the vector in memory), `len` (the length), and `cap` (the capacity). However, this time we declare `ptr` as having type `*const u8`

```rust
pub struct Vecu8 {
    ptr: *const u8,
    len: usize,
    cap: usize,
}
```

Figure 4.6: Vecu8 struct
(initial version, without Prusti specifications)

(see Fig. 4.6) instead of `*const T`[1]. As a result, `Vecu8`'s elements are fixed to be of type `u8`, which allows us to specialise our specifications of the vector and its methods according to `u8`'s properties.

In Chapter 3 we explained the type invariant of Rust vectors (and used it for our struct `Vec` in that example). However, the code here leverages the fact that once they have been initialised, `u8`-typed elements do not become invalid even when they are dropped (as explained in Sec. 4.1). Therefore, if a vector is for example initialised up to its capacity and cleared afterwards (like what happens in `Block::new`), then even though the length will be zero, the whole allocation will still be initialised.

In order to make use of the knowledge that a vector can be initialised beyond its length, we need to detach the definition of the access permissions in the vector's invariant from the `len` field. Only then can the initialisation remain in the same state even when `len` is modified.

We achieve this by adding a Ghost field `init_pt`, just like we did in the previous chapter for `BrokenVec`. This additional field again allows us to define the initialisation separately from `len`. Fig. 4.7 shows the resulting `Vecu8` struct with its invariant, where we use `init_pt` instead of `len` as the delimiter between the `own_range` and `raw_range`.

Note that there is a difference between the invariant here, and the one we used for `BrokenVec` in Chapter 3: here, we have the additional requirement that `len` must be smaller or equal to `init_pt`, meaning that the vector must be initialised at least up to its length. This property was not required for `BrokenVec`, where we allowed the vector to contain uninitialised memory within its length, which normally should not happen.

Vecu8's methods With this definition of `Vecu8` and its invariant we can write specifications for its methods `clear` and `set_len` accordingly.

Fig. 4.8 shows our specifications for the clear function. To illustrate the fact that these specifications are specialised for vectors with an element type implement-

[1] In the definition of `Vec`'s `ptr` in Chapter 3, `T` was a type alias modelling the generic type parameter of `Vec<T>` and could therefore be defined as any arbitrary type.

```
#[structural_invariant(
    std::mem::size_of::<u8>() * self.cap <= (isize::MAX as usize)
    && !self.ptr.is_null()
    && self.len <= (self.init_pt).to_usize()
    && (self.init_pt).to_usize() <= self.cap
    && (
        self.cap != 0 ==> (
            own_range!(
                self.ptr,
                0,
                (self.init_pt).to_usize())
            && raw_range(
                self.ptr,
                std::mem::size_of::<u8>(),
                (self.init_pt).to_usize(),
                self.cap)
            && raw_dealloc!(
                *self.ptr,
                std::mem::size_of::<u8>() * self.cap,
                std::mem::align_of::<u8>()))
    )
)]
pub struct Vecu8 {
    ptr: *const u8,
    len: usize,
    cap: usize,
    init_pt: Int,
}
```

Figure 4.7: Vecu8 and its invariant

ing the `Copy` trait, and are therefore not valid for the general `Vec<T>` type, we renamed the clear function ‘`clear_for_copy_types`’. We leave the function body unimplemented and mark the function as trusted, specifying its behaviour through Prusti postconditions. Like the normal clear method it sets the length to zero and preserves the capacity. However, `init_pt` preserves its value, meaning that the `Vecu8` remains initialised up to that point, despite being empty.

When it comes to the `set_len` method (Fig. 4.8, bottom half), we have already seen in the previous chapter that this is an unsafe function which only alters the `len` field without caring about the initialisation state of the vector. The `len` field takes the value of `new_len`, while `cap` and `init_pt` remain the same.

Unlike in the previous chapter, however, we do not ignore the second safety condition stated in `set_len`’s documentation [16]: this time we require that `new_len` must be smaller or equal to `init_pt`. This precondition is enough to cover both of the safety conditions, since by transitivity it implies that `new_len` is also

```rust
impl Vecu8 {
    /* ... */

    #[trusted]
    #[structural_ensures(self.len == 0)]
    #[structural_ensures(self.cap == old(self.cap))]
    #[structural_ensures(self.init_pt == old(self.init_pt))]
    fn clear_for_copy_types(&mut self) {
        unimplemented!();
    }

    /* Safety:
     - new_len must be less than or equal to cap.
     - The elements at old_len..new_len must be initialized. */
    #[structural_requires(new_len <= (self.init_pt).to_usize())]
    #[structural_ensures(self.len == new_len)]
    #[structural_ensures(self.cap == old(self.cap))]
    #[structural_ensures(self.init_pt == old(self.init_pt))]
    unsafe fn set_len(&mut self, new_len: usize) {
        take_lifetime!(self, lft_self);
        open_mut_ref!(lft_self, *self, self_witness);
        unpack!(*self);
        self.len = new_len;
        pack!(*self);
        close_mut_ref!(*self, self_witness);

    }

    /* ... */
}
```

Figure 4.8: Prusti specifications for Vecu8::clear_for_copy_types and Vecu8::set_len

smaller or equal to `cap`.

Block and its constructor Let us now have a look at the function where Blocks are created: `Block::new`. The original code uses the macro `vec!` to create the vectors `uncompressed` and `compressed`, giving them each a length and zero-initialising them up to their respective lengths right away. Since the `vec!` macro is not supported by Prusti, however, we replaced it with an unimplemented, trusted function `Vecu8::new_init` (see Fig. 4.9).

The postcondition states that the length of the resulting vector takes the value of the `length` argument of the function, and that the vector is also initialised up to that point. We need to annotate the function with `#[no_panic_ensures_postcondition]` to allow Prusti to assume the postcondition for its memory safety

```
impl Vecu8 {
    #[trusted]
    #[no_panic_ensures_postcondition]
    #[ensures(result.len == length)]
    #[ensures(result.init_pt == Int::new_usize(length))]
    fn new_init(length: usize, val: u8) -> Self {
        unimplemented!();
    }

    /* ... */
}
```

Figure 4.9: Vecu8::new_init with Prusti specifications

```
impl Block {
    /// Creates an empty block.
    pub fn new() -> Self {
        /* Initialize vectors so that we do not
           have problems with uninitialized memory. */
        /* vec![0; MAX_BLOCK_SIZE]; */
        let mut uncompressed = Vecu8::new_init(MAX_BLOCK_SIZE, 0);
        uncompressed.clear_for_copy_types();

        /* vec![0; MAX_COMPRESSED_SIZE + FOOTER_SIZE]; */
        let mut compressed = Vecu8::new_init(
            MAX_COMPRESSED_SIZE + FOOTER_SIZE, 0);
        compressed.clear_for_copy_types();

        Self {
            uncompressed,
            compressed,
        }
    }

    /* ... */
}
```

Figure 4.10: Block::new

proof (in the non-panicking case), as we need the `result` keyword and thus cannot use a structural postcondition for this function.

Fig. 4.10 shows our implementation of `Block::new`. After the two vectors have been created and initialised they are both cleared using `clear_for_copy_types`, meaning that their initialisation stays intact. Therefore, when the Block to be returned is created out of these two vectors, they are both initialised up to their

```
#[structural_invariant(
    (self.uncompressed.init_pt).to_usize() == MAX_BLOCK_SIZE &&
    (self.compressed.init_pt).to_usize() == MAX_COMPRESSED_SIZE +
                                            FOOTER_SIZE
)]
pub struct Block {
    // Uncompressed contents,
    // max size = [MAX_BLOCK_SIZE](constant.MAX_BLOCK_SIZE.html).
    uncompressed: Vecu8, // Vec<u8>,
    // Compressed contents + footer (empty if uncompressed),
    // max size = `MAX_COMPRESSED_SIZE + FOOTER_SIZE`.
    compressed: Vecu8, // Vec<u8>,
}
```

Figure 4.11: Block with its invariant

respective maximum capacities.

Since dropping elements of these two vectors will not invalidate their values, this initialisation state will not change throughout the rest of the Block's lifetime in the program. We can therefore write an invariant for `Block` where we declare this unchanging initialisation state of the two vectors, as shown in Fig. 4.11

The Block::load method Now that we have defined `Block`'s invariant, we can also prepare its `load` method for verification. The method starts with an assert statement making sure that both vectors are empty upon entry into the function. This assertion is equivalent to checking a precondition before starting the actual execution of the function, therefore we put this condition into a Prusti precondition to make sure it holds when the function is called (see Fig. 4.12).

For this purpose we need to implement the `is_empty` method for our `Vecu8` and write specifications for it, as shown in Fig. 4.13. We write a postcondition to ensure the correct result. Then, we need to tell Prusti that this function is pure and that it terminates, so that we can use it inside our Prusti precondition for `load`.

After the assert statement, there is a big part of the function that does not really matter to our verification (which we also omitted when we showed the original code of the function in Fig. 4.3). It suffices if we continue at line 21 of that figure, where the `block_size` variable is defined.

There, it is enough for us to know that `analyze_extra_fields` returns a `u16` (if it is successful). We therefore model `analyze_extra_fields` as an unimplemented, trusted function with return type `u16`, shown in Fig. 4.14. Since it is unimplemented, its argument does not matter, so it can be omitted. As a result, the `buffer` field of `Block` is not used anywhere in our function anymore and this is why it was

```rust
impl Block {

    /* ... */

    /* Cannot load into a non-empty block */
    #[requires(
        self.uncompressed.is_empty() &&
        self.compressed.is_empty()
    )]
    pub fn load(&mut self) -> Result<(), BlockError> {
        assert!(
            self.uncompressed.is_empty() &&
            self.compressed.is_empty(),
            "Cannot load into a non-empty block");

        let block_size = analyze_extra_fields() as usize + 1;

        if block_size > MAX_BLOCK_SIZE
        ||  block_size < HEADER_SIZE + MIN_EXTRA_SIZE {
            return Err(BlockError{});
        }

        unsafe {
            // Include footer in self.compressed
            // to read footer in one go.
            self.compressed.set_len(
                block_size - HEADER_SIZE - MIN_EXTRA_SIZE
            );
        }

        match stream_read_exact(&mut self.compressed) {
            Ok(_) => Ok(()),
            Err(e) => Err(e),
        }
    }
}
```

Figure 4.12: Block::load

```
impl Vecu8 {
    /* ... */

    #[pure]
    #[terminates]
    #[ensures(result == (self.len == 0))]
    fn is_empty(&self) -> bool {
        self.len == 0
    }
}
```

Figure 4.13: Vecu8::is_empty

```
#[trusted]
fn analyze_extra_fields() -> u16 {
    unimplemented!();
}
```

Figure 4.14: analyze_extra_fields

no problem to omit the `buffer` field in our definition of the `Block` struct at the beginning of this section.

After the definition of `block_size` comes the safety check. It makes sure the new length of `compressed` does not exceed its capacity. Otherwise, an error[2] is returned before this new length is actually set.

Finally, `compressed` is handed to the reader function. In order to avoid traits, which are not fully supported by Prusti yet, we created a separate function `stream_read_exact` (see Fig. 4.15) to replace the call of `stream.read_exact` in the original code in Fig. 4.3. Since the concrete implementation of the reader function is unknown, we left our function unimplemented, annotating it as trusted. However, we did change its signature: since Prusti does not support slice types, we changed the type of the function's argument `buf` from the original `&mut [u8]` to `&mut Vecu8`. Furthermore, we also used a `match`-expression in order to avoid the `?`-operator, just as we did for the previous example in Chapter 3.

Our redefinition of the reader function allows us to simplify `Block::load`'s signature in the way that is shown in Fig. 4.12: we can omit all the arguments apart from `&mut self` because none of the other arguments are used anywhere in our simplified implementation of the method.

[2]Note that in original code, `BlockError` was defined as an enum type, with three variants. However, we simplified this type to a unit-like struct `BlockError`, i.e. a struct without any fields, since it is not relevant for our verification to know exactly what kind of error is returned.

```rust
#[trusted]
fn stream_read_exact(buf: &mut Vecu8) -> Result<(), BlockError>{
    unimplemented!();
}
```

Figure 4.15: stream_read_exact

4.2.2 Verifying the fixed code

We start this section with a short overview summarising the specifications we wrote for this example. Then we show the result of Prusti verifying the example with these specifications.

First of all, we defined a specialised vector type, `Vecu8` in Fig. 4.7, with fixed element type `u8`, and a Ghost field `init_pt`, which is used to declare the access rights to the vector independently of its `len` field.

We defined the function `new_init` (Fig. 4.9) as a replacement for Rust's `vec!` macro. It sets both `len` and `init_pt` to the length given to it as an argument. Therefore, the vectors created by this function start out with the 'normal' vector invariant, where these two fields are still the same.

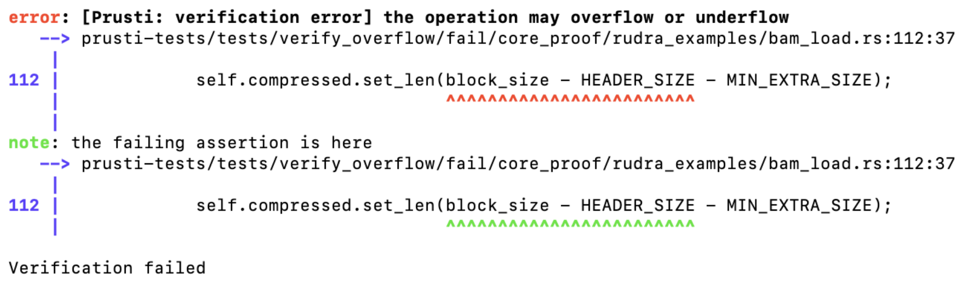
`Vecu8`'s methods `clear_for_copy_types` and `set_len`, are shown in Fig. 4.8. Both of them only alter the `len` field while leaving the initialisation as it was. This is normal for unsafe function `set_len`, which does not care about initialisation. However, this specification of `clear_for_copy_types` is not generally true for the `clear` function and only holds in this special case, thanks to the elements being of type `u8`. `Vecu8`'s method `is_empty` (Fig. 4.13) is a pure function used for the precondition of `Block::load`.

Our simplified `Block` struct and its invariant are shown in Fig. 4.11. The invariant makes sure that both fields `uncompressed` and `compressed` are initialised in the entire range of their respective initial lengths, i.e. the lengths they are given at the time of their creation. `Block::new` in Fig. 4.10 creates new blocks that abide by the rules of this invariant. No additional Prusti specifications were needed for this function.

Finally, `Block::load` (Fig. 4.12), has the aforementioned precondition making sure both vectors are empty at the start. It also calls `analyze_extra_fields` (Fig. 4.14) and `stream_read_exact`(Fig. 4.15), which we simplified and left unimplemented, annotating them both as trusted for our verification.

Verification result Verifying the code with the modifications and Prusti specifications described above was successful.

In the next subsection, we will use these same specifications to make sure that verification is *not* successful anymore if the overflow bug is present in the code.



```
error: [Prusti: verification error] the operation may overflow or underflow
   --> prusti-tests/tests/verify_overflow/fail/core_proof/rudra_examples/bam_load.rs:112:37
    |
112 |             self.compressed.set_len(block_size - HEADER_SIZE - MIN_EXTRA_SIZE);
    |                                     ^^^^^^^^^^^^^^^^^^^^^^^^
    |
note: the failing assertion is here
   --> prusti-tests/tests/verify_overflow/fail/core_proof/rudra_examples/bam_load.rs:112:37
    |
112 |             self.compressed.set_len(block_size - HEADER_SIZE - MIN_EXTRA_SIZE);
    |                                     ^^^^^^^^^^^^^^^^^^^^^^^^

Verification failed
```

Figure 4.16: Prusti's output when we verify the code containing the overflow bug

4.2.3 Verifying the buggy code

In this section, we roll back the fix of the overflow bug in the `Block::load` function, meaning we return back to the old version of the safety check, using

```
if block_size > MAX_BLOCK_SIZE {...}
```

again, instead of the improved

```
if block_size > MAX_BLOCK_SIZE
|| block_size < HEADER_SIZE + MIN_EXTRA_SIZE {...}.
```

The output that we got from running Prusti on this buggy version of the code, with the same specifications as before, is shown in Fig. 4.16.

Prusti gives us a verification error and refers to a potential overflow exactly at the place of the problematic subtraction operation. Therefore, we have successfully written specifications which allow Prusti to verify the fixed code, but do not enable Prusti to (wrongly) prove the code containing the overflow bug.

4.3 Similar code patterns

Normally, the invariant for `std::vec::Vec` uses the field `len` to indicate the borderline between `own_range` and `raw_range` permissions. However, the code in this example only works because it makes use of a special property of the type used for the vector elements. As a consequence, specifying the 'normal' vector invariant in Prusti would not be enough to verify this example.

Therefore whenever code leverages a type's properties to make the program work, Prusti specifications probably need to be adapted. The specifications need to describe the functions' and types' behaviour in the special case, and will not be applicable to the general case anymore. Only then can Prusti make use of the special properties needed to verify the program.

