

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 good idea*¹².

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

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

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.

Why are typestates useful? Diving deeper on why 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, this 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, code running in production 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 Scanner[Open] s = new Scanner(System.in); // open the stream
2 s.nextLine();                             // read
3 Scanner[Closed] s = s.close();             // close the stream
4 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`.

¹<https://git.io/JLdDc>

²<https://www.chromium.org/Home/chromium-security/memory-safety>

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 Rust³, an effort to bring Rust to the safety critical domain.

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

By definition, tpestates 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.

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

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

Typestating Rust

Rust previously had a tpestate 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 tpestate system but rather at leveraging Rust’s type system to allow for tpestated objects. In short, I propose a DSL built with procedural macros which takes advantage of Rust’s generics and affine type system capabilities.

The Tpestate Pattern As of now, it is possible for a developer to make use of tpestates 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 regarding the current computation, or not, and simply be a 0-sized structure. States can be grouped into sets using traits, these sets can be further restricted using the sealed trait pattern⁴. An example implementation of the tpestate pattern is available in <https://github.com/rustype/http-parser>.

Proposal The code necessary to implement the tpestate pattern requires a lot of boilerplate, furthermore, we want to be able to prove tpestate 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 tpestates, thus solving both the boilerplate problem and the requirement for the ability to prove certain properties.

Currently, a prototype is under development, the development of such prototype falls under the scope of my MSc thesis, which is centered around the topic of tpestates and Rust. The prototype can be found in <https://github.com/rustype/tpestate-rs>.

```
1 tpestate! {
2     struct Drone {x: f32, y: f32}
3     fn ping_coords(&self) → (f32, f32);
4     state Idle [last_landing: Time] {
5         transition take_off(self) → Hovering;
6     }
7     state Hovering {
8         fn take_picture(&self, dst: &str);
9         transition land(self) → Idle;
```

³<https://ferrous-systems.com/blog/sealed-rust-the-plan/>

⁴<https://rust-lang.github.io/api-guidelines/future-proofing.html#sealed-traits-protect-against-downstream-implementations-c-sealed>

10

}

11

}