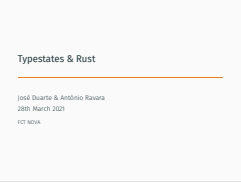
Typestates & Rust

José Duarte & António Ravara

28th March 2021

FCT NOVA



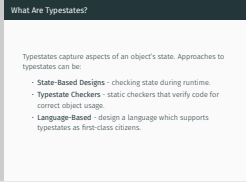
Hello everyone! My name is José Duarte and today I will be talking about using typestates in Rust. I'll present:

- A brief definition of typestates.
- Why they are useful.
- And finally I'll discuss their relationship with Rust and my proposal to integrate them in the ecosystem.

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.



The aim of typestates is to use the type system to capture aspects of an object state, essentially lifting state to the type level.

There are several approaches to typestates, the one I'll be presenting resides between typestate checkers and a "typestatable" language.

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?
Consider the File example:
Scanner s = new Scanner(System.in);
s.nextLine();   // ok
s.close();   // ok
s.nextLine();   // IllegalStateException

The first concern many may have is ”Why should I care?”, lets start by introducing a simple example. Consider the code on the slide, the developer creates the scanner, reads the line and closes, the developer then tries to read a line again causing an `IllegalStateException`. This should be avoidable without effort from the programmer, its 2021, we want the computer to do the job for us.

Why Are Typestates Useful?

What we really want:

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


Why Are Typestates Useful?
What we really want:
| s.nextLine();
| ^^^^^^^^^^^^
= error: `s` was closed in line 3

Ideally we want something like shown, a compile time error telling the programmer: *You closed the stream dummy!*

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`.

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.

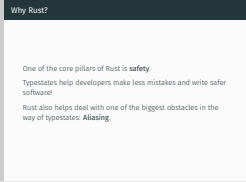
Enter typestates! By lifting state to the type level, the open stream is different from the closed one! The closed stream interface should be different from the open one and thus, we will get the error in the previous slide!

Why Rust?

One of the core pillars of Rust is **safety**.

Typestates help developers make less mistakes and write safer software!

Rust also helps deal with one of the biggest obstacles in the way of typestates: **Aliasing**.



One of Rust's main focus is safety, it provides mechanisms for memory and concurrency, so why not development safety?
Typestates aid the developer in the process of writing bug-free code!
Another key point in favor of Rust resides in its aliasing control, disallowing multiple mutable borrows enables typestates!

Weren’t Typestates Removed Rust?

Rust *used to* have a form of typestate, manifested as *pre* and *post* conditions. It was removed due to not pulling its weight.

However! In the words of Rust’s author:

That said, I really want to encourage use of typestate, not just because ”it’s a cool word / concept” but rather because it’s really assert in slightly more formal attire, and I love assert in my code.

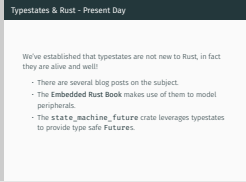
Weren’t Typestates Removed Rust?
Rust used to have a form of typestate, manifested as pre and post conditions. It was removed due to not pulling its weight.
However! In the words of Rust’s author:
That said, I really want to encourage use of typestate, not just because ”it’s a cool word / concept” but rather because it’s really assert in slightly more formal attire, and I love assert in my code.

As some may know, typestates were a feature of pre-1.0 Rust.
They were taken down since their cost-benefit ratio was deemed not worth it. However, Rust’s author expressed support for typestates!

Typestates & Rust - Present Day

We've established that typestates are not new to Rust, in fact they are alive and well!

- 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.



While they were removed, typestates live on in the form of blog posts, references in Rust books and crates!

Typestates & Rust - Intuition

We are able to model typestates with the Rust typesystem.

The key intuition is: Take ownership and return the new state.

```
impl OldState {
    fn transition(self) -> NewState {...}
}
```


Typestates & Rust - Intuition
We are able to model typestates with the Rust typesystem.
The key intuition is: Take ownership and return the new state.
impl OldState {
fn transition(self) -> NewState {...}
}

Currently we are able to model typestates within Rust, dealing with aliasing is simple, we take ownership of the old state, transition to the new one and return!

Typestates & Rust - Advantages

- The compiler helps you track state (compiler knows the types + types express state ⇒ compiler knows the state).
- You can statically restrict functionality based on state, instead of hoping you don't forget a runtime check.

