Specifying and Verifying Sequences and Array Algorithms in a Rust Verifier

Master Thesis Project Description

Johannes Schilling
Supervised by Prof. Dr. Peter Müller,
Aurel Bílý, Federico Poli
Department of Computer Science
ETH Zürich
Zürich, Switzerland

Co-Supervised by
Prof. Dr. Wolfgang Schröder-Preikschat,
Simon Schuster, Phillip Raffeck
Department of Computer Science
FAU Erlangen-Nuremberg
Erlangen, Germany

I. Introduction

Rust is a relatively recent systems programming language, originally developed by Mozilla. Among its features are fast execution speed and a strong, static type system, paired with a novel ownership tracking system to enforce memory and thread safety.

These strong guarantees greatly simplify verification efforts, as shown by Astrauskas et al. [1]. This works especially well for the basic foundation of safety and non-interference properties. For advanced, functional properties, further annotations for use with their verification tool Prusti, are required.

In this thesis we will build on these previous efforts and expand their work to support Rust's built-in sequence types and establish verification mechanisms for functional properties of Rust `trait`s, the interface specification mechanism in Rust. These building blocks will allow us to verify functional correctness of e.g. sort algorithms on arrays and slices.

A. Rust

While Rust's strong type system is used in a novel way towards implementing safety guarantees, it is also very expressive from a conventional type system view.

Interface Polymorphism with Traits. So-called *trait*s allow specifying common behaviors or APIs. Every type implementing a trait can be used similarly with respect to the interface that the trait describes. Rust uses traits – among many other uses – for specifying the behavior of operators like equality checking or ordering (`Eq`, `Ord` traits) or element access in vectors (`Index`, `IndexMut`). Implementing a trait for a user-defined type allows using the corresponding operator with instances of that user-defined type. That means that early during compilation, the compiler will translate various operators in a program to calls to the respective trait functions, e.g. `Ord::cmp` or `IndexMut::index_mut`. See listings 1 and 2 for an example of the translation.

```
1  if a > b {
2    v[0] = 13;
3  }
```

Listing 1: Original code

```
1  if a.cmp(b) == Ordering::Greater {
2    let tmp = v.index_mut(0);
3    *tmp = 13;
4  }
```

Listing 2: Operators unfolded into trait method calls

B. Prusti

The Prusti Verifier [1] allows verifying functional and safety properties of Rust programs.

Rust's model of static guarantees provides a very good foundation for verification. Where in other languages a lot of work goes towards proving basic safety and non-interference of different code parts before being able to even start verifying functional properties, for Rust a number of properties are already given by the language semantics.

Prusti extends Rust with a way to provide functional specifications of functions and also verify their validity. It works by encoding Rust programs and their specifications to Viper [2], an intermediate verification language and framework, which ultimately proves their validity.

A number of basic properties can be derived from the semantics that the Rust language gives to its constructs, such as non-aliasing of two `&mut` exclusive references.

While many constructs are already supported, the built-in Sequence types are not supported yet. To support them in Prusti, we must implement the translation of the Rust compiler's intermediate representation to corresponding Viper predicates.

Additional properties often need dedicated syntax and parsing thereof in addition to producing Viper encoding.

II. Core Goals

Part of this Master Thesis is to make a plan and collect usage examples beforehand, in order to have more input to the design of the solutions and to support the evaluation later.

Specifying and verifying sequences and sort algorithms exercises a number of functionalities that the Prusti verifier does not yet have, namely specifying predicates including quantification and implication, support for arrays and slices and verifying invariants on traits behavior.

A. Collecting Examples of Code using Slices, Sorting or relying on API Invariants

The first task is to collect examples of Rust code using the features we are interested in here – Rust slices or arrays and sorting them – and properties of traits that can't be ascertained using just the type system. These examples will be useful in test cases and for the final evaluation, and also to possibly catch missing corner cases in the design early.

One interesting case from the Rust standard library is the `Hash` trait, which specifies that if a type implements both `Eq` and `Hash`, then the implementation must make sure that instances comparing as equal under `Eq::eq()` must produce the same `Hash::hash()` value.

B. Adding Syntax for Predicate Functions

In order to specify properties that can be reused later e.g. when specifying transitivity of the `Ord` trait, we want to be able to specify predicates in a Rust function-like syntax, but using Prusti's extensions to Rust's syntax. This allows for example quantifications and implication. Predicates will be translated to Viper `function`s, which will never be executed at runtime. A possible syntax for these predicates is shown in listing 3.

```
#[predicate]
fn sorted() -> bool {
  forall(|i : usize| (0 < i < list.len()
      ==> list[i-1] <= list[i])
}
```

Listing 3: Transitivity Predicate

C. Adding Support for Arrays and Slices

Specification and Verification of code using Rust arrays or slices is not yet supported in Prusti. The task here will be to add Viper predicates such as `Array(self: Ref)` and `Slice(self: Ref)` and to add support for them throughout the whole verification pipeline.

We aim to support arrays and slices of primitive and generic types, simple operations (at least `get`, `set`, `len`), creation of arrays and creation of slices as windows into arrays or other slices.

