Typestates & Rust

José Duarte

28th March 2021

FCT NOVA

What Are Typestates?

Typestates capture aspects of an object's state. Approaches to typestates can be:

- **State-Based Designs** - checking state during runtime.
- **Typestate Checkers** - static checkers that verify code for correct object usage.
- **Language-Based** - design a language which supports typestates as first-class citizens.

Why Are Typestates Useful?

Consider the `File` example:

```
Scanner s = new Scanner(System.in);
s.nextLine();   // ok
s.close();      // ok
s.nextLine();   // IllegalStateException
```

Why Are Typestates Useful?

What we really want:

```
| s.nextLine();
| ^^^^^^^^^^^^
= error: `s` was closed in line 3
```

Why Are Typestates Useful?

To achieve it, typestates lift state to the type level:

```
Scanner[Open] s = new Scanner(System.in);
s.nextLine();
Scanner[Closed] s = s.close();
s.nextLine();
```

The last line now fails to typecheck since `Scanner[Closed]` does not have function `nextLine`.

Typestates & Rust

Typestates are not new to Rust.

- There are several blog posts on the subject.
- The Embedded Rust Book makes use of them to model peripherals.
- The `state_machine_future` crate leverages typestates to provide type safe `Future`s.

Typestates & Rust - Problems

- Typestates require a lot of boilerplate.
- Existing crates either do not provide compile-time errors or are limited in scope.

Typestates & Rust - Boilerplate

```
    boilerplate demo
```

How Do We Solve It?

Macros! They are designed to write the boilerplate for us!

The solution

Currently, we have implemented a small proof of concept which allows a user to define a typestated structure as:

```
typestate!(
    strict Drone [Idle, Hovering, Flying] {
        x: f32,
        y: f32
    }
);
```

The Solution

The previous code will generate:

- A `Drone` structure.
- The state structures `Idle, Hovering, Flying`.
- Some relevant `trait`s and `impl`s.

Limitations & Moving Forward

`typestate!` is implemented with `macro_rules`, this has the downsides of being limited and barely readable.

Ideally we want to write a full-fledged DSL, reducing the amount of code the developer has to write.

A Peek Into The Future

```
ts! {
    struct Drone { x: f32, y: f32 }

    fn ping_coordinates(&self) -> (f32, f32);

    state Idle [last_landing: Time] {
        transition take_off(self) -> Hovering;
    }

    state Hovering {
        fn take_picture(&self, dst: &str);
        transition land(self) -> Idle;
    }
}
```