Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Verifying Safe Clients of Unsafe Code and Trait Implementations in Rust

Master's Thesis

Jakob Beckmann

bjakob@student.ethz.ch

Programming Methodology Group
Institute for Programming Languages and Systems
ETH Zürich

Supervisors:
Federico Poli, Dr. Christoph Matheja
Prof. Dr. Peter Müller

September 28, 2020

Acknowledgements

I would like to thank my supervisors, Federico Poli and Dr. Christoph Matheja, for their support and guidance. Their patience during our dicussions and the feedback they provided are truly appreciated. I would also like to thank Prof. Peter Müller, both for the opportunity to work on this thesis, but also for his support in my decision leading up to it. Finally I would like to express my gratitude towards the entire Programming Methodology group, for their constructive questions during my presentations, and the ensuing discussions helping me obtain a clearer view of the path my thesis should take. Thank you.

Contents

Acknowledgements iii

1 Introduction 1

1.1 Problem Statement . . . 1
1.2 Contributions . . . 2
1.3 Overview . . . 3
1.4 Conventions . . . 4

2 Background 7

2.1 Rust Programming Language . . . 7
2.1.1 Borrows . . . 8
2.1.2 Traits . . . 9
2.1.3 Unsafe Rust . . . 10
2.2 Formal Verification . . . 11
2.3 Prusti . . . 12
2.4 Viper . . . 15

3 Traits 19

3.1 Background . . . 19
3.2 Use-case Analysis . . . 21
3.3 Traits with Method-Dependent Unsafety . . . 23
3.4 Marker Traits . . . 24
3.4.1 Invariants . . . 25
3.4.2 Contract Refinement . . . 30
3.5 Evaluation . . . 32
3.5.1 Non-marker Traits . . . 33
3.5.2 Marker Traits . . . 33
3.6 Summary and Shortcomings . . . 36

4 Interior Mutability **37**

4.1 Background . . . 37
4.2 Problem statement . . . 39
4.3 Approaches . . . 40
4.3.1 Simple `Mutex` . . . 42
4.3.2 Rust Specification . . . 56
4.3.3 Rely/Guarantee . . . 57
4.3.4 `RwLock` . . . 63
4.3.5 `RefCell` . . . 63
4.3.6 `Cell` . . . 72
4.4 Implementation Challenges . . . 72

5 Contract Derivation **75**

5.1 Background . . . 75
5.2 Problem Statement . . . 76
5.3 Approaches . . . 78
5.3.1 Operator Overloading . . . 78
5.3.2 `PartialEq<Rhs>` . . . 82
5.3.3 `Eq` . . . 84
5.3.4 `PartialOrd<Rhs>` . . . 84
5.3.5 `Ord` . . . 84
5.3.6 `Default` . . . 86
5.3.7 `Clone` . . . 86
5.3.8 `Copy` . . . 88
5.3.9 `Hash` . . . 89
5.4 Summary and Shortcomings . . . 89

6 Conclusion **91**

6.1 Future Work . . . 91
6.1.1 Atomic Types and Other Wrappers . . . 91
6.1.2 Proper Concurrency Support . . . 92
6.1.3 Improved Support for Universal Quantification in Invariants for Hyperproperties . . . 92

6.1.4 Unsafe Code 93

Bibliography **95**

A Unmodified Listings for Prusti Example **A-1**

B Encoding of `Dummy` **B-1**

C Full Encoding of Stateful `RefCell` **C-1**

D Encoding of `read()` function **D-1**

Chapter 1

Introduction

Rust [Williams and Rust Team, 2020] is a relatively new programming language with a type system and ownership model which guarantees memory and thread safety. Rust has become very popular over the last years. In fact, the 2019 Stack Overflow developer survey ranked Rust as the most loved programming language among respondents for the fourth year in a row[1].

Thanks to some of its guarantees and its rise in popularity, Rust has become a very interesting target for verification. For instance, verification can benefit from Rust intrinsic rules preventing mutable aliasing. Prusti [Astrauskas et al., 2019], a verification frontend, is a plug-in to the Rust compiler, allowing a programmer to annotate functions with specifications directly in the Rust source code. Prusti is highly reliant on some language properties of Rust, allowing it to keep the specifications very simple.

However, in order to allow some low level operations, a second language, *Unsafe Rust*, is embedded into Rust. This other language is more permissive than standard Rust, at the expense of safety guarantees. Code needs to be declared *unsafe* when it relies on a property that the compiler cannot check. Indeed, unsafe Rust relaxes some guarantees on which Prusti relies, making Prusti unsound for verifying some programs containing unsafe code.

1.1 Problem Statement

The goal of this thesis is to enable sound verification of clients using libraries that use unsafe Rust code. The two main focuses are unsafe traits (Chapter 3) and code exhibiting interior mutability (Chapter 4).

Rust traits are similar to Java interfaces. Listing 1.1 shows the declaration of an unsafe trait from the standard library. Prusti returns an error when verifying this trait as it has no way of reasoning about marker traits, and does not consider the additional implications of the trait being unsafe. Marker traits are traits

[1] `https://insights.stackoverflow.com/survey/2019`

used to declare a property or even hyperproperty on the type that implements them [Blandy and Orendorff, 2017]. This hyperproperty is usually defined in the documentation, thus resulting in empty bodies in the trait declaration.

```rust
pub unsafe trait TrustedLen: Iterator {}
```

Listing 1.1 [rust] declaration of TrustedLen trait taken from the standard library.

In regards to interior mutability, Prusti allows verification, but it is unsound. Listing 1.2 shows a small program verifying without issues even though the assertion on line 16 is guaranteed to fail at runtime. Such unsoundness makes it difficult to trust Prusti in its verification task. Listing 1.2 will be further discussed in Chapter 4.

```rust
#[pure]
#[trusted]
fn getter(cell: &Cell<i32>) -> i32 {
    cell.get()
}

fn foo(cell: &Cell<i32>) {
    cell.set(5);
}

fn main() {
    let cell = Cell::new(4);
    let v1 = getter(&cell);
    foo(&cell);
    let v2 = getter(&cell);
    assert!(v1 == v2);
}
```

Listing 1.2 [rust] example unsoundness of Prusti. This program verifies but panics at runtime.

1.2 Contributions

This thesis presents a solution for reasoning about marker traits, and adds support for unsafe traits to Prusti. Moreover, it presents solutions to soundly handle interior mutability in Rust. On top of that, it discusses how contracts for certain traits can be derived directly by Prusti, and how operator overloading can be

handled in a more consistent way. More specifically, the thesis contributes the following:

1. Trait annotations allow to attach verifiable meaning to marker traits, including a new strategy to enable more flexible invariants throughout Prusti. This enables Prusti to reason about marker traits, which was completely impossible before. A fully featured working prototype including two new annotation types is added to Prusti.
2. Added support for traits marked unsafe. This includes normal and marker unsafe traits. Full support for this is added to Prusti.
3. Complete solutions able to handle various common cases of interior mutability. These proposals include a Rust annotation specification, property typechecking, and a full Viper encoding. This was achieved for:
   - `Mutex`, a mutually exclusive lock. Two Viper encodings are presented, one backed by a state machine ensuring simple invariants, the other uses a rely/guarantee mechanism to support more detailed verification, at the cost of complexity.
   - `RwLock`, a read/write lock. Similar solutions to `Mutex` are presented.
   - `Cell` and `RefCell`, simple wrappers exhibiting interior mutability. For both these types, a solution using a user defined state machine is illustrated, which allows to verify arbitrarily fine grained properties about the contents of the wrapper. An additional approach is presented which allows to omit some annotations, making it easier to use, at the cost of implementation complexity.
4. An extension to operator overloading in Prusti, allowing to encode operators on custom types in a consistent way.
5. An approach to allow deriving contracts for derivable traits. This enables to attach meaning to Rust `#[derive]` clauses. To complement this solution, a discussion on Rust's move and copy semantics helps to understand the implications for Prusti when deriving traits such as `Copy`.
6. An investigation on how higher level properties such as reflexivity of operators, or even consistent implementations between different traits (such as `PartialOrd` and `Ord`) can be automatically ensured by encoding them into type "snapshots" in Viper.

1.3 Overview

First, Chapter 2 will introduce the required background knowledge to understand the remainder of the thesis. Then, Chapter 3 discusses unsafe traits in more detail, with a strong focus on marker traits. Chapter 4 introduces interior mutability and several solutions that apply to standard library wrappers exhibiting interior mutability. It first discusses interior mutability in general, moving on

to describe the solutions for each wrapper individually. Chapter 5 presents a theoretical solution to derive trait contracts for some standard traits, which also improves Prusti's handling of operator overloading. Finally, the work is concluded with Chapter 6, which focuses on how these solutions can be extended, or how complementary Prusti features could improve their usability.

1.4 Conventions

Throughout this thesis, the "current" Prusti version refers to tag `rustc-2018-06-07` on the `master` branch of the official repository[2].

The following typographical conventions are used for the remainder of this thesis.

`Constant width`
: is used for inline code listings referring to program elements, such as variables, functions, constants, keywords, or data types. It is also used to denote external programs such as the Rust compiler or Prusti executable.

Italic font
: is used to introduce new concepts.

Bold font
: is used to heavily emphasize aspects of the text.

Block quotes

> Used for text taken from Rust documentation.

Block listings such as Figure 1.1 are used to show larger snippets of code. In general, code listings with a red background are used for listings that either don't compile, or are rejected by `prusti`. Code listings with a green background are used to explicitly indicate they are correct with some respect. This can mean they are accepted by `rustc`, verify with `prusti`, or run without panicking. Moreover, most code snippets are simplified, on order to provide more clarity. For instance, crate import statements and empty `main()` functions tend to be omitted.

[2] https://github.com/viperproject/prusti-dev/tree/rustc-2018-06-07

```rust
1 fn main() {
2     println!("hello world!");
3 }
```

```rust
1 fn main() {
2     println!(undefined_var);
3 }
```

```rust
1 fn main() {
2     assert!(1 == 1);
3 }
```

Figure 1.1 `[rust]` sample code listings.

Chapter 2

Background

This chapter presents the Rust programming language, with all aspects necessary for the understanding of this thesis. It then briefly introduces formal verification and explains how Prusti performs its verification task for a sample Rust program. This introduction provides the reader with adequate background to properly understand the remaining chapters. Background relevant to only small parts of the thesis will be presented in later sections as required.

2.1 Rust Programming Language

Rust [Williams and Rust Team, 2020] is a modern system programming language with a strong focus on speed, safety, and concurrency. Since memory safety is a primary goal of the language, the compiler enforces it on several levels [Blandy and Orendorff, 2017]. For instance, Rust prevents common software bugs such as access errors via dangling pointers or buffer over-reads, memory leaks due to improper memory management or corruption, and common concurrency issues such as data races. Indeed, the language design guarantees both memory and thread safety at compile time using a rich type system and an *ownership model*.

This section briefly discusses those features of Rust that are most relevant for this thesis, namely *ownership*, *moves*, and *borrows*. A comprehensive introduction to Rust can be found in [Blandy and Orendorff, 2017].

The Rust ownership system establishes that every memory location is owned by exactly one variable. When the owning variable goes out of scope (exceeds its lifetime), the memory is disposed of. This makes garbage collection unnecessary without the effort of manual memory management. Moves transfer ownership from one owner to another. Lastly, borrows allow code to temporarily reference a value without affecting its ownership [Klabnik and Nichols, 2020]. As borrows are primordial to this thesis, they will be discussed in further detail in the next section.

2.1.1 Borrows

Borrows come in two distinct forms: *shared* borrows and *mutable* borrows. Shared borrows are used to pass a reference to some code that does not allow the modification of the referenced entity, while mutable borrows allow such modifications. The *borrow checker* enforces strict rules on the usage of borrows to prevent unexpected side effects through aliased or dangling references. On a basic level, the borrow checker enforces the following two properties for every reference:

1. A reference cannot outlive its referent.
2. A mutable reference cannot be aliased. In other words, any mutable reference has to be unique, while shared borrows can coexist.

The principle of a reference outliving its referent is an example of a dangling pointer. A simple instance illustrating the problems this can produce is a function returning a reference to one of its local variables:

```rust
fn foo() -> &i32{
    let a = 5;
    &a
}
```

Listing 2.1 [✗ rust] example of borrow checker restriction: the code snippet returns a reference to a memory location that no longer exists after the call terminates.

Function `foo()` in Listing 2.1 assigns a local variable. This variable is allocated on the stack and will no longer be valid after the call to `foo()` terminates. Therefore returning a reference to it should be prevented, as the reference points to a location on the stack no longer representing the desired value. The borrow checker is able to understand that the lifetime of the reference is not contained in the lifetime of the referent, hence outliving it and producing a dangling reference. Thus `rustc` rejects the code.

Mutability of references is another subject the borrow checker worries about. Listing 2.2 shows a code snippet rejected by `rustc` as it tries to alias a mutable reference. `self.queue.iter()` in line 7 borrows `self` immutably to get an iterator over its internal `queue`. The `iter()` method performs no data copies, hence requiring that the underlying data structure does not change during the iteration. The calls to `process_message` jeopardize this assumption, as they could, in theory, modify the queue. The borrow checker is able to catch this and rejects this code at compile time. This listing is a simple example of the guarantees provided by the Rust type system, namely that `self` will not change for the duration the iterator provides shared references to the (transitively held) contents of `self`.

```
1  struct Server {
2      queue: Vec<i32>
3  }
4
5  impl Server {
6      fn process(&mut self) {
7          for message in self.queue.iter() {
8              // message is a shared borrow
9              self.process_message(message);
10         }
11     }
12
13     fn process_message(&mut self, msg: i32) {
14         ...
15     }
16 }
```

Listing 2.2 [✗ rust] example of borrow checker restriction: the code snippet tries to borrow `self` both mutably and immutably.

2.1.2 Traits

Traits are comparable to Java interfaces, in that they allow to define shared behavior across types. They are ubiquitous in Rust, reducing code duplication and providing better abstraction. Most importantly, traits enable using generics in a statically safe way.

Listing 2.3 illustrates two ways to use traits: as *bounds* for generic type parameters, and as *trait objects*. At the top of the code, a trait `Drawable` is declared. Any type implementing this trait has to provide an implementation of the `draw()` method declared in line 2. If desired, traits can also define a default implementation on the methods they declare. In such a case the implementing type is not required to provide an implementation for the method, but can choose to overwrite the default.

The trait can then be used as a bound for a generic parameter such as the function `draw_all_same_type()` in line 5. This function can be called on a vector containing elements of any single type implementing the trait. This is slightly different from the function `draw_all_diff_type()` in line 11. The latter function uses trait objects, allowing it to be called with a vector containing potentially different types, all implementing `Drawable`. Note that actually only `draw_all_same_type()` is polymorphic (parametric polymorphism), whereas the polymorphic nature of `draw_all_diff_type()` originates from the vector (dynamic binding on `item.draw()`), not the function signature itself.

```rust
trait Drawable {
    fn draw(&self);
}

fn draw_all_same_type<T: Drawable>(items: &Vec<T>) {
    for item in items.iter() {
        item.draw();
    }
}

fn draw_all_diff_type(items: &Vec<Box<Drawable>>) {
    for item in items.iter() {
        item.draw();
    }
}
```

Listing 2.3 [rust] example of traits used as a generic bound, and as a trait object, for polymorphism.

2.1.3 Unsafe Rust

The Rust language ensures that programs are memory safe and free of data races, via types, lifetimes, bound checks. It is referred to as *safe* Rust. *Unsafe* Rust, a second language embedded into safe Rust, allows to perform additional actions normally prohibited by the compiler to provide the programmer more freedom [Klabnik and Beingessner, 2020a]. This freedom is required for some of Rust's use cases. For instance, system software regularly requires to perform raw pointer arithmetic and dereferencing, such as for memory mapped registers of devices. However, due to the lack of formal definition of unsafe code and the small amount of official use-case examples, unsafe Rust is very difficult to use properly. In fact, in a survey from [Evans et al., 2020], 15% of developers use unsafe Rust simply because they needed to make the code compile or because it was faster to write code with unsafe Rust. Nonetheless, unsafe Rust is commonly used, with nearly a third of major crates using unsafe code directly, and over half having dependencies containing unsafe code [Evans et al., 2020].

In general, `unsafe` marks any Rust code whose adherence to the type system the compiler cannot check, and thus needs to be ensured by the programmer. The `unsafe` keyword can mark blocks, functions, and traits as unsafe. Marking a block or function as `unsafe` allows the programmer to perform, amongst others,

the following listed actions within the block or function:

1. dereference raw pointers,
2. call functions or methods that are marked as `unsafe`, and
3. implement `unsafe` *traits*.

Listing 2.4 illustrates how a block (in this case an expression), can be marked `unsafe` to dereference a raw pointer.

```rust
let val = 5;
let val_ptr = &val as *const i32;
let deref = unsafe {
    *val_ptr
};
```

Listing 2.4 `[rust]` example of an `unsafe` block used to dereference a raw pointer.

`unsafe` can also mark trait declarations and implementations as unsafe. In such cases, the semantics of `unsafe` are slightly different. Marking a trait declaration `unsafe` means that its *implementation* needs to ensure some properties not verifiable by the compiler to uphold Rust's type system. When implementing a trait whose declaration is marked `unsafe`, the implementation block also needs to be marked `unsafe`. In this case, `unsafe` only serves as a safeguard to ensure the implementer took notice that the trait is marked `unsafe` and all previously mentioned properties are correctly applied in the code. One possible approach to automatically check that this is the case is via *formal verification*.

2.2 Formal Verification

Formal verification of software allows to mathematically prove that the meaning of a program satisfies its specification [Loeckx and Sieber, 1987]. In other words, it aims at proving the correctness of a program. As stated in [Apt et al., 2009], correctness in this context means the programs behave according to some desirable properties, such as the delivery of correct and intended results, but also higher level properties such as deadlock freedom of concurrent programs.

This type of correctness verification is infamous due to its complex specifications and logic required to reason about mutable memory in programs. This is especially true when the language semantics include pointers, referencing, heap memory, and aliasing of mutable state [Bornat, 2000, O'Hearn et al., 2001].

The verification is performed on abstract mathematical models of the system. An alternative approach to model-based verification is *Hoare Logic* [Hoare, 1969].

The system revolves around *Hoare triples* $\{P\}Q\{R\}$, stating the connection between a precondition P, a program Q, and a postcondition R satisfied by the result of Q's execution on a state satisfying P. In other words, a Hoare triple essentially represents a program with a contract.

An example verifier using Hoare triples for verification is the *Viper* infrastructure, on which *Prusti* is based.

2.3 Prusti

Prusti [Astrauskas et al., 2019] leverages the Rust type system to enable formal verification of programs without the need for overly complex specifications. Prusti serves as a plug-in to the Rust compiler, allowing behavioral contracts to be specified directly in Rust source files.

Programmers annotate their code with specifications and compile it with the `rustc` extended compiler, which verifies the provided contracts. The vast majority of annotations, e.g. *preconditions*, *invariants*, and *postconditions*, are put on function or method level.

Listing 2.5 shows a small program annotated with such specifications. On line 4, a *pure* function is declared with no precondition and two `#[ensures="..."]` attributes. A function is considered pure if its result is fully deterministic on input, and if it is side-effect free. Moreover, the postcondition is given by the conjunction of all contracts provided in `#[ensures="..."]` attributes. Such a postcondition declares a promise made by the callee regarding its internal behavior and return values, which the caller can rely on. Analogously, Prusti allows to specify preconditions via `#[requires="..."]` attributes to define conditions under which a function or method can be called. Therefore, the precondition is a contract that the caller should fulfill at every call-site, on which the callee can rely.

Line 13 defines the `Dummy` type with a type invariant declaring its two internal fields should always be equal. Such an invariant is a condition that should hold in all *visible* states, that is, any client using the type can assume its invariant before and after a function call. The remaining code is standard Rust.

As most attributes used for contract specification in Prusti are Rust expressions, they need to be checked for correct typing. Prusti achieves this by generating ghost code during the parsing of the input program. This ghost code is then typechecked as part of the modified program by the Rust compiler. The program resulting from the modification of Listing 2.5 can be seen in Listing 2.6. For instance, with Listing 2.5 it is crucial to verify that `self.d1` and `self.d2` can be compared with the equal sign. Technically, this would be equivalent to checking that the type of `self.d1` implements the `PartialEq<T>`[1] trait, where `T` is the

[1] https://doc.rust-lang.org/std/cmp/trait.PartialEq.html

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

#[invariant="self.d1 == self.d2"]
struct Dummy {
    d1: i32,
    d2: i32,
}

fn test(d: &Dummy) {
    let val = max(d.d1, d.d2);
    assert!(val == d.d1);
}
```

Listing 2.5 [rust] example Rust program to verify.

type of `self.d2`, and that the trait is in scope. More specifically it means that `PartialEq<i32>` is implemented on `i32`, which is the case via the standard library. Prusti takes advantage of the fact that the Rust compiler already does this for normal expressions.

A lot of code generated by Prusti is not directly relevant to understand the verification process. As a consequence, most of the code presented in Listing 2.6 as well as later in Listing 2.7 is heavily simplified via redactions, and potentially modified. Appendix A offers the full, unmodified, code listings for reference.

Once the program extended by the ghost code successfully typechecks, the verification can be performed. To achieve this, Prusti translates the required parts of the extended program to Viper code and uses the Viper infrastructure to attempt a verification.

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

fn max__spec() -> () {
    fn max__spec__pre(a: i32, b: i32) -> () { }
    fn max__spec__post(a: i32, b: i32, result: i32) -> () {
        || -> bool { result >= a };
        || -> bool { result >= b };
        || -> bool { result == a || result == b };
    }
}

#[invariant = "self.d1 == self.d2"]
struct Dummy {
    d1: i32,
    d2: i32,
}

impl Dummy {
    fn Dummy__spec(self) -> () {
        || -> bool { self.d1 == self.d2 };
    }
}

fn test(d: &Dummy) {
    let val = max(d.d1, d.d2);
    assert!(val == d . d1);
}
```

Listing 2.6 [rust] simplified Rust program generated by Prusti for spec generation and type checking.