4.4 Suggestion for Prusti: specialised specifications

In the case of the example above, the Prusti specifications we defined on our function `clear_for_copy_types` are not generally valid for `std::vec::Vec::clear`. They are only valid in the special case of the vector being specialised for element types implementing the `Copy` trait, where we can make use of the knowledge that the dropped elements do not become invalid.

Let us compare our *specialised* specifications for `clear_for_copy_types` in the case where `T: Copy` with the specifications that would be appropriate if `T` was allowed to be any type (we call this version of the `clear` function `clear_for_general_types` here):

```
#[structural_ensures(self.len == 0)]
#[structural_ensures(self.cap == old(self.cap))]
#[structural_ensures(self.init_pt == old(self.init_pt))]
fn clear_for_copy_types(&mut self) {...}

#[structural_ensures(self.len == 0)]
#[structural_ensures(self.cap == old(self.cap))]
#[structural_ensures(self.init_pt == 0)]
fn clear_for_general_types(&mut self) {...}
```

Their behaviour with regard to `len` and `cap` are the same in both cases, but the specifications differ for the Ghost field `init_pt`. In the case of `clear_for_copy_types` we know that the values that were initialised before remain initialised even after this function is executed. This is not true in the general case, where the type passed to the struct does not necessarily implement the `Copy` trait. The dropped values can become invalid; certainly we do not *know* them to be valid anymore, therefore `init_pt` has to be zero after the execution of `clear_for_general_types`.

Based on this observation we would consider it useful if Prusti allowed users to specialise specifications (such as the postconditions above) depending on the traits that a certain type implements. Then we could write specifications for the general case as well as specialised ones that are used whenever additional information is available, e.g. because of a type parameter implementing a certain trait.

In the case of this example we would then write our specifications for `Vec<T>::clear` for both the specialised and general case. If `T` implements `Copy`, then the specialised specifications of `clear` can be used for verifying, and otherwise the general ones are used.

Instead of only being able to write specifications suiting the most general use case of a function, data structure, type etc., this feature of specialised Prusti specifications would allow users to utilise additional knowledge for our verification if any is available. This way the behaviour of functions and data structures in special cases will better be accounted for.

Chapter 5

Example 3: std::vec::Vec::from_iter

This chapter is about a function in the Rust standard library, `Vec::from_iter` in `SpecFromIter<T, I> for Vec<T>`, where a double free bug was found [20]. The bug was subsequently fixed [21].

5.1 Context, features, and concepts

We start this section by introducing the `std::vec::IntoIter` struct and explaining how it works. We proceed by giving an overview of the function `Vec::from_iter` in `SpecFromIter<T, I> for Vec<T>`. We show what the double free bug found in this function was about, as well as how it could be fixed.

The IntoIter Struct The struct `std::vec::IntoIter` is created by the `Vec::into_iter` method, which is provided by the `IntoIterator` trait [22]. The `cap` field indicates the amount of allocated space and corresponds to the capacity of the vector the `IntoIter` was created from. `buf` points to the start of the `IntoIter`'s allocated memory, while `end` indicates the end of the initialised elements of the `IntoIter`, pointing one past the last element of the original vector (meaning the difference between `end` and `buf` is equal to the length of the original vector). The pointer `ptr` is used to move along the iterator `IntoIter` when for example the `next` method is called. The remaining two fields, `phantom` and `alloc` will not be relevant for our discussion in this chapter, which is why we will not go into detail about them.

As we have seen in the previous two examples, the type invariant of `std::vec::Vec` states that the elements within the length of the vector are initialised. This condition is transferred to `IntoIter` when it is built, meaning that starting from the element pointed to by `buf` up until one before where `end` points, all elements are initialised at the time of the creation of `IntoIter`. At this moment in time, `buf` and `ptr` both still point to the element at the start of the allocation. As the `ptr` pointer advances, the elements between `buf` and `ptr` are the elements that have already been read, while the ones between `ptr` and `end` are still initialised.

The Vec::from_iter function Fig. 5.1 shows the `Vec::from_iter` function in `SpecFromIter<T, I> for Vec<T>` as it is today. It takes as an argument an object `iterator` of some type `I` which implements various traits, among others the `Iterator` trait, as well as `SourceIter<Source: AsVecIntoIter>`, which allows it to be turned into an `IntoIter`. The two calls of `iterator.as_inner().as_into_iter()` (on line 15 and on line 33 of Fig. 5.1) do exactly that, they return the object as a type `IntoIter`.

Before the second of these calls, `iterator` is passed to `SpecInPlaceCollect::collect_in_place` on lines 29-30, which goes through the remaining elements between `ptr` and `end` and collects some of them at the beginning of the iterator, starting at `buf`. The number of elements collected is the returned value `len`. Now, the first `len` elements are initialised too, in addition to the elements between `ptr` and `end`.

In this state, `src` is defined as `iterator` returned as an `IntoIter` (line 33). The length that was returned by the `collect_in_place` function is to be the length of the new vector created on line 39. But before this vector can be created and returned, the initialised elements remaining between `ptr` and `end` need to be dropped. Furthermore, since the new vector is to be created through `Vec::from_raw_parts` from the memory where `src` is allocated, `src` needs to forget about this allocation. Otherwise it would be deallocated when `src` is dropped at the end of the function, which would pose a problem for the returned vector, as it still uses this same allocation.

Originally, these two tasks of dropping the remaining elements and forgetting the allocation were done by first calling `IntoIter::drop_remaining` and then calling `IntoIter::forget_allocation`, both shown in Fig. 5.2. Fig. 5.3 shows how the last few lines of `from_iter` looked (instead of lines 37-41 in Fig. 5.1).

`drop_remaining` calls `self.as_mut_slice()`, which returns the slice of memory between `self.ptr` and `self.end`. This slice is then dropped and `self.ptr` set to `self.end` to indicate that there are no elements left to read in the iterator.

`forget_allocation` overwrites the individual fields of the `IntoIter` to make `src` forget about the allocation, thus preventing the allocation from being dropped once the `IntoIter` goes out of scope.

The Bug There is a detail that was overlooked in the version of the code described above. Drops might panic, and if a panic occurs during the process of dropping the slice in `drop_remaining`, the code will exit and the `IntoIter` will be dropped. Any elements of the slice that have already been dropped are dropped again because they are also part of the `IntoIter`'s allocation. This double free

```rust
// excerpt from rust/library/alloc/src/vec/in_place_collect.rs
impl<T, I> SpecFromIter<T, I> for Vec<T>
where
    I: Iterator<Item = T>
    + SourceIter<Source: AsVecIntoIter>
    + InPlaceIterableMarker,
{
    default fn from_iter(mut iterator: I) -> Self {
        if T::IS_ZST || (/* ... */) {
            // fallback to more generic implementations
            return SpecFromIterNested::from_iter(iterator);
        }

        let (src_buf, src_ptr, dst_buf, dst_end, cap) = unsafe {
            let inner = iterator.as_inner().as_into_iter();
            (
                inner.buf.as_ptr(),
                inner.ptr,
                inner.buf.as_ptr() as *mut T,
                inner.end as *const T,
                inner.cap,
            )
        };

        // SAFETY:
        // `dst_buf` and `dst_end` are
        // the start and end of the buffer.
        let len = unsafe {
            SpecInPlaceCollect::collect_in_place(
                &mut iterator, dst_buf, dst_end)
        };

        let src = unsafe { iterator.as_inner().as_into_iter() };

        /* ... Some assertions ... */

        src.forget_allocation_drop_remaining();

        let vec = unsafe { Vec::from_raw_parts(dst_buf, len, cap) };

        vec
    }
}
```

Figure 5.1: Vec::from_iter original code

```rust
impl<T, A: Allocator> IntoIter<T, A> {

    /* ... */

    pub(super) fn drop_remaining(&mut self) {
        unsafe {
            ptr::drop_in_place(self.as_mut_slice());
        }
        self.ptr = self.end;
    }

    /// Relinquishes the backing allocation, equivalent to
    /// `ptr::write(&mut self, Vec::new().into_iter())`
    pub(super) fn forget_allocation(&mut self) {
        self.cap = 0;
        self.buf =
            unsafe { NonNull::new_unchecked(RawVec::NEW.ptr()) };
        self.ptr = self.buf.as_ptr();
        self.end = self.buf.as_ptr();
    }
}
```

Figure 5.2: drop_remaining and forget_allocation as separate functions

```rust
impl<T, I> SpecFromIter<T, I> for Vec<T>
where
    I: Iterator<Item = T> + SourceIterMarker,
{
    default fn from_iter(mut iterator: I) -> Self {
        /* ... */

        // drop any remaining values at the tail of the source
        src.drop_remaining();
        // but prevent drop of the allocation itself
        // once IntoIter goes out of scope
        src.forget_allocation();

        let vec = unsafe { Vec::from_raw_parts(dst_buf, len, cap) };

        vec
    }
}
```

Figure 5.3: The last few lines of from_iter before the bug fix

```rust
impl<T, A: Allocator> IntoIter<T, A> {

    /* ... */

    #[cfg(not(no_global_oom_handling))]
    pub(super) fn forget_allocation_drop_remaining(&mut self) {
        let remaining = self.as_raw_mut_slice();

        // overwrite the individual fields instead of creating a new
        // struct and then overwriting &mut self.
        // this creates less assembly
        self.cap = 0;
        self.buf =
            unsafe { NonNull::new_unchecked(RawVec::NEW.ptr()) };
        self.ptr = self.buf.as_ptr();
        self.end = self.buf.as_ptr();

        // Dropping the remaining elements can panic,
        // so this needs to be done only
        // after updating the other fields.
        unsafe {
            ptr::drop_in_place(remaining);
        }
    }
}
```

Figure 5.4: IntoIter::forget_allocation_drop_remaining original code

bug [20] results from the fact that both the `IntoIter` and the slice to be dropped share parts of the same allocation.

The Bug Fix The solution to this problem was to merge these two functions into one and switch up the order of doing things a little. `IntoIter::forget_allocation_drop_remaining` (shown in Fig. 5.4) only stores the slice in a local variable at *first*, then proceeds by first overwriting the fields of the `IntoIter` like `forget_allocation` used to do. Now that the `IntoIter` does not refer to the same allocation as the slice anymore, it is safe to drop the slice remembered in local variable `remaining`. If a panic occurs while dropping `remaining`, dropping the `IntoIter` will not lead to any double frees anymore as it does not share any memory with the slice anymore.

5.2 Verification

This section is about the verifying the `from_iter` function introduced in the previous section. In Sec. 5.2.1, we show how we build up the specifications needed for this verification, then summarise the final result of this process in Sec. 5.2.2. Unfortunately, we have not been able to completely verify the fixed code yet. Consequently, no section on testing our specifications on the buggy version of the code is included in here.

5.2.1 Specification process

In this section we show how we went about verifying the `from_iter` function together with `IntoIter::forget_allocation_drop_remaining`.

Vec and its invariant The `Vec` struct we use in this example and its invariant are identical to the one we used in Sec. 3.2.1, Fig. 3.2, including the type alias `T` replacing the generic type parameter. The getter methods `len` and `capacity` are also identical to the ones used in Sec. 3.2.1. In addition to what we used in previous examples we also need `Vec::from_raw_parts` for this one; this function will be shown in more detail later on.

Defining IntoIter and its invariant We define our own `IntoIter` struct, leaving out the fields that we do not need (see Fig. 5.5). Instead of `ptr::NonNull` we use a normal `*const T` pointer for `buf` but add the non-null condition to the invariant (line 5).

Fig. 5.5 also shows the rest of the invariant we use to describe our `IntoIter`. It is very similar to the one we used for `Vec` in Fig. 3.2, but adapted to `IntoIter` and the fact that instead of the `len` field a pointer is used here, as described in the following.

Because we are talking about several pointers this time, we need to add the conditions on lines 7-8, stating that they all belong to the same allocation. To be able to describe the order of the pointers in lines 10-13, we have to use `address_from(pointer, base_pointer)`, which returns the distance from `base_pointer` to `pointer` as a value of type `Int`[1]. These lines therefore mean that `buf` points to an address that is smaller or equal to the one pointed to by `ptr`, which is smaller or equal to the one pointed to by `end`, and they are all within `cap` distance from `buf`.

Lines 17-37 specify the permission ranges for the `IntoIter`. They are similar to the ones we defined for `Vec` in Fig. 3.2 (in the case where `T` was not zero-sized), but here we again use `address_from` to calculate the length of the respective ranges. Different from `Vec`, we have three ranges here instead of two, namely first a `raw_range` between `buf` and `ptr` (where the elements have already been read as `ptr` advanced), then an `own_range` from `ptr` to `end` (where the remaining initialised elements are), and finally another `raw_range` for the rest of the allocated space. The `raw_dealloc` part is the same as for `Vec` before.

[1] `address_from` is a Prusti equivalent of the `offset_from` function used for raw pointers in Rust.

```
#[structural_invariant(
    Int::new_usize(std::mem::size_of::<T>())
        * Int::new_usize(self.cap)
        <= Int::new_usize(isize::MAX as usize)
    && !self.buf.is_null()

    && same_allocation(self.buf, self.ptr)
    && same_allocation(self.buf, self.end)

    && Int::new_isize(0) <= address_from(self.ptr, self.buf)
    && address_from(self.ptr, self.buf)
        <= address_from(self.end, self.buf)
    && address_from(self.end, self.buf) <= Int::new_usize(self.cap)

    && std::mem::size_of::<T>() != 0

    self.cap != 0 ==> (true
        && raw_range(
            self.buf,
            std::mem::size_of::<T>(),
            0,
            address_from(self.ptr, self.buf).to_usize())
        && own_range!(
            self.buf,
            address_from(self.ptr,self.buf).to_usize(),
            address_from(self.end, self.buf).to_usize())
        && raw_range(
            self.buf,
            std::mem::size_of::<T>(),
            address_from(self.end, self.buf).to_usize(),
            self.cap)

        && raw_dealloc!(
            *self.buf,
            std::mem::size_of::<T>() * self.cap,
            std::mem::align_of::<T>())
    )
)]
struct IntoIter {
    buf: *const T,
    cap: usize,
    ptr: *const T,
    end: *const T,
}
```

Figure 5.5: IntoIter with its invariant

The case of T being a zero-sized type is excluded in this example by line 15. We do not need this case in this example, because in case of T being a ZST a more generic implementation of from_iter is used instead of the one discussed here – see Fig. 5.1, lines 9-12.

The from_iter Function and its Prusti Specifications For the Rudra bug and its fix that we want to verify only the last few lines of the from_iter function are actually relevant. We therefore simplify the code by writing a function from_iter_ending that only contains these last few lines, namely lines 37-41 of Fig. 5.1. In order to be able to do this we need to model src's state at the start of this code section, i.e. after collect_in_place has already been called and has returned len.

Remember that in addition to the elements between ptr and end, the first len elements of the IntoIter are also initialised at this moment of the execution, while any elements in between those two regions are not, just like the ones lying beyond the end pointer. Since Prusti does not allow us to define a second invariant for the same struct to describe this new situation, we decided to define a new struct altogether, a special variant of the IntoIter struct, which we call IntoIterAfterCollect. We can then use this as the input type to our function from_iter_ending.

Defining IntoIterAfterCollect Our struct IntoIterAfterCollect differs from IntoIter in that it contains an additional field len, representing the value that is returned by collect_in_place and stored in a local variable in the original code. The reason we put it into the struct here is so that we can use it for our permission specifications in the struct's invariant, as we want to be able to model the fact that, after collect_in_place returns, the first len elements will also be initialised.

Fig. 5.6 shows how we defined the invariant on IntoIterAfterCollect. It is almost identical to the one for IntoIter, except for a few changes in connection to the len field: line 10 states that the distance from buf to ptr must be larger than len. On lines 17-20, we have an additional own_range for the first len elements.

Now we can use an input prep of this new type IntoIterAfterCollect for from_iter_ending, modelling the state of the IntoIter as it comes back from the collect_in_place function. Before we can hand prep to IntoIter::forget_allocation_drop_remaining, we need to make a "normal" IntoIter out of prep again. This transformation is done on lines 9-14 in Fig. 5.7 by building a new IntoIter object src and assigning all the pointers accordingly (prep.len is forgotten, of course).

