Verification of Rust Generics, Typestates, and Traits

Master Thesis Project Description

Matthias Erdin

Supervised by Prof. Dr. Peter Müller
Vytautas Astrauskas, Federico Poli

Department of Computer Science
ETH Zürich
Zürich, Switzerland

September 6, 2018

1 Introduction

Rust [1] is a systems programming language that aims to be a safe replacement for C and C++. The Rust features most relevant for this thesis are the *ownership* type system with *generic types* and *traits*. These concepts are briefly illustrated in this section.

Rust *generics* are similar to generics in other languages such as Java, Scala, and C#. Generics allow using type parameters as placeholders for some concrete type in the definition of functions, structs and other entities. Examples are `struct Vec<T>` as dynamic array of some type `T`, or `fn max<T: Ord>(v1: T, v2: T) -> T`, a function that returns the larger of two values of some *ordered* type `T`. As in Java, Scala, and C#, generics are checked once at the definition side. At compile time however, Rust *monomorphizes* generics for performance reasons, meaning that for each instantiation with a concrete type, the parameter is replaced with the concrete type, producing separate machine code for each separate concrete type.

In Rust, memory is managed through the ownership type system. This system enables Rust to make memory safety guarantees without needing a garbage collector. An important rule is that each memory location has a single owner. As a side effect of that, this system can be used to encode *typestates*, such that the compiler enforces that methods are called only in appropriate states. Consider the following example:

```rust
// socket states
struct Bound;
struct Listening;
struct Connected;
// ...

// base structure for any state
struct Socket<S> {
    fd: i32; // normal fields for socket type
    state: S; // empty state field, does not use memory
}

// functions that operate on sockets in bound state
impl Socket<Bound> {
    fn listen(mut self) -> Socket<Listening> {
        // ...
    }
}

// functions that operate on sockets in listening state
impl Socket<Listening> {
    fn accept(&mut self) -> Socket<Connected> {
        // ...
    }
}
```

Note the difference in the first parameters of the functions; `fn listen` *consumes*, i.e. takes ownership, of a socket in bound state, and returns a new socket in listening state. This is a common Rust idiom to express state change. On the other hand, `fn accept` expects a reference, and returns a new socket in connected state. Because it does not consume its argument, the passed socket can be said to remain in its original state.

The main motivation for using typestates as shown is to restrict method calls to those appropriate for the type's current state at compile time. Our goal is to find and develop convenient means to enable verification of such idiomatic code.

Rust *traits* are comparable to *interfaces* in languages such as C# and Java. Much like interfaces, traits primarily specify methods and their signatures that implementers must provide, which is checked by the compiler. Methods of traits are sometimes accompanied by pre- and postcondition in the documentation, which implementations are expected to uphold. We would like to be able to specify and verify such pre- and postconditions.

In this thesis, we aim to grow the set of Rust code that can be verified. To that end, the existing Rust frontend [2] for the Viper verification

infrastructure [3] is extended. These extensions should allow verification of structs, enums, generics, pre- and postconditions for trait methods and idiomatic Rust code with typestates. Further possibilities are finding more concise and expressive ways to deal with typestates that are not encoded into Rust types, and enabling verification of certain trait properties that are not captured by pre- and postconditions of methods.

2 Design Considerations

In this thesis we aim to consider the following design aspects:

2.1 Soundness

The developed extensions should be sound. This means in particular that successful verification implies that specifications then actually hold at runtime. Also, functions that were verified with generic type parameters should work with any concrete types allowed by the Rust type system.

2.2 Usability

Throughout the thesis, an important aspect is *usability*, meaning that the tools and techniques developed should fit well to existing idioms and real-world code that uses traits and typestates. In the end, using the developed methods should feel natural to the Rust programmer.

2.3 Modularity

Modularity refers to the ability to treat individual components separately, for the sake of decreased interdependence of such components and reduction of overall system complexity. Concretely, functions should, if possible, still be verifiable independently, as is currently the case with the Rust verification frontend. Client code should be verifiable against supplier *specifications*. Invariants that only refer to private fields should not need be exposed to functions and methods that cannot access these fields.

2.4 Expressiveness

Expressiveness refers the range of ideas that can be represented and communicated. Compactness is also relevant. Mechanisms developed should match well to common use cases, and allow expressing ideas concisely. Ideally, all common Rust idioms for expressing typestates are supported.

2.5 Extensibility

Extensibility refers to the degree to which future growth is taken into consideration by system design. In this thesis, the focus will be on extensibility with respect to new states for existing stateful types.

Because traits can be organized in hierarchies, and because there might be a use case for requiring not a single state, but a subset of states as precondition, there is some similarity to the problem of extensibility with respect to typestates and inheritance [4]. Rust does not feature inheritance, and it models aliasing in the type system directly, so not the same challenges apply. It is unclear to what degree extensibility of typestates themselves is needed and useful. Finding an answer to that question is part of this thesis.