2.4 Viper

Viper [Müller et al., 2016] is a verification infrastructure including an intermediate verification language called Silver. For simplicity, Viper will henceforth be used for both the verification infrastructure and the intermediate language. Viper is used as a backend for many verification tools, including Prusti. It supports verification of concurrent, heap manipulating programs, using *implicit dynamic frames* [Smans et al., 2009]. For a comprehensive introduction consult the Viper paper [Müller et al., 2016] or the online tutorial[2]. On an abstract level, Viper can be seen as a language providing a way to write an extension of Hoare triples and to verify their validity (see Section 4.3.1.1 on Hoare logic).

Listing 2.7 shows a modified snippet of the Viper code generated by Prusti from Listing 2.5, illustrating only the relevant parts of the code. The `field` declarations in lines 1 to 3 define fields potentially accessible from any object in the program. Viper does not have any notion of classes, and thus every reference can, in theory, access every declared field. This access is controlled via *permissions*.

Viper represents permissions by a tuple consisting of a heap location and a *permission amount*. The permission amount is a fraction between zero and one inclusive, denoting the exclusivity of the permission. A full, or *exclusive*, permission consisting of a permission amount of one, allows the program state holding the permission to read and modify the associated heap location. Such a permission can be seen in line 10. In contrast, any non-zero permission amount less than one only allows reading, while one of zero allows no access at all. A permission allowing only for reading can be seen in line 19, where the permission is guaranteed to be between `none` (zero) and `write` (one).

Prusti encodes Rust types as Viper `predicate`s representing access to the type's fields. This is also the case for Rust's primitive types as seen in line 9 for `i32`. Predicates provide a name to a parametrized assertion, representing it by a resource, and can thus also be equipped with a permission. Predicates are not directly equivalent to their body, but the resource and assertion can be freely exchanged via the `fold` and `unfold` statements. In line 21, the `Dummy` predicate is unfolded in the expression following the `in` keyword using `unfolding`. `unfolding` allows to temporarily unfold a predicate for a single expression.

Finally, the type invariant for `Dummy` is encoded as a function in line 18 which performs the equality check between the two fields. This function can be called and asserted whenever the invariant for type `Dummy` should be checked.

Contracts on methods and functions in Rust do not need to be translated to Viper functions like the type invariant, as Viper can express functional contracts natively. This is illustrated in line 27 for the `max()` function. Viper defines the `requires` and `ensures` keywords, which the semantics of Prusti's `#[requires`

[2] http://viper.ethz.ch/tutorial/

="..."] and #[ensures="..."] are based on, respectively. These contracts are verified automatically by Viper, in contrast to type invariants that need to be explicitly asserted at relevant points in the Viper code.

The primary way permissions are passed from caller to callee, and potentially returned, is via requires or ensures statements as in lines 19 and 29 (line 29 does not contain permissions, but ensures can return permissions to the caller). However, permissions can also be explicitly *inhaled* and *exhaled*. inhale A adds the permissions denoted by A to the program state and assumes that all value constraints from A hold. exhale A asserts that all permissions denoted by A are currently held, that the value constraints in A hold, and removes the permissions.

Viper allows the declaration of both methods and functions. Functions differ from methods in that they define side-effect free expressions. Thus they are functions in the mathematical sense. Functions can also be abstract, in which case they have no body. The read() function in line 5 of Listing 2.7 is an example of this.

The translation of test() is omitted, as it requires a lot of predicate handling, and its conceptual understanding is not vital to this thesis.

```
field f_d1: Ref
field f_d2: Ref
field val_int: Int

function read(): Perm
  ensures result > none
  ensures result < write

predicate i32(self: Ref) {
  acc(self.val_int, write)
}

predicate Dummy(self: Ref) {
  acc(self.f_d1, write) && (acc(i32(self.f_d1), write) &&
  (acc(self.f_d2, write) && acc(i32(self.f_d2), write)))
}

function Dummy__spec(self: Ref): Bool
  requires acc(Dummy(self), read())
{
  (unfolding acc(Dummy(self), read()) in
    (unfolding acc(i32(self.f_d2), read()) in
      (unfolding acc(i32(self.f_d1), read()) in
        self.f_d1.val_int == self.f_d2.val_int)))
}

function max(_pure_1: Int, _pure_2: Int): Int
  requires true
  ensures result >= _pure_1 && result >= _pure_2 &&
          (result == _pure_1 || result == _pure_2)
{
  (!(_pure_1 < _pure_2) ? _pure_1 : _pure_2)
}
```

Listing 2.7 [viper] simplified Viper program generated by Prusti for verification.

Chapter 3

Traits

This chapter presents the necessary background on `unsafe` traits, followed by an analysis of use-cases for `unsafe` traits, as well as the design of a solution to these use-cases. The analysis is based on publicly available data from commonly used crates on both GitHub[1] and crates.io[2]. For each analyzed use-case, this thesis discusses both a possible design and technical challenges, which had to be addressed when implementing the design in Prusti. Finally, an evaluation aggregates the new capabilities of Prusti.

3.1 Background

A common idiom for Rust libraries is to expose functions taking generic parameters bound by traits such as the declaration in Listing 3.1. The trait bound used in this snippet is `std::str::pattern::Searcher`[3], an experimental trait from the Rust standard library.

```rust
1 fn lib_print_next<T: Searcher>(searcher: &mut T) { ... }
```

Listing 3.1 `[rust]` function declaration using a trait bound on its generic parameter to allow users to pass custom types to it.

Such code allows users of the library to both implement the trait on their own type and use the library function on it. However, it can result in the invalidation of Rust's strong guarantees, when the function implementation relies on some behavior from the trait methods in order to adhere to Rust's type system. This is usually the case when the library function internally uses unsafe code and makes assumptions on return values from the trait methods. For instance, the documentation of `Searcher` used as the generic bound in Listing 3.1, states that:

[1] https://github.com/
[2] https://crates.io/
[3] https://doc.rust-lang.org/std/str/pattern/trait.Searcher.html

> The trait is marked unsafe because the indices returned by the `next()` methods are required to lie on valid utf8 boundaries in the haystack. This enables consumers of this trait to slice the haystack without additional runtime checks.

Listing 3.2 exemplifies this issue due to an assumption on the return value of `next()`. If the indices contained in the `SearchStep::Match` object do not lie on UTF–8 boundaries, the unsafe call to `hs.get_unchecked(start..end)` in line 5 returns a string sliced at invalid boundaries. Rust considers this *undefined behavior*, and hence cannot provide any guarantees.

```rust
fn lib_print_next<T: Searcher>(searcher: &mut T) {
    let hs = searcher.haystack();
    let slice = match searcher.next() {
        SearchStep::Match(start, end) => {
            unsafe { hs.get_unchecked(start..end) }
        },
        _ => unreachable!()
    };
    println!("{}", slice);
}
```

Listing 3.2 [rust] library function relying on trait behavior that might cause inconsistencies in the type system.

Thus, the implementation of `lib_print_next()` is only safe if the `Searcher` trait adheres to always returning valid UTF–8 boundaries for the `haystack`. For the sake of static safety, Rust recommends marking the declaration of a trait used in such a way as `unsafe` [Klabnik and Beingessner, 2020b]. The assumptions upon which safety depends, i.e. the contract of the trait, should also be clearly documented. As a consequence, when a trait declaration is marked `unsafe`, any implementer of the trait needs to mark the implementation as `unsafe` as well, signaling that the implementation adheres to the documented contract. This is desired due to the lack of a standard language to document the contract, and the resulting lack of capabilities of the compiler to verify the code's adherence to the documented contract. One goal of this thesis is to provide the first steps towards formally checking such contracts on trait level.

The code in Listing 3.3 shows an implementation of the `Searcher` trait that does **not** respect the contract provided in the trait's documentation. This snippet is still accepted by `rustc`, but breaks the type system and exhibits undefined behavior when used in conjunction with Listing 3.2.

```rust
unsafe impl Searcher for WrongSearch {
    ...
    fn next(&mut self) -> SearchStep {
        SearchStep::Match(0, 1)
    }
    ...
}
```

Listing 3.3 [rust] Searcher implementation on custom type WrongSearch breaking the type system when used with Listing 3.2.

3.2 Use-case Analysis

The goal of this analysis is **not** to gain a comprehensive overview of the usages of unsafe traits. [Evans et al., 2020], [Qin et al., 2020], and [Astrauskas et al., 2020] present analyses on the prevalence and usage of unsafe code and investigate the usage of unsafe traits as well. The purpose of the analysis in this thesis is to get an informal overview of examples unsafe traits. The analysis is warranted due to the very small number of unsafe trait declarations in Rust code. In fact only 1.2% of crates on crates.io even declares an unsafe trait [Evans et al., 2020].

The intention underlying this analysis is to detect and evaluate trade-offs faced in the design of a solution. This section mainly focuses on presenting the general methodology of the analysis. Moreover, it gives a short overview of the outcome, while the details are presented in succeeding sections when relevant.

The data collection was performed by selecting the fifteen most starred GitHub repositories filtered by language. As this project focuses on how unsafe code is abstracted in libraries, the dependencies of these repositories were used as data input, instead of the repositories themselves. In fact, the dependencies were filtered according to their prevalence, with seven remaining. The set of combined dependencies was then supplemented with the fifteen most downloaded crates from crates.io[4] to form the main input for the analysis. The same dataset is used as a base for the interior mutability use-case analysis presented in the next chapter.

Investigating the prevalence of unsafe trait implementations and safe abstractions of unsafe code, we noticed that around half the libraries use unsafe code. However, the majority of use-cases are memory transmutation[5], calls to external C libraries, and raw pointer dereferencing. Of the analyzed crates, only three implemented unsafe traits[6] declared in the standard library, and none declared

[4] as of 10 Apr 2020 at 13:14

[5] https://doc.rust-lang.org/std/mem/fn.transmute.html

[6] lazy-static, regex, and syn

publicly exposed `unsafe` traits. Indeed, this seems to indicate that public `unsafe` traits are not commonly used in these libraries. Additional use-cases for `unsafe` traits are gathered from Grep.app[7], both to understand how these are used in practice and why none of these cases appeared in the original dataset.

The data from Grep.app illustrates that `unsafe` traits are used with two main ideas in mind:

1. Performance, e.g. to avoid bound checks on slices, as can be observed in the `Searcher` trait. While performance optimization using unsafe code is quite a niche, some libraries heavily rely on unsafe code to achieve better performance [Astrauskas et al., 2020].
2. Tight coupling with other languages. This also aligns with a statement from [Evans et al., 2020] mentioning 45% of developers use the `unsafe` keyword for non-syscall external C functions. Using `unsafe` to interact with external functions does not imply that a trait needs to be marked `unsafe`, but shows the extent to which unsafe code in general is used for interacting with other languages. These other languages tend to rely on other internal properties than Rust, which can be asserted via traits. However, these properties cannot be checked by the compiler, and thus the traits need to be marked `unsafe`. An example is the `PyBorrowFlagLayout` trait from the `PyO3` crate, which requires all types implementing the trait to have a specific layout in memory.

In both cases, the use of `unsafe` traits is encapsulated within the crate as much as possible, in order to expose little unsafety to the user of the crate. This is especially true when using `unsafe` traits for tight coupling with other languages, as the correct usage of such traits can require intricate knowledge of the external language. Moreover, it appears that many uses of `unsafe` traits result from improper understanding of them, such as marking the trait `unsafe` when it declares an `unsafe` function[8], or marking it `unsafe` to help prevent accidental implementations[9].

The additional dataset from Grep.app provides insight in the kind of properties making traits `unsafe`. Generalizing, unsafety characteristics can be clustered into two categories:

1. General correctness requirements of the trait's own declared behavior, including hyperproperties that go beyond the behavior of an individual trait method.

[7] `https://grep.app/`

[8] `https://github.com/phaazon/luminance-rs/blob/2bdab0a7f3ac27f20c020d812a3f7d0818ad0086/luminance/src/backend/tess.rs#L9`

[9] `https://github.com/rustyscreeps/screeps-game-api/blob/00069a3f6336bf92024b9525182278ac40f6bf4b/src/objects.rs#L5`

2. Requirements external to the trait itself. These requirements can refer to correctness, restricted usage, structural memory requirements, among many others. In such cases, the trait is used to mark a type fulfilling these requirements, without necessarily providing additional behavior.

These two categories will be discussed individually in Sections 3.3 and 3.4.

3.3 Traits with Method-Dependent Unsafety

Section 3.1 already introduced the `Searcher` trait and illustrated its unsafe characteristics. Interestingly, its unsafety comes from a need for correctness in the implementation of its declared methods. In other words, the contract making the trait `unsafe` is solely linked to the behavior of a single method declared by the trait. This is the case in the vast majority of `unsafe` traits. Fortunately, this makes verification of `unsafe` trait implementations with Prusti quite simple, as the main goal of Prusti is to verify pre- and postconditions. Thus, the only real challenge for verification of `unsafe` trait implementations falling in this category is to allow specifying the required correctness condition of the trait. In the case of `Searcher`, this essentially comes down to formulating a postcondition to its `next()` method (see Listing 3.4). Unfortunately, this is not so simple in many cases, and a lot of Rust code needs to be added to enable defining the contract.

```rust
1 #[ensures="match result {
2     SearchStep::Match(start, end) => {
3         let haystack = self.haystack();
4         is_on_utf8_boundary(haystack, start) &&
5             is_on_utf8_boundary(haystack, end)
6     },
7     _ => true
8 }"]
9 fn next(&mut self) -> SearchStep;
```

Listing 3.4 `[rust]` the specified contract ensures the method is used in a safe way, assuming there is a static pure function `is_on_utf8_boundary()` returning whether an index is on a valid UTF–8 boundary for some haystack.

On a technical level, adding support for such specifications in Prusti is simply a question of adding `unsafe` traits to the list of supported features and treating them like regular traits. Small additional technicalities are also introduced with the additions from the succeeding sections. These will be discussed where relevant.

Still, many `unsafe` trait contracts are not directly related to the behavior of the methods the trait declares. In such cases, the traits are used to mark a type

implementing them as having some behavior external to the trait itself, hence the name "marker traits".

3.4 Marker Traits

Marker traits are traits used to restrict a type in useful ways according to the type's intrinsic properties. These traits rarely have bodies, in which case their sole purpose is to mark a type. However, marker traits can also define some methods, in which case they serve both as a marker and as a regular trait.

Marker traits are a key part of idiomatic Rust to enable programmers to convey additional meaning to existing code. For example, consider the `PartialEq<T>`[10] trait discussed in a preceding section. It allows a programmer to define a partial equivalence relation between any two entities via the `eq()` method. The `i32` primitive type implements this trait with itself as the associated type (i.e. `PartialEq<i32>`). However, equality between integers is in fact a total equivalence as any integer is equal to itself. Thus the code within `eq()` conveys more meaning than partial equality. As a consequence, the `Eq`[11] marker trait is also implemented on the type. This trait defines no functionality other than adding the semantic meaning that `PartialEq<i32>::eq()` is, in fact, an equivalence relation.

Confusion easily arises when discussing marker traits in conjunction with unsafe code. For instance, it can be difficult to recognize why the `Eq` trait is not marked `unsafe`, considering it defines behavior which is not verifiable by the compiler. While it is true that programmers cannot rely on the correct implementation of a total order on any type implementing `Eq`, the type system's safety does not rely on this correctness. Hence, no undefined behavior arises from an incorrect implementation. Therefore the trait does not need to be marked `unsafe`.

Nevertheless, in some cases, the compiler does depends on the implicit contract expressed by a marker trait, hence making it `unsafe`. The standard library's `TrustedLen` trait is an example of such a case, and portrayed later in this chapter.

Marker traits provide flexibility without requiring code duplication. However, they increase the difficulty of verification as the additional meaning provided by them should be considered in the verification process. Examine Listing 3.5 that should verify because the marker trait `Eq` guarantees that `==` is an equivalence relation and thus, in particular, reflexive.

Prior to this thesis, Prusti did not take marker traits into account. Therefore their entire purpose, which is typically only stated informally in the trait's documentation, is lost in the verification process. Extending Prusti to allow

[10] https://doc.rust-lang.org/std/cmp/trait.PartialEq.html
[11] https://doc.rust-lang.org/std/cmp/trait.Eq.html

```rust
fn should_verify<T>(a: &T) where T: Eq {
    assert!(a == a);  // equivalent to a.eq(&a)
}
```

Listing 3.5 `[rust]` any value that implements `Eq` should be equal to itself, and therefore this property should be verifiable.

annotations on trait level, such that meaning can be attached to traits that define no visible behavior on Rust level would have the following three advantages:

1. Allow reliance on their informally stated contract during verification. This enables the verification of a larger set of programs, as previously lost semantic meaning is then considered.
2. Ensure only types actually adhering to the contract of the trait can implement them. This would allow to remove unsafety from traits for which the unsafety contract can be expressed in Prusti.
3. Help promote the proper usage and understanding of (`unsafe`) marker traits, as they are no longer a pure documentation feature, but can automatically be checked for correctness.

This thesis proposes two additional features to be added to support the annotation of marker traits with behavioral specifications: *trait invariants* and *contract refinement*.

3.4.1 Invariants

The design and implementation of invariants on attached to traits is discussed first.

3.4.1.1 Design

As seen in Section 2.3, Prusti already supports invariants on types. These determine properties that hold in every visible state for some instance of a type. This thesis proposes to extend the idea of invariants and make them applicable to traits. In other words, trait invariants would provide a type invariant to any type implementing the trait. This is different from supporting the same type of invariant on the type itself, as trait invariants allow for far more flexibility in their expressions, such as referencing private fields on the implementing type, and referencing methods unknown to exist at the trait level. These additional features lead to substantial technical challenges, which will be explained later in the chapter. For instance, consider Listing 3.6 as a way to annotate a trait with such an invariant.

```rust
#[invariant="self.eq(&self)"]
pub trait Eq: PartialEq<Self> {}
```

Listing 3.6 [rust] example definition of a trait invariant for the Eq trait.

With this declaration, any type implementing the trait obtains the invariant. On a first glance, this seems equivalent to having the invariant on the type itself. However, traits are only applicable when in scope. To illustrate this, consider the two modules shown in Listing 3.7, defining a trait and a type implementing the trait.

```rust
mod trait_mod {
    // references private fields of implementing type
    #[invariant="self.i1 == self.i2"]
    pub trait Equal {}
}

mod struct_mod {
    use super::trait_mod::Equal;

    pub struct Dummy {
        pub i1: i32,
        pub i2: i32
    }

    // invariant applies
    impl Equal for Dummy {}
}
```

Listing 3.7 [rust] two modules defining a trait and a type implementing the trait.

When using the type from a different module such as in Listing 3.8, the invariant from `Equal` is not applicable, as the trait is not in scope. Thus the verification should fail. This could be desirable because a function that is unaware of the trait should not be able to rely on the trait's properties. However, in practice, it can lead to significant problems discussed shortly.

Yet, simply bringing the trait into scope via a `use` statement makes the invariant applicable, hence allowing the verification of the snippet as shown in Listing 3.9.

Moreover, one needs to consider the expected behavior at call-site for functions taking arguments of types implementing a trait. For instance, consider a function

```rust
use struct_mod::Dummy;

fn tester(d: &Dummy) {
    assert!(d.i1 == d.i2);
}
```

Listing 3.8 [✗ rust] verification cannot access out of scope trait invariants, even when a type implements the trait.

```rust
use trait_mod::Equal;
use struct_mod::Dummy;

fn tester(d: &Dummy) {
    assert!(d.i1 == d.i2);
}
```

Listing 3.9 [✓ rust] when a trait is in scope, all its invariants apply to all types that implement the trait.

taking a parameter of type `Dummy`, where the `Equal` trait is not in scope. Such a function should be allowed to have an implementation which breaks the invariant, as it is out of scope. However, on the call side, the trait might be in scope, hence the call should be illegal as it would jeopardize the invariant. Prusti does not verify invariants in every visible state. In fact, it only verifies the invariants on (most) function boundaries. Therefore it can occur that invariants cannot be relied on, or can even be violated without throwing an error. Listing 3.10 illustrates this.

Since Prusti is not stringent in how invariants are checked, this needs to be taken into consideration for the trait invariant implementation, ensuring the invariant is checked at the call-site. This check needs to be performed after the postcondition check, as a call to a method guaranteeing the invariant via its postcondition should be allowed. It's very complex to ensure this is implemented consistently and thus sound. Due to this challenge, scoping is not implemented in the prototype.

Finally, one needs to consider how invariants are joined. Normal type invariants are trivially joined via conjunction, which is also the chosen approach for trait invariants. This keeps Prusti's behavior predictable. Moreover, the error reporting for trait invariants needs to be of high quality, as trait invariants can also render some type impossible to implement, if two implemented trait invariants are in direct opposition to one another. Situations such as these need to properly refer to the invariants that are violated, and to the code violating them, in order to

```rust
extern crate prusti_contracts;

#[invariant="self.0 >= 0"]
struct MyTuple(i32);

fn main() {
    let mut s = MyTuple(-1);
    // no error is raised
    s.0 = -200;
    // still no error raised
    assert!(s.0 >= 0);
    // error: "asserted expression might not hold"
}
```

Listing 3.10 [rust] Prusti rejects this due to the assertion in line 11, even though this is the only line that makes sense given the invariant. The two lines violating the invariants are not treated as problematic by Prusti.

make the problem apparent to the programmer in a simple way.

