Bridging the gap between Typestates and Rust in production code

José Duarte
António Ravara (Advisor)

March 2021

NOVA School of Science and Technology

Bridging the gap between Typestates and Rust in production code
José Duarte
António Ravara (Advisor)
March 2021
NOVA School of Science and Technology

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.

Outline

Introduction

State of the Art

Case Study

Plan

Outline
Introduction
State of the Art
Case Study
Plan

Durante a apresentação irei introduzir o tema, rever sumáriamente o estado da arte, apresentar a proposta de trabalho e por fim rever o plano de trabalho da mesma.

Outline

Outline

Introduction

Context

Problem

Objectives

State of the Art

Case Study

Plan

Context

Software plays a crucial role in our lives.

- From web browsers, to word processors and more!

As software becomes more important, bugs become more expensive.

- Losing work due to a bug in the save procedure is not nice.
- A bug in the firmware for a pacemaker may cost a life.

Context

Software plays a crucial role in our lives.

- From web browsers, to word processors and more!

As software becomes more important, bugs become more expensive.

- Losing work due to a bug in the save procedure is not nice.
- A bug in the firmware for a pacemaker may cost a life.

Problem

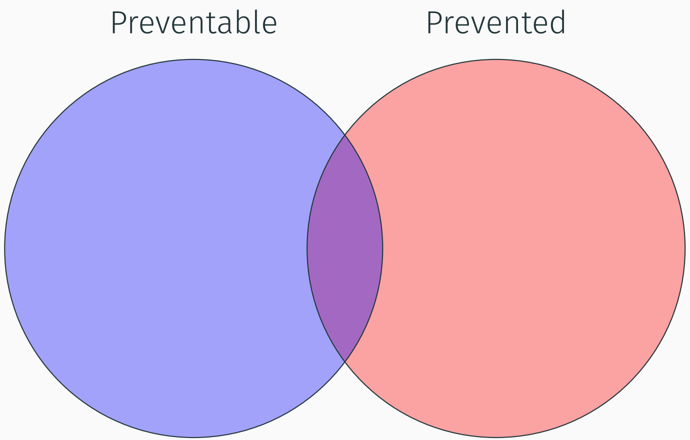
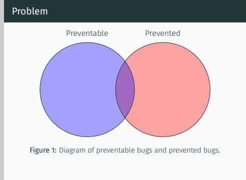


Figure 1: Diagram of preventable bugs and prevented bugs.



Problem - with Rust

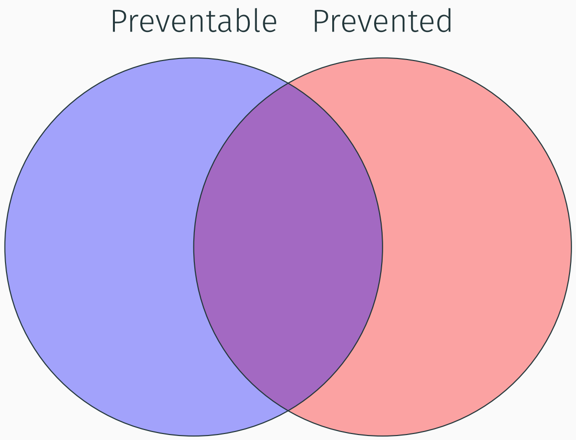
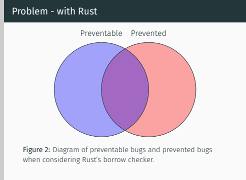


Figure 2: Diagram of preventable bugs and prevented bugs when considering Rust’s borrow checker.



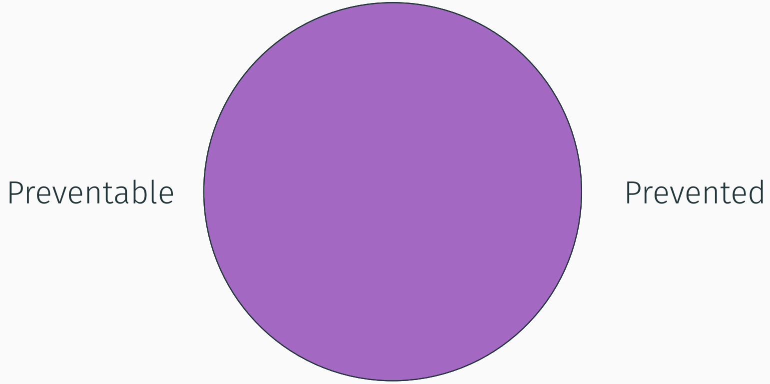


Figure 3: The ideal diagram of preventable bugs and prevented bugs, where all bugs are prevented.