3 Core Goals

3.1 Collect Samples

The first task is to collect Rust code samples that can be used for the following design and implementation tasks. Also, research papers about typestates are used as a source for samples. The challenge is to find good examples that use different kinds of entities, namely functions, structs, enums, generics, and trait methods that have pre- and postconditions. Ideally, samples cover a range of sophistication. These samples should be helpful in the following design and implementation steps.

While collecting samples, the basics and relevant terms and features of Rust are learned.

3.2 Support for Generics

Initially, support for *generics* in general must be implemented. The main challenge is to ensure that generics are verified upfront such that any instantiation that is legal according to the Rust type system upholds the specifications at runtime. Another likely challenge will be the handling of any unforeseen interactions with other language features. Attention is required with respect to the efficiency of the encoding.

This step also serves to get comfortable working with the existing infrastructure.

3.3 Invariants for Structs and Enums

As the next enhancement to the Rust verification frontend, the possibility to write invariants for structs and enums is implemented. In particular, handling public versus private invariants is looked at.

Writing meaningful invariants in contexts with type parameters should then also be possible. One idea is to allow type comparisons in specifications; it has to be investigated what the limitations of that approach would be.

The challenges are at least the following: The design should yield an encoding that is compact and works in the general case, especially in contexts with type parameters. It must be determined what specifications are allowed to express, e.g., whether a struct invariant is allowed to refer to content of private fields; such decisions must take into account how permissions can be handled in the Viper encoding. Also, it must be defined at which points invariants must hold, and means provided by which methods can indicate that at certain points of execution some invariants may only partially hold; and similar concerns [5].

3.4 Specifications for Trait Methods

The next step would be to attach pre- and postconditions to trait methods, against which implementations will be checked. This also applies to generics with trait bounds. One challenge is to define whether it useful that implementations can weaken precondition or strengthen postconditions, and what design best captures common use cases.

Note that this does *not* include any properties of traits that are not captured by method pre- and postconditions. Basic support for such properties would be covered with extension 4.3.

3.5 Invariants for Typestates Based on Rust Type System

One of the programming styles in Rust is encoding typestates with generics, like `Socket<T: SocketState>`. The main task here is generally to find and implement techniques to attach invariants to the common styles used in existing Rust code for modeling typestates.

The main challenges here are to figure out syntax and semantics that work for a wide range of common idioms, and that will be easy to use and feel natural to the Rust programmer. The focus clearly is on *usability*, but also on *extensibility* with respect to typestates.

4 Extensions

The following are possible extensions to the core goals. The aim is to complete 2 extensions.

4.1 Verification of Real-World Code

The tools developed could be applied to various real-world code. One possibility might be validating parts of the *rustls* [6] library that implements

TLS in Rust utilizing typestates.

4.2 Typestates via Specification Only

The main motivation for encoding typestates using specifications only is that existing code may use types that have states, but without having employed the idiom that uses Rust types to encode state. Such code can be enhanced with specification-only typestate information without breaking backwards compatibility.

Encoding typestates in the Rust type system is advantageous in that the Rust compiler checks and enforces states. It can be cumbersome however, because it complicates code structure and requires runtime operations that may or may not be fully optimized away. It is desirable that the Viper frontend for Rust allows to express typestates even if they are not reflected by types of the program. Such extensions must harmonize well with existing typestate idioms that use Rust types to encode state, such that both approaches are usable simultaneously.

The focus here should be on *expressiveness*. See the *Appendix* for an illustrative comparison of the two approaches.

4.3 Specifications for Intrinsic Properties of Traits

Support for trait method pre- and postconditions was established in core goal 3.4. This can be extended such that certain trait properties that are not captured by method pre- and postconditions can be verified also.

In Rust, traits are also commonly used to mark types with *intrinsic properties* [7] or to impose specific requirements. As an example, the Rust standard library provides the trait `PartialEq` [8]:

```
pub trait PartialEq<Rhs = Self>
{
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool { ... }
}
```

The library reference however also defines requirements expressed as invariants on this trait that implementations must obey, specified as follows [8]:

> *Formally, the equality must be (for all* `a`*,* `b` *and* `c`*):*
>
> - *symmetric:* `a == b` *implies* `b == a`*; and*
> - *transitive:* `a == b` *and* `b == c` *implies* `a == c`.

Moreover, the standard library provides a so-called *marker trait* `Eq` [9], derived from `PartialEq`, that does not define further methods at all, but imposes additional requirements only [9]:

> *[...] in addition to* `a == b` *and* `a != b` *being strict inverses, the equality must be (for all* `a`*,* `b` *and* `c`*):*
>
> - *reflexive:* `a == a`*;*
> - *symmetric:* `a == b` *implies* `b == a`*; and*
> - *transitive:* `a == b` *and* `b == c` *implies* `a == c`*.*

Note well that the invariants imposed by the `PartialEq` and `Eq` traits are *hyperproperties*. General support for verification of such hyperproperties is out of scope for this thesis.