3.4.1.2 Implementation

The implementation of invariant support for traits has two major visions: (1) *ease of use*, to align with Prusti's general vision, and (2) *flexibility*, to provide as much functionality as possible via the feature.

In order to provide ease of use, trait invariants will be usable exactly like type invariants from the user's perspective. This makes its usage fully uniform and thus the feature is easy to pick up.

As seen in the use-case analysis, many `unsafe` trait implementations refer to type internals in their behavioral specification. Moreover, some `unsafe` marker traits do not have super-traits, and hence have no defined behavior they can rely on. To maximize flexibility, contracts on such traits should be supported in a safe way. On a technical level, this means that contracts should be able to refer to type internals that are not necessarily known to exist when the trait is declared. The names of the fields are assumed to be known at trait level for simplicity. The solution is easily extended to allow renaming of fields using additional annotations. As an example of this, consider Listing 3.7. The contract defined on the trait in line 3 refers to two fields on `self` that are not guaranteed to exist on the implementing type. Moreover, even if both fields exist, they might not be comparable. On top of that, invariants also allow function calls, in order to not fully rely on fields. In order to safely allow such invariants, one must

thoroughly type-check them on the individual types implementing the trait.

To allow type checking for each type implementing the trait, the parser generates code on the type itself, representing the invariant. For instance for Listing 3.7, the following code shown in Listing 3.11 is generated during the parsing stage.

```rust
impl Dummy {
    fn Dummy__Equal__spec(self) -> () {
        || -> { self.i1 == self.i2 };
    }
}
```

Listing 3.11 [✓ rust] generated code for trait invariant type checking for Listing 3.7.

Generating the code directly on the type instead of on the trait has several advantages:

- One can use private fields and methods in the invariant declaration, as these are guaranteed to be available in implementation blocks of the type.
- Allow incremental compilation when the trait is in another crate or module. If the code was generated in the trait declaration, the implementation of a trait for some type would require recompilation of both the module defining the trait and the module implementing the trait.
- One can pass an implicit `Self` type by value without requiring the `Sized` trait bound on the trait itself. This is an implementation detail, which would, however, heavily impact the ease of use of trait invariants. The generated code in Listing 3.11 uses a parameter `self` passed by value. In order to allow this, Rust needs to ensure `self` has a known size at compile time. This is simple to ensure when `self` has a concrete type as in the presented listing. However, if the same function would be defined on a trait, the compiler would not be able to ensure that the implementing type has a known size at compile time. Therefore an additional trait bound would be necessary on the trait (namely `Sized`) to make sure this is actually the case.

However, generating this code also has several technical challenges. First and foremost, generating such code depends on, at least, the parser having fully run. This follows from the fact that a link between a trait implementation and its declaration is required to obtain the declared trait invariant for some type. In fact, for traits declared in another module, this would require a large part of type-checking to have already run. However, the code also needs to be generated during a parser run, in order to be checked for type errors. To solve this, Prusti is modified to perform two full parser runs. During the first one, no code is generated, but a register is built containing relevant information such as trait attributes,

links between trait implementations and declarations, and more. The second parser run can then generate code using information from this register allowing it to access information technically only available after its own completion. The second parser run does not affect overall performance of Prusti, as Rust's parsing is essentially instantaneous, and the observed time spent parsing Rust in Prusti is negligible compared to the time required to generate and verify Viper code.

Ideally the register would in fact use both a parsing and analysis run from the compiler, as it would allow for more robust type information, as well as inter-module type information. However, as this would require an even larger restructuring of Prusti's codebase and the limited time available to implement the solution, that approach was not chosen. Moreover, the selected design is conceptually easily extensible to support such behavior.

Second, as these invariants are dependent on the set of traits currently in scope, Prusti needs to encode them separately in Viper. Then, in every place the invariant for a type needs to be checked or assumed, the invariants of all implemented traits in scope are conjoined. In order to perform this conjunction, the register from the first parser pass is used. The actual encoding of the assertions in Viper is identical to the encoding of regular type invariants.

3.4.2 Contract Refinement

Another needed feature to properly support `unsafe` traits is contract refinement. Consider the `TrustedLen` [Rust Team, 2020d] trait from the Rust standard library. `TrustedLen` is a marker trait placing additional requirements on a method of its super-trait (`Iterator` [Rust Team, 2020c]). Its unsafety comes from the fact that the refined behavior is relied upon by the compiler, hence potentially creating undefined behavior when implemented on a type which does not uphold the contract defined in the trait's documentation.

Specifically, the trait's documentation states:

> An iterator that reports an accurate length using `size_hint`.
>
> The iterator reports a size hint where it is either exact (lower bound is equal to upper bound), or the upper bound is `None`. The upper bound must only be None if the actual iterator length is larger than `usize::MAX`. In that case, the lower bound must be `usize::MAX`, resulting in a `.size_hint` of `(usize::MAX, None)`.
>
> The iterator must produce exactly the number of elements it reported or diverge before reaching the end.

Unsafety This trait must only be implemented when the contract is upheld. Consumers of this trait must inspect `.size_hint`'s upper bound.

Therefore the unsafety originates from a change in the contract of a function not declared in the trait itself, but one of its super-traits.

3.4.2.1 Design

The preceding example clearly illustrates the need for contract refinement on trait level. In other words, it shows the need for some form of contract definition allowing to modify the behavioral specifications of some method in the super-traits. To achieve this, this thesis proposes the introduction of two new attributes: `#[refine_requires]` and `#[refine_ensures]`. The attributes allow to modify the pre- and postconditions of some method in the trait's bound. Thus, in contracts to ordinary `#[requires]` and `#[ensures]` attributes, these attributes change the contract of a method that is not part of the given trait.

For instance, consider Listing 3.12, allowing to provide an additional postcondition to `Iterator::size_hint`.

```
1 #[refine_ensures(Iterator::size_hint="
2     if let Some(upper) = result.1 {
3         result.0 == upper
4     } else {
5         result.0 == usize::MAX
6     }")]
7 pub trait TrustedLen: Iterator {}
```

Listing 3.12 `[rust]` standard library definition of `TrustedLen` with annotation for contact refinement. It might be worth noting that the here defined contract for `TrustedLen` does not remove the unsafety from the trait implementation, as the provided contract does not ensure the returned size hint is correct.

The design is kept robust by following subtyping rules: annotations can only make preconditions weaker and postconditions stronger. This is achieved by only allowing additions to the original contracts. In other words, any refinement is joined with the original contract via implication for preconditions, and conjunction for postconditions. This makes the behavior from trait level contract refinement very similar to normal trait contract refinement, and thus simple to grasp [Erdin, 2019].

3.4.2.2 Implementation

The described design presents several challenges in its implementation. First and foremost, Prusti needs to typecheck all specifications on the method level in the super-trait. However, the link to the super-trait is not available at parse time. This is solved similarly as for the invariants, with two full parser passes. During the second pass, the specification can be typechecked directly on the trait originally declaring the method whose specification is refined. Verifying the refined contract directly on the declaring method has several advantages: it coincides with existing specification typechecking, hence allowing reuse of existing code, improving maintainability and extensibility. Moreover, it allows to aggregate all specifications related to a method in a single place, rather than having them scattered across the code space.

The second challenge in this implementation is trait scoping. This refers to the same challenge as for invariants. Despite the problem seeming very similar, the technical aspect is drastically different. Prusti's general design allows to annotate every AST item (type declaration, function definition, etc.) with a single specification. This specification can then be retrieved during the compiler's analysis stage via some unique ID. Unfortunately, as a single trait declaration can in theory declare specification refinements for several independent methods, these would require unique specification IDs. This makes a solution such as declaring a single specification ID on trait declaration level (as is done for invariants) infeasible to solve scoping semantics. Such a solution would require the trait declaration to have a set of IDs, one for each contract refinement, which would be filtered for the relevant ones wherever the specification predicate is used and the trait is in scope. While such a solution is theoretically possible, it would require a major design change in Prusti's code generation and specification handling.

On top of that, it should be considered that marker traits define hyperproperties on the type they implement. Thus disrespecting scope actually has the advantage of providing stronger guarantees, without generating false positives in the verification process. To illustrate this, consider some synthetic type `Dummy` implementing `TrustedLen` with full contract refinement. In Listing 3.13 `TrustedLen` is not in scope. However, having the type implement the trait anywhere in the code base essentially guarantees the meta behavior on the type. Therefore verifying the snippet is perfectly sound.

Finally, as scoping was not fully implemented for invariants, this makes general trait level specifications uniform in behavior.

3.5 Evaluation

This section presents a short evaluation of the designed and implemented solutions for the handling of `unsafe` traits on simple examples.

```rust
use std::iter::Iterator;

#[ensure="result"]
fn test_dummy(d: &Dummy) -> bool {
    let (lower, upper) = d.size_hint();
    if upper.is_some() {
        lower == upper.unwrap()
    } else {
        lower == usize::MAX
    }
}
```

Listing 3.13 [✓ rust] verifies even though TrustedLen is not in scope.

3.5.1 Non-marker Traits

Regular unsafe traits not serving as marker traits are treated like normal traits. Therefore all supported features from normal traits are supported for unsafe traits, implying that every contract expressible in Prusti's specification language can be appended to unsafe traits. As a consequence, traits such as Searcher can be fully verified with a contract similar to the one presented in Listing 3.4. This type of trait is rarely used in practice. However, due to the general nature of the proposed solution, it is evident that most such traits can be devoid of unsafety with carefully crafted functional specifications on the methods rendering them unsafe. Unfortunately, it does require a lot of additional Rust code to enable the contract definition in many cases.

3.5.2 Marker Traits

The verification of marker traits can be seen as unrelated to trait unsafety. Every contract ensuring correctness on a safe marker trait can also ensure safety and correctness on an unsafe marker trait, as the unsafety originates from a reliance on the correctness of the trait's usage.

3.5.2.1 Invariants

Invariants are not a feature exclusive to marker traits, but tend to be more useful when handling traits used as markers. This comes from the fact that marker traits usually assert some intrinsic property on their implemented types, and that the asserted property typically translates to some invariant on the type itself.

An example found during the analysis that would require such an invariant is

the `PyBorrowFlagLayout`[12] trait from the PyO3 crate. This trait requires all types implementing it to have some specific layout in memory. This is verifiable, but requires a lot of additional Rust code to enable verification. For instance, in order to capture the structure and annotations of a Rust type, macros can be used, which would need to be applied to every type implementing the `PyBorrowFlagLayout` trait. The information provided from the macros could then be used to verify that every type does indeed adhere to the contract, using trait invariants. This will not be shown due to its complexity and invasiveness, while it is possible.

The standard's library `Eq` is an example of a safe marker trait which benefits from invariants. It allows to define that every instance of the implementing type is equal to itself. Defining an invariant on the trait not only allows to verify code such as the previously shown Listing 3.5, but also ensures implementing `Eq` on types that do not implement a total order on `PartialEq<Self>::eq()` will be rejected as soon as an instance is used that does not respect reflexivity.

To illustrate the capabilities of trait invariants, they are also evaluated on a set of synthetic trait declarations. For instance, consider the `std::ops::Add`[13] trait from the standard library allowing to overload the addition operator. This trait sets no restrictions on the behavior of the addition operator. Assume a mathematics library allows to restrict the behavior of types that adhere to some properties of group theory by using marker traits. For instance, consider additive identities. An additive identity is a value that, when added to any value of the set, results in the same value it was added to. Listing 3.14 shows a marker trait declaration indicating that the default value can be used as an additive identity.

```rust
1 trait DefaultIsIdentity: std::ops::Add + Default {}
```

Listing 3.14 [rust] example declaration of a trait declaring the default value of a type as an additive identity.

The use-case of such traits is that some types define sensible defaults that all adhere to some property. These types can then be used while relying on this property. However, without trait invariants, the marker trait can also be implemented on any type not adhering to the property, causing incorrectness in the codebase. An invariant such as shown in Listing 3.15 allows to ensure the marker trait is only implemented on types which actually follow this property.

[12] https://github.com/PyO3/pyo3/blob/51171f7475cdc671693c5c6f48350569e6dfac62/src/type_object.rs#L46
[13] https://doc.rust-lang.org/std/ops/trait.Add.html

```rust
#[invariant="self + Self::default() == self &&
             Self::default() + self == self"]
trait DefaultIsIdentity: std::ops::Add + Default {}
```

Listing 3.15 [rust] example contract for marker trait declaring the default as an additive identity.

3.5.2.2 Contract Refinement

Likewise to invariants, contract refinement is not a feature only useful to `unsafe` traits. Some normal marker traits also restrict the behavior of their super-traits, such as the `ExactSizeIterator`[14] trait, which restricts the result of `Iterator::size_hint` to return the exact size of the iterator. `ExactSizeIterator` is an interesting trait, not only due to its contract, but also because it does not conform to the general definition of a marker trait, in the sense that **it does** define additional methods. However, all of these methods have sensible implementations based on the intrinsic property the trait defines for its implementing types. Thus it is still considered a marker trait.

Similar to `ExactSizeIterator`, `TrustedLen`[15] defines an `unsafe` alternative to traits whose size is known if they have bounded size. With the implemented feature of contract refinement, as long as a contract for `Iterator::size_hint` can be formulated, both the contracts of `ExactSizeIterator` and `TrustedLen` can be defined and used for verification.

Another example from the standard library is `FusedIterator`[16]. The documentation of this trait states that is should be implemented on an iterator that always continues to return `None` when exhausted. A sample contract for this can be seen in Listing 3.16.

```rust
#[refine_ensures(Iterator::next="
    old(self.peek()) == None ==> self.peek() == None
    ")]
pub trait FusedIterator: Iterator + std::iter::Peekable {}
```

Listing 3.16 [rust] example contract for `FuzedIterator` trait.

Note that an additional trait bound was added to allow specifying the contract. Using the `next()` method would not have been valid, as it is not pure and modifies the iterator.

[14] https://doc.rust-lang.org/std/iter/trait.ExactSizeIterator.html
[15] https://doc.rust-lang.org/std/iter/trait.TrustedLen.html
[16] https://doc.rust-lang.org/std/iter/trait.FusedIterator.html

3.6 Summary and Shortcomings

This chapter discussed a prototype implementation of a feature enabling to reason about marker traits, and remove unsafety from traits on a conceptional level. With the automatic verification of `unsafe` trait contracts, the Prusti-enabled compiler can indeed check the properties of `unsafe` traits, rendering them "safe". However, many properties that are easily expressed in plain text in the documentation can be quite hard to state as Rust expressions. This limits the applicability of the implemented solutions to translate the documentation of `unsafe` traits into Prusti contracts. This thesis did not investigate on a quantitative level how often such contracts occur, but from the authors experience, it affects a significant share of `unsafe` traits. Several improvements to Prusti expressions will be discussed in the conclusion, as ideas for future work.

Chapter 4

Interior Mutability

This chapter first presents another use case of unsafe code, interior mutability, and the problems it can cause to Prusti. It then provides an analysis of the main usages of interior mutability, which enabled the design for soundly supporting most use-cases of interior mutability in Prusti, explained thereafter. Finally, while the author did not implement the approach, technical challenges one will likely face when incorporating these designs to Prusti will be discussed.

4.1 Background

Sometimes, a program entity needs to be modified even when having multiple aliases. Implementations of smart pointers, such as reference counted shared pointers, are an example of this. Another are complex data structures that use caching on logically immutable operations. Such modifications go against Rust's referencing rules presented in Section 2.1. Rust supports *interior mutability* to deal with such scenarios, allowing changes on immutable variables or shared references. Listing 4.1 exhibits interior mutability in line 6.

Rust's `std::cell::UnsafeCell`[1] is the only type allowing **safe** modification via shared references [Jasper et al., 2020]. Of course, since `unsafe` blocks are allowed to dereference raw pointers, it is also possible to modify an immutable object in such a way. However, this is not considered safe by Rust, as `rustc` assumes all referenced objects containing no `UnsafeCell` do not change over the lifetime of the reference. Listing 4.1 illustrates interior mutability using the `std::cell::Cell`[2] wrapper for `UnsafeCell`.

In Listing 4.1, function `foo()` modifies the `cell` using a shared reference by calling the `set()` method. Thus the assertion in line 14 will cause a runtime panic. This voids the common assumption that arguments passed by shared reference remain unchanged by the function call. Notice that any code using `UnsafeCell` still needs to respect the referencing rules introduced in Section 2.1. Concretely,

[1] https://doc.rust-lang.org/std/cell/struct.UnsafeCell.html
[2] https://doc.rust-lang.org/std/cell/struct.Cell.html

```rust
fn getter(cell: &Cell<i32>) -> i32 {
    cell.get()
}

fn foo(cell: &Cell<i32>) {
    cell.set(5);
}

fn main() {
    let cell = Cell::new(4);
    let v1 = getter(&cell);
    foo(&cell);
    let v2 = getter(&cell);
    assert!(v1 == v2);  // Panics at runtime
}
```

Listing 4.1 [✗ rust] example of interior mutability.

the code wrapping the UnsafeCell still needs to ensure that any mutable borrow to the contents of the UnsafeCell is unique, and that any reference (both shared or mutable), cannot outlive the contents of the UnsafeCell. Listing 4.2 shows how `std::cell::RefCell`[3], another safe wrapper of UnsafeCell providing references to its contents, will panic if one tries to create two coexisting mutable references to the UnsafeCell's contents. This example illustrates how Rust's type safety checks are moved from compile time to runtime when using UnsafeCells.

```rust
struct Dummy { }

fn main() {
    let cell = RefCell::new(Dummy {});
    let mut_ref_1 = cell.borrow_mut();
    let mut_ref_2 = cell.borrow_mut();  // Panics at runtime
}
```

Listing 4.2 [✗ rust] example usage of `std::cell::RefCell`.

Not all safe abstractions of UnsafeCell cause runtime errors in critical situations. The mutual exclusion primitive `std::sync::Mutex`[4], for instance, ensures the referencing rules to the contents of its internal UnsafeCell via blocking.

[3] https://doc.rust-lang.org/std/cell/struct.RefCell.html
[4] https://doc.rust-lang.org/std/sync/struct.Mutex.html

Moreover, RwLock[5], another type of lock, is also a safe abstraction of interior mutability using blocking to avoid panics at runtime.

4.2 Problem statement

Interior mutability is problematic for Prusti, as Prusti treats UnsafeCell as any other type, and assumes its contents do not change when referenced immutably. However, this is unsound: it allows to successfully verify false assertions. Listing 4.3 shows a slight modification of Listing 4.1, declaring the function getter() as pure. As it returns the wrapped value, it is fully deterministic on input and has no side-effects, therefore it is reasonable to declare the method as pure. #[trusted] is required as cell.get() is not declared as pure in the standard library, and its call would thus be illegal in a pure function body. Therefore Prusti is forced to trust the developer that the function getter() is indeed pure, without actually verifying its body. Unsurprisingly, the listing still panics at runtime (in line 16) as the value inside the cell is modified during the call to foo(). However, Prusti verifies the snippet: it assumes the shared reference passed to foo() cannot be used to modify the underlying data. As a consequence, it concludes both getter() calls to take the same argument, and thus to return the same value.

```rust
1  #[pure]
2  #[trusted]
3  fn getter(cell: &Cell<i32>) -> i32 {
4      cell.get()
5  }
6
7  fn foo(cell: &Cell<i32>) {
8      cell.set(5);
9  }
10
11 fn main() {
12     let cell = Cell::new(4);
13     let v1 = getter(&cell);
14     foo(&cell);
15     let v2 = getter(&cell);
16     assert!(v1 == v2);  // Panics but verifies
17 }
```

Listing 4.3 [✓ rust] example of interior mutability with Prusti annotations.

Similar phenomena are exhibited for all other wrappers of UnsafeCell. This

[5] https://doc.rust-lang.org/std/sync/struct.RwLock.html

chapter discusses approaches for fixing this unsoundness when dealing with wrappers of `UnsafeCell` while providing as many guarantees as possible about the underlying mutable data.

4.3 Approaches

Direct handling of `UnsafeCell` uses unsafe code; its verification is out of scope for this thesis. Instead, our goal is the client side verification of safe abstractions using unsafe code. When considering interior mutability, this implies the verification of safe wrappers of `UnsafeCell`. In this chapter we will focus on the four wrappers presented so far:

1. `std::cell::Cell`,
2. `std::cell::RefCell`,
3. `std::sync::Mutex`, and
4. `std::sync::RwLock`.

We first justify our choice to focus on these wrappers. In order to ensure it makes sense to focus on these wrappers defined in the standard library, rather than any custom user-defined ones, an analysis was conducted on the data from Section 3.2 to understand the most common usages of interior mutability and the abstractions used. Our analysis indicates that interior mutability is, nearly exclusively, used via the above safe wrappers provided by the Rust standard library. `UnsafeCell`s tend to not be used directly. In fact, only a single analyzed crate[6] made use of `UnsafeCell` without passing through a wrapper. Even in this case, the developers first used a `RefCell` to achieve their implementation goals, and later switched to `UnsafeCell` only for performance reasons.

Rust's safe wrappers to `UnsafeCell` all provide some form of interior mutability. Nevertheless, the high level reasoning that can be applied to them can differ significantly. For instance, `RefCell` can be thought of in a fundamentally different way than `Mutex`, due to the fact that `RefCell` cannot be used in multi-threaded contexts. This allows more fine-grained reasoning on the contents of `RefCell` compared to `Mutex`.

Consider Listing 4.4, defining a function taking a shared reference to a `RefCell`, and declaring a precondition on the contents of the cell. While not formally defined in the example, reading the value from a `RefCell` in the precondition is always safe when the function safely uses the cell. If the function borrows, mutably or immutably, from the cell in its body without panicking at runtime, then replacing the body with the borrow from the precondition at the call-site also does not panic. Moreover, since reading the contents of the cell solely depends

[6] `rand`

on the cell itself, it becomes apparent that this action can be modelled as a pure function.