Typestates & Rust - Advantages
- The compiler helps you track state (compiler knows the types + types express state ⇒ compiler knows the state).
- You can statically restrict functionality based on state, instead of hoping you don't forget a runtime check.

This approach has some advantages:

- Since state is lifted to the type level, the compiler will aid you tracking your application's state.
- State is reflected on APIs and thus, an API will change as its state changes, restricting impossible function calls, relieving the programmer from being required to remember an `if` to ensure the application doesn't panic.

Typestates & Rust - Problems

- Typestates require a lot of boilerplate.
- Collections do not work that well with typestates.
- Existing crates either do not provide compile-time errors or are limited in scope.

Typestates & Rust - Problems
- Typestates require a lot of boilerplate.
- Collections do not work that well with typestates.
- Existing crates either do not provide compile-time errors or are limited in scope.

However, typestates are not without problems!

- They require a lot of boilerplate.
- Existing collections do not work with typestates.
- Crates that provide close mechanisms do not provide compile-time errors or are limited in scope.

Typestates & Rust - Boilerplate

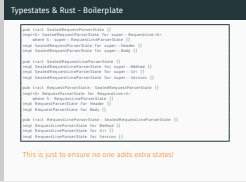
```
pub trait SealedRequestParserState {}
impl<S> SealedRequestParserState for super::RequestLine<S>
    where S: super::RequestLineParserState {}
impl SealedRequestParserState for super::Header {}
impl SealedRequestParserState for super::Body {}

pub trait SealedRequestLineParserState {}
impl SealedRequestLineParserState for super::Method {}
impl SealedRequestLineParserState for super::Uri {}
impl SealedRequestLineParserState for super::Version {}

pub trait RequestParserState: SealedRequestParserState {}
impl<S> RequestParserState for RequestLine<S>
    where S: RequestLineParserState {}
impl RequestParserState for Header {}
impl RequestParserState for Body {}

pub trait RequestLineParserState: SealedRequestLineParserState {}
impl RequestLineParserState for Method {}
impl RequestLineParserState for Uri {}
impl RequestLineParserState for Version {}
```

This is just to ensure no one adds extra states!



Typestates & Rust - Boilerplate

```
let parser = HttpRequestParser
    ::<RequestLine<Method>>::start(packet);
let parser = parser.parse()?;
let parser = parser.parse()?;
let parser = parser.parse()?;
let parser = parser.parse()?;
let request = parser.parse();
println!("{:#?}", request);
```

You also need to re-declare the variable as state evolves.

Typestates & Rust - Boilerplate
let parser = HttpRequestParser
::<RequestLine<Method>>::start(packet);
let parser = parser.parse()?;
let parser = parser.parse()?;
let parser = parser.parse()?;
let parser = parser.parse()?;
let request = parser.parse();
println!("{:#?}", request);
You also need to re-declare the variable as state evolves.

How Do We Solve It?

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

How Do We Solve It?
Macros! They are designed to write the boilerplate for us!

To solve the boilerplate problem we can use macros! That's what they were built for!

Rust's macros allow us to write a DSL, which we can then "compile" back into Rust.

The Solution

Currently, I 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
Currently, I 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
}
);

I've implemented a small PoC using `macro_rules`, it is severely limited but already takes care of some boilerplate!

The Solution

The previous code will generate:

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

The Solution
The previous code will generate:
- A Drone<State> structure.
- The state structures Idle, Hovering, Flying.
- Some relevant traits and impls.

The previous snippet will provide you with a Drone structure, generic over state, the state structures and some `trait`s and `impl`s to implement the sealed ”hierarchy”.

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.

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.

The current implementation is severely limited and barely readable, ideally, the final product will be a complete DSL, which allows the programmer to specify a typestates and possible transitions, the programmer will then only be required to add the `impl`s.

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;
    }
}
```


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;
}
}

Our first draft looks as follows.

`struct`s are declared as regular Rust structures, functions available to every state are declared freely, states and their relationship are declared as follows, the previous state is declared in square brackets, the next is declared with the `transition` keyword and a function signature.

The syntax is not fixed, as a matter of fact nothing is and feedback would be much appreciated.

More Information

You can track progress and provide feedback at `github.com/rustype/typestate-rs`.

Code for slides 12 & 13 can be found at `github.com/rustype/http-parser`.

More Information

You can track progress and provide feedback at `github.com/rustype/typestate-rs`.

Code for slides 12 & 13 can be found at `github.com/rustype/http-parser`.