For us to have access to the knowledge of IntoIterAfterCollect's invariant, and specifically to be able to hand prep's permissions over to src, we need to unpack prep.

```
#[structural_invariant(
    (Int::new_usize(self.cap) *
        Int::new_usize(std::mem::size_of::<T>())
        <= Int::new_isize(isize::MAX))
    && !self.buf.is_null()

    && same_allocation(self.buf, self.ptr)
    && same_allocation(self.buf, self.end)

    && Int::new_usize(self.len) <= address_from(self.ptr, self.buf)
    && address_from(self.ptr, self.buf)
        <= address_from(self.end, self.buf)
    && address_from(self.end, self.buf) <= Int::new_usize(self.cap)

    && std::mem::size_of::<T>() != 0
    && (self.cap != 0 ==> (true
            && own_range!(
                self.buf,
                0,
                self.len)
            && raw_range(
                self.buf,
                std::mem::size_of::<T>(),
                self.len,
                address_from(self.ptr, self.buf).to_usize())
            && own_range!(
                self.ptr,
                0,
                address_from(self.end, self.ptr).to_usize())
            && raw_range(
                self.ptr,
                std::mem::size_of::<T>(),
                address_from(self.end, self.ptr).to_usize(),
                self.cap
                    - address_from(self.ptr, self.buf).to_usize())

            && raw_dealloc!(
                *self.buf,
                std::mem::size_of::<T>() * self.cap,
                std::mem::align_of::<T>())
    ))
)]
```

Figure 5.6: IntoIterAfterCollect with its invariant
(continued on the next page)

```rust
struct IntoIterAfterCollect {
    buf: *const T,  /* corresponds to `buf` in original IntoIter */
    len: usize,     /* the new length returned by collect_in_place */
    ptr: *const T,  /* corresponds to `ptr` in original IntoIter */
    end: *const T,  /* corresponds to `end` in original IntoIter */
    cap: usize,     /* corresponds to `cap` of original IntoIter */
}
```

Figure 5.6: IntoIterAfterCollect with its invariant (cont.)

```rust
fn from_iter_ending(mut prep: IntoIterAfterCollect) -> Vec {
    let stash_ptr = prep.buf;
    let stash_len = prep.len;
    let cap = prep.cap;

    unpack!(prep);
    stash_range!(stash_ptr, 0, stash_len, stash1);

    let mut src = IntoIter {
        buf: prep.buf,
        ptr: prep.ptr,
        end: prep.end,
        cap: prep.cap,
    };

    /* destroying prep */
    prep.cap = 0;
    prep.len = 0;
    prep.buf = new_raw_vec();
    prep.ptr = prep.buf;
    prep.end = prep.buf;
    pack!(prep);

    unsafe {src.forget_allocation_drop_remaining();}
    restore_stash_range!(stash_ptr, 0, stash1);

    todo!();
    // let vec = unsafe {
    //     Vec::from_raw_parts(
    //         stash_ptr as *mut T,
    //         stash_len,
    //         cap)
    // };
    // vec
}
```

Figure 5.7: from_iter_ending with Prusti specifications

```
/* replaces `NonNull::new_unchecked(RawVec::NEW.ptr())` */
#[trusted]
#[no_panic]
#[no_panic_ensures_postcondition]
#[ensures(!result.is_null())]
fn new_raw_vec() -> *mut T {
    unimplemented!();
}
```

Figure 5.8: Function new_raw_vec

However, there is a problem with handing the permissions of `prep` over to `src`: compared to `IntoIter`, `IntoIterAfterCollect` has an additional owned range at the beginning. In the original code, `forget_allocation_drop_remaining` will simply not know about this since it is given an `IntoIter` and the task to drop the initialised elements between `ptr` and `end`. However, since we declare the permissions explicitly in Prusti, we have to take care of the first range of owned memory. We use the `stash_range!` Prusti command, which hides the indicated `own_range` and makes it look like there is a `raw_range` in that memory region instead. With the first `own_range` stashed away like this, we can now build our `IntoIter` with fitting permissions without problems.

There remains just one last detail we need to deal with before we call `forget_allocation_drop_remaining`: we now have two objects of different types in the same memory region, namely `prep` and `src`. Once `forget_allocation_drop_remaining` drops the initialised region of `src` and returns `src` with only a `raw_range`, the two structs will have different permissions even though they share the same allocation. In order to avoid such a conflict, we make the `prep` variable forget about this allocation, but without dropping it. We achieve this by overwriting `prep`'s fields (see Fig. 5.7, lines 17-21), just as the original code already does for the `IntoIter` in `forget_allocation_drop_remaining`, and just like we did in Chapter 3 when we transformed the vector back and forth between `Vec` and `BrokenVec`.

To simplify getting a new pointer for `prep.buf`, we replaced

```
unsafe{ NonNull::new_unchecked(RawVec::NEW.ptr()) }
```

with an unimplemented, trusted function `new_raw_vec` (Fig. 5.8). In the original function, a constant is returned, which means that a panic is impossible and we can annotate the function with `#[no_panic]`. Adding the annotation `#[no_panic_ensures_postcondition]` makes sure that the postcondition can be assumed at the call site of `new_raw_vec`. This way it is guaranteed that the pointer is never null. After overwriting the fields, we `pack!` the invariant of this completely altered `prep`.

IntoIter's forget_allocation_drop_remaining method Now, `src` is ready to be handed to `forget_allocation_drop_remaining`, which is shown in Fig. 5.9. Since

```rust
impl IntoIter {
    unsafe fn forget_allocation_drop_remaining(&mut self) {

        let (remaining_ptr, remaining_len)
            = (self.ptr as *mut T, self.len());

        take_lifetime!(self, lft_self);
        open_mut_ref!(lft_self, *self, self_witness);
        unpack!(*self);

        // overwrite the individual fields instead of creating a new
        // struct and then overwriting &mut self.
        // this creates less assembly
        self.cap = 0;
        self.buf = new_raw_vec() as *const T;
        self.ptr = self.buf;
        self.end = self.buf;

        // Dropping the remaining elements can panic, so this
        // needs to be done only after updating the other fields.
        unsafe {
            drop_in_place(remaining_ptr, remaining_len);
        }
        pack!(*self);
        close_mut_ref!(*self, self_witness);
    }

    /* original: provided by trait `ExactSizeIterator`
       which vec::IntoIterator implements */
    #[non_verified_pure]
    #[pure]
    #[terminates]
    #[no_panic]
    #[no_panic_ensures_postcondition]
    fn len(&self) -> usize {
        unsafe {self.end.offset_from(self.ptr) as usize}
    }
}
```

Figure 5.9: IntoIter's methods forget_allocation_drop_remaining and len

```
#[trusted]
#[structural_requires(
    own_range!(to_drop,0,len))]
#[structural_ensures(
    raw_range(to_drop,std::mem::size_of::<T>(),0,len))]
unsafe fn drop_in_place(to_drop: *mut T, len: usize) {
    unimplemented!();
}
```

Figure 5.10: Function drop_in_place

Prusti does not support slices, we replaced the call `self.as_raw_mut_slice()` on line 7 of Fig. 5.4 by instead simply remembering the pair of pointer and length of the slice that would have been returned by this function[2]. In the original code, the `len` method used to obtain the length here is provided by the trait `ExactSizeIterator` which `IntoIter` implements. For simplicity, and to avoid traits, we model it simply as a method of `IntoIter` directly (see Fig. 5.9 at the bottom). In our case, where `T` is never zero-sized, the method returns the distance from `ptr` to `end`, which we obtain in our implementation of the function by using `offset_from` (the Rust function that is modelled in Prusti by the `address_from` function we have seen before).

Next, the `IntoIter` object's invariant needs to be unpacked in order to allow the fields to be overwritten and to get access to the permissions that are to be dropped. This time, since `self` is behind a reference, we need the statements on lines 7-8 of Fig. 5.9 to open the reference before we can unpack the invariant.

Once the fields are overwritten, the `own_range` that was originally held by `self` gets dropped. We modelled `drop_in_place` (Fig. 5.10) as an unimplemented, trusted function taking the `own_range` of the first `len` elements starting from `to_drop`, and returning the `raw_range` for the same memory area.

We end the `forget_allocation_drop_remaining` function by packing the invariant of the (now completely overwritten) `IntoIter` and closing the reference again. Then, we add a postcondition, giving back a `raw_range` for the whole length of the original allocation, now that the initialised elements in the middle have been dropped. Fig. 5.11 shows this postcondition: the `raw_range` permissions and `raw_dealloc` condition (lines 6-15) that were taken out of `old(self)`'s invariant are now handed back to `from_iter` on their own. Note that `old()` is used on all mentions of fields in this postcondition, as it refers back to the state of `self` when it entered the function, before all its fields were overwritten.

These returned permissions will be used by `from_iter` to build the new vector from, using `Vec::from_raw_parts`. This vector should be initialised up to the length that

[2]In fact, this pointer and length are what are used to define the slice in the first place.

had been returned by `collect_in_place`, meaning that we need to restore the stashed permissions that we hid away before we built `src` from `prep`.

However, in order to be able to restore the `own_range` on `src`, there is a condition that needs to hold: the elements of `src` in the stashed memory region must not have changed. We can use `bytes!` to add this condition to the postcondition of `forget_allocation_drop_remaining`; see lines 17-31 in Fig. 5.11.

The problem is that using `bytes!` does not work while the permissions of `self` are defined in the invariant of `IntoIter`, due to a precondition of `bytes!` itself. So instead, we have to move the `IntoIter`'s permissions from the invariant into the precondition of `forget_allocation_drop_remaining`. Fig. 5.12 shows the resulting precondition of `forget_allocation_drop_remaining`. At the same time, we remove these lines from the invariant of `IntoIter`.

Function from_iter_ending after forget_allocation_drop_remaining returns After `forget_allocation_drop_remaining` returns, we restore the stashed permissions via `restore_stash_range!` (line 25 in Fig. 5.7).

Once the permissions are unstashed again in `from_iter_ending`, we can finally build the new vector by calling the function `Vec::from_raw_parts`, whose implementation and specifications are shown in Fig. 5.13. Its precondition requires all the conditions of `Vec`'s invariant to hold (so that the function can build a `Vec` with this invariant). The postconditions simply ensure that the length and capacity of the returned `Vec` correspond to the values given as arguments.

5.2.2 Final verification specifications and results

Let us summarise how we modelled the relevant functions and their specifications now. We have our version of the `IntoIter` struct and the final version of its invariant in Fig. 5.14, which does not contain access permissions anymore as they have been moved to the pre-condition of `IntoIter::forget_allocation_drop_remaining` (see below).

Our special variant of this struct, `IntoIterAfterCollect` in Fig. 5.6, is used to model the state of the iterator after `collect_in_place` has returned. It is used as an input to `from_iter_ending`, where only the last few lines of the original `from_iter` function are included. In comparison to `IntoIter`, this struct has an additional field `len`, used for indicating its first range of initialised memory, as seen in its invariant.

The implementation of `IntoIter::forget_allocation_drop_remaining` is shown in Fig. 5.9, with Prusti annotations to unpack and re-pack the invariant. It returns permissions to the old range of the `IntoIter`, but for `from_iter` to be able to restore the stashed range it needs to also guarantee that the bytes in the stashed region have not changed. Due to the use of `bytes!` in the postcondition of `forget_allocation_drop_remaining` (Fig. 5.11), the access permissions to the `IntoIter`'s

```rust
impl IntoIter {
    #[no_panic_ensures_postcondition]
    #[structural_ensures(
        std::mem::size_of::<T>() != 0
        && old(self.cap) != 0 ==> (true
            && raw_range(
                old(self.buf),
                std::mem::size_of::<T>(),
                0,
                old(self.cap))

            && raw_dealloc!(
                *self.buf,
                std::mem::size_of::<T>() * old(self.cap),
                std::mem::align_of::<T>())

            && forall(|index: Int|
                (Int::new(0) <= index
                && index < address_from(old(self.ptr),old(self.buf)))
                ==>
                old(bytes!(
                        *(address_offset(old(self.buf), index)),
                        std::mem::size_of::<T>()))
                 == bytes!(
                        *(address_offset(old(self.buf), index)),
                        std::mem::size_of::<T>()),
                triggers=[
                    (bytes!(
                        *(address_offset(old(self.buf), index)),
                        std::mem::size_of::<T>()),),]
            )

        )
    )]
    unsafe fn forget_allocation_drop_remaining(&mut self) {
        /* ... */
    }
}
```

Figure 5.11: Postcondition of forget_allocation_drop_remaining

```
impl IntoIter {
    #[no_panic_ensures_postcondition]
    #[structural_requires(
        self.cap != 0 ==> (true
            && raw_range(
                self.buf,
                std::mem::size_of::<T>(),
                0,
                address_from(self.ptr,self.buf).to_usize())
            && own_range!(
                self.buf,
                address_from(self.ptr,self.buf).to_usize(),
                address_from(self.end,self.buf).to_usize())
            && raw_range(
                self.buf,
                std::mem::size_of::<T>(),
                address_from(self.end,self.buf).to_usize(),
                self.cap)

            && raw_dealloc!(
                *self.buf,
                std::mem::size_of::<T>() * self.cap,
                std::mem::align_of::<T>())
        )
    )]
    #[structural_ensures(
        /* ... */
    )]
    unsafe fn forget_allocation_drop_remaining(&mut self) {
        /* ... */
    }
}
```

Figure 5.12: Precondition of `forget_allocation_drop_remaining`

```rust
impl Vec {
    #[no_panic]
    #[no_panic_ensures_postcondition]
    #[structural_requires(
        std::mem::size_of::<T>() * capacity <= (isize::MAX as usize)
        && !ptr.is_null()
        && length <= capacity
        && (
            if std::mem::size_of::<T>() != 0 {
                (capacity != 0 ==> (
                    own_range!(
                        ptr, 0, length)
                    && raw_range(
                        ptr,
                        std::mem::size_of::<T>(),
                        length,
                        capacity)
                    && raw_dealloc!(
                        *ptr,
                        std::mem::size_of::<T>() * capacity,
                        std::mem::align_of::<T>())
                ))
            } else {
                capacity == usize::MAX
                && own_range!(
                    ptr, 0, length)
                && raw_range(
                    ptr,
                    std::mem::size_of::<T>(),
                    length,
                    capacity)
            }
        )
    )]
    #[ensures(result.len() == length)]
    #[ensures(result.capacity() == capacity)]
    pub unsafe fn from_raw_parts(ptr: *mut T, length: usize,
                                 capacity: usize) -> Vec {
        Vec {
            ptr,
            len: length,
            cap: capacity,
        }
    }
}
```

Figure 5.13: Vec::from_raw_parts with its pre- and postconditions

```
#[structural_invariant(
    Int::new_usize(std::mem::size_of::<T>())
        * Int::new_usize(self.cap)
        <= Int::new_usize(isize::MAX as usize)
    && !self.buf.is_null()

    && same_allocation(self.buf, self.ptr)
    && same_allocation(self.buf, self.end)

    && Int::new_isize(0)  <= address_from(self.ptr, self.buf)
    && address_from(self.ptr, self.buf)
        <= address_from(self.end, self.buf)
    && address_from(self.end, self.buf) <= Int::new_usize(self.cap)

    && std::mem::size_of::<T>() != 0
)]
struct IntoIter {
    buf: *const T,
    cap: usize,
    ptr: *const T,
    end: *const T,
}
```

Figure 5.14: IntoIter and its invariant, final version without permission ranges

memory allocation had to be moved from its invariant to the precondition of `forget_allocation_drop_remaining` (Fig. 5.12), and as a consequence `forget_allocation_drop_remaining` had to be made an `unsafe` function.

Lastly, our function `from_iter_ending` in Fig. 5.7 models the last few lines of the original `Vec::from_iter`. It takes an object of type `IntoIterAfterCollect` as an input and constructs a normal `IntoIter` from it after stashing away the initialised range of the first `len` elements that a normal `IntoIter` would not have. It then proceeds by overwriting the fields of the input in order to make it forget about the allocation now used by `src`.

The newly created `IntoIter src` is handed to `forget_allocation_drop_remaining`, where the `IntoIter` forgets about the allocation as well and the middle range of initialised memory is dropped. After `forget_allocation_drop_remaining` returns, the stashed permissions are restored so that the new vector can be built using `Vec::from_raw_parts`, and with `len` as its length. This vector is the return value of the function.

Verification result Unfortunately, we have not been able to verify this example so far. The `raw_dealloc!`-condition we defined in the invariant of `IntoIterAfterCollect` (Fig. 5.6, lines 37-40) caused a panic. Since it was not directly

```
#[no_panic_ensures_postcondition]
#[ensures(result.0 == self.ptr as *mut T)]
#[ensures(result.1 == address_from(self.end, self.ptr).to_usize())]
unsafe fn as_raw_mut_slice(&mut self) -> (*mut T, usize) {
    (self.ptr as *mut T, self.len())
}
```

Figure 5.15: Function as_raw_mut_slice

clear from the error reported what the problem was, we temporarily removed the `raw_dealloc!`-permissions not only from the invariant of `IntoIterAfterCollect`, but also from the pre- and postcondition of `forget_allocation_drop_remaining`, which relied on the permissions of the invariant.

With these altered specifications, we tried verifying the program again. Now, there seems to be a problem with restoring the stashed range, as we get an internal Prusti error on line 25. Nonetheless, we have been able to verify `from_iter_ending` up to this point, i.e. up to line 24, with the rest of the code starting from line 25 commented out.

Furthermore, remember that we replaced the call to method `as_raw_mut_slice` in `forget_allocation_drop_remaining` by directly assigning a pair of pointer and length (see Fig. 5.9). Initially, we wanted to use a function that would simply return exactly this pair (Fig. 5.15). Unfortunately, Prusti was not able to prove in this case that we have enough permissions to drop the `remaining`-range on line 22 despite the postcondition we wrote for this function. Therefore, we decided to use the pair directly instead.