The first major feature of `RefCell` visible in this listing that does not apply to `Mutex` is in the precondition. Declaring a precondition based on the contents of a `Mutex`, even if possible, makes little sense, as another thread holding the same mutex might void the hypothesis set up by the precondition before executing the first statement in the function body. Some forms of reasoning can accommodate for this (see Section 4.3.3), but in any case, an assertion such as in line 6 should not verify. In the context of the `RefCell` however, it should.

The same issue leads to another weaker guarantee for `Mutex`. Namely, if one considers a borrow from a cell to be the equivalent to a lock of a mutex, then the assertion in line 11 would not hold for a mutex unless strong assumptions about other threads are imposed, for the same reason stated for the precondition, as another thread might have interfered. On the other hand, in the scenario illustrated by the listing, the assertion can be verified, as no interference can occur between the write in line 7 and the read in line 11, as long as the cell is not passed as an argument to some function call between the drop and the new borrow in line 10.

```rust
struct Dummy { inner: i32 }

#[requires="<cell contents>.inner == 42"]
fn foobar(cell: &RefCell<Dummy>) {
    let mut dummy = cell.borrow_mut();
    assert!(dummy.inner == 42);
    dummy.inner = 1042;
    drop(dummy);

    let dummy = cell.borrow();
    assert!(dummy.inner == 1042);
}
```

Listing 4.4 [rust] possible reasoning for `RefCell`, in contrast to `Mutex`.

As illustrated with the preceding comparison, Rust's wrappers for `UnsafeCell` allow for different types of reasoning, for instance due to concurrency considerations (cells versus locks). Moreover, copy semantics (`Cell` versus `RefCell`), or permission based concurrent reads (`Mutex` versus `RwLock`) can also play a role. As a consequence of these differences, the thesis will present distinct handling and encodings for the wrappers. It starts by presenting the encoding for `Mutex`, then moving to the other wrapper. The presented solutions are easily extensible to any other wrappers exhibiting interior mutability.

The following sections will strongly focus on `Mutex` as the analogy to the abstraction used to encode interior mutability is easier to understand with a lock-like data-structure. First, Hoare Logic is briefly introduced in slightly more detail. This is followed by the first approach encoding a small state machine inspired by a solution to concurrent data accesses to shared memory. Then, a second approach illustrates how modifications to the first one can bring more consistency to the encoding. Finally, a rely/guarantee mechanism complementing the second solution is described.

4.3.1 Simple `Mutex`

When considering locks, reasoning on a formal level can be tricky, as interference from other threads tend to make guarantees on the protected data very difficult to uphold. More commonly, formal verification tries to ensure a program is deadlock free, since Rust explicitly does not guarantee deadlock freedom. As this section presents an encoding for locks, it is important to directly declare the exact goals that a solution should aim for, including whether a state encoding a deadlock should be considered invalid or not. Hoare logic shall be used to formalize an approach to solve this problem.

4.3.1.1 Hoare Logic

Hoare logic was already briefly introduced in Section 2.2. As our formalization uses Hoare logic notation, this section will briefly introduce some more aspects of it.

Definition 4.1 provides a more formal description of Hoare triples. Indeed it includes the fact that Hoare logic allows to prove *partial* correctness, thus only providing correctness if termination is guaranteed.

Definition 4.1. A Hoare triple $\{P\}Q\{R\}$ is "valid" if a program Q executing on a state s results in a state s', and if $s \vDash P$ (s satisfies P), then $s' \vDash Q$ (s' satisfies Q).

Furthermore, Hoare logic defines rules on triples to provide a proof system. For instance, the rule of consequence states

$$\frac{P_1 \rightarrow P_2 \quad , \quad \{P_2\}Q\{R_2\} \quad , \quad R_2 \rightarrow R_1}{\{P_1\}Q\{R_1\}} \tag{4.1}$$

Equation 4.1 states that it is possible to strengthen the precondition and/or to weaken the postcondition. Informally, this means a function can be called in states providing stronner guarantees than the ones required in its precondition, and weaker guarantees can be asserted than the ones provided by its postcondition.

Viper encodes an extension of Hoare logic to reason modularly about methods, and how their contracts relate to one another. In fact, Listing 4.5 is essentially nothing else than a declaration of the Hoare triple $\{P\}$ `foo()` $\{R\}$.

```
method foo()
  requires P
  ensures  R
{
  // ...
}
```

Listing 4.5 `[viper]` example method declaration to showcase its relation to Hoare triples.

4.3.1.2 Deadlocks

The notion of a deadlock is non-trivial to reason about in Hoare logic, as it does not truly represent a program state. Consider Listing 4.6, in which the state after the second `lock()` in line 3 will never be reached, thus by Definition 4.1, the triple is technically valid. This is directly visible from the fact that asserting `false` in line 4 should indeed verify, as the assertion would never be reached.

```
assume P
lock(x)  // acquires some lock x
lock(x)  // deadlock: x is already held by this thread
assert R
```

Listing 4.6 `[viper]` Viper encoding of a Hoare triple representing a deadlock. In Viper, an `assume` statement assumes the properties stated in the expression. An `assert` statement checks the permissions and value properties expressed in the assertion, throwing an error if they are not guaranteed to hold.

However, even if an assertion with expression `false` should verify after the deadlock, it can be argued that the deadlock itself represents a state, specifically a state where no other action is allowed, nor providing any guarantees. Using this argumentation, the deadlock is allowed to occur, but no action is allowed to follow due to the very nature of the deadlock. Thus Listing 4.6 should verify only if `R` does not rely on the program state at all.

For simplicity, any deadlock state can be considered invalid, as programs causing deadlocks as an intended final step of computation are not a realistic use-case. The approach handling deadlock states as undesirable is supported by

literature such as [Leino et al., 2009]. Moreover, in a study from [Qin et al., 2020], nearly 80% of bugs related to the failure to acquire a `Mutex` come from a double lock. An example of such a bug can be seen in Listing 4.7. In the listing, before the patch was applied, the lifetime of the temporary reference returned by `client.read()` in line 3 spans the entire `match` block. Thus the write in line 7 creates a double lock. After the patch, the lifetime of the temporary reference only lives until `result` is created, releasing the lock directly in line 4. This implies the write in line 7 can safely occur.

```rust
1   fn do_request() {
2       // client: Arc<RwLock<Inner>>
3 -     match connect(client.read().unwrap().m) {
4 +     let result = connect(client.read().unwrap().m);
5 +     match result {
6           Ok(_) => {
7               let mutinner = client.write().unwrap();
8               inner.m = mbrs;
9           }
10          Err(_) => {}
11      };
12  }
```

Listing 4.7 [rust] patch fixing a bug related to a double-lock from TiKW, taken from [Qin et al., 2020]

The change characterizing a deadlock as an invalid state modifies the definition of a valid Hoare triple to the one stated in Definition 4.2.

Definition 4.2. A Hoare triple $\{P\}Q\{R\}$ is said "valid" if a program Q executing on a state s results in a state s' where s' is not a deadlock state, and if $s \vDash P$ (s satisfies P), then $s' \vDash Q$ (s' satisfies Q).

In the following subsection, we present two encodings of `Mutex` to account for interior mutability in (a simplified version of) Prusti's Viper encoding.

4.3.1.3 First Viper Encoding

While interior mutability differs a lot from normal Rust behavior, it is similar to concurrent accesses of shared data structures in languages not using ownership-based type systems. At any time, another thread can modify data, so long at the data was shared with the thread in the first place. Similarly, any function call can modify data, so long as the data was passed to the function in the first place. In both cases, the thread or client can only rule out a modification when reading

the data. This makes reasoning about such data difficult, as changes can occur constantly, and the data can rarely guarantee to remain identical between two states.

Our goal is to exploit this analogy between interior mutability and concurrent accesses to shared data to derive a solution for the sound verification of interior mutable data. Many such solutions exist, especially attached to *Concurrent Sepration Logic* [Brookes and O'Hearn, 2016], a version of separation logic used to reason about concurrent programs. A flexible approach to reason about concurrent accesses to shared data are "*Concurrent Abstract Predicates (CAP)*" [Dinsdale-Young et al., 2010]. This approach formally defines Hoare triples with abstract predicates in their prestates and poststates to reason about actions on a concurrent data structure. Moreover, it defines the possible actions that can be performed on the data structure itself while a specific predicate holds. For instance, for a simple lock, the `LOCK` and `UNLOCK` actions can be defined as shown in Equations 4.2 and 4.3. Equation 4.2 states that given a prestate where it is known that x is a lock, the `lock` method can be called, which will result in x being *Locked*. The knowledge that x is a lock is not lost during the call. Equation 4.3 states that given that x is *Locked*, it can be unlocked by the current thread.

$$\{isLock(x)\} \quad \texttt{lock(x)} \quad \{isLock(x) * Locked(x)\} \tag{4.2}$$

$$\{Locked(x)\} \quad \texttt{unlock(x)} \quad \{emp\} \tag{4.3}$$

Equation 4.2 uses the separating conjunction $*$ in its poststate. The separating conjunction $A * B$ is a feature of *separation logic* [Reynolds, 2002]. It's intuitive meaning is that $A * B$ indicates that the program heap can be split into two disjoint parts in which A and B hold respectively.

As an example use-case for the separating conjunction, Equation 4.4 states that for two programs relying on disjoint prestates in the heap, and resulting in disjoint poststates, their independent sequential execution has the same outcome as their parallel execution (|| operator). In fact, this equation describes the proof rule for disjoint concurrency used for *Concurrent Separation Logic* [O'Hearn, 2007].

$$\frac{\{P_1\}Q_1\{R_1\} \quad \{P_2\}Q_2\{R_2\}}{\{P_1 * P_2\}\; Q_1||Q_2\; \{R_1 * R_2\}} \tag{4.4}$$

Returning to the analogy, the abstract predicates from Equations 4.2 and 4.3 can be used to define an axiomatic base for reasoning. For instance, the fact that x is a lock should be freely sharable (Equation 4.5), while no two disjoint heaps should be allowed to simultaneously claim to hold the lock (Equation 4.6).

$$isLock(x) \iff isLock(x) * isLock(x) \tag{4.5}$$
$$Locked(x) * Locked(x) \iff \texttt{false} \tag{4.6}$$

The predicates introduced in Equations 4.2 to 4.3 can be interpreted concretely. In [Dinsdale-Young et al., 2010], the concrete interpretation is for a compare-and-swap lock using a semaphore. Thus, in the paper, $isLock(x)$ is interpreted as representing a permission to acquire the lock, as well as an expression stating that either the lock is currently held by a thread (semaphore is set to 1), or it is not and can be released if acquired (semaphore is set to 0 and permission to unlock can be acquired). Thus, any program state in which $isLock(x)$ holds can call `lock()`, and the lock's invariant holds (it is either locked, or can be acquired). The second predicate, $Locked(x)$, is interpreted as the insurance that the lock is held (semaphore is set to 1) and that the lock can be released (permission to `UNLOCK` is held by the current thread). On top of that, the paper gives concrete definitions to the `LOCK` and `UNLOCK` actions. The `LOCK` action is understood as a transition of the semaphore's value from 0 to 1 and the acquisition of the permission to `UNLOCK`. Similarly, the `UNLOCK` action is defined as a transition of the semaphore's value from 1 to 0 and the loss of permission to `UNLOCK`. This can essentially be seen as a *state machine* with two states and transitions between them. This abstraction can be used to encode a finite state machine in Viper to reason about interior mutability.

4.3.1.4 State Machine Encoding

A finite-state machine is a computational model representing an abstract machine that is in exactly one of a finite number of states at any given time. The machine can change states via so called "state transitions" in response to some event or inputs. The definition of such a machine is comprised of the set of possible states, the initial state the machine starts in, and the possible transitions between states and their triggers.

Figure 4.1 shows a graphical representation of a state machine with two states as from [Dinsdale-Young et al., 2010]. Its initial state has either the semaphore set to zero and a permission to `UNLOCK`, or the semaphore set to one. Moreover, the permission to `LOCK` is held. From this state, the `LOCK` action can be performed, bringing the state machine into a state where the semaphore is guaranteed to be one, and the permission to `UNLOCK` is held by the current thread. The former state can be reinstated by performing the `UNLOCK` action.

Before being able to encode this state machine to handle interior mutability in Viper, one has to understand a major difference between Rust and C-style locks. Rust locks, such as `Mutex`, protect data rather than critical regions. Therefore, instead of providing a `lock()` and `unlock()` method to ensure no two threads

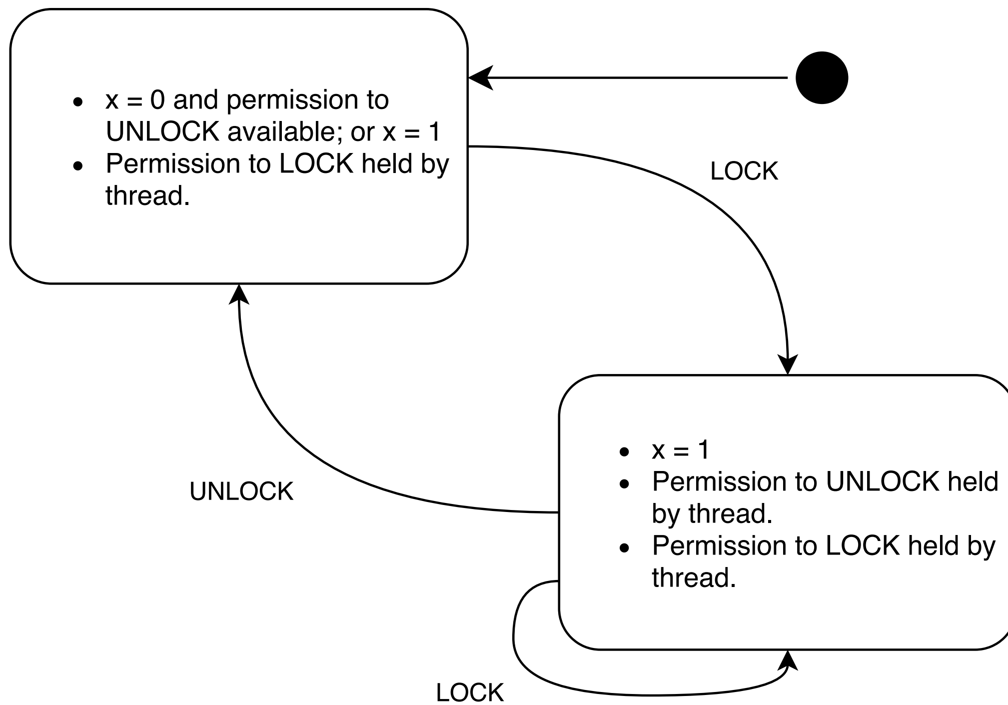


Figure 4.1 Graphical representation of the state machine for a compare-and-swap lock with semaphore x from [Dinsdale-Young et al., 2010].

can enter a set of critical regions at the same time, Rust locks provide only a `lock()` method returning a reference to the protected data. In order to ensure the reference no longer exists upon release of the lock, a *Resource Acquisition Is Initialisation (RAII)*[7] guard is used wrapping the returned reference. This goes hand in hand with Rust's ownership model and ensures that once the guard is dropped by the thread, the lock is automatically released. As a consequence, just as it is assumed the semaphore can only be changed by one thread at a time using an atomic compare-and-swap operation, the protected data from an `Mutex` can only be modified by a single thread at a time. Thus, for a `Mutex`, one can set an invariant on the data it protects and modify the state machine from Figure 4.1 into the one shown in Figure 4.2. Note the difference in the transformation from the state of the semaphore in the former figure to the access of the protected data, and its state, in the latter one.

State machines can be encoded in Viper by providing abstract predicates for each state, and modeling state transitions using abstract methods taking the permissions of the active state as a precondition, and ensuring the permissions to the target state in the postcondition. A Rust method can then be linked to a state transition of one of its arguments by adding the required source state in

[7] https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

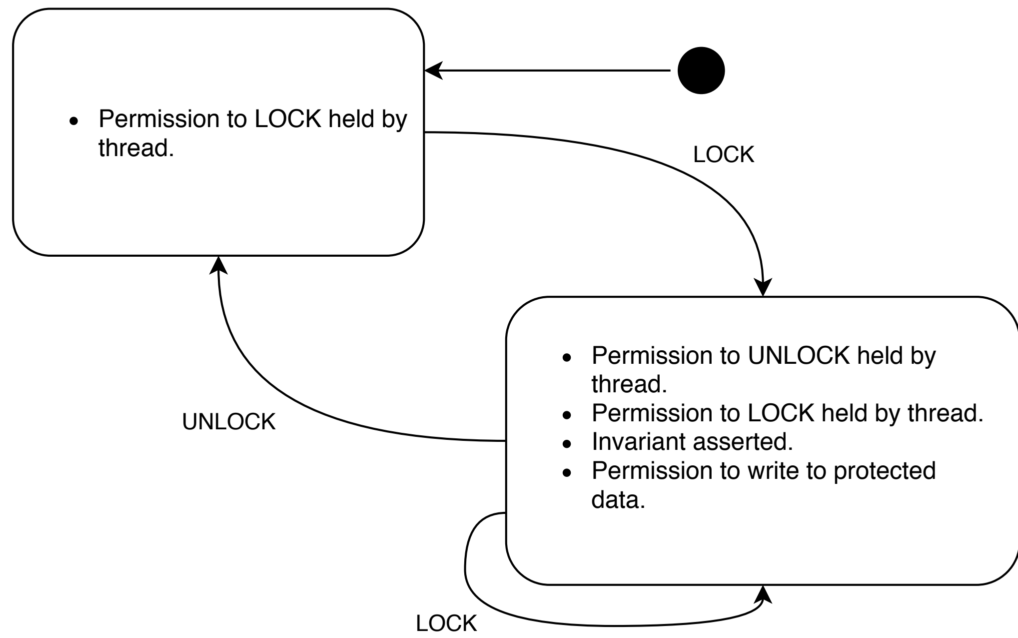
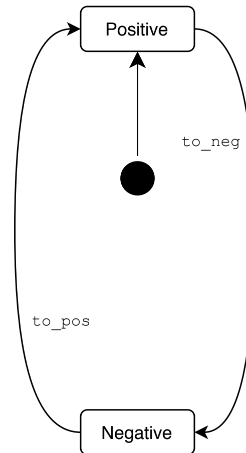


Figure 4.2 Graphical representation of the state machine for `Mutex`.

the encoded method's precondition, and the target state in the postcondition. Thus the "call event" of the Rust function defines the transition trigger of the state machine. Finally, the initial state of the machine can be set by inhaling the predicate encoding the initial state, or using an encoded abstract method for the creation of the lock which sets the initial state. An example of the encoding of a simple state machine can be seen in Listing 4.8.

```
predicate Positive(machine: Ref)

predicate Negative(machine: Ref)

method to_pos(machine: Ref)
  requires Negative(machine)
  ensures  Positive(machine)

method to_neg(machine: Ref)
  requires Positive(machine)
  ensures  Negative(machine)

method create_machine() returns
   (machine: Ref)
  ensures Positive(machine)
```

Listing 4.8 `[viper]` example state machine with two states, `Positive` and `Negative`, with defined transitions between both states and an initial state of `Positive`.



In Listing 4.8, assume the Viper source encodes a Rust snippet containing two functions `to_pos()` and `to_neg()`. Then the state transition is encoded directly in the pre- and postconditions of their respective Viper encodings in the listing. Listing 4.9 shows a sample client, creating the state machine in its initial state, and calling one of the encoded Rust functions. The new state of the machine can then be asserted in line 9 to verify that the transition took place. Moreover, note that calling a function that would require a source state that is not the current one will fail during verification, as not enough permissions to the abstract predicate are present to uphold the precondition of the called abstract method.

The encoding presented thus far provides states and transitions, but lacks the capabilities of attaching meaning to each state. For instance, the current state `Positive` in Listing 4.8 has no meaning attached to it, therefore the Viper program cannot reason about what it means for the machine to be in a `Positive` state.

In order to attach meaning to a state, each state is associated with an invariant. For instance, suppose the state machine from Listing 4.8 tracks the state of some integer value. In such a case, the value can be asserted to be `val > 0` or `val < 0` for `Positive` and `Negative` states respectively. The exact moments these expressions should be assumed and asserted depends on the use case of the state machine. As an example, in the `Mutex` case, this needs to be done at lock and release time.

```
1  method client() {
2    var machine: Ref
3    machine := create_machine()
4
5    // to_pos(machine)      --> fails as the source state
6    //                          is not correct
7
8    to_neg(machine)
9    assert(Negative(machine))
10 }
```

Listing 4.9 `[viper]` client side example of a method call triggering a state transition for the underlying state machine.

Now that it is clear how state machines can be encoded in Viper, we return to the state machine defined in Figure 4.2. Assuming a `Mutex` wrapping a `Dummy` (see Appendix B), the Viper encoding can be represented as in Listing 4.10. The chosen invariant is that the `inner` field of the `Dummy` should be positive. In the listing, the invariant can be seen as part of the $Locked(x)$ predicate. This is due to the fact that the concrete interpretation of the $Locked(x)$ predicate for `Mutex` is an access to the protected data, and the establishing of the invariant.

Note that Listing 4.10 implicitly considers the interpretations of the predicates as declared in the Equations 4.5 and 4.6. The fact that several threads can have knowledge of the `Mutex` simultaneously is implicit from the non-exclusive read permissions to the `Mutex` abstract predicate. Simultaneously, this read permission is also used as the encoding for the permission to lock, since the `Mutex` needs to be referenced to allow locking it. Therefore the `lock()` method states this permission in its precondition. The `lock()` encoding is simplified in this example, as it does not take into account that the Rust `lock()` function actually returns a `LockResult`, instead of a reference to the wrapped data directly. However, as the handling of `Result` types is already solved in Prusti, this is a small detail that does not affect the validity of the presented solution.