Problem - Ideal
Preventable
Prevented
Figure 3: The ideal diagram of preventable bugs and prevented bugs, where all bugs are prevented.

Objectives

A library which brings *practical* typestates to Rust.

- Minimal learning overhead.
- Zero-cost abstraction.
- Scalable to large projects.

Objectives

A library which brings *practical* typestates to Rust.

- Minimal learning overhead.
- Zero-cost abstraction.
- Scalable to large projects.

Outline

Outline
Introduction
State of the Art
Session Types
Typestates
Case Study
Plan

Session Types

Session Types

Typestates

Bridging the gap between Typestates and Rust in production code
└─State of the Art
└─Typestates
└─Typestates

Outline

Outline

Introduction

State of the Art

Case Study

Problem

Solution

Approach

Workflow

Plan

Problem

Error happens at runtime, possibly crashing the program.

```rust
fn main() {
    let protocol = Protocol::new();
    protocol.step1();
    protocol.step3(); // runtime error
    protocol.step2();
}
```

Our tools should work for us, not make us work for them.

Problem

Error happens at runtime, possibly crashing the program.

```rust
fn main() {
    let protocol = Protocol::new();
    protocol.step1();
    protocol.step3(); // runtime error
    protocol.step2();
}
```

Our tools should work for us, not make us work for them.

Solution

Ideally, we want to catch the error at compile-time.

```
fn main() {
    let protocol = Protocol::new();
    protocol.step1();
    protocol.step3();
            ^^^^^^^
            | error: cannot call `step3`
    protocol.step2();
}
```


Solution

Ideally, we want to catch the error at compile-time.

```
fn main() {
    let protocol = Protocol::new();
    protocol.step1();
    protocol.step3();
            ^^^^^^^
            | error: cannot call `step3`
    protocol.step2();
}
```

Approach - Overview

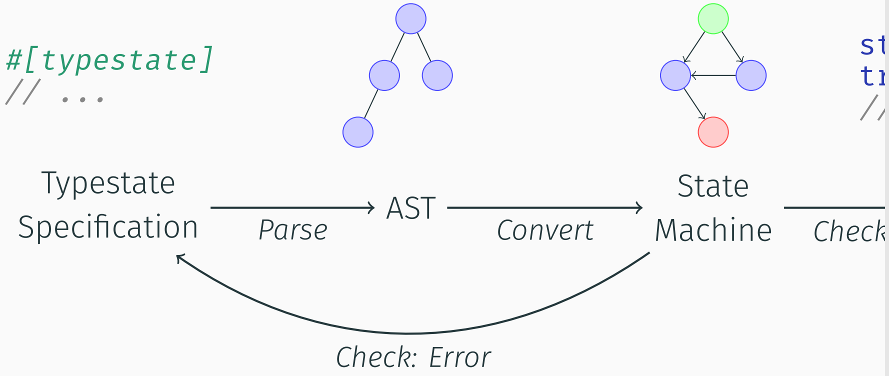
We can exploit the Rust typesystem to emulate typestates, however this approach requires boilerplate.

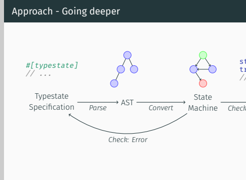
Use macros!

- Integral part of the language, requiring no new experience.
- Able to throw errors during compile-time.
- Rewrite the annotated code, generating boilerplate for the user.

Approach - Overview
We can exploit the Rust typesystem to emulate typestates, however this approach requires boilerplate.
Use macros!
- Integral part of the language, requiring no new experience.
- Able to throw errors during compile-time.
- Rewrite the annotated code, generating boilerplate for the user.

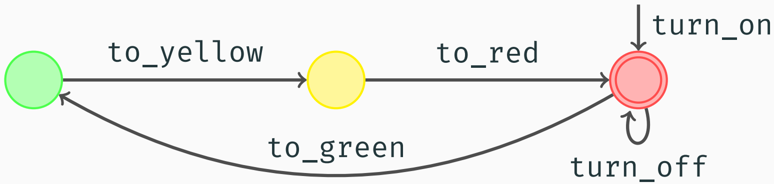
Approach - Going deeper

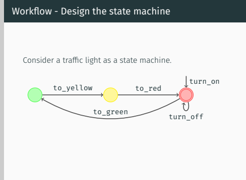




Workflow - Design the state machine

Consider a traffic light as a state machine.