Potentially our code should actually work the way we described it, but error reporting in this case does not make for a good enough debugging experience yet for us to have found out what is causing these errors, hence we have not been able to make any further progress thus far.

5.3 Similar examples and verification patterns

In this example we have a type, `IntoIter`, that we need to model in two different states, namely its normal state, and in the state it is in after `collect_in_place`. We use the same trick as in Chapter 3 to work around the fact that we cannot define multiple invariants on one struct: we define an extra struct `IntoIterAfterCollect`, whose invariant describes the state after `collect_in_place`, while the invariant of `IntoIter` specifies the normal state.

However, having two structs means that in order to switch from `IntoIterAfter-Collect` to `IntoIter`, we need to make the `IntoIterAfterCollect` forget about the allocation by overwriting all its fields (just as we did for `Vec` and `BrokenVec` in Chapter 3, and just like `forget_allocation_drop_remaining` forgets about the allocation of `src` in the actual code of this example.

Another interesting aspect of this example is the fact that we need to hide some of the permissions using Prusti's `stash_range!` macro before we can call the `forget_allocation_drop_remaining` method. Later, we need to restore them using `restore_stash_range!`. The reason we need to hide the `own_range` at the first `len` elements is because `fadr` only expects initialised elements between the `ptr` and `end` pointers, but the first `len` elements are actually also initialised in this case. Despite not expecting the front area of the vector to be initialised, `forget_allocation_drop_remaining` preserves these elements in their original state, making it possible to restore the hidden permissions after `forget_allocation_drop_remaining` returns, and to continue the execution of `from_iter` *with* the `own_range!` at the front of the vector.

This verification technique can therefore be applied for any code where a similar situation occurs: whenever a function to be called expects uninitialised data in some range where the data is actually initialised, permissions can be hidden using `stash_range!` so as to make the data look like what the callee function expects. If the data of the stashed range is preserved throughout the execution of this function, then the permissions can be restored once it returns, allowing the caller function to proceed with its execution knowing again that the data in this range is initialised.

5.4 Suggestions for Prusti

As we already stated in Sec. 3.4, it would be very practical if Prusti allowed several switchable invariants to be specified on one struct. Then we would not need to define an extra struct and exchange one struct for the other, having to always overwrite the fields of one of them.

Furthermore, if there was a way of making `bytes!` work with permissions defined in the invariant of `IntoIter`, and not only if they are defined in the precondition of `forget_allocation_drop_remaining`, then we would not have to turn `forget_allocation_drop_remaining` into an unsafe function during our verification. `forget_allocation_drop_remaining` remaining a safe function is desirable, since functions that are safe in the program should remain so during the verification.

Chapter 6

Example 4: std::vec::Vec::dedup_by

Our final verification example is another one from the standard library: it is about `Vec::dedup_by`. This function is not one of those where Rudra found a bug, but we chose it because it was similar to another function on the Rudra list: the `String::retain` method.

`String::retain` iterates through a `String` and retains only the characters specified by a predicate that was given as an argument. During this computation, the type invariant of `String` is broken temporarily, but it is restored again by the time the end of the `String` is reached and the function returns. The bug that was found in that function was that the programmers had forgotten to make sure that the invariant is restored in the case of a panic as well.

`std::string::String` consists of a vector of bytes, and its type invariant states that it must always be UTF-8 encoded. If we were to verify the `String::retain` method, we would therefore have to prove that this UTF-8 property holds at the end of the execution, as well as when the `String` is dropped due to a panic. Proving that the UTF-8 property holds would be a complicated and tedious task, but at the same time, UTF-8 is not the main point of this verification.

So instead of `String::retain`, we chose to verify `Vec::dedup_by`, which works in a very similar way to how `String::retain` does and uses the same programming concepts. However, in contrast to `retain` it operates on `Vec`, a struct with a much simpler-to-prove type invariant than `String`.

In Sec. 6.1, we will present the original Rust code of the `Vec::dedup_by` method and explain the drop guard concept used to guarantee memory safety in the case of a panic during the execution of `dedup_by`. Since verifying this function with Prusti was not straightforward in this case, we decided to model this example in Viper instead, which will be shown in Sec. 6.2.1. Sec. 6.3 notes that there is a widespread use of the drop guard concept, allowing ideas and techniques for verifying this example to be applied to many similar examples as well. Finally, we conclude the chapter by discussing what Prusti features would be needed to verify the `dedup_by` method in Prusti.

6.1 Context, features, and concepts

The Vec::dedup_by method The `dedup_by` method takes a closure `same_bucket` as an argument. This closure defines some equality relation on the element type of the vector. Given the references to two vector elements it decides whether the two elements are 'equal' with respect to this relation.

The `dedup_by` method iterates through the vector, compares elements, and removes any consecutive duplicates according to this equality relation [23] [24]. In the following we are going to present the original code of the function (Figures 6.1-6.3[1]) and explain how it works.

First of all, the code takes care of the special case: if the vector contains at most one element, then there are no elements to compare. The function does not need to do anything and can return immediately (see Fig. 6.2, lines 6-9).

In the general case, the function goes on to define a struct `FillGapOnDrop` (Fig. 6.2), the so-called 'drop guard', which is used to ensure memory safety even when there is a panic and all of the allocated objects of the function are dropped. On line 10 in Fig. 6.1 an instance of `FillGapOnDrop`, `gap`, is defined. It stores a mutable reference to the vector itself, as well as two indices, `read` and `write`, both initialised to 1.

The main part of the computation takes place in the while-loop. In every iteration, the `same_bucket` equality function is used to compare a new element with the most recent 'accepted' element (Fig. 6.1, line 24). Here, `read_ptr` points to the new element we read, at index `gap.read`, while `prev_ptr` points to one element before index `write`. It is the last element of the ones we decided to keep in the vector so far.

If the two compared elements are equal according to `same_bucket`'s definition, then `gap.read` is incremented and the element at the `read_ptr` is dropped. If they are *not* equal according to `same_bucket`, however, then we keep both elements in the vector. Since there might be a gap between the element at `prev_ptr` and the `read_ptr` due to elements that had to be dropped previously, we need to copy the new element at `read_ptr` to the front at `write_ptr` (line 37) to make sure all the retained elements are in one consecutive range. Because of this new element being accepted and written at `gap.write`, not only `gap.read`, but also `gap.write` has to be incremented in this case.

[1]We have moved the special case for $l \leq 1$, as well as the definition of the drop guard struct `FillGapOnDrop` and its drop handler into separate Figures 6.2 and 6.3, while the main part of the function is shown in Fig. 6.1.

```rust
impl<T, A: Allocator> Vec<T, A> {
    pub fn dedup_by<F>(&mut self, mut same_bucket: F)
    where
        F: FnMut(&mut T, &mut T) -> bool,
    {
        /* ... (special case l <= 1) */
        /* ... (struct FillGapOnDrop) */
        /* ... (impl Drop for FillGapOnDrop) */

        let mut gap = FillGapOnDrop { read: 1, write: 1, vec: self };
        let ptr = gap.vec.as_mut_ptr();

        /* Drop items while going through Vec, it should be more
         * efficient than doing slice partition_dedup + truncate */
        /* SAFETY:
         * Because of the invariant, read_ptr, prev_ptr and write_ptr
         * are always in-bounds and read_ptr never aliases prev_ptr
         */
        unsafe {
            while gap.read < len {
                let read_ptr = ptr.add(gap.read);
                let prev_ptr = ptr.add(gap.write.wrapping_sub(1));

                if same_bucket(&mut *read_ptr, &mut *prev_ptr) {
                    // Increase `gap.read` now
                    // since the drop may panic.
                    gap.read += 1;
                    /* We have found duplicate, drop it in-place */
                    ptr::drop_in_place(read_ptr);
                } else {
                    let write_ptr = ptr.add(gap.write);

                    /* Because `read_ptr` can be equal to
                     * `write_ptr`, we either have to use `copy`
                     * or conditional `copy_nonoverlapping`.
                     * Looks like the first option is faster. */
                    ptr::copy(read_ptr, write_ptr, 1);

                    /* We have filled that place, so go further */
                    gap.write += 1;
                    gap.read += 1;
                }
            }
```

Figure 6.1: The Vec::dedup_by method [23]
(continued on the next page)

```
            /* Technically we could let `gap` clean up with its Drop,
             * but when `same_bucket` is guaranteed to not panic,
             * this bloats a little the codegen,
             * so we just do it manually */
            gap.vec.set_len(gap.write);
            mem::forget(gap);
        }
    }
}
```

Figure 6.1: The Vec::dedup_by method [23] (cont.)

```
impl<T, A: Allocator> Vec<T, A> {
    pub fn dedup_by<F>(&mut self, mut same_bucket: F)
    where
        F: FnMut(&mut T, &mut T) -> bool,
    {
        let len = self.len();
        if len <= 1 {
            return;
        }

        /* INVARIANT: vec.len() > read >= write > write-1 >= 0 */
        struct FillGapOnDrop<'a, T, A: core::alloc::Allocator> {
            /* Offset of the element we want to check
             * if it is duplicate */
            read: usize,

            /* Offset of the place where we want to place
             * the non-duplicate when we find it. */
            write: usize,

            /* The Vec that would need correction
             * if `same_bucket` panicked */
            vec: &'a mut Vec<T, A>,
        }

        /* impl Drop for FillGapOnDrop */
        /* ... dedup_by computation ... */
    }
}
```

Figure 6.2: The special case $l \leq 1$ of dedup_by and the drop guard FillGapOnDrop

Consider the situation at this point with regard to initialisation: the first few elements, up to and including the one at `prev_ptr`, are the elements we decided to keep in the vector. They are initialised, valid elements. The range starting from `read_ptr`, i.e. `[gap.read..gap.vec.len()]` holds the elements that are still to be read and compared, i.e. the ones for which it has not been decided yet whether they are kept or dropped. They are still in the original state from when the function was called, so they are also initialised, valid vector elements.

However, as soon as an element has to be dropped, a gap appears between these two ranges of the vector, where the elements have either been dropped or moved. Consequently, the elements in this gap range might not be initialised even though according to the type invariant the whole range `vec[0..gap.vec.len()]` of the vector should always be initialised. The vector's invariant is therefore broken if such a gap appears.

This broken invariant is not a problem if the while-loop is able to run panic-free to completion, i.e. until the read index `gap.read` reaches the end of the vector. At this point, all the elements have been checked, and the ones to retain are located in the range `[0..gap.write]`. By setting the length of the vector to `gap.write` with the `set_len` method on line 49, the vector's invariant can be restored and the function can therefore return with the invariant intact.

However, if a panic occurs during one of the iterations, there are additional measures to be taken in order to ensure memory safety. Without the drop guard, the vector would be dropped as the function is exited. Dropping the vector in its broken state where some of the elements might not be initialised could result in undefined behaviour.

The drop guard's purpose is to prevent that the vector is dropped in its broken state. It stores a mutable reference to the vector itself as well as the write and read indices that indicate the start and end of the gap in the vector, respectively (Fig. 6.2. When a panic occurs and all objects held by the function are freed, `gap`'s `drop` method gets executed. The task of this function is to get the vector into a state where it is safe to drop the vector, i.e. a state where the type invariant is no longer violated.

Fig. 6.3 shows the implementation of `FillGapOnDrop`'s drop method. It copies the left-over elements (i.e. the ones that `dedup_by` could not read anymore before the panic) from `vec[read..]` to `vec[write..(write + items_left)]` so that they are located just after the ones that were accepted and collected at the front of the vector during the regular execution. After this copy operation there is one consecutive region of valid, initialised elements at the front of the vector, while the rest is at the back. By subtracting the number of dropped elements from the length of the vector and setting this new length of the vector using `set_len`, the vector's type invariant can be restored. Therefore, after `gap`'s drop handler completes its execution, it is now safe to drop the vector itself, without the risk of any double frees occurring.

```rust
impl<T, A: Allocator> Vec<T, A> {
    pub fn dedup_by<F>(&mut self, mut same_bucket: F)
    where
        F: FnMut(&mut T, &mut T) -> bool,
    {
        /* ... */

        impl<'a, T, A: core::alloc::Allocator> Drop
        for FillGapOnDrop<'a, T, A> {
            fn drop(&mut self) {
                /* This code gets executed when `same_bucket`
                 * panics */
                /* SAFETY: invariant guarantees that `read - write`
                 * and `len - read` never overflow and that the copy
                 * is always in-bounds. */
                unsafe {
                    let ptr = self.vec.as_mut_ptr();
                    let len = self.vec.len();

                    /* How many items were left when `same_bucket`
                     * panicked. Basically vec[read..].len() */
                    let items_left = len.wrapping_sub(self.read);

                    /* Pointer to first item in
                     * vec[write..write+items_left] slice */
                    let dropped_ptr = ptr.add(self.write);
                    /* Pointer to first item in vec[read..] slice */
                    let valid_ptr = ptr.add(self.read);

                    /* Copy `vec[read..]` to
                     * `vec[write..write+items_left]`.
                     * The slices can overlap,
                     * so `copy_nonoverlapping` cannot be used */
                    ptr::copy(valid_ptr, dropped_ptr, items_left);

                    /* How many items have been already dropped
                     * Basically vec[read..write].len() */
                    let dropped = self.read.wrapping_sub(self.write);

                    self.vec.set_len(len - dropped);
                }
            }
        }
        /* ... */
    }
}
```

Figure 6.3: The drop handler of FillGapOnDrop

Note that since we only need this drop handler to be executed in the case of a panic, we call `mem::forget` at the end of the happy path (when the invariant holds again). This function makes `dedup_by` forget about the `gap` object and thus prevents `gap` from being dropped when it goes out of scope at the end of `dedup_by`.

6.2 Verification of Vec::dedup_by in Viper

The next task now would be to verify `Vec::dedup_by` in Prusti. However, when it comes to verification with Prusti, there are some uncertainties regarding what Prusti specifications we would need to write in order to verify this code. For example, it is unclear how to write specifications for the copy functions, especially the 'normal' `ptr::copy` function where the source and destination ranges are allowed to overlap. This potential overlap makes it more complicated and difficult to handle the access permissions for these two ranges of the vector appropriately. In order to get a better understanding of how to specify code like this and to see what we would need in Prusti to verify this example, we decided to model and verify the example in Viper instead.

In order to be able to implement and verify `Vec::dedup_by` in Viper, we needed to find a way to represent all the necessary data structures (most importantly of course the `Vec` itself). Further, we had to model memory and addresses, defining the access permissions required to access locations in our model of memory.

In Sec. 6.2.1 we discuss our implementation of the `dedup_by` method in Viper, and explain the important aspects of our surrounding Viper model as we make use of it for our main implementation and verification. In some cases, we will refer to the appendix for more detailed implementation and information. In Sec. 6.2.2 we will compare the memory model we used for our Viper model in Sec. 6.2.1 with the Rust memory model with regard to certain aspects. Then we will present some ideas for alternative Viper models to better approximate Rust's model.

6.2.1 Modelling dedup_by in Viper

The `dedup_by` method operates on an instance of type `Vec`, whose type invariant we have seen in previous chapters: the capacity is the number of elements allocated, and the vector is initialised up to its length. In Sec. 6.1 we explained how this invariant gets temporarily broken during the execution of the `dedup_by` method – but we also saw that the invariant gets restored before the `Vec` instance either returns (happy path) or is dropped (in case of a panic). We therefore have to make sure in our Viper model that the method starts *and ends* with a vector with an intact invariant.

Vec model Fig. 6.4 shows how we model the `Vec` type in Viper: we define three fields, `pointer`, `length`, and `capacity`, as well as a predicate `OwnVec`, holding the

```
 1  /** Vector Fields */
 2  field pointer: Address
 3  field length: Int
 4  field capacity: Int
 5
 6  predicate OwnVec (self: Ref) {
 7    /* Vec basics:  */
 8    Vec_basics(self) &&
 9
10    /* Vec access ranges: */
11    (own_range(self.pointer, 0, self.length)) &&
12    (raw_range(self.pointer, self.length, self.capacity, size_of()))
13  }
14
15  define Vec_basics(self)
16    acc(self.pointer) &&
17    acc(self.length) &&
18    acc(self.capacity) &&
19
20    get_offset(self.pointer) == 0 &&
21    0 <= self.length &&
22    self.length <= self.capacity &&
23    self.capacity == alloc_size(get_allocation(self.pointer))
```

Figure 6.4: Model for the Vec struct

vector's properties and permissions. For convenience, we use a macro, `Vec_basics`, to hold the basic properties of the vector.

A Viper predicate like `OwnVec` is a way to name an assertion. It can have parameters that are used in the assertion held in its body [25]. The parameter of `OwnVec` is of the Viper built-in type `Ref`. Values of this type (except null) represent potential receiver objects for field accesses. Permissions to a field `f` are denoted `acc(x.f)`, where `x` is a `Ref` value. Therefore, lines 16-18 in Fig. 6.5 denote that `OwnVec` holds access permissions to all three vector fields. Furthermore, the usual size relations hold for `length` and `capacity`, as shown on lines 21-22.