Moreover, the fact that the `Mutex` can only be locked once at a time (Equation 4.6) is seen from the exclusive permission to `Locked` returned from the `lock()` method, and consumed by the `unlock()` method. However, in contrast to the formalization in the CAP paper, Viper is very permissive with permissions inhaled on call-site via postconditions. In fact, it allows to inhale more than a single full permission to a predicate. Therefore, even though, via the exact definition of exclusive permission, the encoding is correct, it will still verify Listing 4.11. As a consequence, the encoding allows to verify an invalid Hoare triple as defined in Definition 4.2.

```
1  predicate Mutex(x: Ref)
2
3  predicate Locked(x: Ref) {
4    // write access to data &&                        invariant
5    acc(Dummy(x), write) && unfolding Dummy(x) in x.inner > 0
6  }
7
8  method lock(mutex: Ref)
9    requires acc(Mutex(mutex), read())
10   ensures  acc(Mutex(mutex), read())
11   ensures  acc(Locked(mutex), write)
12
13 method unlock(mutex: Ref)
14   requires acc(Locked(mutex), write)
```

Listing 4.10 [viper] encoding of Mutex with simple state machine and state invariant. Note that the Mutex and the data itself (Dummy) is encoded as the same object in Viper. This is achievable due to the strong possibility for abstraction enabled by the Viper object model. For encodings of Dummy and read() see Appendices B and D.

The prevention of a double lock such as in Listing 4.11 can be achieved by adding an assertion after the lock at call-site to verify that the permission held to the Locked predicate is less or equal to one. Even so, the encoding would be inelegant and quite abstract with respect to the Rust code, as, for instance, the conceptual behavior of the lock() Viper method is significantly different to Rust's lock() function. It is even moreso visible by the need for encoding an unlock() method, without having such a function in Rust. The second approach presented below tries to accommodate to this, by reasoning about Mutex as a simple wrapper returning a reference to the enclosing data. This has the advantage that the permission handling to the wrapped data is directly handled by the lock encoding as opposed to the Locked predicate.

4.3.1.5 Second Viper Encoding

The second approach attempts to model the Viper encoding to resemble more the behavior of the Rust code for Mutex. Concretely, it reasons about locking as returning a temporary reference to the wrapped data from the lock. The returning of temporary references is typically encoded in Viper using *magic wands*.

```
method client()
  requires Mutex(x)
{
  lock(x)
  lock(x)
}
```

Listing 4.11 [✓ viper] encoding allows to "double lock" and therefore does not catch obvious deadlock situations.

Magic Wands Rust functions that return temporary references to parts of their arguments are encoded to Viper using magic wands. The separating implication, or "magic wand" is a connective from separation logic that can be understood as follows: a separating conjunction, or "magic wand" $A \mathrel{-\!\!*} B$ encodes that if a given heap is extended with a disjoint heap satisfying A, the resulting heap satisfies B.

The magic wand connective is also part of the Viper verification language, where its semantic meaning is identical. However, Viper magic wands `A --* B` actually represent a resource. This resource needs to be combined with the resource in its left hand side (argument `A`), which exchanges it with its right hand side (argument `B`). Informally, when considering permissions, this means the right hand side permission can be obtained by giving up the left hand side permission (which needs to be held previously). To illustrate its meaning within an example, consider Listing 4.12. In the snippet, a predicate `A` and a magic wand connective `A --* B` are inhaled. In line 4, using the `apply` statement, these resources are exchanged for `B`. Hence the `assert` in line 5 succeeds. If the `assert` had been placed before the `apply`, it would have failed.

```
inhale A
inhale A --* B
// assert B    -->  would fail here
apply A --* B
assert B
```

Listing 4.12 [viper] example of a magic wand used in Viper.

Such wands are extensively used to encode temporary references returned by Rust functions, borrowing from their arguments. For instance, Figure 4.3 shows such a Rust function and its (simplified) Viper encoding.

In a similar fashion to the example from Figure 4.3, a magic wand can encode the write permission to the inner data provided from locking the `Mutex`, and can return a lock capability when revoking this permission. A "lock capability" can be

```
1 struct Dummy { inner: i32 }
2
3 fn foo(d: &mut Dummy) -> &mut i32 {
4     &mut d.inner
5 }
```

```
1 method foo(x: Ref)
2   requires Dummy(x)
3   ensures i32(result)
4           && (i32(result) --* Dummy(x))
```

Figure 4.3 Sample Viper encoding of a function returning a reference to a field of its argument. Rust code at the top and its Viper encoding below. The encoding used for `Dummy` can be seen in Appendix B.

modelled using a predicate, that is required to be true to call a function. In Viper this is encoded as an abstract predicate used in the precondition of the method that requires the capability (the `lock()` method in this case).

Listing 4.13 shows two abstract predicate declarations, one encoding the fact that a reference is a `Mutex`, while the other encodes the lock capability. The `LOCK` action can then be encoded as shown in line 5 in the listing. Moreover, the listing encodes `Mutex<Dummy>`, and couples the wrapped data inside the lock with the lock itself, as is done in Rust. Finally, using a lock capability (`CanLock`) rather than a *Locked* predicate automatically ensures no deadlocks can occur, as the lock can only be called when the `CanLock` capability is returned upon release of the previous lock. It is worth noting that the `lock()` method has lost its polymorphic nature, and now requires an encoding for every concrete type `T` in `Mutex<T>`. However, the *CanLock* encoding is now polymorphic, versus its non-polymorphic counterpart encoding *Locked*.

With this modified encoding, the required Viper code to model the call-site is a call to `lock()`, while the call-site encoding of the release of the lock is simply an application of the magic wand, as illustrated in Listing 4.14.

The above approaches only encode the lock and release actions on the `Mutex`. This is equivalent to the encoding of the `Mutex::lock()` function, and the dropping of the `MutexGuard` to release the lock. The construction of the lock is encoded by simply inhaling the two predicates encoding the fact that a reference is a `Mutex` and the lock capability. Finally, `Mutex::try_lock()` can be encoded nearly identically to `Mutex::lock`, while `Mutex::into_inner()` and `Mutex::get_mut()` are straightforward to encode, as they do not make use of interior mutability, and require a full permission to the `Mutex`. `Mutex::is_poisoned()` can only be

```
predicate Mutex(mutex: Ref)

predicate CanLock(mutex: Ref)

method lock(mutex: Ref) returns (data: Ref)
  requires acc(Mutex(mutex), read()) &&
           acc(CanLock(mutex), write)
  ensures  acc(Mutex(mutex), read()) &&
           acc(Dummy(data), write) &&
           (acc(Dummy(data), write) --*
             acc(CanLock(mutex), write))
```

Listing 4.13 `[viper]` abstract predicates for `Mutex` and lock capability, and the `lock` action encoded as a Viper abstract method. The Viper encoding of `Dummy` can be seen in Appendix B.

```
apply (acc(Dummy(data), write) --* acc(CanLock(mutex), write))
```

Listing 4.14 `[viper]` call-site encoding of the lock release. The Viper encoding of `Dummy` can be seen in Appendix B.

assumed to return a havocked boolean, as it is runtime dependent whether another thread could have potentially panicked while holding the lock.

It is worth noting that while the encoding favors detection of deadlocks, it does not indeed perform a verification on whether the program is deadlock free. This is mainly due to the fact that when a `Mutex` is passed to another thread, a `CanLock` predicate is passsed along to ensure the other thread can acquire the lock. Thus two threads can enter a deadlock by sharing two locks and trying to acquire both locks in opposite order.

Finally, the invariant can be directly encoded as part of the `lock()` method. This approach is opposed to encoding the invariant as part of the `Dummy` predicate. However, as `lock()` is already dependent on the type inside the `Mutex`, encoding it directly in the method enables `Dummy` to remain agnostic on whether it is used inside a lock or not. This results in an encoding of `lock()` as shown in Listing 4.15.

Using this state machine approach in conjunction with the invariant on the acquired state, code snippets such as Listing 4.16 can be verified. The specification syntax of the invariant in line 1 and how it is typechecked will be explained in Section 4.3.2. The listing verifies successfuly, as it is guaranteed that the value of the `inner` field is strictly positive when locking in line 5. This leads to the assertion in line 7 to be guaranteed to pass. Finally, the field value can be modified,

```
method lock(cell: Ref) returns (data: Ref)
  requires acc(Mutex(cell), read()) &&
           acc(CanLock(cell), write)
  ensures  acc(Mutex(cell), read()) &&
           acc(Dummy(data), write) &&
             unfolding Dummy(data) in data.inner > 0
  ensures  ((acc(Dummy(data), write) &&
             unfolding Dummy(data) in data.inner > 0) --*
            acc(CanLock(cell), write))
```

Listing 4.15 [viper] encoding of `lock()` method with an invariant of `data.inner > 0`.

even breaking the invariant in line 8, but satisfies the invariant upon release. Since the invariant is upheld upon release, the code verifies.

```
#[invariant="data.inner > 0"]
type MyMutex = Mutex<Dummy>;

fn client(mutex: &MyMutex) {
    let mut dummy = mutex.lock().unwrap();
    let val = 42 * dummy.inner;
    assert!(val > 0);
    dummy.inner = -10;
    // ...
    dummy.inner = 42;
}
```

Listing 4.16 [✓ rust] example of `Mutex` usage that can be verified using the first Viper encoding.

4.3.1.6 Other `Mutex` Methods

The previous sections only discussed the encoding of the `Mutex::lock()` method, and the encoding of the dropping of the `MutexGuard` returned from said method. However, `Mutex` defines other functions as well.

The `try_lock()` function can be encoded in a very similar fashion as `lock()`, as the two functions essentially perform the same action, blocking being the only difference. However, the blocking behavior is not visible from a static perspective.

The `new()` function is encoded by an inhale of the two predicates encoding the

Mutex. The invariant would also need to be established after construction, similar to how type invariants can be established. The two functions `into_inner()` and `get_mut()` do not exhibit interior mutability, and thus can be encoded nearly as they would be for regular types. Of course, `get_mut()`, which returns a temporary mutable reference to the wrapped data, would also need to assume the invariant upon return of the reference, and assert it when dropping the reference. However, since the function takes a mutable reference to the Mutex, this is not interior mutability and thus Prusti already handled it correctly. The encoding only needs to be adapted to the new encoding of the Mutex type and be able to handle the invariant.

Lastly, the `is_poisoned()` function checks whether the Mutex is poisoned. A lock is considered poisoned when a thread holding the lock panics. Reasoning about the value returned from `is_poisoned()` is very difficult, even at runtime. Even if the function returns that Mutex is not poisoned, the value cannot be trusted as the lock can become poisoned between the retrieval of this value and its usage. Moreover, reasoning statically about potential panics together with possible thread interleaving is a tremendous challenge. As a consequence, `is_poisoned()` is encoded as a function returning a havocked boolean value, providing no guarantee.

4.3.2 Rust Specification

In order to declare the specification for a stateful Mutex, which can be encoded in the way described previously, a programmer needs to declare the invariant that holds when the Mutex is acquired. As the invariant refers the data protected by the Mutex, it needs to be possible to express this in the invariant. In order to achieve this, the new `data` keyword is added to the Rust expressions used in invariants. How this keyword affects typechecking by the Rust parser and its translation to Viper will be explained shortly. Moreover, as a `Mutex<T>` is a type defined in the standard library, the invariant cannot be declared at type declaration level. Therefore the invariant to attached to a type alias declared in Rust. Listing 4.17 shows an example of this.

```rust
1 struct Dummy { inner: i32 }
2
3 #[invariant="data.inner > 0"]
4 type MyMutex = Mutex<Dummy>;
```

Listing 4.17 [rust] Rust definition of a stateful Mutex.

Having declared such a specification, the invariant can be translated into a static function by the parser. The result of this translation is seen in Listing 4.18.

Note this differs from normal invariants on types and traits, as the invariant is not part of an implementation block, the type of the parameter to the `spec__MyMutex()` is provided explicitly, and the closure expressing the invariant contains more code than simply the expression defined in the `#[invariant="..."]` attribute. Neither of these changes is especially problematic on the implementation level. First, the fact that the ghost code is generated outside an `impl` block actually makes the generation simpler. Second, the exact type of the parameter to the typechecking function is known, as the invariant is defined on a type alias, which can be used for the parameter type. The fact that the argument is passed by value does not restrict the usage of the invariant, as `Mutex` is always `Sized`, even when its type parameter is not. Finally, the additional code needed within the closure is always the same thanks to the newly introduced `data` keyword. Moreover, this code is guaranteed to not create other problems, as `unwrap()` can only cause runtime panics, and the typechecking function will never be called within the codebase (ghost code).

```rust
struct Dummy { inner: i32 }

type MyMutex = Mutex<Dummy>;

fn spec__MyMutex(mutex: MyMutex) -> () {
    || -> bool {
        let data = mutex.lock().unwrap();
        data.inner > 0
    };
}
```

Listing 4.18 `[rust]` code generated by the parser for the invariant listed in 4.17.

The encoding presented so far only allows to append an invariant to the wrapped data inside the `Mutex`. This can be useful in some cases but does not provide strong enough guarantees to verify most use cases. Therefore the encoding is strengthened via a *rely/guarantee* mechanism.

4.3.3 Rely/Guarantee

Rely/guarantee is a compositional verification method for shared memory concurrency [Owicki and Gries, 1976] [Vafeiadis, 2008]. The specification consists of four components (R, G, P, Q).

- The predicates P and Q are the precondition and postcondition of the entire execution of a thread. These have the same interpretation as in Hoare Logic.

- R (rely) and G (guarantee) encode the properties of all atomic operations that can be performed by an interfering thread (environment), and the thread itself, respectively. Both are two-state predicates relating the state before an atomic operation to the state after the operation has completed. Thus the rely condition models the interference the program can tolerate from its environment, and the guarantee condition models the interference the program imposes on its environment.

In order for a rely/guarantee specification to be well formed, P and Q need to be stable under the rely condition. Here, Q is said to be stable under R if $(Q; R) \Rightarrow Q$. In other words, applying a change from the rely condition R on a state satisfying Q results in another state satisfying Q.

In the case of a lock, any execution between an acquisition and its corresponding release is considered atomic. Thus the rely condition encodes the possible changes to the protected data from interfering threads, whereas the guarantee condition encodes the changes the thread itself is allowed to impose on the protected data. Figure 4.4 shows a graphical representation of the rely/guarantee mechanism.

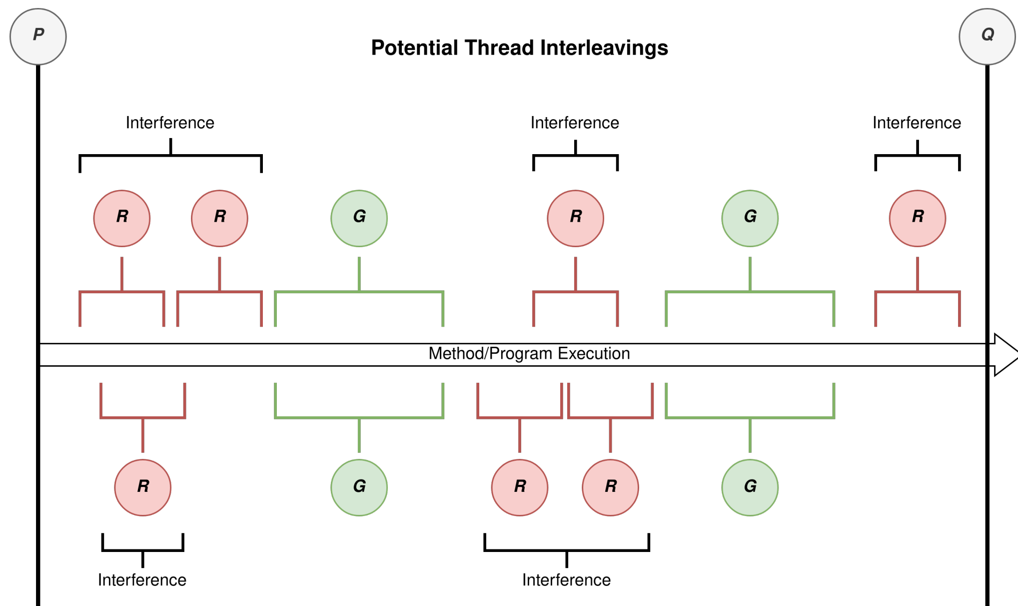


Figure 4.4 Graphical representation of a program (method) using rely/guarantee to enable verification. Note that this diagram only shows two possible thread interleavings for interference. Rely/guarantee enables to reason about the program no matter what the thread interleaving is.

To illustrate this for `Mutex` we will use a shared counter as an example. The counter n has the constraint that it is monotonically increasing. Thus the rely condition can be encoded as $n \rightsquigarrow n + 1$, where $\rightsquigarrow$ denotes the application of an

atomic operation, in this case by an interfering thread. Thus a stable, verifiable method could be one assuming that $n \geq 42$ in its precondition, asserting that $n \geq 32$ in its postcondition, and performing the operation $n \rightsquigarrow n - 10$ during an acquisition of the lock protecting the counter, hence defining the guarantee condition. Listing 4.19 show an example program which performs $n \rightsquigarrow n - 10$. The Rust specification for the pre- and postconditions are temporary placeholders and the rely condition is not specified.

```rust
1 #[requires="<mutex data>.inner >= 42"]
2 #[ensures="<mutex data>.inner >= 32"]
3 fn client(mutex: &Mutex<Dummy>) {
4     let mut dummy = mutex.lock().unwrap();
5     dummy.inner -= 10;
6 }
```

Listing 4.19 `[rust]` sample program with guarantee condition $n \rightsquigarrow n - 10$. This is verifiable with a rely condition $n \rightsquigarrow n + 1$.

In order to present the Viper encoding of this counter, a `Dummy` type is used with its `inner` field being the counter. First, consider Listing 4.20 showing a client implementing the proposed guarantee. Interference should be simulated before the lock acquisition and after release, in in lines 8 and 16, in order to model the rely condition. Moreover, a way needs to be found to express the pre- and postconditions on the state of the value wrapped by the `Mutex`. This can be challenging as no permissions are held to the wrapped value until the lock is acquired in line 10. In order to enable the declaration of contracts on the value protected by the lock, which is required by the rely/guarantee mechanism, a ghost value is used. This ghost value is a copy of the protected data, to which the current thread has full write access, and which is used to track the value in contracts and apply interference.

```
1  method client(x: Ref)
2    requires acc(Mutex(x), read())
3    requires acc(CanLock(x), write)
4    // precondition: x.inner >= 42
5    ensures  acc(Mutex(x), read())
6    // postcondition: x.inner >= 32
7  {
8    // interference
9    var data: Ref
10   data := lock(x)
11
12   data.inner := data.inner - 10
13
14   apply (acc(Dummy(data), write) --*
15             acc(CanLock(mutex), write))
16   // interference
17 }
```

Listing 4.20 [viper] beginning of an encoding of a client using a rely/guarantee Mutex. fold and unfold statements are omitted for brevity.

Interference from other threads is encoded as shown in Listing 4.21. The function defined in line 6 is simply a helper function used to access the `inner` field of `Dummy` types. The interesting part of this snippet is line 4. The `interference` method encodes any number of executions of the action by interfering threads as defined in the rely condition. This `interference` method can then be applied to the ghost value at any time to model the interference. More specifically, it will be applied shortly before the lock acquisition, modeling any interference that could have occurred between the assertion of the precondition and the lock acquisition. On top of that, it will be applied again shortly after releasing the lock, modelling any interference between the release and the assertion of the postcondition. Thus can be seen in Listing 4.22 in lines 10 and 20.

```
1 method interference(dummy: Ref)
2   requires acc(Dummy(dummy), write)
3   ensures  acc(Dummy(dummy), write)
4   ensures  old(get_inner(dummy)) <= get_inner(dummy)
5
6 function get_inner(dummy: Ref): Int
7   requires acc(Dummy(dummy), write)
8   ensures unfolding Dummy(dummy) in result == dummy.inner
```

Listing 4.21 `[viper]` encoding of the `interference` method simulating thread interference and thus the potential effects the rely can have on the value of the `Mutex`.

The ghost value also enabled to define the pre- and postconditions as shown in lines 4 and 7. The remaining work is to link the ghost value to the actual value returned from the lock. This is performed via the `assume` in line 13 and the assignment in line 17.

Listing 4.22 finalizes the encoding of methods using rely/guarantee, and shows how these can verify more complex properties than simple invariants.

Additionally, note that the call-site of `client()` does not need to be modified, even though interference can occur between any two calls taking the same `Mutex`. However, the body of `client()` already guarantees stability of the pre- and post-condition with respect to the rely condition, and therefore applying interference between any two calls taking the same `Mutex` will not modify the properties that can be asserted.

