Rust & Typestates

January 9, 2021

José Duarte & António Ravara NOVA School of Science & Technology, Portugal

Introduction

Systems programming is one of the most demanding domains of computer science where bugs and their respective consequences are costly to both service providers and consumers. Languages like C/C++ have dominated the systems programming landscape for years and one of the main problems of both is the lack of memory management, leaving such responsibility to the developer, which in turn has proven to be *a less than ideal* solution [1] [2].

To address such problem, several tools and languages have been and continue to be developed, so far, Rust has been the only one to achieve *mainstream* status. Rust aims to provide memory safety without affecting performance or productivity, to achieve such ambitious goal, Rust validates code with the borrow checker, which then enforces memory safety rules.

Why Rust? One of Rust's core values is safety, such value manifests itself in the form of memory safety, and the provision of tools to prevent concurrency problems. The mindset is also imbued in the community, an example would be Sealed Rust[3], an effort to bring Rust to the safety critical domain.

While Rust addresses memory safety, not all bugs are memory-related, when implementing a protocol, the developer has its specification at hand, yet, bugs still arise due to human error, ideally we ought to have mechanisms that validate our code against some kind of specification. Automata is a powerful tool to materialize the relationships between states and operations, as shown by Figure 1.

Typestates enable the programmer to track the state of an object throughout its lifetime with help of the type system and while Rust has no direct support for typestates, its type system is powerful enough to allow them to be implemented within it, effectively blurring the line between specification and implementation by baking the possible state transitions in the type system.

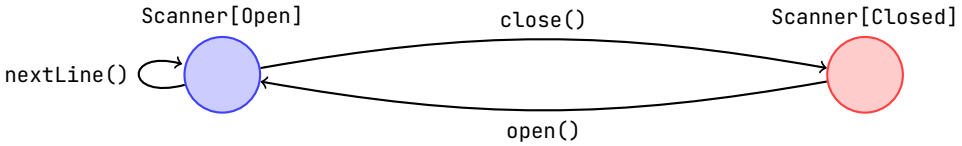

Figure 1: The `Scanner` typestate state machine.

What are typestates? In a nutshell, typestates can be thought of a way to constrain APIs as the program state evolves. More formally, typestates belong to the behavioral types category and are built on the idea of lifting state to the type level, since state becomes part of the type system, the compiler will be able to reason about state, effectively helping the developer track state and validate certain assumptions.

How are typestates useful? Diving deeper on how do typestates help the developer, I provide a simple yet classic example. Consider a stream, whether it be a file or a socket, to be read, the stream must first be open, only then can it be read and finally it must be closed.

```
1 Scanner s = new Scanner(System.in); // open the stream
2 s.nextLine();                       // read
3 s.close();                          // close the stream
4 s.nextLine();                       // IllegalStateException
```

In the example above the developer tries to read a line after closing the stream, when ran, the example yields a `IllegalStateException` since you cannot read from a closed source.

The fact that this code compiles without warnings (even when `-Xlint:all` is used) is problematic, since the error can only be caught at runtime. While the presented example is simple, production-code is not, and code paths that raise runtime errors may be untested until it reaches the hands of the user.

Using typestates solves the above problem by establishing a distinction between the open and closed `Scanner`, consider the following example:

[1] https://git.io/JLdDc
[2] https://www.chromium.org/Home/chromium-security/memory-safety
[3] https://ferrous-systems.com/blog/sealed-rust-the-plan/

```
Scanner[Open] s = new Scanner(System.in);   // open the stream
s.nextLine();                               // read
Scanner[Closed] s = s.close();              // close the stream
s.nextLine();                               // compile-time error
```

The compiler is now able to provide the developer with an error at compile-time since it now knows that the `Scanner` is closed and thus, it does not have a function `nextLine`.

Alias Control Besides the pro-safety mentality of Rust, another key detail which makes Rust a solid candidate for typestates is its alias control.

By definition, typestates are incompatible with aliasing, this is due to the fact that if an object is being used by N clients, when a client mutates an object, all other clients guarantees are broken. Back to the stream example, if a client closes the stream, all other clients may crash since they may try to read from a stream which is now closed.

While Rust cannot enforce a truly linear typesystem (in which objects must be used *exactly once*), it is able to enforce an affine type system (or *at most once* usage), as demonstrated bellow.

```
let x = 0
let y = x;
println!("{}", x); // error, value moved in line 2
```

Such type system allows us to emulate typestates by forcing moves whenever a state transition is required to happen, this way, the previous state is "*destroyed*" and cannot be mutated, risking the typestate guarantees.

Typestating Rust

Rust previously had a typestate system which was removed in version 0.4 since it "did not pull its own weight". The idea I propose does is not aimed at bringing back the old typestate system but rather at leveraging Rust's type system to allow for typestated objects. In short, I propose a DSL built with procedural macros which takes advantage of Rust's generics and affine type system capabilities.

The Typestate Pattern As of now, it is possible for a developer to make use of typestates in Rust, the key idea is to write functions which consume the current object and return a new object, the function signature should be similar to `fn transition(self: OldState, args...) → NewState`.

Each state can then carry information, or not, being a simple 0-sized structure. States can be grouped into sets using traits, these sets can be further restricted using the sealed trait pattern[4]. An example implementation of the typestate pattern is available in https://github.com/rustype/http-parser.

Proposal The code necessary to implement the typestate pattern requires a lot of boilerplate, furthermore, we want to be able to prove typestate properties (e.g. that all states are productive). The former challenge can be solved through the use of macros, and the latter is not novel in the literature, to address it we just need to extract the state machine from the code and apply existing algorithms.

However, to do so without writing a static checker from the ground up we are required to use Rust's procedural function-like macros, these enable the creation of a DSL for typestates, thus solving both the boilerplate problem and the requirement for the ability to prove certain properties.

Existing work on the Rust ecosystem regarding typestate-like mechanisms is limited, we've reviewed several automata crates and only one provided strong compile-time guarantees, such crate was the `state_machine_future` crate, the crate, however, requires a runtime such as Tokio, due to being centered around asynchronous computations.

Our approach aims to extend the `state_machine_future` capabilities with an expressive DSL and the possibility to be used without a runtime. Currently, a prototype is under development and can be found in https://github.com/rustype/typestate-rs.

[4] https://git.io/JLbry