Project Description

Verifying Safe Clients of Unsafe Code and Trait Implementations in Rust

Jakob Beckmann
ETH Züirich
April 21, 2020

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

1 Introduction

Rust is a modern system programming language with a strong focus on speed, safety, and concurrency. One of its main features is that the compiler is able to automatically detect memory-related bugs such as dangling pointers [1]. This is performed using an ownership based type system, where each instance of a type has a unique direct owner.

Inbuilt mutability rules also allow Rust programmers to avoid problems related to unexpected side effects through aliased references. On a high level, the compiler enforces the existence of a unique reference if the reference allows for changes on the instance (mutable borrow), or enforces the immutability of all references (immutable borrows) at any point in the source code.

Another major feature of Rust are traits. They serve as interfaces defining shared behaviour across types, with potentially distinct implementation of the behaviour for each type implementing the trait. They are ubiquitous in Rust, reducing code duplication and providing better abtraction. Most importantly, traits are an enabler to use generics in a statically safe way.

The Rust type system enforced by the compiler is quite conservative. In fact, it can be too restrictive to perform a lot of work required by system software. The underlying computer hardware that system software

relies on, the operating system that needs to be directly interacted with, among other things, are inherintly unsafe. Mozilla introduced unsafe Rust to cope with these issues. Rust blocks or functions can be marked unsafe to loosen compiler checks and allow some illegal actions in safe Rust such as dereferencing raw pointers [2].

Unsafe Rust comes in several flavours, such as unsafe code blocks, unsafe functions, or unsafe traits. In all cases, marking something as unsafe declares the existence of contracts the compiler can't check, and that the programmer should ensure that these contracts are upheld. In this context, contracts refer to code safety, and not correctness.

A common practice in Rust is to isolate the usage of unsafe Rust as much as possible and provide safe external interfaces to the isolated code. This has the advantage to provide all guarantees of the Rust compiler in the entire program, with only the small separated unsafe code to be manually verified for functional correctness. Particularly, it allows to design safe clients for libraries making use of unsafe code internally. This practice is extensively seen in the Rust standard library, where many internals rely on unsafe code.

The high degree of guarantees provided by the Rust compiler for safe code can be leveraged to make formal verification of software simpler. More concretely, one can mostly focus on verifying functional properties while fully relying on the compiler's guarantees. This is the goal of Prusti [3].

Prusti provides a set of directives that can be added to Rust programs as annotations to specify desired functional properties. Prusti then uses the annotations to prove the correctness of the program. Internally this is achieved by translating the Rust program into the Viper intermediate verification language [4].

As unsafe code relaxes some guarantees of safe Rust code that Prusti relies upon, it's not supported. The main goal of this project is to enable sound verification of safe code using libraries that create safe abstractions over unsafe code. Moreover, support for usage of unsafe traits should be added to Prusti. This involves improving current support for traits and considering the implications of marking traits as unsafe.

2 Unsafe Code Problematics

One implication of unsafe Rust is that it allows for interior mutability. In other words, it is possible to change the contents pointed to by an immutable reference. In practice, this is used for various reasons such as reference counted variables, implementation details of logically immutable methods, and more. This is problematic for Prusti, as it

relies on immutable references to not be used to modify the content they point to. However, using a safe abstraction of unsafe code, this can't be garanteed in general.

For instance, the `std::cell::Cell` struct [5] is an example of a safe abstraction of an object that uses interior mutibility:

```
use std::cell::Cell;

fn main() {
    let cell = Cell::new(0);
    assert_eq!(cell.get(), 0);
    // here we pass an immutable reference
    foo(&cell);
    // PANIC: we assume the contents to not have changed
    assert_eq!(cell.get(), 0);
}

fn foo(cell: &Cell<i32>) {
    // modify the internal data of an immutable reference
    cell.set(5);
}
```

Listing 1: Interior Mutability

In listing 1, no code is marked as unsafe, as the abstraction is perfectly safe. However, due to usage of unsafe code in the implementation of `std::cell::Cell`, it's possible to change its contents by holding an immutable reference of the `Cell`. More specifically, the `set` method uses an immutable reference, but mutates the `Cell`. This makes reasoning about references fundamentally flawed, should one assume immutable references to never mutate. The main goal of this project is to find an approach to make the verification of code such as this listing possible.

Another major use case of unsafe Rust code are unsafe traits. A trait is declared as unsafe if implementors should read the trait's documentation to ensure their implementation maintains a contract the trait requires to run safely [6].

An common example of unsafe traits is the `std::iter::TrustedLen` trait [7]. This trait is a "Marker Trait" meaning it contains no implementation, but modifies the behavioural meaning of another trait (namely its supertrait). Concretely `std::iter::TrustedLen` modifies the meaning of the `size_hint` method of the `std::iter::Iterator` trait [8]. Should a developer implement this trait on a custom iterator without adhering to the contract specified in `std::iter::TrustedLen`'s documentation, it might result in undefined behaviour when using some functions on the iterator.