4.3.3.1 Rust Specification

The Rust specification of the introduced rely/guarantee mechanism is very difficult to formally define on a conceptual level. Consider the previous example with the counter. In order to specify the example in Prusti, the specification problem can

```
method client(x: Ref, y: Ref)
  requires acc(Mutex(x), read())
  requires acc(CanLock(x), write)
  requires acc(Dummy(y), write) &&
             unfolding Dummy(y) in y.inner >= 42
  ensures  acc(Mutex(x), read())
  ensures  acc(Dummy(y), write) &&
             unfolding Dummy(y) in y.inner >= 32
{
  interference(y)
  var data: Ref
  data := lock(x)
  assume data.inner == y.inner

  data.inner := data.inner - 10

  y.inner := data.inner
  apply (acc(Dummy(data), write) --*
             acc(CanLock(mutex), write))
  interference(y)
}
```

Listing 4.22 `[viper]` beginning of an encoding of a client using a rely/guarantee `Mutex`. `fold` and `unfold` statement are omitted for brevity.

be split into three distinct parts:

1. In the pre- and postcondition, one needs a way to refer to the data contained a `Mutex` passed as an arbitrary argument to the function. This can be done via a new helper function in Prusti specification such as `data(arg3)` to refer to the data contained in the `Mutex` passed as the `arg3` parameter. This is mostly an implementation problem, and conceptually quite trivial.
2. The rely condition needs to be specified somewhere. The first conceptual issue with this is to ensure that the condition does indeed model the interference from other threads properly. However, even assuming that this does not need to be verified, expressing this condition in Prusti is challenging, as Prusti essentially has no support for concurrency. Therefore, Prusti does not define how threads should be reasoned about conceptually, or how one can refer to a specific thread in a contract.
3. Finally, a function encoding a guarantee condition such as the one from the previous example can only be called by a single thread, as the guarantee condition is not contained in the rely condition. Thus having two threads

being allowed to call the function would intrinsically break the rely condition. The reason for this can be seen from Listing 4.22. If the method being encoded into `client()` is called from two different threads on the same `Mutex`, the rely condition cannot be accurate, as at least one other thread performs the action $n \rightsquigarrow n - 10$. Such a restriction also needs to be specified somehow in Prusti contracts. Again, this is not possible with a concrete model on how threading is modelled in Prusti contracts.

Moreover, it is unknown how the Viper encoding for moving values to other threads via closures will look, and whether this will require new specifications on Rust level.

4.3.4 `RwLock`

The only major difference between `RwLock` and `Mutex` is that it allows multiple concurrent reads simultaneously. Therefore, a `RwLock` can be seen as a concurrent version of the `RefCell` (introduced in the next section), which also allows several `borrow()` values to coexist. As the remaining behavior of `RwLock` is the same as for `Mutex`, it can be encoded in the same way. The encoding of `RwLock::write()` is essentially the same as `Mutex::lock()`, while the encoding of `RwLock::read()` is the same as `RwLock::write()` but taking only a read permission to the `CanLock` capability. Using such a read permission enables the property that several `read()` values can coexist, while not allowing a `write()` value to be obtained.

4.3.5 `RefCell`

`RefCell`s allow more fine grained reasoning than `Mutex`. Only a single variable can mutably reference the contents of a `RefCell` at any point in the program, or several variables can reference it immutably. This sounds similar to `Mutex`, which ensures mutual exclusion. However, when reasoning about `RefCell`s, one does not need to consider concurrent accesses to the data, and therefore interference from other threads is not a possibility. This can be seen in the presented encoding of `Mutex`, except that the state machine encoding the `Mutex` (Figure 4.2) could not make any assumptions on the state of the wrapped data unless the `Mutex` was acquired by the current thread. This is not the case for `RefCell`s, as only the current thread can hold a reference to it.

The lack of interference, in principle, allows to constantly track the state of the wrapped value inside a `RefCell`. However, tracking everything precisely becomes an issue when considering calls to external libraries. Therefore, while a similar encoding to the one presented for `Mutex` in Section 4.3.1 is possible, a more fine-grained is desirable. In order to achieve more fine grained verification, the idea of a state machine can be extended.

4.3.5.1 Rust Specification

In order to provide as much flexibility as possible, the presented solution allows to define a state machine with an arbitrary finite number of states, and an invariant for each state. This machine is then encoded in a similar fashion than for `Mutex` to Viper to enable verification. In order to specify such a state machine in Rust, the `#[state="..."]` attribute is presented, with the grammar shown in Figure 4.5.

```
⟨declaration⟩   ::= '[state="' ⟨body⟩ '"]'

⟨body⟩          ::= ⟨name⟩
                 |  ⟨name⟩ '=>' ⟨invariant⟩

⟨name⟩          ::= ⟨identifier⟩

⟨invariant⟩     ::= ⟨expression⟩
```

Figure 4.5 BNF for new state attribute.

Using this grammar, all states of the state machine can be defined, with their corresponding invariant, which provides meaning to the state. For instance, for a simple `RefCell` wrapping a `Dummy` that can take two states (`Positive` and `Negative`), a sample specification is shown in Listing 4.23. The order the states are declared in has no effect, with the exception that the first state is used as the initial state upon construction of the `RefCell`. Thus every constructed `RefCell` will need to guarantee the state invariant of the first declared state right after construction.

```rust
struct Dummy { inner: i32 }

#[state="Positive => data.inner > 0"]
#[state="Negative => data.inner < 0"]
type MyCell = RefCell<Dummy>;
```

Listing 4.23 `[rust]` sample declaration of a state machine for a `RefCell` with two states, `Positive` and `Negative`.

Listing 4.23 shows only two states for the `RefCell`. However, any number of states can be defined. Moreover, the invariants for the states need not be disjoint as is the case in the listing. This choice, both in number of states and precision of

their invariants, allows to make the verification arbitrarily fine-grained, as needed by the use-case.

The invariants provided for each state are typechecked in similar fashion to the invariant for `Mutex`. In other words, for every state, a single closure is generated, which contains both the code borrowing from the `RefCell`, and the invariant. Whether all closures are contained within a single static function, or whether each invariant is placed within a single static function is not relevant. Moreover, similarly to `Mutex`, the call to `borrow_mut()` is guaranteed to not create problems, at it can only panic at runtime, and the closures are never called.

The second thing, after states, that needs to be specified in Rust are state transitions. On the cell level, it can only transition between states when it is mutably borrowed from. On the function level, any function taking a `RefCell` as argument, either directly or transitively, can change the state of the `RefCell`.

On the cell level, every borrow needs to be marked to declare a state transition. This is performed via a new `#[transition="..."]` attribute. Its grammar is very simple and therefore not explicitly stated here. It essentially simply states the source state and the target state. Listing 4.24 shows an example of such a transition annotation in line 5. The same listing shows how state transitions are expressed on function levels, namely in the pre- and postconditions, such as in lines 1 and 2.

```rust
#[requires="Positive(cell)"]
#[ensures="Negative(cell)"]
fn foo(cell: &MyCell) {
    // ...
    #[transition="Positive => Negative"]
    let val = cell.borrow_mut();
    val.inner = -42;
    // ...
}
```

Listing 4.24 `[rust]` state transition declarations both on cell and function level for `RefCell`. The transition annotation can be omitted if no transition occurs.

4.3.5.2 Viper Encoding

The Viper encoding of the `RefCell` itself is extremely similar to the `Mutex` one, with only changes in naming. Listing 4.25 shows the two abstract predicates encoding the `RefCell`. Note the inclusion of a predicate `CanBorrowMut`. This predicate serves the same purpose as `CanLock`, preventing double mutable borrows as in Listing 4.2. The actual encoding of the state machine is done the same way as

was shown in the section introducing it (Section 4.3.1.4). Finally, the `borrow_mut()` function and the drop of the returned wrapped reference are encoded the same way as for `Mutex`. This can also be seen in Listing 4.25.

```
1 predicate RefCell(cell: Ref)
2
3 predicate CanBorrowMut(cell: Ref)
4
5 method borrow_mut(cell: Ref) returns (data: Ref)
6   requires acc(RefCell(cell), read()) &&
7            acc(CanBorrowMut(cell), write)
8   ensures  acc(RefCell(cell), read()) &&
9            acc(Dummy(data), write) &&
10           (acc(Dummy(data), write) --*
11             acc(CanBorrowMut(cell), write))
```

Listing 4.25 `[viper]` encoding of `RefCell` and its `borrow_mut()` method.

Function level state transitions are encoded as expected, by simply adding the abstract predicate representing the state in the pre- or postcondition. Cell level state transitions are more involved as for `Mutex`. In the `Mutex` case, the transition in the state occurred at lock and release time. In the `RefCell` case, transitions occur during the borrow, and the state can be preserved until the next borrow. Moreover, the invariant that needs to be assumed and asserted at the start and end of the borrow, respectively, is state dependent. Listing 4.26 shows the encoding of Listing 4.24 into Viper.

First, the function level state transition can be seen in lines 4 and 5. These will be inhaled and exhaled on call-sites and therefore track the state transitions of the `RefCell` on the side of the caller. Moreover, they ensure the state transition(s) performed within the method's body reflect the transition presented to the caller. Next, the call to `borrow_mut()` can be seen in line 13. The call can be performed due to permissions passed in the precondition in line 3. Using these "permissions to borrow mutably" ensures `foo()` cannot be called while its argument is borrowed mutably from, catching the runtime panic that would occur at verification time. Right after the borrow in line 13, the state invariants are assumed. Note that all state invariants are assumed via implication. Contrary to `Mutex`, the active state is known for `RefCell` as no other thread can interfere with its state. Thus it would also be an option to have Prusti track the current state of the `RefCell` and only assume its specific invariant. Moreover, attributes mark transitions in in Rust (unless no state transition occurs), making such tracking reasonably simple to implement. Here all invariants are assumed via implication as it provides a sound solution which is easier to implemented in Prusti. After the assumptions from lines 15 and 16, the invariant of the current state can be used. The assertion

in line 19 is not part of the encoding of Listing 4.24, but only illustrates that the invariant from the positive state of the `RefCell` can be used after the borrow. Finally, the drop of the reference returned from `borrow_mut()` is encoded. This includes the state transition encoding in lines 24 and 25, and the assertions guaranteeing the state change respected the invariants set by the states (lines 28 and 29). Here, again, only the invariant of the current state could be asserted, if the solution is implemented in Prusti with state tracking. Once this is performed, the magic wand from the postcondition of `borrow_mut()` is applied, giving up the permissions to the wrapped data of the `RefCell`, and returning permissions to borrow mutably.

4.3.5.3 Implicit Transitions

The solution presented so far requires a `#[transition="..."]` annotation every time a transition is performed via a mutable borrow. This provides a high level of control and ensures no accidental transitions are performed, but it is also very explicit. Since Prusti's goal is to enable an easy approach to verification, a second solution will be presented, where state transitions are implicitly tracked by Prusti.

```
method foo(cell: Ref)
  requires acc(RefCell(cell), read()) &&
           acc(CanBorrowMut(cell), write)
  requires Positive(cell)
  ensures  Negative(cell)
  ensures  acc(RefCell(cell), read()) &&
           acc(CanBorrowMut(cell), write)
{
  // ...

  // borrow_mut and invariant inhales
  var dummy: Ref
  dummy := borrow_mut(cell)
  unfold Dummy(dummy)
  assume Positive(cell) ==> dummy.inner > 0
  assume Negative(cell) ==> dummy.inner < 0

  // can use rely
  assert dummy.inner > 0

  dummy.inner := -42

  // release
  exhale Positive(cell)    // state change
  inhale Negative(cell)

  // ensure state change is correct
  assert (perm(Positive(cell)) > none) ==> dummy.inner > 0
  assert (perm(Negative(cell)) > none) ==> dummy.inner < 0

  fold Dummy(dummy)
  apply acc(Dummy(dummy), write) --*
          acc(CanBorrowMut(cell), write)

  // ...
}
```

Listing 4.26 [viper] encoding of RefCell state transitions in Viper.

```rust
#[requires="Positive(cell)"]
#[ensures="Negative(cell)"]
fn foo(cell: &MyCell) {
    let val = cell.borrow_mut();
    val.inner = -42;
    bar(cell);
}

#[requires="Negative(cell)"]
#[ensures="Negative(cell)"]
fn bar(cell: &MyCell) { /* ... */ }
```

Listing 4.27 `[rust]` sample method body with no `#[transition="..."]` annotations on each state transition.

Since transitions will no longer be annotated directly at every borrow, a new mechanism needs to ensure that only allowed transitions can take place. This is achieved by specifying the set of all legal state transitions right next to the state declarations. An example of such a declaration can be seen in Listing 4.28. In the listing, the declared state machine allows transitions from a state to itself, and from `Positive` to `Negative`. However, the transition back from `Negative` to `Positive` is not declared and therefore not allowed.

```rust
struct Dummy { inner: i32 }

#[state="Positive => data.inner > 0"]
#[state="Negative => data.inner < 0"]
#[transition="Positive => Positive"]
#[transition="Negative => Negative"]
#[transition="Positive => Negative"]
type MyCell = RefCell<Dummy>;
```

Listing 4.28 `[rust]` sample declaration of a state machine for a `RefCell` with two states and three legal transitions.

Using the declared set of legal state transitions, Prusti can keep track of the set of potential state transitions throughout the verification of a method body. Concretely, the precondition defines an initial set of states that the finite state machine can be in. Then the state can be changed via either direct borrows, or method calls taking the `RefCell` as an argument. In the former case, Prusti can assume the invariants of the set of potential states, exhale the set of current states, and inhale implications for all states reachable from the current set of potential states. Listing 4.29 shows the encoding for such a transition based

on the Rust method shown in Listing 4.27. Note that in line 1, the invariant assumption uses an implication with the current state on the left hand side. This is not required when the set of potential states is a single state. However, for a larger set, it is required as Prusti is ignorant on which state could actually be currently active. This is easily seen when considering another borrow after line 9. If another mutable borrow occurs within the same method, Prusti is unaware which state (`Positive` or `Negative`) the state machine might be in. On the other hand, thanks to the invariants in the `inhale` statements from lines 5 and 6, Viper will only have a `Negative` resource. At this point, using two implications similar to the one in line 1 will ensure that only the invariant of `Negative` is actually assumed.

```
1  assume Positive(cell) ==> dummy.inner > 0  // rely on invariant
2  exhale Positive(cell)  // exhale set of current potential states
3
4  // both Positive and Negative are reachable from Positive
5  inhale (dummy.inner > 0) ==> Positive(cell)
6  inhale (dummy.inner < 0) ==> Negative(cell)
7
8  fold Dummy(dummy)
9  apply acc(Dummy(dummy), write) --* acc(canBorrowMut(cell), write)
10 // New set of potential states (Positive + Negative)
```

Listing 4.29 `[viper]` encoding of `RefCell` implicit state transition in Viper.

In the latter case, when calling a method taking the `RefCell` as an argument, the transition performed by the method is known, and therefore it is sufficient to verify that the state set declared in the called method's precondition is in fact in the set of current potential states at call-time, and exhale all other potential states. Prusti's regular contract handling would then take care of exhaling the state from the called method's precondition, and inhale the state from the postcondition. In the example provided by Listing 4.27, this would imply verifying that `Negative` is in the set of potential states at call-time of `bar()`, and exhaling `Positive` (as it is also in the set of potential states at call-time).

Finally, when the end of the method body to be verified is reached, it is enough to check whether the set of states declared in the postcondition is a subset of the potential states at the end of the method body. In the example Listing 4.27, the set of potential current states at the end of the method body is only `Negative`, which is also the state declared in `foo()`'s postcondition. Thus the method would verify.

4.3.5.4 Other RefCell Methods

First and foremost, the borrow() function can be encoded similarly to the borrow_mut() function, with the only difference being the amount of permissions to CanBorrowMut it takes in the precondition. It takes only a read permission, thus ensuring more than a single immutable borrow can occur simultaneously. However, this still safely prevents a mutable borrow to coexist with an immutable one, which is desired. The fact that a full write permission to the wrapped data is returned from borrow() is not problematic, as Rust already ensures that all code that gets encoded as acting on the immutable borrow does not modify the value. try_borrow() and try_borrow_mut() are encoded similarly.

The new() method is encoded by an inhale of the RefCell and CanBorrowMut predicates. The functions take(), replace(), and swap() can all be expressed in terms of borrow_mut(), and therefore are encoded accordingly. get_mut(), undo_leak(), and into_inner() do not exhibit interior mutability, and therefore, similarly to Mutex, can be encoded in a similar fashion to normal Rust functions, with the exception that they need to assume the invariant of the active state.

Finally, the methods as_ptr(), replace_with(), and try_borrow_unguarded () are unsupported due to use of raw pointers, closures, and unsafe functions respectively.

4.3.5.5 RefCells in Other Types

Rust functions taking objects that contain a RefCell also have the opportunity to modify the value contained in the RefCell during the call. Therefore, it should be possible to refer to the contained RefCell in the pre- and postconditions. This is performed using the standard dot notation for field dereferencing from Rust. Listing 4.30 illustrates this with an example, with the pre- and postconditions referencing the RefCell contained in container in lines 5 and 6.

```rust
struct Container {
    cell: MyCell,
}

#[requires="Positive(container.cell)"]
#[ensures="Negative(container.cell)"]
fn foo(container: &Container) {
    // ...
}
```

Listing 4.30 [rust] state transition declarations for RefCells contained within another type.

4.3.6 `Cell`

One major difference between `Cell` and `RefCell` is that the former never returns references to its inner value. The only two functions returning the value contained in the cell (`get()` and `update()`) use copy semantics to return the value. Thus the returned value cannot be used to modify the cell it was returned from. This implies that the encoding of a `Cell`'s methods is quite straightforward. Every function that allows to modify the inner value (`set()`, `replace()`, and `swap()`) can be encoded by an exhale of the predicate encoding the source state, and an inhale of the destination state encoding. Moreover, the state invariants need to be asserted to ensure the state transition is respected by the `set()` function. When such a function call does not trigger a state change, only the assertions need to appear in the Viper encoding. Functions querying the contents of the cell, and returning a copy of the data to the caller can be encoded by an inhale of the state invariants. The actual encoding of the `Cell` is solely the state machine presented in the section on `RefCell`s. Figure 4.6 shows an example encoding of a client method making basic use of a `Cell<Dummy>` and the corresponding Viper encoding.

4.4 Implementation Challenges

The implementation of the presented solution is made relatively simple by design. However, some challenges are bound to emerge. For instance, the invariants asserted for the first `Mutex` encoding, or the state invariants for other encodings, refer to the contents of the wrapper. However, the identifier used for these contents are provided by Rust code. Therefore, it is required to either modify the invariants as required by the location they are used in. Another approach is to not use the Rust identifier in the Viper encoding for returned references from the `lock()`/`borrow_mut()` functions. Instead, predefined identifiers can be used. However this might restrict the usage of the wrappers to not accept parallel acquisition of two locks, or other similar situations. No matter the design choice, it will require a change in the way identifiers are handled, or a change toward dynamic encoding of invariants based on call-site information.

Another challenge is error reporting. The designed solutions provide great flexibility in the verification of programs. However, they also increase the complexity in both usage of Prusti and encoding to Viper. Complex Viper encodings usually tend to make error reporting more complex as well, as failures in the verification of the Viper code need to be reported back on Rust source code level. This mapping back to the original source code can be quite difficult for some of the designed solutions.

Finally, a challenge introduced to the implementation for the proposed solutions is stability. Prusti cannot ensure the developer enters valid and correct

specifications for every Rust source code. However, it tries to make the specification of incorrect contracts difficult, as an additional safeguard. Examples of such safeguards include but are not limited to typechecking of contracts, or verification of self-framing of pre- and postconditions. Such checks help ensure the programmer does indeed verify the desired properties. Such checks can be difficult to implement for some solutions such as the rely/guarantee mechanism for `Mutex` and `RwLock`. For instance, referring back to the example in Section 4.3.3, it is difficult to ensure the developer does indeed provide a definition of interference (`n >= old(n)`) that accurately models a repeated application of the rely condition ($n \rightsquigarrow n+1$). Should a developer indeed make the mistake to provide a interference definition which does not properly model this, it could lead to very confusing errors or verification of a program specifying a broken the rely/guarantee model. Reducing such situation to a minimum via safeguards can be very challenging, even if not necessary for the correct implementation of the designed solution.

```
#[state="Positive => data.inner > 0"]
#[state="Negative => data.inner > 0"]
type MyCell = Cell<Dummy>;

pub fn main() {
    let cell = MyCell::new(Dummy { inner: 42 });

    #[transition="Positive => Negative"]
    cell.set(Dummy { inner: -10 });

    let mut d = cell.get();
    assert!(d.inner < 0);

    d.inner = 42;
}
```

```
method main() {
  var cell: Ref
  inhale Positive(cell)     // new

  var tmp: Ref              // set
  inhale Dummy(tmp)
  unfold Dummy(tmp)
  tmp.inner := -10

  exhale Positive(cell)     // transition
  inhale Negative(cell)
  assert (perm(Positive(cell)) > none) ==> tmp.inner > 0
  assert (perm(Negative(cell)) > none) ==> tmp.inner < 0

  var d: Ref                // get
  inhale Dummy(d)
  unfold Dummy(d)
  assume Positive(cell) ==> d.inner > 0
  assume Negative(cell) ==> d.inner < 0

  assert d.inner < 0        // assert

  d.inner := 42             // assignment
}
```

Figure 4.6 Sample Viper encoding of a Rust function using a `Cell`. Rust code at the top and its Viper encoding below. The Viper code is simplified compared to the actual code generated by Prusti. The encoding used for `Dummy` can be seen in Appendix B.

Chapter 5

Contract Derivation

Rust allows to derive some traits. In other words, it allows to annotate custom types with a `#[derive]` attribute, which tells the compiler to implement a specified trait in some standard form on the custom type. In such a case, even though the compiler provides an implementation for the trait, Prusti is not able to deal with the trait implementation. The goal of this chapter is to showcase a solution enabling Prusti to derive a contract for the implementation provided by the compiler. Moreover, the chapter discusses how traits performing operator overloading can be handled in a generic fashion.

5.1 Background

As mentioned above, `#[derive]` attributes can be used to tell the compiler to implement a standard form of a trait on a custom type. Listing 5.1 shows how such an attribute can be used to derive the implementation for `PartialEq` and `Eq` on the `Dummy` type.

```rust
#[derive(PartialEq, Eq)]
struct Dummy {
    inner: i32,
}
```

Listing 5.1 `[rust]` example of trait derivation on custom type `Dummy`.