We do not aim for a generic solution for verying intrinsic properties of traits, but to group use cases into classes and find solutions to some of these classes. Examples of such classes of properties would be:

- `PartialEq` and `Eq`, whose invariants are hyperproperties
- `Sync` and `Send`, which are related to concurrency and safety
- `Copy`, which is related to optimization and receives special treatment by the compiler.

5 Schedule

Calculating with 20 weeks (not counting holidays and pre-description):

1 week	Core Goal: Collect Samples
3 weeks	Core Goal: Support for Generics
2 weeks	Core Goal: Invariants for Structs and Enums
2 weeks	Core Goal: Specifications for Trait Methods
3 weeks	Core Goal: Invariants for Typestates [...]
4 weeks	Extensions (2)
5 weeks	Writing Report and Preparing Final Presentation

Considering holidays and variable work intensity during the semester yields the following expected times of completion:

2018-09-07	Core Goal: Collect Samples
2018-09-28	Core Goal: Support for Generics
2018-10-18	Core Goal: Invariants for Structs and Enums
2018-11-02	Core Goal: Specifications for Trait Methods
2018-11-30	Core Goal: Invariants for Typestates [...]
2019-01-10	Extensions (2)
2019-02-08	Writing Report and Preparing Final Presentation

References

[1] "The Rust programming language." `https://www.rust-lang.org/`. Accessed on 2018-08-31.

[2] "A Viper front-end for Rust." `https://github.com/viperproject/prusti`. Accessed on 2018-08-31.

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

[4] R. DeLine and M. Fähndrich, "Typestates for objects," in *ECOOP 2004 – Object-Oriented Programming* (M. Odersky, ed.), (Berlin, Heidelberg), pp. 465–490, Springer Berlin Heidelberg, 2004.

[5] A. J. Summers, S. Drossopoulou, and P. Müller, "The need for flexible object invariants," in *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2009.

[6] "A modern TLS library in Rust." `https://github.com/ctz/rustls`. Accessed on 2018-08-31.

[7] "The Rust standard library: Module std::marker." `https://doc.rust-lang.org/std/marker/index.html`. Accessed on 2018-08-31.

[8] "The Rust standard library: Trait std::cmp::PartialEq." `https://doc.rust-lang.org/std/cmp/trait.PartialEq.html`. Accessed on 2018-08-31.

[9] "The Rust standard library: Trait std::cmp::Eq." `https://doc.rust-lang.org/std/cmp/trait.Eq.html`. Accessed on 2018-08-31.

Appendix

This following code snippet uses the `Type<S: State>` idiom for encoding typestate via Rust types. Note the boilerplate in each state-changing method, and also note that the client must be explicit about state changes, in that after each state change a new variable must be used; albeit identically named in this example.

```
use std::marker::PhantomData;

struct New;
struct Bound;
struct Listening;

struct Socket<S> {
    fd: i32,
    state: PhantomData<S>,
}

impl<S> Socket<S> {
    fn print_fd(&self) {
        println!("{}", self.fd);
    }
}

impl Socket<New> {
    fn new() -> Self {
        Socket {
            fd: raw_socket(),
            state: PhantomData,
        }
    }

    fn bind(self) -> Socket<Bound> {
        raw_bind(self.fd);
        Socket {
            fd: self.fd,
            state: PhantomData,
        }
    }
}

impl Socket<Bound> {
    fn listen(self) -> Socket<Listening> {
```

```
        raw_listen(self.fd);
        Socket {
            fd: self.fd,
            state: PhantomData,
        }
    }
}

fn main() {
    let f = Socket::new();
    let f = f.bind();
    let f = f.listen();

    f.print_fd();
}
```

The following code snippet is the hypothetical variant that uses specification-only state information instead of the `Type<S: State>` idiom to encode state in the previous snippet.

```
#[states="new,bound,listening"]
struct Socket {
    fd: i32
}

impl Socket {
    fn print_fd(&self) {
        println!("{}", self.fd);
    }

    #[state="result(=>new)"]
    fn new() -> Self {
        Socket {
            fd: raw_socket(),
        }
    }

    #[state="self(new=>bound)"]
    fn bind(&mut self) -> {
        raw_bind(self.fd);
    }

    #[state="self(bound=>listening)"]
    fn listen(&mut self) {
```

```
25          raw_listen(self.fd);
26      }
27  }
28
29  fn main() {
30      let f = Socket::new();
31      f.bind();
32      f.listen();
33
34      f.print_fd();
35  }
```