The challenge for Prusti in this case is threefold. First, Prusti only supports annotations on functions, methods, and structures, hence not allowing any functional specification on marker traits (traits without function declarations). This project aims at introducing a solution to this, potentially in the form of trait level annotations. Second, some form of specification inheritence needs to be considered for traits whose contract depends on their supertrait. In this case, `std::iter::TrustedLen` needs to be aware of `std::iter::Iterator`'s functional specification in order to specify restrictions on the `size_hint` method. Such trait contract refinement isn't supported by Prusti. Third, Prusti's specification language is not expressive enough to express some restrictions on implementations of unsafe traits. In the case of `std::iter::TrustedLen`, Prusti offers no way to refer to the contract of a method contained in another trait. Therefore, it would currently be impossible to change its contract, such as strengthening the postcondition in this case. This project also aims at solving this problem.

3 Core Goals

3.1 Investigation

The first task is to collect interesting examples of Rust programs that should be verified by Prusti as an outcome of the thesis. The challenge here is to find a set of examples that both reflects the usage of unsafe code across Rust projects, and contains enough variety to meaningfully evaluate the final result of the thesis.

3.2 Analysis

During the analysis, the reasoning about unsafe Rust code is described. This includes cases of unsafe code that can't be abstracted as fully safe Rust code. Listing 1 is an example of such a case. Unsafe traits exposed to client code such as `std::iter::TrustedLen` are another example of this.

The aim of this goal is to fully understand the problem and solutions to it. The outcome of the analysis should be a full theoretical understanding of the formal problems created by the set of examples collected during the investigation.

3.3 Design

This task aims at designing a technique solving the project's main vision using the theoretical understanding from the analysis step. The technique should allow to verify common cases of safe clients that use libraries implemented with unsafe code and trait implementations.

For the `TrustedLen` trait example, this would probably involve the developer defining the trait (library developer) to provide annotations on the trait itself in order to define the functional properties of the trait for Prusti. Such directives on traits are currently not supported by Prusti and would need to be added.

The challenges for this task is to maintain the "ease of use" ideology of Prusti. In particular, designing a solution that covers as many use cases as possible while not being invasive for the developer. Moreover, it will require to be familiar with Viper to determine how some predicates can be expressed in it.

3.4 Implementation

The prototype implementation of the designed technique.

This task requires to be familiar with the Prusti infrastructure.

3.5 Evaluation

Ideally using the examples from the investigation, evaluate the implemented solution. The set of examples from the investigation might get complemented with new ones as use cases emerge during analysis or design.

4 Extensions

4.1 Soundness for Unsupported Types

Many types are currently not supported by Prusti and potentially unsound. A possible extension goal would be to determine what types are unsound and modify their encoding in Prusti to force soundness in case they are not directly used. In other words, some types internally rely on unsupported types that are not exposed publicly and represent an implementation detail. The goal of this extension would be to modify the encoding of these unsupported "hidden" types to ensure the sound verification of clients making use of the wrapper type.

The challenge in this task is to get intimately familiar with type encodings in Prusti, and explore how modifications in said encoding can make it sound.

4.2 Clone Trait

The Rust compiler can derive trait implementations under certain conditions. An example of this is the `std::clone::Clone` trait [9]. An interesting goal would be to not only automatically infer the implementation of the trait, but also the corresponding contract that the trait

is defining. In order to investigate this approach, the extension goal would focus only on the Clone trait.

4.3 Library Annotations

In order to verify programs making use of libraries, Prusti typically requires a lot of glue code wrapping library functions with local functions just to provide Prusti annotations. This process is tedious and bad for code maintenance. The goal of this extension would be to define, design, and implement a prototype to provide contract annotations for external library functions without requiring glue code. For instance, this could be achieved by defining a semantic for an external file declaring all contracts for externally exposed functions of the library.

5 Schedule

The schedule presented below will be used for guidance. It assumes a start date towards end of March or beginning of April. The timespans provided are only approximate. The cumulative time is around 20 weeks, leaving ample time should some section be more work than expected.

Investigation	2 weeks
Analysis	3 weeks (5 cumulative)
Design	2 weeks (7 cumulative)
Implementation	3 weeks (10 cumulative)
Evaluation	1 week (11 cumulative)
Extensions	5 weeks (16 cumulative)
Writing Report and Presentation	4 weeks (20 cumulative)

References

1. Rust Programming Language. https://www.rust-lang.org/. [Online; accessed 23-March-2020].

2. Steve Klabnik and Carol Nichols with contributions from the Rust Community. *The Rust Programming Language*. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html, 1 edition. [Online; accessed 23-March-2020].

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

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

5. The Rust Standard Library. Struct std::cell::Cell. https://doc.rust-lang.org/std/cell/struct.Cell.html. [Online; accessed 23-March-2020].

6. Alexis Beingessner with contributions from the Rust Community. *The Rustonomicon*. https://doc.rust-lang.org/nomicon/index.html, 1 edition. [Online; accessed 23-March-2020].

7. The Rust Standard Library. Trait std::iter::TrustedLen. https://doc.rust-lang.org/std/iter/trait.TrustedLen.html. [Online; accessed 23-March-2020].

8. The Rust Standard Library. Trait std::iter::Iterator. https://doc.rust-lang.org/std/iter/trait.Iterator.html. [Online; accessed 23-March-2020].

9. The Rust Standard Library. Trait std::clone::Clone. https://doc.rust-lang.org/std/clone/trait.Clone.html. [Online; accessed 23-March-2020].