The documentation of `PartialEq` [Rust Team, 2020b] states that:

> This trait can be used with `#[derive]`. When derived on structs, two instances are equal if all fields are equal, and not equal if any fields are not equal. When derived on enums, each variant is equal to itself and not equal to the other variants.

Thus, by deriving the trait in Listing 5.1, two instances of `Dummy` are equal

if and only if their `inner` fields contain equal integers. As a consequence, a structure can only derive `PartialEq` if all its fields implement `PartialEq`, and similarly for `Eq`. In other words, the compiler will generate an implementation for `PartialEq::eq()` that satisfies the contract defined in the documentation. On the other hand, as the `Eq` trait is a marker trait, deriving it does not change the code generated by the compiler for the implementation of `Eq`.

Trait derivations are very useful to reduce boilerplate code for standard implementations of common traits from the standard library. Crates can also define derivable traits. For instance, Rust's most used serialization crate `serde`[1] defines the derivable traits `Serialize` and `Deserialize`. As of the 23rd of August 2020, the following list of standard library traits were derivable[2]:

- `PartialEq`,
- `Eq`,
- `PartialOrd`,
- `Ord`,
- `Clone`,
- `Copy`,
- `Hash`,
- `Default`,
- `Debug`.

5.2 Problem Statement

As trait implementations can be derived by the compiler, Prusti should be able to derive trait contracts. The goal of this chapter is to present a solution on how the contracts for the traits listed in the previous section can be derived, such that they provide the strongest possible guarantees for verification.

Contract derivation has several advantages. It reduces the specification effort for standardized contracts. Standard contracts are actually very common, as program entities tend to represent a single logical entity at some determined abstraction level. Furthermore, the very purpose of traits is to provide a common logical interface that is type agnostic. Therefore it makes sense that most trait contracts can be standardized on the type they are implemented on.

Second, trait derivation is concise in terms of the required syntax, while adding considerable functionality to Rust programs. Similarly, providing meaning through contracts to the derived traits would improve verification without any additional required specification from the user.

Third, as contract derivation would be performed automatically by Prusti, it should be less prone to specification errors than manually adjoined contracts.

Furthermore, this chapter discusses operator overloading. In Rust, operator overloading is performed by implementing specific traits on a type that provide a function representing the operator. This is connected to the derivation of contracts for derivable traits, as nearly half of derivable traits from the standard library actually perform operator overloading.

[1] https://serde.rs/
[2] https://doc.rust-lang.org/rust-by-example/trait/derive.html

For most operators (with the exception of equality), Prusti ignores the overload and cannot assume anything on the result of applying an operator on a custom type. Listing 5.2 shows an example with the "less or equal" operator implemented via the `PartialOrd` trait. The function definition in line 9 is used to overload all comparison operators other than equality. It's manual implementation simply returns the ordering between the `inner` fields. Thus an instance of `Dummy` is less or equal to another if its `inner` field is less or equal to the `inner` field of the other instance. Thus, the assertion in line 25 is guaranteed to hold. However, Prusti does not encode the overload and therefore cannot infer anything on the result of the comparison between `m1` and `m2`, hence why the verification fails.

```rust
1  use std::cmp::Ordering;
2
3  #[derive(PartialEq)]
4  struct Dummy {
5      inner: i32,
6  }
7
8  impl PartialOrd<Dummy> for Dummy {
9      fn partial_cmp(&self, other: &Dummy) -> Option<Ordering> {
10         if self.inner < other.inner {
11             Some(Ordering::Less)
12         } else if self.inner > other.inner {
13             Some(Ordering::Greater)
14         } else {
15             Some(Ordering::Equal)
16         }
17     }
18 }
19
20
21 fn main() {
22     let m1 = Dummy { inner: 42 };
23     let m2 = Dummy { inner: 42 };
24
25     assert!(m1 <= m2);    // does not verify!
26 }
```

Listing 5.2 [✗ `rust`] partial ordering not supported by Prusti. The code does not verify even though the assertion is guaranteed to hold.

This chapter will also discuss how Prusti can handle such overloads.

5.3 Approaches

The section starts by discussing how Prusti could reason about operator overloads, and then moves on to presenting the approaches chosen to encode the derived contracts for traits. As traits from the standard library are the most commonly derived (see list of traits from Section 5.1), the solutions in this thesis will focus on them. The approaches for the traits are generally quite different, but the heuristic guiding the reasoning is rather consistent. Therefore the solution could be extended to other derivable traits if needed.

5.3.1 Operator Overloading

Prusti already performs operator overloading for the `PartialEq` trait when it is derived. In order to encode the overload, it uses Viper *domains*. Domains allow the declaration of new types with attached functions and axioms. Domains are fully abstract, in that their functions have no body or preconditions, and their behavior is only defined via the domain axioms. An example declaration for a domain is shown in Listing 5.3, taken from the Viper tutorial.

```
1 domain MyDomain {
2   function foo(): Int
3   function bar(x: Bool): Bool
4
5   axiom axFoo { foo() > 0 }
6   axiom axBar { bar(true) }
7   axiom axFoobar { bar(false) ==> foo() == 3 }
8 }
```

Listing 5.3 `[viper]` example declaration for Viper domains, taken from the Viper tutorial.

Prusti uses such domains in order to express equality between custom types. It achieves this by creating a bijection between the type's Viper encoding, and a so called "snapshot" of the type, expressed as a domain. The snapshots have several advantages over regular type encodings. First, they can be returned from Viper functions, allowing for more flexibility in the usage of custom types in relation to functions. Second, comparing equality of snapshots compares the snapshots themselves, whereas comparing references checks whether they point to the same object. Listing 5.4 shows an encoding of a type and its domain.

```
field inner: Int

predicate Dummy(self: Ref) {
  acc(self.inner, write)
}

domain SnapDummy {
  function consDummy(i: Int): SnapDummy

  axiom SnapDummyInjectivity {
    forall i1: Int, i2: Int ::
    { consDummy(i1), consDummy(i2) }
    consDummy(i1) == consDummy(i2) ==> (i1 == i2)
  }

  axiom SnapDummySurjectivity {
    forall s: SnapDummy :: (
      ( forall i: Int :: {consDummy(i)}
       s != consDummy(i) ) ==> false
    )
  }
}

function snapDummy(self: Ref): SnapDummy
  requires acc(Dummy(self), read())
{
  unfolding acc(Dummy(self), read()) in
    consDummy(self.inner)
}
```

Listing 5.4 `[viper]` example encoding of a type's snapshot for a `Dummy` type with declaration `Dummy { inner: i32 }` when `PartialEq` is derived.

Note that the snippet defines an injective and surjective property between the snapshot of `Dummy` and its internal integer in lines 10 and 16. These allow to create a bijective mapping between the wrapped integer and the wrapping `Dummy`, enabling to consider the `Dummy` equal if the wrapped integers are equal. Finally, line 24 declares a constructor to build a snapshot from the normal Viper encoding of `Dummy`. This allows to create snapshots from an object any time one is needed.

To illustrate our approach to operator overloading, we will use `PartialOrd` from Listing 5.2. In said case, a domain function is used to encode the behavior of `partial_cmp()`. Axioms are used to provide the domain function with meaning. The axioms' bodies is determined by the implementation of `PartialOrd::partial_cmp()`. Listing 5.5 shows the additional domain function with a set of axioms.

```
1  domain SnapDummy {
2    function cmp(a: SnapDummy, b: SnapDummy): Int
3
4    axiom SnapDummyCmpGT {
5      forall i1: Int, i2: Int ::
6      { consDummy(i1),consDummy(i2) }
7      (i1 > i2) ==> (cmp(consDummy(i1), consDummy(i2)) == 1)
8    }
9
10   axiom SnapDummyCmpLT {
11     forall i1: Int, i2: Int ::
12     { consDummy(i1),consDummy(i2) }
13     (i1 < i2) ==> (cmp(consDummy(i1), consDummy(i2)) == -1)
14   }
15
16   axiom SnapDummyCmpEq {
17     forall i1: Int, i2: Int ::
18     { consDummy(i1),consDummy(i2) }
19     (i1 == i2) ==> (cmp(consDummy(i1), consDummy(i2)) == 0)
20   }
21 }
```

Listing 5.5 `[viper]` Viper encoding of `partial_cmp()` function of `PartialOrd` trait on `Dummy`. See Listing 5.4 for `consDummy()` definition.

For simplicity, the encoding ignores the `Option<T>` wrapping the result, and encodes the `Ordering` as an integer. Concretely, `-1`, `0`, and `1` encode `Ordering::Less`, `Ordering::Equal`, and `Ordering::Greater` respectively.

It is worth noting that encoding Rust's operators as abstract domain functions

with axioms which reflects the implementations of the operators is nearly identical to simply implementing a normal Viper function with the same code. While this is absolutely true in some cases, the domain approach provides a more flexible encoding for future improvements. For instance, domains can be used with universal quantifiers (`forall`), making the expression of properties on the snapshot much simpler than on Viper predicates. For instance, Section 6.1.3 from Chapter 6, proposes better support for universal quantification on custom types to allow expressing invariants such as the one shown in Listing 5.6 below. When encoding equality in a domain, such an invariant can be expressed by a domain axiom without much effort. This would not be the case when encoding equality directly as a standard Viper function.

```rust
#[invariant="forall a: Self :: {<trigger>} a == a"]
pub trait Eq: PartialEq<Self> {}
```

Listing 5.6 `[rust]` true contract to express reflexivity of equality relation on `Eq` trait. The trigger is left parametrized.

Furthermore, the domain approach allows to consolidate the approach for all operator overloads. For instance, regular functions cannot return references, but are allowed to return domain instances. Therefore operators returning custom types, such as the addition operator, **require** an encoding based on domains to enable overloading.

Nevertheless, the presented approach has a couple drawbacks. First and foremost, axioms are expressed as Viper expressions not reading any program state. However, not all Rust code can be expressed as such expressions. Therefore, some implementations of operators might not be directly translatable to an axiom. For instance, any operator performing some caching for performance could be problematic. In this case, a workaround would be to not encode the caching in Viper, as it does not contribute to the externally visible logic of the operator. However, when considering manual implementations of operators, extracting only the logic from the implementation can be extremely challenging. Furthermore, even if the code contains only logic, encoding it can be non-trivial. For instance, the axioms presented in Listing 5.5 are difficult to automatically infer from the implementation in Listing 5.2. Secondly, many operators are currently simply not encoded to Viper by Prusti. Adding an encoding, while allowing more thorough verification, affects performance. From the few test performed by the author, the performance impact of adding such encodings is not very significant. However, this was only tested on synthetic examples, and Rust code making extensive use of operator overloading might be heavily affected by this added encoding. Moreover, performance is heavily dependent on the type of operator. Encoding operators such as addition, which return a new snapshot, seemed to lead to significant performance drops in verification.

Having covered how operator overloading can be handled soundly in Prusti, contract derivation is discussed.

5.3.2 PartialEq<Rhs>

As previously mentioned, Prusti currently supports deriving the contract for PartialEq. However, it is supported only in very specific cases. Indeed, Prusti can derive the contract for PartialEq only if the implementation is derived, and all transitively contained types of the structure also implement PartialEq via derivation. As soon as any (directly or transitively) contained type implements PartialEq manually, the contract can no longer be derived. This is caused by the fact that Prusti simply encodes manual implementations of PartialEq as any regular trait implementation, but does not infer the link between the operator and the defined `eq()` function. Moreover, the problem with manual implementations is that they might not implement a total equality, and therefore a snapshot instance might not be logically equal to itself. Yet, this is a required to uphold the mathematical property that $x = y$ implies $f(x) = f(y)$ for a pure function f. As this property is used in the bijection between domain instances and the Cartesian product of their fields, equality between snapshots cannot be used to encode PartialEq for manual implementation. An example of a valid manual implementation of PartialEq that is not a total equality can be seen in Listing 5.7. It implements transitivity and symmetry.

```rust
struct OtherDummy {
    d1: i32,
    d2: i32,
}

impl PartialEq<OtherDummy> for OtherDummy {
    fn eq(&self, other: &OtherDummy) -> bool {
        self.d1 != self.d2 &&
            other.d1 != other.d2 &&
            self.d1 == other.d1
    }
}
```

Listing 5.7 [rust] partial equality manual implementation.

In order to handle such cases for normal operator overloading, we introduce a new domain function. This new function encodes equality between two snapshots as expressed in the manual implementation of PartialEq. In other words, when PartialEq is not derived but manually implemented, we extract an expression from the trait method's body and use it in the domain's encoding to express the

new domain function's meaning (within an axiom). The encoding of Listing 5.7 can be seen in Listing 5.8. The axiom in line 5 reflects the Rust implementation of `eq()` from Listing 5.7. Then, when encoding any `==` operator as a call to `equalsOtherDummy()`, it enables the verification of code whose behavior relies on a manual implementation of `PartialEq`. The `consOtherDummy()` function is omitted for brevity, but is analogous to the definition of `consDummy()` in Listing 5.4, but with two parameters for both wrapped integers.

```
domain SnapOtherDummy {
  // ...
  function eq(a: SnapOtherDummy, b: SnapOtherDummy): Bool

  axiom SnapOtherDummyEquality {
    forall i1: Int, j1: Int, i2: Int, j2: Int ::
    { consOtherDummy(i1, j1),consOtherDummy(i2, j2) }
    eq(consOtherDummy(i1, j1), consOtherDummy(i2, j2)) <==> (
        (i1 != j1) && (i2 != j2) && (i1 == i2))
  }
}

function equalsOtherDummy(self: Ref, other: Ref): Bool
  requires acc(OtherDummy(self), read())
  requires acc(OtherDummy(other), read())
{
  eq(snapOtherDummy(self),snapOtherDummy(other))
}
```

Listing 5.8 [viper] encoding of abstract `eq()` domain function, including the axiom giving it meaning, and a regular function to be called on references.

Returning to contract derivation, the current implementation could be changed to derive a contract making use of the `eq()` domain function, for consistency. In this case, the body of the axiom in line 5 would need to be modified to have the conjunction of calls to `eq()` between all respective fields on the right hand side of the biconditional. This is important because using equality between domains would no longer be sound when deriving a `PartialEq` trait, as a field might manually implement a partial equality.

5.3.3 Eq

Contract derivation for Eq[3] is straightforward considering the newly introduced trait invariants. In fact, any marker trait can only be annotated with a trait invariant, and hence there is never any work to do when deriving the trait. The trait invariant will always be applied to any type implementing the trait, whether via normal implementation, derivation, or blanket implementation. This solution does however assume, that the standard library defines a valid trait invariant on the Eq trait, or that the contract is somehow injected.

5.3.4 PartialOrd<Rhs>

The PartialOrd documentation[4] states:

> This trait can be used with #[derive]. When derived on structs, it will produce a lexicographic ordering based on the top-to-bottom declaration order of the struct's members. When derived on enums, variants are ordered by their top-to-bottom declaration order.

Deriving a contract adhering to the documentation is quite feasible. It essentially comes down to deriving the bodies of the axioms declared in Listing 5.5. Deriving the bodies for a lexicographic ordering can be done via a naive heuristic. For instance, the body of the axiom encoding equality is simply the conjunction of the equalities among all fields. Finally, the other two axiom bodies can be obtained by a disjunction of comparisons. Listing 5.9 illustrates this for a OtherDummy type with two integer fields.

It is important to note that Listing 5.9 shows comparisons using operators in the axiom bodies. This would however be done via the domain functions if a field would be a non-primitive type. The domain function for the field type is guaranteed to exist, as the field needs to implement PartialOrd as well, lest the Rust compiler would reject the trait derivation.

5.3.5 Ord

The standard implementation of PartialOrd provides a total ordering. Therefore the contract for Ord::cmp()[5] is essentially the same as for PartialOrd::partialcmp(). In fact, removing the Option<T> wrapper from the result is the only required change to obtain the correct cmp() contract.

However, additional axioms can be added that some additional correctness properties are upheld. These are not necessary for sound verification, but would

[3]https://doc.rust-lang.org/std/cmp/trait.Eq.html
[4]https://doc.rust-lang.org/std/cmp/trait.PartialOrd.html#derivable
[5]https://doc.rust-lang.org/std/cmp/trait.Ord.html

```
axiom SnapOtherDummyCmpGT {
  forall i1: Int, j1: Int, i2: Int, j2: Int ::
  { consOtherDummy(i1, j1),consOtherDummy(i2, j2) }
  (i1 > i2 || (i1 == i2 && j1 > j2)) ==>
    (cmp(consOtherDummy(i1, j1), consOtherDummy(i2, j2)) == 1)
}

axiom SnapOtherDummyCmpLT {
  forall i1: Int, j1: Int, i2: Int, j2: Int ::
  { consOtherDummy(i1, j1),consOtherDummy(i2, j2) }
  (i1 < i2 || (i1 == i2 && j1 < j2)) ==>
    (cmp(consOtherDummy(i1, j1), consOtherDummy(i2, j2)) == -1)
}

axiom SnapOtherDummyCmpEq {
  forall i1: Int, j1: Int, i2: Int, j2: Int ::
  { consOtherDummy(i1, j1),consOtherDummy(i2, j2) }
  (i1 == i2 && j1 == j2) ==>
    (cmp(consOtherDummy(i1, j1), consOtherDummy(i2, j2)) == 0)
}
```

Listing 5.9 `[viper]` axioms representing the lexicographic ordering for the `cmp()` domain function. This is based on the `OtherDummy` type with two integer fields defined in Listing 5.7.

help reduce errors on manual implementation of `PartialEq`, `PartialOrd`, or `Ord`. For instance, the documentation of `Ord` [Rust Team, 2020a] states that:

> Implementations of `PartialEq`, `PartialOrd`, and `Ord` must agree with each other. That is, `a.cmp(b) == Ordering::Equal` if and only if `a == b` and `Some(a.cmp(b)) == a.partial_cmp(b)` for all `a` and `b`. It's easy to accidentally make them disagree by deriving some of the traits and manually implementing others.

This can be ensured using additional axioms relating the `cmp()` and `eq()` domain functions. For instance, Listing 5.10 presents an axiom ensuring `a.cmp(b) == Ordering::Equal` if and only if `a == b`. Again, for simplicity the `Option<T>` is omitted, therefore `cmp()` actually `Ord::cmp()` quite accurately, instead of `PartialOrd::partial_cmp()`. Note that such axioms have nothing to do with contract derivation of `Ord` itself. They simply ensure their proper usage when some of the comparison traits are manually implemented.

```
axiom SnapOtherDummyEqCmpConform {
  forall i1: Int, j1: Int, i2: Int, j2: Int ::
  { consOtherDummy(i1, j1),consOtherDummy(i2, j2) }
  eq(consOtherDummy(i1, j1), consOtherDummy(i2, j2)) <==>
    (cmp(consOtherDummy(i1, j1), consOtherDummy(i2, j2)) == 0)
}
```

Listing 5.10 [viper] example axiom ensuring conformity between implementations of `PartialEq`, `PartialOrd`, and `Ord`.

5.3.6 `Default`

The derived contract for `Default`[6] can be built recursively from the `Default` contracts of the fields. Any standard library type implementing `Default` can have a properly defined contract on what value is returned, as the implementation is known. Moreover, custom types that manually implemented `Default` can choose to define a contract. Thus, by using the combination of these contracts in the derived contract, no guarantees are provided on custom types without a contract, but every other field of the type deriving `Default` is known to guarantee their own `Default` contract. Whether the standard library contracts are injected directly via the derivation mechanism via some lookup, or via a separate process handling the injection of contracts to non-annotated code is implementation specific.

For instance consider Listing 5.11. It declares a `DummyWrapper` with two fields, a `f32` floating pointer number, which defaults to zero, and a `Dummy`, which defines a custom contract.

Using the contract from the standard library and the custom contract, a very precise contract can be provided for `Default`. The derived contract for Listing 5.11 is illustrated in Listing 5.12.

5.3.7 `Clone`

Defining a contract for `Clone`[7] can be more complex than expected. It is commonly understood for a clone to be a duplicated object, hence having a different raw address in memory, that is equal to the original object. The former property is guaranteed by Rust and is easily verifiable. Rust guarantees that even empty types effectively point to different memory locations when cloned. The latter property is problematic, as there is no guarantee that fields of a clonable type are comparable. This makes reasoning about cloning on a formal level quite difficult, as the only property that can be assumed about the type is that its every field

[6] https://doc.rust-lang.org/std/default/trait.Default.html
[7] https://doc.rust-lang.org/std/clone/trait.Clone.html

```rust
struct Dummy {
    inner: i32,
}

impl Default for Dummy {
    #[ensures="result.inner == 1"]
    fn default() -> Dummy {
        Dummy { inner: 1 }
    }
}

#[derive(Default)]
struct DummyWrapper {
    field1: f32,
    dummy: Dummy
}
```

Listing 5.11 [rust] example for contract generation for `Default` trait.

```rust
impl Default for DummyWrapper {
    #[ensures="result.field1 == 0.0"]
    #[ensures="result.dummy.inner == 1"]
    fn default() -> DummyWrapper { ... }
}
```

Listing 5.12 [rust] sample contract for derived `Default` trait from Listing 5.11.

defines a `clone()` method.

In order to argue about equality of fields for the type implementing `Clone`, `Eq` will need to be considered. The different cases are:

1. If the type itself implements `Eq`, then it becomes trivial to ensure that the cloned object is equal to the original one in `clone()`'s postcondition. This postcondition is type agnostic (i.e. its formulation does not depend on the structure of the type implementing `Clone`), and thus a static contract expressing that the result must be equal to the argument can be used and automatically added.
2. If the type itself does not implement `Eq`, it might still be derivable. This would allow to ensure that all fields of a cloned instance are equal to the original instance. However, it is worth noting that this will not imply that the cloned instance itself would be equal to the original, `PartialEq` might

have been manually implemented on the type with a partial equality.