D. Annotation and Verification of Properties of Rust Traits

Rust has two traits related to equality: `PartialEq` and `Eq`. The first one allows for partial equality, for types that do not have a full equivalence relation. For example, in floating point numbers `NaN != NaN`, so floating point types implement `PartialEq` but not `Eq`. The `PartialEq` trait requires that the relation defined is symmetric and transitive. The `Eq` trait defines no additional methods, but inherits `PartialEq` and additionally requires reflexivity as a property, making it an equivalence relation.

These properties can only be described in the documentation in Rust. In this thesis, we want to design and implement a way to

Specify the requirements that a trait has for its implementations

Verify that a trait implementation satisfies the requirements of the trait, i.e. upholds the guarantees that the trait makes.

While not set in stone, a possible syntax for this could be as shown in listing 4.

```
#[api_invariant(Eq::transitivity)]
trait Eq {
  #[pure]
  fn eq(a: &Self, b: &Self) -> bool;

  #[predicate]
  fn transitivity() -> bool {
    forall |a : Self, b: Self, c: Self| {
      (eq(a, b) && eq(b, c)) ==> eq(a, c)
    }
  }
}

struct Foo { ... }

impl Eq for Foo {
  fn eq(a: &Self, b: &Self) -> bool { ... }
}
```

Listing 4: API Invariant of the `Eq` trait

When verifying, Prusti must instantiate the API invariant for each type that implements the trait. For example, `Eq::transitivity` must hold for the `Foo::eq` implementation.

E. Evaluation

The last step before writing the thesis report is to evaluate the implementation. The samples collected earlier are a valuable resource. We will evaluate whether it is actually possible to apply the techniques implemented, and how well. Time taken to verify, occurrences of unexpected runtime errors and correctness of the results will be of interest. Verifying a simple sort algorithm should be possible, given all the building blocks described previously.

III. Extension Goals

A. Verifying Functional Correctness of the Rust Standard Library Sort Algorithm[1]

The sort algorithm used by the Rust standard library is a modified mergesort, borrowing some ideas from other algorithms. While the code uses `unsafe`, it does so only for performance optimizations, i.e. we can still verify a functionally equivalent version.

B. Specification and Verification of `Vec<T>`

Arrays and slices are the Rust language's built-in types for fixed-size sequences and variable-length views into such sequences, respectively. The type for dynamically self-expanding sequences (i.e. owning its backing memory) is `Vec<T>`.

This extension goal is about specifying `Vec<T>` so that its instances become more usable in verification as well. Some parts can already be specified using trusted wrappers and the `#[extern_spec]` mechanism, but that is limited and not as user-friendly.

This should be somewhat analogous once the built-in types work, but may contain additional pitfalls. Particularly, this extension goal is not about complete support of generic types.

C. Method for Importing external Prusti specifications

While this thesis focuses mainly on additions to Prusti to support more verification use cases, there might be a significant body of Rust standard library code that could be – at least partially – supported already without further mechanisms implemented in Prusti, but lacks some annotations.

While the `#[extern_spec]` mechanism itself already exists, its feature set is limited. In particular, existing annotations currently have to be copied to each file using them. To allow more of the Rust standard library or external crates to be verified, the `#[extern_spec]` mechanism could be extended, laying the foundations for a library of verification annotations.

D. Method for Specifying a Conversion of Rust Types to Viper Types

The Viper verifier has a builtin `Seq[T]` type of sequences with element type `T`. Rust has different types that conceptually represent a sequence of same-typed elements: both `[T]` (arrays and slices), and `Vec<T>` (called *vector*), as well as user-defined custom types. Similar conceptual equivalences could exist for other types like sets or multisets.

There is already work in progress on Ghost Type Encodings. What is missing is the concrete translation. This extension goal is about creating a mechanism to describe a translation from instances of concrete Rust types like `[T]` to Viper/Silver types like `Seq[T]`, so Viper's operations for those can be applied for the corresponding Rust types as well.

It would also be really useful to have conversions for Rust's sequence types to Viper's `multiset`, in order to specify that sorting doesn't change a sequence's members.

[1] Excluding the use of unsafe for performance reasons and other small changes

E. Pure Model Functions, Model Fields

This extension goal is about looking into a way to provide model functions and model fields on data structures by annotating them with `#[model]` attribute and making them available for specification and verification.

The example in listing 5 shows such an example. The actual interface for `Iterator<T>` has a method `next(&mut self) -> Option<T>` which returns `None` once no more elements are available, but notably does not allow random access to single elements. The model state does allow that, which might be useful in order to express invariants about the elements that are impossible or much more difficult to express otherwise.

```
1  trait Iterator<T> {
2    #[model]
3    #[pure]
4    fn at_index(idx: isize) -> T;
5  }
6
7  struct VecIterator<T> {
8    vals: &Vec<T>,
9  }
10
11 impl<T> Iterator<T> for VecIterator<T> {
12   #[model]
13   #[pure]
14   fn at_index(idx: isize) -> T {
15     self.vals[idx]
16   }
17 }
```

Listing 5: Ghost Code Example using Iterator

IV. Schedule

1 Week Core Goal: Collect Samples
2 Week Core Goal: Add Syntax for Predicate Functions
4 Weeks Core Goal: Add Support for Arrays and Slices
4 Weeks Core Goal: Annotation and Verification of Properties of Rust Traits
4 Weeks Extension Goals
1 Week Evaluation
4 Weeks Write Report, Prepare Final Presentation

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)*, vol. 3, no. OOPSLA. ACM, 2019, pp. 147:1–147:30.

[2] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62.