Viper domains allow users to define custom types [25]. The type of the `pointer` field, `Address`, is such a custom type, that we defined in order to represent addresses in our Viper model. Each address in our model is intended as the location of one byte in memory. It consists of an allocation and an integer offset from the base address of this allocation.

To model allocations, we defined another domain type, `Allocation`. An `Allocation` has a size, which can be obtained through its domain function `alloc_size`.

The domain functions of `Address` are the constructor `create_address`, which takes

```
predicate Raw_Byte(addr: Address)

predicate MemoryBlock(addr: Address, ty_sz: Int)
{
  forall a: Address :: {Raw_Byte(a)}
    get_allocation(a) == get_allocation(addr) &&
    get_offset(addr) <= get_offset(a) &&
    get_offset(a) < (get_offset(addr) + ty_sz)
    ==> Raw_Byte(a)
}

function valid_element(address: Address, size: Int): Bool
  requires acc(MemoryBlock(address, size), wildcard)

predicate Own(addr: Address)
{
  MemoryBlock(addr, size_of()) &&
  valid_element(addr,size_of())
}
```

Figure 6.5: Raw_Byte, MemoryBlock, valid_element, and Own

as input an `Allocation` and an `Int` and returns the corresponding `Address`, and the two destructors, `get_allocation`, which returns the address's allocation, and `get_offset`, which returns the byte offset of the address within its allocation (i.e. the address's distance from the allocation's base address, in number of bytes)[2].

Since a vector corresponds to one allocation in memory, `pointer` must point to the start of the allocation (i.e. the address with offset zero), while the capacity corresponds to the size of the allocation pointed to by `pointer`. This property is defined on lines 20 and 23.

On lines 11-12 we declare the access permissions held by the vector in a similar way to how we did it in Prusti in previous chapters: the vector is initialised up to `self.length`, thus holds the right to read the elements in this range, while it only holds raw access rights to the rest of the allocation. The two macros `own_range` and `raw_range` that we use here will be presented in the following paragraph.

Memory access permissions and ZSTs Since each address in our model is intended to refer to one byte in memory, we start by defining a predicate that gives access to a single byte at a given address: `Raw_Byte`, defined as a predicate without a body, taking an `Address` as argument (see Fig. 6.5).

Vector elements can consist of several bytes (or of no bytes, in the case of a zero-sized element type). It is therefore useful to define a predicate that gives access to

[2]For more detailed information about our address and allocation model, please refer to Sec. A.3

```
function offset(addr: Address, idx: Int) : Address
{
  create_addr(
    get_allocation(addr),
    (get_offset(addr) + mul(size_of(), idx))
  )
}
```

Figure 6.6: Function offset

a range of several raw bytes, a `MemoryBlock`. As shown in Fig. 6.5, this predicate takes an additional integer argument `ty_sz` and provides `Raw_Byte`-access to this number `ty_sz` of bytes, starting from `Address addr`.

The Boolean function `valid_element` returns true if the contents of a given memory block represent a valid element (of the vector element type). Combining access to a memory block and the knowledge that the element at this place is valid, the `Own` predicate is used to assert the right to not only write to this memory, but to also read it. Different from the `MemoryBlock` predicate, `Own` does not get the size of the block as an argument. Rather, it obtains it from the `size_of` function directly.

We use the `size_of` function to model obtaining the size of the vector's element type, similarly to how Rust's `std::mem::size_of<T>` can be used to obtain the size of an arbitrary type `T`. Here, we defined `size_of` as an abstract function with a postcondition ensuring that the result is greater or equal to zero:

```
function size_of(): Int
  ensures result >= 0
```

For convenience we would like to have a way of describing access permissions for a whole range of a vector's elements, similar to Prusti's `raw_range` and `own_range!`. We first define a helper function `offset` (Fig. 6.6) to calculate the offset from an address (similar to what Rust's `pointer::offset` does). Note that here, the offset `idx` is given in units of the vector's element type (not in bytes): given the address `addr` of a vector element and an offset `idx`, it computes the address of the `idx`th element after the element at `addr`.

Multiplication is undecidable, and as a consequence, the prover sometimes needs manual guidance when multiplications are included in the code. Therefore, we defined our own function `mul`, which allows us to help the verifier with manual instructions (e.g. postconditions or lemmas).

Now we want to define our range access permissions in the style of Prusti's `raw_range`. Our first idea is to define a macro as follows:

```
define raw_range(ptr, from, to, ty_sz)
  forall o: Int :: { offset(ptr, o) }
    from <= o && o < to ==> MemoryBlock(offset(ptr, o), ty_sz)
```

With this macro we can obtain access to a whole range of memory blocks. Unfortunately, this definition of the macro is not compatible with zero-sized types (ZSTs), and we will explain why below.

Consider a vector of zero-sized-typed elements stored at an address a with an allocation *alloc* and byte-offset x within this allocation. This vector's elements do not occupy any space, and as a consequence all of them are located at the same address a, as illustrated by applying the `offset` function to $a = \text{create\_addr}(\text{alloc}, x)$ for any offset *idx*:

$$\begin{aligned}\text{offset}(a, \text{idx}) &= \text{offset}(\text{create\_addr}(\text{alloc}, x), \text{idx}) \\ &= \text{create\_addr}(\text{alloc}, x + \text{idx} * 0) \\ &= \text{create\_addr}(\text{alloc}, x) \\ &= a\end{aligned}$$

As a consequence, for all indices in the definition of the `raw_range` macro above, we have that $\text{offset}(a, \text{idx}) = a$ i.e. for each index the same address is handed to `MemoryBlock` as an argument. This means that the receiver expression of the quantified permissions, `MemoryBlock(offset(ptr, o), ty_sz)`, is not injective, which Viper does not allow.

In order to make the receiver expression injective even for zero-sized types, we introduce a new predicate `Idx_MemoryBlock` as an additional 'level' between the `MemoryBlock` predicate and the `raw_range` macro, and adapt the macro's definition accordingly (see Fig. 6.7). Analogously, we define a macro `own_range` for ranges of initialised vector elements, and the predicate `Idx_Own` between the `Own` predicate and this macro.

Pre- and postcondition of dedup_by Fig. 6.8 shows our model of the `dedup_by` method in Viper. The loop body, where the main part of the computation takes place, will be discussed in detail shortly; for now, we omit it for brevity.

We put the `OwnVec` predicate in both the pre- and the postcondition of the method to model how the invariant of `Vec` holds both at the start and at the end of `dedup-by`. A Viper predicate instance is not equivalent to its body, therefore holding `OwnVec` is not enough to access the permissions and assertions held inside its body [25]. In order to be able to make use of them inside our `dedup_by` method, we use `unfold` to exchange the `OwnVec` instance for its body. This unfolding operation is similar to the `unpack!` operation in Prusti that we have used in the previous examples. At the end of the method we need to perform the inverse operation, folding `OwnVec`'s body back into the `OwnVec` predicate. This `fold` operation is thus similar to Prusti's `pack!`.

Special case $l \leq 1$ In the special case where the length of the vector is smaller than or equal to one, there are no elements to compare, meaning we do not need

```
predicate Idx_MemoryBlock(addr: Address, idx: Int, ty_sz: Int)
{
  MemoryBlock(offset(addr,idx),ty_sz)
}

predicate Idx_Own(addr: Address, idx: Int)
{
  Own(offset(addr,idx))
}

define own_range(ptr,from,to)
  forall o: Int :: { Idx_Own(ptr,o) }
    from <= o && o < to ==> Idx_Own(ptr,o)

define raw_range(ptr,from,to,ty_sz)
  forall o: Int :: { Idx_MemoryBlock(ptr,o,ty_sz) }
    from <= o && o < to ==> Idx_MemoryBlock(ptr,o,ty_sz)
```

Figure 6.7: Predicates Idx_MemoryBlock and Idx_Own, and macros raw_range and own_range

to do anything in this `if`-clause and can directly continue on to the `fold` statement at the very end of the method.

The general case and the while-loop invariant The `else`-clause in Fig. 6.8 contains the general case (i.e. `l > 1`), with the while-loop. In case of a panic, the execution will jump directly out of the loop to the `exit_with_panic` label (more about this later), so the `set_len` instruction is only executed if the while-loop was able to successfully execute until the end. By setting the correct new length, the vector's invariant is restored, allowing us to fold the `OwnVec` predicate.

In order to verify our while-loop we need to define an invariant for it. Lines 20-22 hold the access permissions for all of the three vector fields, and ensure that they remain unchanged throughout the execution of the loop.

The chain of inequalities on line 23 corresponds to the invariant described in the comments for the struct `FillGapOnDrop` in the original code (see Fig. 6.2, line 11). However, we had to tweak the first of the conditions slightly because Viper invariants are not only required to hold when going from one iteration to the next, but also at the end of a loop, when the invariant's knowledge is transferred back to the enclosing context. Since `l == r` is possible when the loop exits, we need to put `l >= r` in the invariant instead of `l > r`. Note that `l == r` is *only* possible upon exiting the loop after the whole loop was able to run completely to the end, without any panic occurring.

Finally, lines 24-26 describe the access permissions within the range `[0..l]` of the vector during the execution of the loop. As we can see, these permissions

```
method dedup_by(self: Ref)
  requires OwnVec(self)
  ensures  OwnVec(self)
{
  unfold OwnVec(self)

  var vec_ptr: Address := self.pointer
  var cap: Int := self.capacity
  var l : Int := self.length

  if (l <= 1) {
    /* Do Nothing */
  } else {
    var r : Int := 1
    var w : Int := 1

    /* Lemmas for offset fn */

    while(r < l)
      invariant acc(self.pointer)  && self.pointer  == vec_ptr
      invariant acc(self.length)   && self.length   == l
      invariant acc(self.capacity) && self.capacity == cap
      invariant l >= r  && r >= w && w > w - 1 && w - 1 >= 0
      invariant own_range(self.pointer,0,w)
      invariant raw_range(self.pointer,w,r,size_of())
      invariant own_range(self.pointer,r,l)
    {
      /* LOOP BODY */
    }
    set_len(self,w)
    label exit_with_panic
  }
  fold OwnVec(self)
}
```

Figure 6.8: Method dedup_by
Loop body omitted

```
method same_bucket(a: Address, b: Address)
  returns (same: Bool, panic: Bool)
  requires acc(Own(a), 1/2) && acc(Own(b), 1/2)
  ensures  acc(Own(a), 1/2) && acc(Own(b), 1/2)
```

Figure 6.9: Abstract method same_bucket

correspond exactly to the broken state of the vector invariant during the execution of the while loop: if the invariant was intact, we would have own-permissions for this whole range; here, however, there is a gap in the range `[w..r]`, where we only hold raw permissions. This gap needs to be closed before the function returns, which is enforced by the `fold` statement at the end of the method: as long as the permissions stated in the predicate's body are not restored, this folding operation fails.

Starting the loop iteration At the start of each iteration of this while-loop, the `same_bucket` function compares a new element (placed at 'read' index `r`) with the latest accepted element (located at index `prev`, one before 'write' index `w`).

Fig. 6.9 shows our Viper implementation of `same_bucket`. In the original Rust code, `same_bucket` is a closure given to `dedup_by` as an argument, but since there are no closures in Viper we turn `same_bucket` into its own, separate method. As the exact behaviour of the closure is unknown, we define `same_bucket` as an abstract method (i.e. a method with no body) and only describe the access permissions it takes and returns.

Our `same_bucket` method requires the `Own` predicate for both its arguments (i.e. it has the permission to read (and write) the vector elements at addresses `a` and `b`), and fully returns these permissions to the caller afterwards.

The `own_range` in the loop invariant consists of `Idx_Own` predicates. We need to unfold the `Idx_Own` predicates at the addresses of the two elements to be compared, so as to obtain `Own` predicates instead (see lines 7-8 in Fig. 6.10). Only then do we have the right permissions to be able to call `same_bucket`. After `same_bucket` returns, the `Own` predicates have to be folded into `Idx_Own` predicates again (lines 14-15).

In order to be able to model a panic occurring in `same_bucket`, we let the method return a second Boolean, `panic`, to indicate whether or not a panic occurred. This Boolean return value allows us to handle the panicking and non-panicking cases of the execution via control flow in `dedup_by`. Fig. 6.11 shows the framework of the control flow in the while-loop in order to illustrate the various paths the execution can take.

If a panic 'occurs' in `same_bucket`, then the 'panic-path' is taken, where first the gap in the vector's permissions needs to be fixed, before execution breaks out of

```
while(r < l)
  /* INVARIANT */
{
  var prev: Int := w - 1
  var same: Bool; var panic: Bool

  unfold Idx_Own(self.pointer, r)
  unfold Idx_Own(self.pointer, prev)

  /* vec(idx) returns (start) address
   of element of vec at index idx */
  same, panic := same_bucket(vec(r), vec(prev))

  fold Idx_Own(self.pointer, r)
  fold Idx_Own(self.pointer, prev)

  /* ... */
}
/* ... */
```

Figure 6.10: Call site of same_bucket

```
while(r < l)
{
  /* same_bucket */

  if (panic) {
    /* PANIC-PATH */
    goto exit_with_panic

  } else {
    /* HAPPY PATH */

    if (same) {
      /* SAME-PATH */

    } else {
      /* DIFFERENT-PATH */

    }
  }
}
/* ... */
label exit_with_panic; /* Panic & Happy Paths join */
```

Figure 6.11: Skeleton of the while-loop

```
/* HAPPY PATH */
if (same) {
  r := r + 1
  unfold Idx_Own(self.pointer, r - 1)
  drop_in_place(vec(r-1))
  fold Idx_MemoryBlock(self.pointer, r - 1,size_of())
} else {
  /* DIFFERENT-PATH */
}
```

Figure 6.12: Dropping the duplicate element

```
method drop_in_place(addr: Address)
  requires Own(addr)
  ensures  MemoryBlock(addr,size_of())
```

Figure 6.13: Method drop_in_place

the loop early and goes directly to the `exit_with_panic` label, where it joins the happy path again.

The 'happy path' is split depending on the result of comparing the two elements in `same_bucket`. If the two vector elements are 'the same', the one at index `r` needs to be dropped. If the elements are not 'the same', the element at `r` is accepted and copied to the right spot at index `w` (if it was not already there).

Dropping the duplicate Fig. 6.12 shows what we do when duplicate elements are detected and we need to drop the element at index `r`. First, the index is incremented, then we call the `drop_in_place` method, which is shown in Fig. 6.13. It is an abstract method that takes an instance of the `Own` predicate for the element at address `addr`, but only returns a `MemoryBlock` predicate instance instead, thus modelling that this element has been dropped. Just as for `same_bucket`, we need to unfold the `Idx_Own` instance at the address of the element that we have to drop in order to obtain the corresponding `Own` instance needed to call the `drop_in_place` method. Afterwards, we fold the `MemoryBlock` we obtain from the method's postcondition into an `Idx_MemoryBlock` instead, then continue with the next iteration of the loop.

Copying the retained element to the right place Remember that if the vector element at index `r` is not a duplicate of the one at index `prev`, it is kept in the vector. If none of the previously checked elements had to be dropped (`r == w`), then the element is already at the right place (see the `else`-clause on lines 32-34 of Fig. 6.14).

```
/* HAPPY PATH */
if (same) {
  /* SAME-PATH */
  /* ... */
} else {
  /* DIFFERENT-PATH */
  if (r != w) {
    assert w < r

    /** Stash point */
    /* own(src-range) --> raw(src-range) */
    exhale own_range(self.pointer,r,r+1)
    inhale raw_range(self.pointer,r,r+1,size_of())
    label stash_point

    /* `MemoryBlock`s --> `Raw_Byte`s */
    range_unfold(self.pointer,r,r+1,size_of())
    range_unfold(self.pointer,w,w+1,size_of())

    copy_nonoverlapping(vec(r), vec(w), 1)

    /* `Raw_Byte`s --> `MemoryBlock`s */
    range_fold(self.pointer,r,r+1,size_of())
    range_fold(self.pointer,w,w+1,size_of())

    /* restore_stash */
    exhale raw_range(self.pointer,w,w+1,size_of()) &&
           bytes(vec(w),mul(1,size_of()))
           == old[stash_point](bytes(vec(r),mul(1,size_of())))
    inhale own_range(self.pointer,w,w+1)

  } else {
    assert r == w /* already in right place */
  }
  w := w + 1
  r := r + 1
}
```

Figure 6.14: Retaining the read element

```
method copy_byte(src: Address, dst: Address)
  requires Raw_Byte(src) && Raw_Byte(dst)
  ensures  Raw_Byte(src) && Raw_Byte(dst)
  ensures  bytes(dst,1) == old(bytes(src,1))
```

Figure 6.15: Method copy_byte

However, if there is a gap in the vector due to previously dropped elements (`r != w`), then the element at `r` needs to be copied to its new spot at index `w`, which is done in the `if`-clause in Fig. 6.14.

Unlike in the original code, we make the distinction between these two cases (`r == w` and `r != w`), because doing so allows us to use `copy_nonoverlapping` for copying the element instead of the more general `copy`, where the source and destination ranges may overlap and where therefore writing specifications is more complicated.

