Annotating the Rust Standard Library with Specifications for Use in a Rust Verifier

Master's Thesis Project Description

Julian Dunskus
Supervised by Aurel Bílý, Prof. Dr. Peter Müller
Department of Computer Science, ETH Zürich
Zürich, Switzerland

July 2022

1 Introduction

Prusti [1] is a Rust [2] verifier developed by the Programming Methodology Group at ETH Zurich. It is based on the Viper [3] verification infrastructure: It translates input to a Viper AST and verifies that using Viper, then associates the results with the original Rust source. It can verify `assert!(...)` assertions in code as well as pre-/postconditions and invariants on functions (specified via Rust's attribute system).

In order for Prusti to understand a piece of code, it has to know how any functions called from it behave, which it can gather from annotations users place on their declared functions. However, real-world Rust code depends heavily on external methods and types that are not part of the user's codebase, so specifications for their behavior need to be available.

The most central such repository of external code is the Rust Standard Library, known as the `std` crate, providing essential features like optionals (`Option`), error handling (`Result`), and functional primitives (`map`, `filter`, etc.). Being able to verify programs using this functionality with minimal setup would lower the barrier to entry for using Prusti and make it easier to apply to existing codebases, proliferating verification of Rust code.

Currently, if users want to verify programs using this code, they must search for or write these specifications themselves. An example of such a specification for

(part of) `std::option::Option` follows—our goal is to eliminate the need to write or gather such code for the standard library.

```
extern crate prusti_contracts;
use prusti_contracts::*;

#[extern_spec]
impl<T> std::option::Option<T> {
    #[pure]
    #[ensures(matches!(*self, Some(_)) == result)]
    pub fn is_some(&self) -> bool;

    #[pure]
    #[ensures(self.is_some() == !result)]
    pub fn is_none(&self) -> bool;

    #[requires(self.is_some())]
    pub fn unwrap(self) -> T;

    // ...
}
```

2 Approach

2.1 Core Goals

1. Identify most-used parts of the standard library.
2. Gather specifications for the standard library.
3. Make these specifications easily accessible.
4. Evaluate coverage and/or utility of specifications.

The bulk of the work will be in writing these aforementioned specifications by hand. While some opportunities may arise to process documentation or unit tests for automatic extraction of specifications, we are not expecting to rely on that. Although comparatively small, the Rust Standard Library is still large enough that it would be very ambitious to cover 100%, so we will focus our efforts on the most important parts. In order to identify these parts and adequately prioritize our work, we will evaluate which methods and types are most used in real-world code, likely using Qrates [4] or some custom processing.

In addition to writing the specifications, we will also publish them: It should be made simple to use the specifications we gather in new codebases, e.g. by adding a crate we upload to `crates.io`. Specifications for any Rust library will be gathered in a central repository, versioned separately from any individual crate. If possible,

these specifications will handle multiple versions of a crate, otherwise this will remain as future work or require a workaround for now.

We will determine measures to quantify how much our specifications improve the end-user experience, e.g. coverage of methods or verifiability of tests in the standard library, perhaps combined with the frequency data discussed previously.

2.2 Extension Goals

1. Specify crates other than `std`.
2. Switching between abstractions/models depending on use case.
3. Verify some popular crates.
4. Verify stated specifications by running Prusti on their source code.

While the standard library is of course the most central crate, there are others that see wide usage and would be helpful to provide specifications for. Other popular crates that might benefit from specifications include `futures`, a powerful abstraction for asynchronous code, and `itertools`, which extends iterators with additional functionality.

Sometimes, there are multiple incomparable ways to abstract a mechanism, where different use cases require different abstractions to verify. Where it makes sense, we could ideally offer multiple models for users to switch between at will. On a crate level, this could work by enabling different dependency features when including our specifications crate, though finer-grained control is desirable. An example of such a case is array sorting, where one can either represent it as permuting the elements or as a multiset preserving its contents.

Currently, Prusti's test setup includes running it on the top 500 crates from crates.io, testing to make sure it does not crash. With the specifications we gather, we could perhaps start actually verifying some of these crates. This was previously infeasible because commonly-used basic types like `Option` or `Result` were treated as a black box.

Lastly, we can verify that our gathered specifications are valid by analyzing their source code with Prusti, ideally resulting in rigorous proof for some of them.

2.3 Related Work

There has been similar work done for other languages that provides valuable experience in how to approach these problems. In Python, users can add type annotations to their code to achieve some level of static typing, and there is a large repository

providing types for different packages called typeshed [5]. Similarly, TypeScript extends JavaScript with a type system, and there is a de-facto standard repository of type annotations for different libraries called Definitely Typed [6].

More specifically to Prusti, there has been work done on specifying iterators, a fundamental component of Rust programming. Not only will this be useful verbatim in providing specifications for the iterators in the standard library, but its models may also serve to inform our own approaches to modeling difficult types we encounter.

3 Schedule

This project will last 6 months, and the schedule is roughly planned as follows:

2 weeks	identify most-used parts of `std` crate
6 weeks	write specifications for (chosen parts of) `std`
2 weeks	publish gathered specifications
3 weeks	evaluate completeness/utility in real-world code
4 weeks	write report

Individual parts may overlap where it makes sense, e.g. when work on one part is blocked until other work outside our direct control completes. Note that the sum is less than 24 weeks: any additional time will be put towards either specifying more of the standard library or extension goal. There are also other miscellaneous tasks like preparing the three presentations.

References

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

[2] Rust Programming Language. `https://www.rust-lang.org`. Accessed 2022-07-18.

[3] 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.

[4] The qrates framework. `https://github.com/rust-corpus/qrates`. Accessed 2022-07-11.

[5] The typeshed repository. https://github.com/python/typeshed. Accessed 2022-07-18.

[6] Definitely Typed: TypeScript type definitions. https://github.com/DefinitelyTyped/DefinitelyTyped. Accessed 2022-07-18.