3. Finally, consider the fields individually. The field's type implements `Clone` either via the standard library, via a `#[derive]` clause, or via manual implementation. In this case, the same reasoning as for `Default` from Section 5.3.6 can be applied. In other words, for each field, if the field implements `Clone` in the standard library, a predetermined contract for it is used. In the case the field derived `Clone`, the derived contract for the field is used. Finally, if the field implements `Clone` manually, the given contract is used if it was provided, otherwise no contract is used for the field. If no contract is used there is simply no guarantee on the state of the field is provided by the derived contract for the type under consideration.

A sound approach would thus be to check which of these scenarios applies in order. On the first matching scenario, a contract for `Clone::clone()` could be derived as explained. As the last scenario is always applicable, every custom type would get a derived contract. Nonetheless, the strength of the guarantees provided by the contract is dependent on what additional properties the type or its fields provide (such as implementing `Eq`).

5.3.8 `Copy`

The `Copy`[8] trait is a marker trait signifying that instances can be duplicated by simply copying bits. This has important implications for Rust, as it determines whether move or copy semantics apply to a type. It also affects verification in that functions taking an object by value consume that object unless the object's type is `Copy`. Thus, unless `Copy` is implemented, they have a side-effect and cannot be considered pure. Whether allowing such functions to be declared as pure actually creates problems within Prusti is uncertain. Indeed, Prusti typechecks contracts in a way that already guarantees that the pre- and postconditions as a whole behave according to Rust's type system. This includes that an object cannot be captured twice in a single postcondition for example. In fact, the author did not find a single way to use object capturing to break verification. However, this is likely partially due to the currently limited support for concurrency in Prusti. Therefore, as no formal proof is defined to demonstrate that allowing pure function to capture objects is safe, it should be disallowed.

Prusti can be adapted to prohibit pure functions taking arguments by value, if the types of these arguments don't all implement `Copy`. Note that this is fully unrelated to the derivation of the `Copy` trait itself. In fact, as `Copy` is a marker trait, its contract would be defined by a trait invariant, thus making it trivial to derive, and similar to `Eq`.

[8] https://doc.rust-lang.org/std/marker/trait.Copy.html

5.3.9 `Hash`

A solution for the `Hash`[9] trait will not be presented in this thesis. This is due to the nature of Rust's `hash()` function. Said function takes a reference to the receiver object which is to be hashed, and a mutable reference to a stateful hasher which will aggregate all hashes it was used to generate. Ideally, one would like to reason about the fact that a hash is a function which is fully deterministic on input. However, this is made difficult by Rust's design for hashing. Concretely, one would need to somehow express that the state change of the hasher is the same for two `hash()` calls taking the same state of the receiver object and the same state of the hasher. The issue about reasoning in such a way is twofold. First, reasoning abstractly about state changes is difficult, unless the type is handled in a special way by Prusti. Second, the actual type of the hasher is unknown when deriving the `Hash` trait, as the `hash()` function takes a generic parameter bound by the `Hasher` trait instead of a concrete type.

Moreover, the main benefit of using the information about how hashes behave is to verify more properties about data structures using hashing such as hashmaps. It is however much simpler to directly treat standard library data structures using hashing internally in a special way by Prusti. For instance, getting the value for equal keys on an immutable hashmap should always return the same value. Such a property is easier to directly encode on the `HashMap::get()`[10] function. In fact, it is enough to treat `get()` as a pure function to ensure this property. Thus, the most common use-cases of `hash()` can be covered using more elegant solutions then by providing a complex new mechanism to reason about `hash()`.

5.4 Summary and Shortcomings

This chapter presented a new way to encode operator overloads and how contracts can be derived for derivable traits. The section on operator overloading focused on very few traits, but its heuristic can be applied to any additional operators. The solution was designed with flexibility for extension in mind. This comes at the cost of a restriction on the implementation of operators, and performance. Indeed, due to the domain encoding of operators, their implementation needs to be "pure", in that it is deterministic and side-effect free.

The section focusing of contract derivation introduced different approaches to derive contracts for most derivable standard library traits. Such contract derivation provides significant advantage to the simplicity and verification strength of Prusti.

[9] https://doc.rust-lang.org/std/collections/struct.HashMap.html
[10] https://doc.rust-lang.org/std/collections/struct.HashMap.html#method.get

Chapter 6

Conclusion

This thesis has addressed the handling of `unsafe` traits and interior mutability. It has provided a way to reason about marker traits, provide specifications for intrinsic properties of traits, or implicit properties of types implementing the traits. Moreover, it presented a way to reduce the unsafety from using `unsafe` traits. Thus it contributed to making unsafe Rust safe.

On top of that, it provided a way to reason modularly about interior mutable objects. Using a rely/guarantee mechanism, it showed how concurrent data accesses in Rust can be represented and modelled in Viper. Furthermore, the state machine backed encoding for thread-unsafe wrappers to `UnsafeCell` allows for arbitrarily fine-grained specification of both state and transitions of the wrapper. This improves the verification capabilities of Prusti for any program using these data structures, in spite of their interior mutable nature.

Finally, the thesis also presented how operator overloading can be handled via Viper domain encodings, and how those domain encodings can be used to express additional hyperproperties on the type snapshots they encode. Moreover, solutions were presented on how contracts can be derived for the majority of Rust's standard library derivable traits.

6.1 Future Work

This section presents opportunities for future work.

6.1.1 Atomic Types and Other Wrappers

This thesis did not address all interior mutable types. The Rust standard library defines a few more, less used, types which exhibit interior mutability. For instance, most primitive Rust types (such as `usize`[1]) have atomic counterparts (such as `AtomicUsize`[2]). These atomic types all exhibit interior mutability. They are less

[1] https://doc.rust-lang.org/std/primitive.usize.html

[2] https://doc.rust-lang.org/std/sync/atomic/struct.AtomicUsize.html

used than the wrappers subject to this thesis, but are used in many other types internally (e.g. `Arc`[3]). Whether types using the atomic type should be handled separately in Prusti is subject to further discussion, but being able to handle atomic types by themselves is surely useful to enable additional verification of user defined types containing atomics. The verification of these types is made complex due to the control over the memory ordering[4] the developer has when using these types. Therefore the verification, in all likelihood, would need to consider different orderings to make the verification sound.

6.1.2 Proper Concurrency Support

The solutions presented in this thesis have some shortcomings, due to the lack of support for some features in Prusti. One of these deficiencies in Prusti is the lack of support for useful concurrency. This includes support for closures, and fork/join semantics. As of now, this missing support makes the presented solution for `Mutex` and `RwLock` not very useful in practice, simply because their use in sequential code is extremely limited. However, both support for closures and concurrency are currently being implemented.

6.1.3 Improved Support for Universal Quantification in Invariants for Hyperproperties

Many invariants, both on traits and types, use quantified expressions. Currently, only primitive Rust types that have a direct mapping to Viper types (such as `usize`) can be used in quantified expressions. This drastically restricts the expressiveness of Prusti contracts, as it makes the verbalization of relations between instances of custom types nearly impossible. For instance, the trait invariant for `Eq` presented in Listing 3.6 is valid, yet a quantified expression such as the one shown in Listing 6.1 would be more adequate.

```rust
#[invariant="forall a: Self :: {<trigger>} a == a"]
pub trait Eq: PartialEq<Self> {}
```

Listing 6.1 `[rust]` true contract to express reflexivity of equality relation on `Eq` trait. The trigger is left parametrized.

Similarly, some contracts cannot be expressed without the support for such quantified invariants. An example is the intrinsic relation between the `eq()` and `ne()` methods of the `PartialEq<T>` trait. The documentation states that:

[3] https://doc.rust-lang.org/std/?search=Arc
[4] https://doc.rust-lang.org/std/sync/atomic/enum.Ordering.html

> Any manual implementation of `ne` must respect the rule that `eq` is a strict inverse of `ne`; that is, `!(a == b)` if and only if `a != b`.

Trait invariants would support the specification of such properties to ensure the manual implementation of the `ne()` method matches the documented behavior. However, this is not expressible without broader support for quantified expressions. Listing 6.2 illustrates the simplicity of the contracts, should this support be added.

```rust
#[invariant="
   forall a: Self, b: Rhs :: {<trigger>} !(a == b) == (a != b)
"]
pub trait PartialEq<Rhs: ?Sized = Self> {
    fn eq(&self, other: &Rhs) -> bool;

    fn ne(&self, other: &Rhs) -> bool {
        !self.eq(other)
    }
}
```

Listing 6.2 `[rust]` contract to express intrinsic relation between `eq()` and `ne()` methods of the `PartialEq<T>` trait. The trigger is left parametrized.

Similarly, symmetry and transitivity could be defined for the `PartialEq<T>` trait.

Viper domains, as mentioned in previous chapters, would be a great fit to encode such properties, as they can be used in universal quantification in Viper. A precursor to this work was already shown in Section 5.3.5 with the introduction of axioms to ensure consistent implementations of `PartialEq`, `PartialOrd`, and `Ord`. However, the extent to which this approach is useful with respect to the variety of properties that can be encoded with axioms using such universal quantification over the domain was not investigated in this thesis and would present a great opportunity for future work.

6.1.4 Unsafe Code

The ultimate goal for the verification of Rust would be the complete support for verification of unsafe Rust code. Unsafe Rust is generally difficult to reason about, and very prone to complex errors. Software verification is mostly useful in places where bugs are likely to occur, as it is relatively costly to perform, and therefore overkill for parts of the code that are very simple, or where extensive testing can easily be performed. This makes verification of unsafe Rust very desirable. However, as the guarantees provided by unsafe Rust are approximately equivalent

to regular C++, such verification is extremely difficult. Nevertheless, there are bound to be ways to encapsulate unsafe Rust to a larger extent, and reduce the risk attached to it. For instance, `unsafe` blocks not performing pointer arithmetic and dereferencing still have quite strong safety guarantees. Moreover, many `unsafe` code blocks are simply used to call standard library `unsafe` functions, which can potentially be abstractly modelled in such a way to make reasoning about their behavior somewhat useful for verification.

Bibliography

[Apt et al., 2009] Apt, K. R., de Boer, F. S., and Olderog, E.-R. (2009). *Verification of Sequential and Concurrent Programs*, page 3. Springer-Verlag, third edition. Citation on page 11

[Astrauskas et al., 2020] Astrauskas, V., Matheja, C., Poli, F., Müller, P., and Summers, A. J. (2020). How do programmers use unsafe rust? Preprint. Citation on page 21, 22

[Astrauskas et al., 2019] Astrauskas, V., Müller, P., Poli, F., and Summers, A. J. (2019). Leveraging rust types for modular specification and verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 3, pages 147:1–147:30. ACM. Citation on page 1, 12

[Blandy and Orendorff, 2017] Blandy, J. and Orendorff, J. (2017). *Programming Rust: Fast, Safe Systems Development*. O'Reilly Media, first edition. Citation on page 2, 7

[Bornat, 2000] Bornat, R. (2000). Proving pointer programs in hoare logic. In Backhouse, R. and Oliveira, J. N., editors, *Mathematics of Program Construction*, pages 102–126, Berlin, Heidelberg. Springer. Citation on page 11

[Brookes and O'Hearn, 2016] Brookes, S. and O'Hearn, P. W. (2016). Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65. Citation on page 45

[Dinsdale-Young et al., 2010] Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M. J., and Vafeiadis, V. (2010). Concurrent abstract predicates. In D'Hondt, T., editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer. Citation on page 45, 46, 47

[Erdin, 2019] Erdin, M. (2019). Verification of rust generics, typestates, and traits. Master's thesis, Federal Institute of Technology (ETH) Zürich, Zürich, Switzerland. Citation on page 31

[Evans et al., 2020] Evans, A. N., Campbell, B., and Soffa, M. L. (2020). Is rust used safely by software developers? *CoRR*, abs/2007.00752. Citation on page 10, 21, 22

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580. Citation on page 11

[Jasper et al., 2020] Jasper, M., Klabnik, S., Scheel, R., and Huss, E. (2020). *The Rust Reference*, chapter 10.4 Interior Mutability. Accessed 2020-05-19: `https://doc.rust-lang.org/reference/interior-mutability.html`. Citation on page 37

[Klabnik and Beingessner, 2020a] Klabnik, S. and Beingessner, A. (2020a). *The Rustonomicon, The Dark Arts of Unsafe Rust*, chapter 1.2 What Unsafe Can Do. Accessed 2020-05-19: `https://doc.rust-lang.org/nomicon/what-unsafe-does.html`. Citation on page 10

[Klabnik and Beingessner, 2020b] Klabnik, S. and Beingessner, A. (2020b). *The Rustonomicon, The Dark Arts of Unsafe Rust*, chapter 1.1 How Safe and Unsafe Interact. Accessed 2020-08-07: `https://doc.rust-lang.org/nomicon/safe-unsafe-meaning.html`. Citation on page 20

[Klabnik and Nichols, 2020] Klabnik, S. and Nichols, C. (2020). *The Rust Programming Language*, chapter 4.2 References and Borrowing. No Starch Press. Accessed 2020-05-14: `https://doc.rust-lang.org/stable/book/ch04-02-references-and-borrowing.html`. Citation on page 7

[Leino et al., 2009] Leino, K. R. M., Müller, P., and Smans, J. (2009). Verification of concurrent programs with chalice. In Aldini, A., Barthe, G., and Gorrieri, R., editors, *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer. Citation on page 44

[Loeckx and Sieber, 1987] Loeckx, J. and Sieber, K. (1987). *The Foundations of Program Verification*, chapter 6. Vieweg+Teubner Verlag, second edition. Citation on page 11

[Müller et al., 2016] Müller, P., Schwerhoff, M., and Summers, A. J. (2016). Viper: A verification infrastructure for permission-based reasoning. In Jobstmann, B. and Leino, K. R. M., editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag. Citation on page 15

[O'Hearn, 2007] O'Hearn, P. W. (2007). Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307. Citation on page 45

[Owicki and Gries, 1976] Owicki, S. S. and Gries, D. (1976). An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340. Citation on page 57

[O'Hearn et al., 2001] O'Hearn, P., Reynolds, J., and Yang, H. (2001). Local reasoning about programs that alter data structures. In Fribourg, L., editor, *Computer Science Logic. CSL 2001. Lecture Notes in Computer Science*, volume 2142, pages 1–19, Berlin, Heidelberg. Springer. Citation on page 11

[Qin et al., 2020] Qin, B., Chen, Y., Yu, Z., Song, L., and Zhang, Y. (2020). Understanding memory and thread safety practices and issues in real-world rust programs. In Donaldson, A. F. and Torlak, E., editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 763–779. ACM. Citation on page 21, 44

[Reynolds, 2002] Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society. Citation on page 45

[Rust Team, 2020a] Rust Team (2020a). Trait `std::cmp::Ord`. Accessed 2020-09-04: `https://doc.rust-lang.org/std/cmp/trait.Ord.html#how-can-i-implement-ord`. Citation on page 85

[Rust Team, 2020b] Rust Team (2020b). Trait `std::cmp::PartialEq`. Accessed 2020-09-04: `https://doc.rust-lang.org/std/cmp/trait.PartialEq.html#derivable`. Citation on page 75

[Rust Team, 2020c] Rust Team (2020c). Trait `std::iter::Iterator`. Accessed 2020-09-04: `https://doc.rust-lang.org/std/iter/trait.Iterator.html`. Citation on page 30

[Rust Team, 2020d] Rust Team (2020d). Trait `std::iter::TrustedLen` . Accessed 2020-09-04: `https://doc.rust-lang.org/std/iter/trait.TrustedLen.html`. Citation on page 30

[Smans et al., 2009] Smans, J., Jacobs, B., and Piessens, F. (2009). Implicit dynamic frames: Combining dynamic frames and separation logic. In Drossopoulou, S., editor, *European Conference on Object-Oriented Programming*, pages 148–172, Berlin, Heidelberg. Springer. Citation on page 15

[Vafeiadis, 2008] Vafeiadis, V. (2008). Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory. Citation on page 57

[Williams and Rust Team, 2020] Williams, A. and Rust Team (2020). Rust. Accessed 2020-09-04: `https://www.rust-lang.org/`. Citation on page 1, 7

Appendix A

Unmodified Listings for Prusti Example

Listing A.1 shows the original Rust program generated by Prusti from Listing 2.5. Note that Prusti also generates a program file for typechecking. That file is omitted here.

```
1  #![feature(custom_attribute)]
2  #![allow(dead_code)]
3  extern crate prusti_contracts;
4  #[pure]
5  #[ensures = "result >= a && result >= b"]
6  #[ensures = "result == a || result == b"]
7  #[__PRUSTI_SPEC = r#"101"#]
8  fn max(a: i32, b: i32) -> i32 { if a < b { b } else { a } }
9  #[__PRUSTI_SPEC_ONLY = r#"101"#]
10 #[allow(unused_mut)]
11 #[allow(dead_code)]
12 #[allow(non_snake_case)]
13 #[allow(unused_imports)]
14 #[allow(unused_variables)]
15 fn max__spec() -> () {
16     #[__PRUSTI_SPEC_ONLY = r#"101"#]
17     fn max__spec__pre(a: i32, b: i32) -> () { }
18     #[__PRUSTI_SPEC_ONLY = r#"101"#]
19     fn max__spec__post(a: i32, b: i32, result: i32) -> () {
20         #[allow(unused_imports)]
21         use prusti_contracts::internal::*;
22
23         #[__PRUSTI_EXPR_ID = r#"101"#]
24         #[pure]
25         || -> bool { result >= a };
26
```

```
27         #[__PRUSTI_EXPR_ID = r#"102"#]
28         #[pure]
29         || -> bool { result >= b };
30
31         #[__PRUSTI_EXPR_ID = r#"103"#]
32         #[pure]
33         || -> bool { result == a || result == b };
34     }
35 }
36 #[invariant = "self.d1 == self.d2"]
37 #[__PRUSTI_SPEC = r#"102"#]
38 struct Dummy {
39     d1: i32,
40     d2: i32,
41 }
42 impl Dummy {
43     #[__PRUSTI_SPEC_ONLY = r#"102"#]
44     #[allow(unused_mut)]
45     #[allow(dead_code)]
46     #[allow(non_snake_case)]
47     #[allow(unused_imports)]
48     #[allow(unused_variables)]
49     fn Dummy__spec(self) -> () {
50         #[allow(unused_imports)]
51         use prusti_contracts::internal::*;
52
53         #[__PRUSTI_EXPR_ID = r#"104"#]
54         #[pure]
55         || -> bool { self.d1 == self.d2 };
56     }
57 }
58 #[__PRUSTI_SPEC = r#"103"#]
59 fn test(d: &Dummy) {
60     let val = max(d.d1, d.d2);
61     assert!(val == d . d1);
62 }
63 #[__PRUSTI_SPEC = r#"104"#]
64 fn main() { }
```

Listing A.1 [rust] full Rust program generated by Prusti for spec generation and type checking.

Appendix B

Encoding of Dummy

In many examples throughout this thesis, a Rust `Dummy` type is used, whose declaration can be seen in Listing B.1. Its simplified Viper encoding is seen in Listing B.2. Note this is not the true encoding obtained from Prusti, but a simplified one in order to keep the examples shorter and less complex.

```
struct Dummy {
    inner: i32,
}
```

Listing B.1 [rust] declaration of the Dummy examples.

```
field inner: Int

predicate Dummy(self: Ref) {
  acc(self.inner, write)
}
```

Listing B.2 [viper] encoding of Dummy used in many Viper examples.

Appendix C

Full Encoding of Stateful `RefCell`

```
predicate RefCell(cell: Ref)

predicate CanBorrowMut(cell: Ref)

// data.inner > 0
predicate Positive(cell: Ref)

// data.inner < 0
predicate Negative(cell: Ref)

field inner: Int

predicate Dummy(self: Ref) {
  acc(self.inner, write)
}

function read(): Perm
  ensures result > none
  ensures result < write

method borrow_mut(cell: Ref) returns (data: Ref)
  requires acc(RefCell(cell), read()) &&
           acc(CanBorrowMut(cell), write)
  ensures  acc(RefCell(cell), read()) &&
           acc(Dummy(data), write) &&
           (acc(Dummy(data), write) --*
             acc(CanBorrowMut(cell), write))


method client(cell: Ref)
  requires acc(RefCell(cell), read()) &&
```

```
32              acc(CanBorrowMut(cell), write)
33    requires Positive(cell)
34    ensures  Negative(cell)
35  {
36    var dummy: Ref
37
38    // borrow_mut and invariant inhales
39    dummy := borrow_mut(cell)
40    unfold Dummy(dummy)
41    assume Positive(cell) ==> dummy.inner > 0
42    assume Negative(cell) ==> dummy.inner < 0
43
44    // ...
45
46    // can use rely
47    assert dummy.inner > 0
48
49    dummy.inner := -42
50
51
52    // release
53    exhale Positive(cell)    // state change
54    inhale Negative(cell)
55
56    // ensure state change is correct
57    assert (perm(Positive(cell)) > none) ==> dummy.inner > 0
58    assert (perm(Negative(cell)) > none) ==> dummy.inner < 0
59
60    fold Dummy(dummy)
61    apply acc(Dummy(dummy), write) --* acc(CanBorrowMut(cell), write)
62  }
```

Listing C.1 [viper] full encoding of a stateful RefCell with two states: Positive and Negative. The snippet shows the entire encoding of the abstract predicates, and a borrow-release with a state transition that is verified.

Appendix D

Encoding of `read()` function

```
1 function read(): Perm
2   ensures result > none
3   ensures result < write
```

Listing D.1 `[viper]` encoding of `read()` function used in many Viper examples.

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 Safe Clients of Unsafe Code and Trait Implementations in Rust

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

Name(s):	**First name(s):**
Beckmann	Jakob

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

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

Place, date	**Signature(s)**
Zürich, 27.09.2020	J. Beckmann

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