Copying the element with copy_nonoverlapping In our model we implement `copy_nonoverlapping` as a method calling a recursive method `copy_rec_nonoverlapping`, which ultimately uses the (abstract) method `copy_byte` shown in Fig. 6.15.

The `copy_byte` method operates on individual raw bytes, taking `Raw_Byte` access permissions to both the source and destination addresses. Afterwards, the permissions are returned to the caller.

The function `bytes` appearing in the postcondition is a function that takes as arguments an `Address` and an integer `size`, and returns the sequence of `size` bytes starting at the given address[3]. Here it is used to ensure that the byte from `src` was copied to `dst` by `copy_byte`.

Just like `copy_byte` copies single raw bytes from the source to the destination, we modelled all the copy methods to operate on raw bytes, defining their pre- and postconditions via the `Raw_Byte` predicate. The implementation of `copy_nonoverlapping`, for example, is shown in Fig. 6.16[4].

It takes as arguments the starting addresses of both the source and the destination ranges, as well as `size`, an integer indicating the number of vector elements to be copied. Multiplying this number by the size of an element (returned by `size_of`) yields the number of bytes to be copied[5]. If `size` is zero, the method simply

[3]For the detailed implementation of the `bytes` function, refer to Sec. A.3

[4]For the implementation of the `byte_offset` function used in `copy_nonoverlapping`, please refer to Sec. A.3, Fig. A.4

[5]The lemma `lemma_mul_by_zero` ensures that function `mul` returns zero if either of its arguments is zero.

```
1  method copy_nonoverlapping(src: Address, dst: Address, size: Int)
2    requires 0 <= size
3    requires size == 0 || size_of() == 0
4      || (mul(size, size_of()) >= size
5          && mul(size, size_of()) >= size_of())
6    requires size_of() == 0 ? true :
7      forall k: Int :: {byte_offset(src, k)} {byte_offset(dst, k)}
8      0 <= k && k < mul(size, size_of()) ==>
9      Raw_Byte(byte_offset(src, k)) && Raw_Byte(byte_offset(dst, k))
10   ensures  size_of() == 0 ? true :
11     forall k: Int :: {byte_offset(src, k)} {byte_offset(dst, k)}
12     0 <= k && k < mul(size, size_of()) ==>
13     Raw_Byte(byte_offset(src, k)) && Raw_Byte(byte_offset(dst, k))
14   ensures  bytes(dst, mul(size, size_of()))
15     == old(bytes(src, mul(size, size_of())))
16 {
17   lemma_mul_by_zero()
18
19   if (size == 0) {
20     /* SPECIAL CASE: nothing to copy. */
21   } else {
22     assert size > 0
23     copy_rec_nonoverlapping(src, dst, mul(size, size_of()))
24   }
25 }
```

Figure 6.16: Method copy_nonoverlapping

returns; otherwise, the recursive method is called[6]. It gets both addresses as arguments, as well as the (already calculated) number of total bytes to be copied.

The precondition on lines 6-9 requires `Raw_Byte` permissions for both the source and destination ranges, while the postcondition on lines 10-13 returns these permissions back to the caller after the function. The second postcondition, on lines 14-15, states that `copy_nonoverlapping` copies the bytes from the source range over to the destination range.

Obtaining the permissions to copy_nonoverlapping from the loop invariant As we have just seen, the permissions of the copy methods are defined on the level of the `Raw_Byte` predicate. In order to be able to call `copy_nonoverlapping`, we need to get from the `Idx_Own` and `Idx_MemoryBlock` predicates that are held by the invariant down to the `Raw_Byte`-level.

First, on lines 12-14 of Fig. 6.14, we stash away the *own*-permissions we hold for element at index `r`. To stash away the `Idx_Own` predicate instances of our

[6]For the implementation of the recursive function `copy_rec_nonoverlapping`, see Sec. A.3

own_range, we use Viper's exhale statement (line 12, meaning that we give up these permissions. Instead, we inhale the corresponding raw_range (line 13, meaning that the Idx_MemoryBlocks of this range are added to the program state (i.e. we basically *'assume'* that we have these permissions). We mark this point of the execution with the label stash_point (on line 14) so that we can refer back to it when we want to restore the stashed permissions again later. After completing this stashing operation we have now a raw_range for both the source and destination ranges.

Since it is not that easy to unfold a whole *range* of Idx_MemoryBlock predicates at once, we defined a lemma range_unfold. It is defined as an abstract method requiring a raw_range in its precondition and ensuring all the Raw_Byte predicates for the corresponding range in its postcondition[7]. We apply this lemma for both our source and destination ranges (lines 17-18 of Fig. 6.14).

After copy_nonoverlapping returns, we fold the two ranges back again on lines 23-24 with a reverse lemma range_fold[8] defined to take a range of Raw_Byte predicates and return a raw_range again.

On lines 27-30 we want to restore the stashed *own*-permissions, but on the destination range instead of the source range. In order to be allowed to take up *own*-permissions for the destination range we need to prove that the contents of that range are valid elements. We do this by exhaling the condition on lines 28-29, which states that the bytes in the destination range now correspond to the bytes in the source range at the time the permissions were stashed (i.e. at the stash_point label). As this condition is guaranteed by copy_nonoverlapping's postcondition, we are able to exhale it, together with the raw_range. Then we can inhale the own_range for the destination range instead.

After successfully copying the element at index r to its new spot at index w, we increment both indices r and w and continue on with the next iteration of the loop.

Panicking and repairing the gap Now that we have explained the execution of the happy path, we now explain what happens in the panic case. In the original Rust code, when a panic occurs in closure same_bucket, the function is exited and all object instances are dropped, including the drop guard FillGapOnDrop. The drop method of FillGapOnDrop is executed and makes sure that the Vec's invariant is intact (i.e. there is no gap in the own_range anymore) before the vector itself is dropped as well.

In our code, panicking executions are modelled in dedup_by itself, in the if-clause that gets executed when same_bucket returns the second Boolean return value panic as true (see Fig. 6.17). First, we call a method drop_FillGapOnDrop which takes the responsibility of repairing the gap that in the original code was held by

[7]For the definition of the lemma range_unfold, refer to Sec. A.3, Fig. A.6
[8]For the definition of the lemma range_fold, refer to Sec. A.3, Fig. A.7

```
/* WHILE-LOOP BODY */
/* ... */
if (panic) {
  /* PANIC! */
  drop_FillGapOnDrop(self, r, w)
  /* EXIT while-loop */
  goto exit_with_panic

} else {
  /* HAPPY PATH */
}
```

Figure 6.17: The panic path

```
method drop_FillGapOnDrop(self: Ref, r: Int, w: Int)
  requires Vec_basics(self)
  requires self.length > r && r >= w && w > w-1 && w-1 >= 0
  requires own_range(self.pointer, 0, w)
  requires raw_range(self.pointer, w, r, size_of())
  requires own_range(self.pointer, r, self.length)
  ensures  Vec_basics(self)
  ensures  self.length   == old(self.length) - (r-w) &&
           self.capacity == old(self.capacity) &&
           self.pointer  == old(self.pointer)
  ensures  own_range(self.pointer, 0, self.length)
  ensures  raw_range(
              self.pointer,
              self.length,
              old(self.length),
              size_of())
{
  /* METHOD BODY */
}
```

Figure 6.18: Pre- and postconditions of drop_FillGapOnDrop

the drop handler of `FillGapOnDrop`. Once this method has repaired the broken vector invariant, we break out of the loop directly and continue the execution at the label `exit_with_panic`.

The drop_FillGapOnDrop method As we just mentioned, the `drop_FillGapOnDrop` method models the functionality of `FillGapOnDrop`'s drop handler. We show the preconditions it takes, as well as the postconditions we want it to ensure in Fig. 6.18.

First of all, the basic properties defined in the `Vec_basics` macro hold. Remember that these are all the properties held in `OwnVec` apart from the permission ranges. Next, the inequalities in the second precondition correspond to the invariant of the `FillGapOnDrop` struct, annotated as a comment in the original code. Finally, the permissions defined on lines 4-6 describe the permissions of the broken invariant, i.e. with the gap at `[w..r]`. Note that these preconditions correspond to the invariant of the while-loop in `dedup_by`[9].

For the postcondition, the `Vec_basics` should of course hold again. Also, `self .pointer` and `self.capacity` should be unchanged, while `self.length` should have the correct new length, namely the length of the gap subtracted from the original length, as the method should keep all elements that are still valid in the vector. As for the permissions, they should correspond to the ones of the usual vector type invariant, i.e. the vector should now be initialised exactly up to its (new) length.

We can regard this postcondition as the 'target' we need to reach in order to repair the vector's broken type invariant. Fig. 6.19 shows what we do in the method's body in order to reach this target.

The task of this method is to copy the elements at source range `[r..(r + items_ left)]` to the destination range at `[w..(w + items_left)]`. The variable `items_ left` denotes the number of elements for which `dedup_by` did not have the chance to decide whether they should be kept or dropped, i.e. the elements at `[r..l]`. Unfortunately, the source range and destination range might overlap this time, meaning we cannot use `copy_nonoverlapping` but instead need to use the normal `copy` method, whose pre- and postconditions are more complicated to define due to the possible overlap.

Fig. 6.20 shows how we specify these pre- and postconditions for `copy`. This time, a `Raw_Byte` permission in the destination can only be given if this byte is not already contained in the source range. We therefore define a helper function `contains _byte` (shown in Fig. 6.21) to determine whether a specific byte is contained in a certain range of memory. The function takes an address `a` and integer `size` to describe the byte range, as well as a second address `other`. It checks whether `other` corresponds to any address that is located at a byte distance between zero and `size` from address `a`, i.e. whether `other` is contained in the range of `size` bytes starting from `a`.

In order to be able to use the `copy` method in `drop_FillGapOnDrop`, we need to reach the `Raw_Byte` level needed for `copy`, starting from the permissions defined in the precondition, just like we had to when we used `copy_nonoverlapping` in the happy path. Like then, we start by stashing away the `own_range` of the source (see

[9]The only small difference: here, it is `self.length > r`, when in the invariant it was `l >= r`. However, `l == r` is only possible when the execution leaves the while-loop after successfully completing its execution for the whole length of the vector, i.e. `drop_FillGapOnDrop` is never called in a state where `l == r` but only when `l > r`, so its precondition really does match the invariant.

```
method drop_FillGapOnDrop(self: Ref, r: Int, w: Int)
  /* PRECONDITIONS & POSTCONDITIONS */
{
  /* Lemmas for offset fn */

  var l: Int := self.length
  var items_left: Int := l - r
  var dropped_ptr: Address := offset(self.pointer, w)
  var valid_ptr: Address := offset(self.pointer, r)

  /* Stash Point */
  exhale own_range(self.pointer, r, l)
  inhale raw_range(self.pointer, r, l, size_of())
  label stash_point

  range_unfold(self.pointer, r, l, size_of())
  range_unfold_rest(
               self.pointer, w, w + items_left, size_of(),
               self.pointer, r, l)

  copy(valid_ptr, dropped_ptr, items_left)

  range_fold(self.pointer, w, w + items_left, size_of())
  range_fold_rest(
             self.pointer, r, l, size_of(),
             self.pointer, w, w + items_left)

  /* Stash restore */
  exhale raw_range(self.pointer, w, w + items_left, size_of()) &&
         bytes(dropped_ptr, mul(items_left, size_of()))
         == old[stash_point](
           bytes(valid_ptr, mul(items_left, size_of())))
  inhale own_range(self.pointer, w, w + items_left)

  set_len(self, w + items_left)
}
```

Figure 6.19: Method body of drop_FillGapOnDrop

```
method copy(src: Address, dst: Address, size: Int)
  requires 0 <= size
  requires size_of() == 0 ? true :
    forall k: Int :: {byte_offset(src, k)} {byte_offset(dst, k)}
    0 <= k && k < mul(size, size_of()) ==>
    Raw_Byte(byte_offset(src, k)) && (
      !contains_byte(
        src,
        mul(size, size_of()),
        byte_offset(dst, k))
      ==> Raw_Byte(byte_offset(dst, k))
    )
  ensures  size_of() == 0 ? true :
    forall k: Int :: {byte_offset(src, k)} {byte_offset(dst, k)}
    0 <= k && k < mul(size, size_of()) ==>
    Raw_Byte(byte_offset(src, k)) && (
      !contains_byte(
        src,
        mul(size, size_of()),
        byte_offset(dst, k))
      ==> Raw_Byte(byte_offset(dst, k))
    )
  ensures  bytes(dst, mul(size, size_of()))
    == old(bytes(src, mul(size, size_of())))
```

Figure 6.20: Method copy

```
function contains_byte(
    a: Address, size: Int, other: Address): Bool
{
  exists k: Int :: {byte_offset(a,k)}
    0 <= k && k < size && byte_offset(a,k) == other
}
```

Figure 6.21: Function contains_byte

```
function contains(
    a: Address, start: Int, end: Int, other: Address): Bool
{
  exists idx: Int :: { offset(a, idx) }
    idx >= start && idx < end && offset(a, idx) == other
}
```

Figure 6.22: Function contains

lines 12-14 in Fig. 6.19). At the end, we restore it on the destination range instead of the source range (lines 29-33), again using the `bytes`-condition ensured by the `copy` method just like for `copy_nonoverlapping`.

When it comes to unfolding the range, the potential overlap causes this to be more complicated than in the case of `copy_nonoverlapping`. While the source range can be unfolded just as before (line 16), the destination range must only be unfolded in the areas where it does not overlap with the (already unfolded) source range. We define a new lemma `range_unfold_rest`[10] to complement `range_unfold`, and use it to unfold any elements of the destination range that have not been unfolded with the source range already (lines 17-19). This lemma uses a helper function `contains`, shown in Fig. 6.22. It works in a similar way to the `contains_byte` function that we used for `copy` itself, but checks the overlap of whole elements instead of individual bytes. To fold the `Raw_Bytes` up into `Idx_MemoryBlock`-ranges again after the copy operation, we define a reverse lemma, `range_fold_rest`[11] just like we did for `range_unfold`.

After copying the left-over elements to the right place, there is no gap in the permissions of the vector anymore. The only thing left to do is setting the length of the vector to its new value, namely the index up to where the vector is now initialised. With this, the invariant is finally restored. The vector is returned to the `dedup_by` method, where execution jumps immediately out of the loop to `exit_with_panic`. There, the vector predicate `OwnVec` is folded and `dedup_by` returns with the vector's invariant intact.

6.2.2 Discussing different memory models

In the following, we discuss the memory we have used so far and compare how certain aspects are handled in comparison to the Rust memory model. Then we present two alternatives for our memory model, which we have come up with in the pursuit of better approximating the Rust model.

[10] For the definition of the lemma `range_unfold_rest`, refer to Sec. A.3, Fig. A.8
[11] For the definition of the lemma `range_fold_rest`, refer to Sec. A.3, Fig. A.9

The status quo In our Viper model we have modelled a type `Address` for the values of pointers. So far, addresses in our model are constructed from an `Allocation` instance and an `Int` offset via domain function `create_addr`, with `get_allocation` and `get_offset` building an inverse to it together:

$$\begin{aligned} \text{create\_addr} &: \text{Allocation} \times \text{Int} \to \text{Address} \\ \text{get\_allocation} &: \text{Address} \to \text{Allocation} \\ \text{get\_offset} &: \text{Address} \to \text{Int} \end{aligned}$$

For simplicity, we will abstract this in the following and regard addresses as a tuple of an `Allocation` and an `Int`,

$$\text{Address} := \text{Allocation} \times \text{Int}$$

so we will shorten the representation of an address *A = create_addr(alloc ,x)* to *A = (alloc, x)*. The associated offset function looks as follows:

$$\begin{aligned} \text{offset} &: (\text{Allocation} \times \text{Int}) \times \text{Int} \to (\text{Allocation} \times \text{Int}), \\ &\text{offset}((\text{alloc}, x), y) = (\text{alloc}, x + y * \text{size\_of}()), \end{aligned}$$

where size_of() returns the size of one of the elements of the allocated object.

Consider the following Rust code snippet:

```
a1 := alloc()
a2 := realloc(a1)
assert!(a1 != a2)
```

In our current model, this would correspond to the following:

`a1` is a pointer to an allocated object at address A1 $= (\text{alloc1}, 0)$
`a2` is a pointer to an allocated object at address A2 $= (\text{alloc2}, 0)$

In the Rust model, the assert statement may fail since `a2` is obtained via reallocation of `a1` and therefore both locations may point to the same location in memory. However, in our model it is impossible to show that two addresses constructed from different allocations refer to the same memory location.

The reason why `a1 == a2` is impossible in our model is that *create_addr* is an injective function: there cannot be any two addresses A1 $= (\text{alloc1}, x)$, A2 $= (\text{alloc2}, y)$ such that $(\text{alloc1}, x) \neq (\text{alloc2}, y)$ but A1 $=$ A2, even though the two addresses might actually refer to the same location in memory. In other words: every address in our model belongs to *exactly one* allocation, and its offset is always defined relative to the base address of that allocation. Consequently, the assert statement in the example above is guaranteed to always succeed in our model. The goal is to resolve this discrepancy between our model and the model used by Rust.