Workflow - Declaring state in Rust

Using the DSL we first declare the module, main automata and the states:

```rust
#[typestate] mod traffic_light {
    #[automata] struct TrafficLight;
    #[state] struct Green;
    #[state] struct Yellow;
    #[state] struct Red;
    // ...
```

We still need transitions!

Workflow - Declaring state in Rust

Using the DSL we first declare the module, main automata and the states:

```rust
#[typestate] mod traffic_light {
    #[automata] struct TrafficLight;
    #[state] struct Green;
    #[state] struct Yellow;
    #[state] struct Red;
    // ...
```

We still need transitions!

Workflow - Declaring transitions in Rust

All transition functions take ownership of the current state and return the new state.

```rust
    // code from the previous slide ...
    fn to_yellow(self: Green) -> Yellow;
    fn to_red(self: Yellow) -> Red;
    fn to_green(self: Red) -> Green;
    // ...
```

Finally, we need *start* and *end* states.

Workflow - Declaring transitions in Rust

All transition functions take ownership of the current state and return the new state.

```rust
    // code from the previous slide ...
    fn to_yellow(self: Green) -> Yellow;
    fn to_red(self: Yellow) -> Red;
    fn to_green(self: Red) -> Green;
    // ...
```

Finally, we need *start* and *end* states.

Workflow - Declaring *start* and *end* states in Rust

Functions that do not use **self** and *return* a valid state are inferred as the *start* state.

Functions that take **self** and do *not return* a valid state are inferred as the *end* state.

```
    // code from the previous slides ...
    fn turn_on() -> Red;
    fn turn_off(self: Red);
}
```

Our traffic light is ready!

Workflow - Declaring *start* and *end* states in Rust

Functions that do not use **self** and *return* a valid state are inferred as the *start* state.

Functions that take **self** and do *not return* a valid state are inferred as the *end* state.

```
    // code from the previous slides ...
    fn turn_on() -> Red;
    fn turn_off(self: Red);
}
```

Our traffic light is ready!

Workflow - The other features

There are still other important features left. For maintenance purposed, consider that our traffic light is now required to count every **GYR** cycle.

The `TrafficLight` structure is now declared as follows:

```
#[automata] struct TrafficLight { cycles:u64 }
```


Workflow - The other features

There are still other important features left. For maintenance purposed, consider that our traffic light is now required to count every **GYR** cycle.

The `TrafficLight` structure is now declared as follows:

#[automata] struct TrafficLight { cycles:u64 }

Workflow - The other features

How can we check if the light requires maintenance?

We add a **pure** function:

```
fn requires_maintenance(
    &self: TrafficLight
) -> bool;
```

This function is not able to perform mutations due to the immutable reference (&).

Workflow - The other features

How can we check if the light requires maintenance?

We add a **pure** function:

```
fn requires_maintenance(
    &self: TrafficLight
) -> bool;
```

This function is not able to perform mutations due to the immutable reference (&).

Workflow - The other features

How can we check if the light requires maintenance?

We add a **pure** function:

```
fn requires_maintenance(
    self: &TrafficLight
) -> bool;
```

This function is not able to perform mutations due to the immutable reference (&).

Workflow - The other features

How can we check if the light requires maintenance?

We add a **pure** function:

```
fn requires_maintenance(
    self: &TrafficLight
) -> bool;
```

This function is not able to perform mutations due to the immutable reference (&).

Workflow - The other features

After maintenance, how can we reset the counter?

We add an **`impure`** function:

```rust
fn reset_counter(self: &mut TrafficLight);
```

The function is able to perform mutations, due to usage of a mutable reference (`&mut`), but it is unable to transition between states.

Workflow - The other features

After maintenance, how can we reset the counter?

We add an **`impure`** function:

```rust
fn reset_counter(self: &mut TrafficLight);
```

The function is able to perform mutations, due to usage of a mutable reference (`&mut`), but it is unable to transition between states.

Outline

Outline
Introduction
State of the Art
Case Study
Plan

Plan Overview

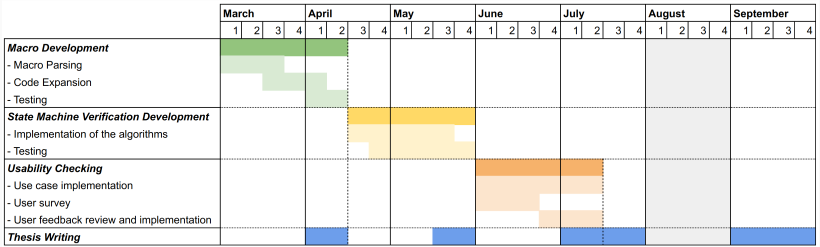


Figure 4: Work plan Gantt chart