However, seeing as we have defined quantified permissions over addresses in our model (such as the ones used in `MemoryBlock`, `raw_range` and `own_range`), Viper in fact *requires* this injectivity of the `create_addr` function, and thus we cannot resolve our problem by simply making the function non-injective. Instead, we need to find an alternative implementation that allows us to compare pointers belonging to different allocations properly like in the Rust model while also still upholding the fulfilment of Viper's requirements for quantified permissions.

Using Integers as Addresses In this alternative we model memory as a single contiguous array of locations and simply use integers as addresses:

$$\text{Address} := \text{Int}$$

The offset function in this model would look as follows:

$$\begin{aligned} \text{offset} &: \text{Int} \times \text{Int} \rightarrow \text{Int} \\ \text{offset}(a, x) &= a + x * \text{size\_of}() \end{aligned}$$

Since all Addresses will now be in the same contiguous space of memory, comparing them can be done simply via integer comparison. In our above example, this would mean:

```
a1 is a pointer to an allocated object at some address x1 : Int,
a2 is a pointer to an allocated object at some address x2 : Int,
```

and it may happen that `x1 == x2`, in which case the assert statement fails.

Since each Address is a different integer value in this model, the injectivity requirement for quantified permissions is also fulfilled. Therefore, this model is now able to handle the assert statement in our previous example in the same way as Rust.

On the other hand, however, this model ignores a different aspect of the Rust pointer model. Consider the following Rust code snippet:

```
a1 := alloc()
a2 := realloc(a1)
if (a1 == a2) {
    unsafe{ *a1 = 3; }
}
```

This is not allowed in Rust since even if `a1 == a2`, `a1` is a pointer belonging to an allocation which has been deallocated, which means it is *not allowed* to be used anymore. However, since our new alternative model only uses integers as addresses and does not keep track of which allocation a pointer belongs to, using `a1` after its allocation has been deallocated cannot be prevented in this model. We therefore need a way to still be able to keep track of the allocation our pointers belong to and the status of these allocations.

Adding a non-injective function to_int to our model For this alternative we combine the two previous ideas: we keep our original definition of `Address`, but we add a non-injective function `to_int` that maps each Address to the integer space:

$$\text{to\_int} : \text{Address} \to \text{Int}$$

It is defined such that it returns the same result if applied to addresses which refer to the same location in memory (regardless of which allocation they are apart of), but returns different results if the addresses refer to different memory locations. It fulfils the following property:

Let

$$\begin{aligned} b &= (\text{alloc}, 0), \\ a &= (\text{alloc}, y) = \text{offset}(b, x), \end{aligned}$$

where b is the base pointer of allocation *alloc*, and $y = x * \text{size\_of}()$. Then:

$$\text{to\_int}(a) = \text{to\_int}(b) + y = \text{to\_int}(b) + x * \text{size\_of}()$$

This new model is able to tell when two pointers point to the same memory location, namely by applying `to_int` to both of the addresses and then comparing the results. Thanks to `to_int`'s non-injectivity, the returned integers may be the same even if the allocations the addresses belong to are not, in which case the assert statement in the example above will fail (as it should). Contrary to our integer memory model, however, this, model is still able to tell which allocation each of the addresses belong to. Therefore, in the second Rust code snipped we showed above, this model recognises that `a1` belongs to a deallocated allocation and is no longer allowed to be used. Furthermore, the injectivity of `create_addr` is also retained in this model so that no problems arise with the quantified permissions in our model.

6.3 Similar examples

Using unsafe Rust code, it is possible to perform computations that temporarily break data structures and their invariants, while restoring them to be whole again at the end of the execution. Drop guards are a very useful technique used to ensure memory safety even in the face of a panic occurring *during* such a computation, when the data structure is in a broken state. Furthermore, drop guards can also come in handy when it comes to preventing memory leakage. Therefore, drop guards are used quite often for these purposes.

Even in the implementation of Rust's `Vec` type alone, `dedup_by` is only one of four functions that use this technique. In the collection of bug examples we analysed for this project, we also encountered several bugs that received a fix involving a drop guard. Therefore, many aspects of remodelling the code and writing specifications for verifying it are not exclusive to the example of `dedup_by`, but can be applied to other Rust functions using the drop guard technique as well.

6.4 Suggestions for Prusti

In order to verify this example with Prusti, we would need to model the vector in two different states again, namely in the state where the vector's type invariant is intact, and in the state where there is a gap of potentially uninitialised memory within the vector. For this purpose it would once again be useful to have the possibility of defining multiple invariants on the vector struct and switching back and forth between them, like what we would already have found useful for examples 1 and 3 and we have described in detail in Sec. 3.4.

As long as this feature of switchable invariants is not available in Prusti, we can use a similar approach as we already used in these two examples, namely defining two versions of the vector struct, e.g. `Vec` and `GapVec`, each with the corresponding invariant. Then we can again change from `Vec` to `GapVec` by first creating a new instance of `GapVec` from the `Vec` instance, then overwriting the fields of the `Vec`, and the other way around.

Additionally, for writing specifications for the `std::ptr::copy` function we would need a function to check the overlap between the source and destination ranges. Then, we would also need a way to specify permissions to memory locations dependent on the outcome of this function, so that permissions are given wherever the is no overlap, but no permissions are given in the areas where the ranges overlap.

Chapter 7

Discussion of Findings

In this chapter we summarise our findings while analysing examples of memory safety bugs in Rust, as well as the outcome of our assessment of Prusti's capabilities in verifying unsafe Rust code. In Sec. 7.1 we present common bug patterns we encountered while choosing the examples to verify. In Sec. 7.2, we recapitulate Prusti features and verification techniques we used successfully for verifying our examples, while in Sec. 7.3 we present some ideas of what additional Prusti features would be useful to have in the future. Sec. 7.4 concludes with an evaluation of Prusti's design and capabilities with regard to verifying memory safety of unsafe Rust code.

7.1 Common bug patterns

In this section we describe our observations of bug patterns that appeared frequently in the collection of bugs we chose our examples from. Keep in mind that this is a very specific subset of Rust bugs, as they are all memory safety bugs found by the Rudra 'Unsafe Dataflow (UD)' analyser. Most of them either have to do with panic safety, or with the exposure of uninitialised memory to outside code. Therefore, the validity of the following observations is limited to this specific subset and not generalisable to the set of all Rust bugs.

What we noticed is that often programmers forgot to take care of certain cases that could come up in the execution of their program. Most often, the programmers expected the code to behave in a certain way. Some of these expectations were about their own code while others were assumptions about user-provided code, e.g. in closures or trait functions. These expectations often led them to overlook 'special' cases that were not covered by their program.

In Chapter 4, for example, we had a function with a safety check to make sure the result of a subtraction would not be too large. The 'normal' way the value could be too large was checked, but they did not realise that the subtraction could overflow, which is another way to end up with a too large result.

A problem we found frequently in the examples we analysed was that of temporarily broken invariants. There were many examples, like `Vec::dedup_by` in Chapter 6 or `String::retain` in Chapter 2, where a type invariant was broken for the duration of a computation, but is restored at the end before the function returns. However, often the programmers forgot to think of what happens when a panic occurs and the function is exited *during* the computation, when the data structure is still in a broken state. These examples could usually be solved by adding a drop guard, like it was the case for the `String::retain` example in Chapter 2.

Furthermore, as previously mentioned in Sections 2.3 and 3.3, there were another large number of bugs that were due to programmers assuming a certain behaviour from a trait function, even when it could be user-implemented. This was especially often the case in relation to the `Read` trait, where most programmers assume that a read function reads some data *into* the reader without reading the previous contents of the reader itself. Despite this being the conventional way of implementing a reader, this behaviour is not guaranteed by the traits documentation. Therefore, the programmers' assumption is incorrect and it is unsafe to call a read function on an uninitialised buffer.

In general, we can learn from our examining all these bugs that reasoning about panic safety is hard. The unwinding paths in the case of a panic, but also the normal drops at the end of a function are invisible to the programmer as they are automatically inserted by the compiler. As a consequence, problems regarding panic safety are easily overlooked. For this reason it is advantageous to have tools like Prusti to aid the programmer in this task of checking panic safety.

7.2 Prusti patterns

In this section we present some patterns and Prusti features we used successfully in verifying our examples, some of which we used frequently.

Unimplemented and trusted One pattern that we used quite often when verifying our examples is the 'unimplemented/trusted'-pattern, where we replaced a function's body with the `unimplemented!` macro and annotated the function itself as `#[trusted]` to make Prusti ignore the unimplemented body. There were several types of situations where this was helpful. One reason to use this pattern is to simplify functions where it suffices to know the signature of the function, or even only the return type, but the exact implementation of its body is irrelevant for the verification. These functions are often 'side' functions that get called by the main function we want to verify. We used this pattern for example in Chapter 4 for `analyze_extra_fields` (see Fig. 4.14), as well as in Chapter 5 for `new_raw_vec` (see Fig. 5.8).

Furthermore, this pattern is also useful to apply to functions where the body is actually not known, which is the case for closures as well as trait functions, as we explain below.

Closures are not supported by Prusti but can easily be replaced by a separate function outside of the main function that takes the closure as an argument. However, being a function argument, the closure's body implementation is unknown, only its signature is known. Applying the 'unimplemented/trusted' pattern allows us to model a closure as a separate function, otherwise this would not be possible. We used this technique for example for the closure in Sec. 3.2.1.

Trait functions are similar: Assume the function we want to verify takes as an argument `a`, which is an instance of a type that implements some trait `T`. When a function from `T` gets called on `a`, we again only know the signature of this trait function, but not its implementation. Since traits are not fully supported by Prusti yet, we need to model trait functions separately as well, just as we did for the closure above, and again this is possible thanks to 'unimplemented/trusted'. We made use of this in Chapter 4 when we modelled the call to a reader's `read_exact` method as a separate function instead (see Fig. 4.15).

Using Ghost fields to detach struct fields from the invariant Another thing we used several times for our verifications are Ghost types. Every time we used them for these examples was for a very specific purpose, namely in order to detach (some) of the struct's fields from the invariant's definition.

We used this method both for our type `BrokenVec` in Chapter 3 (see Fig. 3.4) and for `Vecu8` in Chapter 4 (see Fig. 4.7) to change an invariant that would usually define permissions dependent on all three fields, `ptr`, `len`, and `cap`, into an invariant where the Ghost field is used in `len`'s stead to define permissions that do not depend on the `len` field anymore.

Permissions between pointers In most cases, the permissions we needed to define lay between two integer offsets from a pointer, so the types expected by `own_range!` and `raw_range` were fitting: they both work by taking a pointer to memory and two `usize` integers indicating the start and end of the region given access to, as offsets from the pointer.

However, in one of our examples, namely the one about `Vec::from_iter` in Chapter 5, the permission ranges had to be defined between several pointers instead of between indices. In order to obtain the integer offsets we needed to define these permissions, we had to use the `address_from` function, a Prusti version of Rust's `offset_from` function calculating the distance between two pointers (see Fig. 5.6 and Figures 5.11-5.12).

7.3 Wish list

In this section we present some ideas for potential Prusti features that we would have found useful for verifying our examples. If they were added to Prusti in the future, we think they could facilitate verifying unsafe Rust code.

Multiple invariants In several of our examples we would have liked an easy way to model a struct in several different states.

In Chapter 3 a struct's invariant was temporarily broken, so we needed to model it both in this broken state, as well as in the normal state, where its type invariant was intact. In Chapter 5 we wanted to model a struct's special state after returning from a certain function call, as well as its 'normal' state that it should have when methods are called on it.

Unfortunately, Prusti does not allow multiple invariants defined on a single struct. Our workaround was to define new structs for each of the different invariants that we wanted to model. However, the transformation from a struct `A` to a struct `B` is rather cumbersome. To make an instance `b` of type `B` out of some object `a` of type `A`, we have to proceed as follows: the first step is to build an instance `b` of type `B` that is an equivalent of `a` by using `a`'s field values as well as the properties stated in its invariant for the definition of `b`. Once `b` was defined like this, `a` has to forget about the allocation and permissions they now both share. This can be achieved only by overwriting the fields of `a` with new values. Only then can `b` be used for proceeding with the rest of the program.

We could therefore save a lot of work if Prusti allowed several invariants to be defined on a single struct, and provided a way making it swap the invariant used on an instance of the struct, e.g. with a swap command. Then an object could be defined as an instance of a type with one invariant, but during the computation we can tell Prusti to swap to another invariant if the state of the object changes.

For this feature, it would also be good to allow users to define a 'default invariant' if they want to. This could be used in cases like the one in the example in Chapter 3, where one of the invariants represent the struct's actual type invariant, while the second one represents a broken version of this invariant. In cases like this, the type invariant holds when the instance is created, and it also has to hold again before the instance is returned by the function and exposed to the outside, since all outside code will expect its type invariant to be intact. Therefore, we could use such a 'default invariant' as the invariant used when a new instance of the struct is created, as well as require this invariant to be in use whenever the instance is exposed to outside code that expects this specific invariant to hold.

Specialised specifications In Chapter 4 we saw an example where a type taking a generic type parameter was used (`Vec<T>`), but only with one specific type inserted for the type parameter (`Vec<u8>`). The program was only able to work because it leveraged certain properties of type `u8`. As a consequence, we also had to take into account these special properties while writing our Prusti specifications. The resulting specifications were therefore only valid for this specific version of the vector type, but would be wrong if applied to the general type `Vec<T>`.

It would be helpful if Prusti allowed us to write specifications both for the general case, where it is unknown what type the parameter `T` takes or what properties this type exhibits, as well as for specialised cases, where additional information is available about the properties exhibited by `T`. During verification, Prusti would then choose between the general and 'specialised' specifications based on, for example, the traits implemented by a certain type parameter.

7.4 Prusti's design and capabilities

As mentioned in a previous chapter, Prusti is designed so that it verifies memory safety separately from general correctness and absence of panics. This allows us to isolate the verification of unsafe code from the verification of safe code, just how the `unsafe` keyword isolates unsafe code itself from safe code.

This isolation helps Prusti prevent us from writing nonsensical specifications. For example, Prusti does not allow us to specify memory safety requirements as preconditions for safe functions (i.e., using `structural_requires` on a safe function is not allowed, as it would be a contradiction to the function being safe).

Prusti also prohibits the use of the `result` keyword in structural postconditions. However, if the postcondition about the result is nevertheless needed for the proof of memory safety, then the function can be annotated with `#[no_panic_ensures_postcondition]` in order to enable Prusti to assume the result in the non-panic-situation at least, as we did in several of our examples.

So far, there is little documentation available on how to use Prusti for verifying *unsafe* code. There is no overview in one place of all the available features, and therefore no possibility to look up how they can be used, and what for.

As for error reports, some of them are already quite informative, making it easy to correct faulty usages of specifications (such as the ones described above), whereas others still leave it rather unclear what the problem is.

When it comes to Prusti's performance, we did not encounter any particular problems, but since measuring the performance was not the aim of this project and the number of examples we verified was not large enough to get any meaningful results, we leave this task for future work.

Once error reporting is further developed and more resources like a detailed documentation become available, Prusti will become much more user-friendly and verifying Rust programs with it will be much more efficient. Especially in regard to cleanup code that gets inserted automatically by the compiler (either in case of a panic, or whenever a value goes out of scope), Prusti can be of great help. As discussed in Sec. 7.1, these invisible parts of the computation are particularly tricky for programmers to reason about when it comes to memory safety. Having Prusti as a tool to assist in this challenging task will certainly facilitate guaranteeing the memory safety of unsafe code.

Chapter 8

Related Work

Several studies have been conducted about the usage of unsafe code in Rust programs [26][27]. They have analysed where, when, and for what purposes unsafe code is used by programmers. Although Rust's `unsafe` keyword is meant to be used sparingly, they have come to the conclusion that unsafe code is rather used a lot more often than expected (especially when it comes to enabling interoperability with other programming languages) [26]. Cui et al. [27] studied safety requirements across unsafe boundaries and defined typical safety properties, then categorised existing Rust CVEs according to the properties they defined.

Such an analysis of existing bug reports has also been done by Xu et al. [28]. Based on their analysis they defined patterns of Rust safety bugs, which they used to give recommendations for both the avoidance of new safety bugs, and the detection of already existing safety bugs.

As mentioned in Chapter 2, the Rudra project [1] went one step further: after also defining memory safety bug patterns, they used them for their large-scale automatic analysis of the complete Rust crate directory, as well as the standard library and the Rust compiler, `rustc`. Their approach to ensuring memory safety is therefore to find as many memory bugs as possible and eliminate them.

Rather than bug detection, the goal of our project was to verify Rust code with our Rust verifier Prusti. In contrast to projects like Rudra, which try to achieve memory safety through the detection and elimination of as many bugs as possible, Prusti's aim is to ensure memory safety by proving the complete absence of such memory safety bugs.

Apart from verifying our examples of Rust code with Prusti, another goal of our project was to evaluate Prusti's capabilities with regard to verifying unsafe Rust code, as previous work on Prusti has primarily focused on verifying *safe* Rust [2][29]. The above analyses of the collection of past bug reports [1][28] as well as on the usage of unsafe code in general [26][28] could serve as sources of examples for further evaluating Prusti's usefulness in verifying unsafe code in the future.

Chapter 9

Conclusion

The aim of this project was to assess Prusti's capabilities to verify unsafe Rust code. To achieve this goal we selected several examples of Rust code where memory bugs have been found in the past but which have been fixed since. If verifying the fixed code with Prusti was successful, this choice of examples allowed us to subsequently make sure that the unfixed code, by contrast, fails to verify while using the same specifications.

To enable the verification, a considerable amount of remodelling the code was necessary. For example, we generally defined our own custom structs to model Rust data structures, and often replaced functions with a simplified version of themselves. There are some very useful Prusti features and specification patterns that we used frequently throughout this process, for example the 'unimplemented/trusted' pattern to simplify functions whose body was not relevant to the verification, Ghost types, which we used for detaching certain struct fields from the invariant if necessary, and finally some annotations to bridge the gap between memory safety verification and correctness verification.

Nevertheless, our remodelling efforts have revealed the absence of some functionalities that we would have liked. These observations resulted in some suggestions for potential future Prusti features: switchable invariants (allowing us to switch back and forth between multiple invariants defined on a *single* struct), and specialised specifications (which are defined next to general ones, and which are used by Prusti depending on traits implemented by some type in order to leverage additional knowledge provided by these traits). These features would in our opinion facilitate writing Prusti specifications for verifying unsafe Rust code.

Appendix A

Appendix

A.1 Prusti configuration flags

The following table lists the Prusti configuration flags we used for verifying examples.

Flag	Value
unsafe_core_proof	true
enable_type_invariants	true
create_missing_storage_live	true
smt_qi_eager_threshold	20
purify_with_symbolic_execution	true
trace_with_symbolic_execution	true
symbolic_execution_single_method	true
panic_on_failed_exhale_materialization	false
Only used for ex. 3:	
merge_consecutive_statements	false

A.2 Additional Code for Chapter 3: Example 1: glium::buffer::Content::read

Fig. A.1 shows the getter functions for the length and capacity fields of `Vec` from Fig. 3.2. They are both annotated as pure methods that terminate, allowing them to be used in Prusti specifications elsewhere. They cannot panic (`#[no_panic]`) and therefore their result can be assumed for the proof of memory safety (`#[no_panic_ensures_postcondition]`), not only for the proof of general correctness. `non_verified_pure` tells Prusti that it does not need to verify these functions but can assume its postconditions nevertheless.

```rust
impl Vec {
    #[non_verified_pure]
    #[pure]
    #[terminates]
    #[no_panic]
    #[no_panic_ensures_postcondition]
    pub fn len(&self) -> usize {
        self.len
    }

    #[non_verified_pure]
    #[pure]
    #[terminates]
    #[no_panic]
    #[no_panic_ensures_postcondition]
    pub fn capacity(&self) -> usize {
        self.cap
    }
}
```

Figure A.1: Getter methods len and capacity of struct Vec

```
domain Allocation {
  function alloc_size(a: Allocation) : Int
}

domain Address {
  function create_addr (alloc: Allocation, n: Int) : Address
  function get_allocation (addr: Address) : Allocation
  function get_offset (addr: Address) : Int

  axiom addr_inverse {
    forall alloc: Allocation, i: Int :: {create_addr(alloc, i)}
      get_allocation(create_addr(alloc, i)) == alloc &&
      get_offset(create_addr(alloc, i)) == i
  }

  axiom offs_range {
    forall addr: Address :: {get_offset(addr)}
      get_offset(addr) >= 0
      && get_offset(addr) < alloc_size(get_allocation(addr))
  }

  axiom diff_addr {
    forall a1: Address, a2: Address ::
      a1 != a2 ==>
        get_offset(a1) != get_offset(a2)
        || get_allocation(a1) != get_allocation(a2)
  }

}
```

Figure A.2: Allocation and Address

A.3 Additional Code for Chapter 6: Example 4: std::vec::Vec::dedup_by

Modelling memory and addresses Fig. A.2 shows our address model. Each address refers to one byte within an allocation, at some integer offset from the start of the allocation. The constructor `create_addr` takes an allocation and an offset and creates the corresponding address, while the destructors `get_allocation` and `get_offset` can be used to obtain either one of these two components of an address; the axiom `addr_inverse` ensures that these functions actually work as inverses to `create_addr`. Meanwhile, the `offs_range` axiom restricts the offset of the address to the range of the allocation. Finally, `diff_addr` makes sure that each allocation-integer pair maps to exactly one instance of `Address`.

The bytes function The `bytes` function in Fig. A.3 takes an address and an integer `size` as arguments and returns the sequence of `size` bytes starting from the given address.

```
domain Byte{}

function bytes(addr: Address, size: Int): Seq[Byte]
  ensures |result| == size
  ensures forall a: Address
    :: {bytes(a, 0)}
    bytes(a, 0) == Seq[Byte]()
  ensures forall a: Address, sz: Int, x: Int, y: Int
    :: {bytes(a, sz)[x..y]}
    0 <= x && x <= y && y <= sz ==>
    bytes(a, sz)[x..y] == bytes(byte_offset(a, x), y - x)
  ensures forall a: Address, d: Int, sz: Int
    :: {bytes(byte_offset(a, d), sz)}
    bytes(byte_offset(a, d), sz) == bytes(a, d + sz)[d..(d + sz)]
```

Figure A.3: Domain Byte and the bytes function

The byte_offset function The `byte_offset` function is a helper function used by `copy_nonoverlapping` (see Fig. 6.16) and `copy_rec_nonoverlapping` (see Fig. A.5).

```
function byte_offset(addr: Address, offs: Int): Address
  ensures forall a: Address :: {byte_offset(a,0)}
    byte_offset(a,0) == a
{
  create_addr(get_allocation(addr), (get_offset(addr) + offs))
}
```

Figure A.4: Function byte_offset

Method copy_rec_nonoverlapping The method `copy_rec_nonoverlapping` is used by `copy_nonoverlapping` (see Fig. 6.16) for recursive calls. In the base case it returns without doing anything; otherwise, it uses the method `copy_byte` to copy one single byte (the last of the range), then calls itself recursively. Its code is shown in Fig. A.5.

```
/* num_bytes: size in BYTES, not in type-sized elements */
/* if size_of() == 0 ==> num_bytes == 0 */
method copy_rec_nonoverlapping(
    src: Address, dst: Address, num_bytes: Int)
  requires num_bytes >= 0
  requires forall k: Int :: {byte_offset(src,k)} {byte_offset(dst,k)}
    0 <= k && k < num_bytes ==>
    Raw_Byte(byte_offset(src, k)) && Raw_Byte(byte_offset(dst, k))
  ensures  forall k: Int :: {byte_offset(src,k)} {byte_offset(dst,k)}
    0 <= k && k < num_bytes ==>
    Raw_Byte(byte_offset(src, k)) && Raw_Byte(byte_offset(dst, k))
  ensures  bytes(dst, num_bytes) == old(bytes(src, num_bytes))
{
  if (num_bytes == 0) {
    /* DONE (Base Case) */
  } else {
    var from: Address := byte_offset(src, num_bytes - 1)
    var to:   Address := byte_offset(dst, num_bytes - 1)

    /* copy the last byte */
    copy_byte(from,to)

    /* recursively copy the first num_bytes-1 bytes */
    copy_rec_nonoverlapping(src, dst, num_bytes - 1)

    assert bytes(dst,num_bytes)[0..(num_bytes - 1)]
    == old(bytes(src,num_bytes)[0..(num_bytes - 1)]) /* trigger! */
  }
}
```

Figure A.5: Method copy_rec_nonoverlapping

Unfolding and folding ranges The following lemmas are used to unfold and fold whole memory ranges, i.e. to obtain `Raw_Byte` predicate instances for the bytes within a `raw_range`, or vice-versa.

`range_unfold` and `range_fold` are used for both the source and destination range in `dedup_by` (Fig. 6.14) before and after `copy_nonoverlapping`, respectively.

In `drop_FillGapOnDrop` (Fig. 6.19) they are used in combination with `range_unfold_rest` and `range_fold_rest` before and after `copy`, respectively.

```
method range_unfold(ptr: Address, start: Int, end: Int, el_size: Int)
  requires start >= 0 && end >= 0 && el_size >= 0
  requires raw_range(ptr,start,end,el_size)
  ensures  el_size == 0 ? true :
    forall k: Int :: { byte_offset(offset(ptr,start),k) }
      0 <= k && k < mul(end-start, el_size) ==>
      Raw_Byte(byte_offset(offset(ptr,start),k))
```

Figure A.6: Lemma range_unfold

```
method range_fold(ptr: Address, start: Int, end: Int, el_size: Int)
  requires start >= 0 && end >= 0 && el_size >= 0
  requires el_size == 0 ? true :
    forall k: Int :: {byte_offset(offset(ptr,start),k)}
      0 <= k && k < mul(end - start,el_size) ==>
        Raw_Byte(byte_offset(offset(ptr,start),k))
  ensures raw_range(ptr,start,end,el_size)
```

Figure A.7: Lemma range_fold

```
method range_unfold_rest(
    ptr: Address, start: Int, end: Int, el_size: Int,
    other: Address, other_start: Int, other_end: Int)
  requires start >= 0 && end >= 0 && el_size >= 0
  requires forall o: Int :: { Idx_MemoryBlock(ptr, o, el_size) }
    start <= o && o < end &&
    !contains(other, other_start, other_end, offset(ptr, o))
    ==> Idx_MemoryBlock(ptr, o, el_size)
  ensures el_size == 0 ? true :
    forall k: Int :: { byte_offset(offset(ptr, start), k) }
    0 <= k && k < mul(end - start,el_size) &&
    !contains_byte(
        offset(other, other_start),
        mul(end-start, el_size),
        byte_offset(offset(ptr, start), k)
    )
    ==> Raw_Byte(byte_offset(offset(ptr, start), k))
```

Figure A.8: Lemma range_unfold_rest

```
method range_fold_rest(
    ptr: Address, start: Int, end: Int, el_size: Int,
    other: Address, other_start: Int, other_end: Int)
  requires start >= 0 && end >= 0 && el_size >= 0
  requires el_size == 0 ? true :
    forall k: Int :: { byte_offset(offset(ptr, start), k) }
    0 <= k && k < mul(end - start, el_size) &&
    !contains_byte(
        offset(other, other_start),
        mul(end - start, el_size),
        byte_offset(offset(ptr, start), k)
    )
    ==> Raw_Byte(byte_offset(offset(ptr, start), k))
  ensures  forall o: Int :: { Idx_MemoryBlock(ptr, o, el_size) }
    start <= o && o < end &&
    !contains(other, other_start, other_end, offset(ptr, o))
    ==> Idx_MemoryBlock(ptr, o, el_size)
```

Figure A.9: Lemma range_fold_rest

Bibliography

[1] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "RUDRA: Finding Memory Safety Bugs in Rust at the Ecosystem Scale," in *SOSP '21: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. Association for Computing Machinery, 2021.

[2] 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)*, vol. 3, no. OOPSLA. ACM, 2019, pp. 147:1–147:30. [Online]. Available: http://doi.acm.org/10.1145/3360573

[3] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A Verification Infrastructure for Permission-Based Reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62. [Online]. Available: https://doi.org/10.1007/978-3-662-49122-5_2

[4] Prusti Project, Chair for Programming Methodology, ETH Zürich, "Prusti User Guide: Verification Features," accessed: 2023-09-09. [Online]. Available: https://viperproject.github.io/prusti-dev/user-guide/verify/summary.html

[5] Prusti Project, "viperproject/prusti-dev, Branch 'refactorings'," accessed: 2023-09-03. [Online]. Available: https://github.com/viperproject/prusti-dev/tree/refactorings

[6] Viper Project, "viperproject/silicon, Branch 'meilers_silicarbon_qponly'," accessed: 2023-09-03. [Online]. Available: https://github.com/viperproject/silicon/tree/meilers_silicarbon_qponly/

[7] ——, "viperproject/viper-ide," accessed: 2023-09-03. [Online]. Available: https://github.com/viperproject/viper-ide

[8] Yechan Bae, Ammar Askar, Youngsuk Kim, Jungwon Lim, “Rudra-PoC,” accessed: 2023-08-28. [Online]. Available: https://github.com/sslab-gatech/Rudra-PoC

[9] ——, “Rudra-PoC/README.md,” accessed: 2023-08-28. [Online]. Available: https://github.com/sslab-gatech/Rudra-PoC/blob/master/README.md

[10] ——, “Rudra-PoC/std-bugs.csv,” accessed: 2023-08-28. [Online]. Available: https://github.com/sslab-gatech/Rudra-PoC/blob/master/std-bugs.csv

[11] Pierre Krieger and contributors, “glium::buffer::Content trait, fn read,” accessed: 2023-09-07. [Online]. Available: https://github.com/glium/glium/blob/master/src/buffer/mod.rs#L87

[12] ——, “glium::buffer::Content for [T], fn read,” accessed: 2023-08-25. [Online]. Available: https://github.com/glium/glium/blob/master/src/buffer/mod.rs#L147

[13] Youngsuk Kim, “Uninitialized buffer exposed to user provided fn,” accessed: 2023-08-25. [Online]. Available: https://github.com/glium/glium/issues/1907

[14] Pierre Krieger and contributors, “Documentation of glium::buffer::Content::read,” accessed: 2023-08-25. [Online]. Available: https://docs.rs/glium/0.32.1/glium/buffer/trait.Content.html#tymethod.read

[15] The Rust Team, “Rust Language Documentation for struct std::vec::Vec, paragraph about capacity and reallocation,” accessed: 2023-08-13. [Online]. Available: https://doc.rust-lang.org/std/vec/struct.Vec.html#capacity-and-reallocation

[16] ——, “Rust Language Documentation of std::vec::Vec::set_len,” accessed: 2023-08-14. [Online]. Available: https://doc.rust-lang.org/std/vec/struct.Vec.html#method.set_len

[17] J.-C. Filliâtre, L. Gondelman, and A. Paskevich, “The Spirit of Ghost Code,” *Formal Methods in System Design*, vol. 48, no. 3, pp. 152–174, Jun. 2016, extended version of https://hal.inria.fr/hal-00873187. [Online]. Available: https://hal.science/hal-01396864

[18] The Rust Team, “Rust Language Documentation of method read in std::io::Read,” accessed: 2023-09-08. [Online]. Available: https://doc.rust-lang.org/std/io/trait.Read.html#tymethod.read

[19] ——, “Rust Language Documentation of method read_exact in std::io::Read,” accessed: 2023-09-08. [Online]. Available: https://doc.rust-lang.org/std/io/trait.Read.html#method.read_exact

[20] Yechan Bae, "Double free in Vec::from_iter specialization when drop panics," accessed: 2023-08-19. [Online]. Available: https://github.com/rust-lang/rust/issues/83618

[21] GitHub User the8472, "Fix double-drop in Vec::from_iter(vec.into_iter()) specialization when items drop during panic," accessed: 2023-08-19. [Online]. Available: https://github.com/rust-lang/rust/pull/83629/files

[22] The Rust Team, "Rust Language Documentation of std::vec::IntoIter," accessed: 2023-08-20. [Online]. Available: https://doc.rust-lang.org/std/vec/struct.IntoIter.html

[23] ——, "Code of std::vec::Vec::dedup_by," accessed: 2023-09-01. [Online]. Available: https://doc.rust-lang.org/src/alloc/vec/mod.rs.html#1714-1716

[24] ——, "Rust Language Documentation of std::vec::Vec::dedup_by," accessed: 2023-09-01. [Online]. Available: https://doc.rust-lang.org/std/vec/struct.Vec.html#method.dedup_by

[25] Viper Project Team, "Viper tutorial," accessed: 2023-09-10. [Online]. Available: http://viper.ethz.ch/tutorial/

[26] V. Astrauskas, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "How do programmers use unsafe rust?" in *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, vol. 4, no. OOPSLA. New York, NY, USA: ACM, nov 2020. [Online]. Available: https://www.youtube.com/watch?v=PQWCk3NXn0g

[27] M. Cui, S. Sun, H. Xu, and Y. Zhou, "Is unsafe an achilles' heel? a comprehensive study of safety requirements in unsafe rust programming," 2023.

[28] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. Lyu, "Memory-safety challenge considered solved? an in-depth study with all rust cves," 2021.

[29] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The prusti project: Formal verification for rust (invited)," in *NASA Formal Methods (14th International Symposium)*. Springer, 2022, pp. 88–108. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-031-06773-0_5

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

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

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

Title of work (in block letters):

Verifying Vulnerability Fixes in a Rust Verifier

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

Name(s):	**First name(s):**
Furrer	Olivia

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)**
Zollikon, 10 September 2023	O. Furrer

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