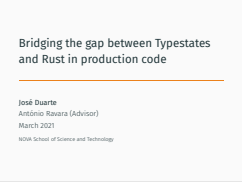
Bridging the gap between Typestates and Rust in production code

José Duarte
António Ravara (Advisor)
March 2021

NOVA School of Science and Technology

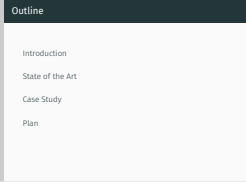
2021-03-02 Bridging the gap between Typestates and Rust in production code



Bom dia Professor(es), sou o José Duarte, e o meu tema de tese é ”Bridging the gap between typestates and Rust in production code”.

O tema é orientado pelo Professor António Ravara.

Outline



Durante a apresentação irei:

- Introduzir o tema.
- Rever sumariamente o estado da arte.
- Rever um caso de estudo simples, apresentando assim o projeto.
- Por fim, falar do plano de trabalho para o semestre.

Outline

Outline

Introduction
Context
Problem
Objectives
What are typestates?

State of the Art

Case Study

Plan

Relativamente à introdução irei:

- Contextualizar o tema da tese.
- Apresentar o problema que a mesma visa endereçar.
- Os objetivos da tese.
- Finalmente, o que são typestates, antes de aprofundar nas secções seguintes.

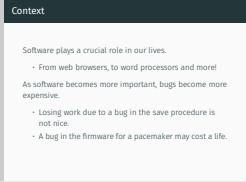
Context

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.



Dado o papel crescente do software nas nossas vidas, os bugs saem cada vez mais caro, tanto às empresas como aos utilizadores.

Perder uma mensagem no Facebook não é grave, mas um dia de trabalho é pior. Imagine-se então um bug num pacemaker, pode causar a morte do utilizador!

Problem

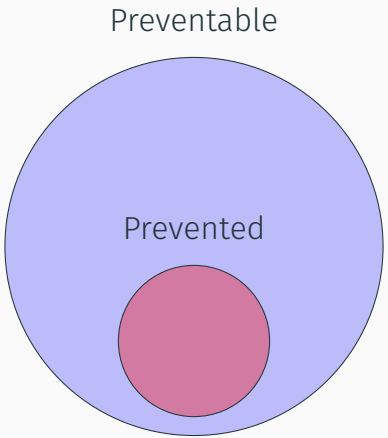


Figure 1: Diagram of preventable bugs and prevented bugs.

Problem
Preventable
Prevented
Figure 1: Diagram of preventable bugs and prevented bugs.

Podemos imaginar todos os tipos de bugs como um conjunto (não mostrado), onde se inserem dois subconjuntos. Os bugs que podemos prevenir e os que, de facto, conseguimos prevenir.

Por exemplo, erros de gestão de memória e mau uso de APIs.

Problem — with Rust

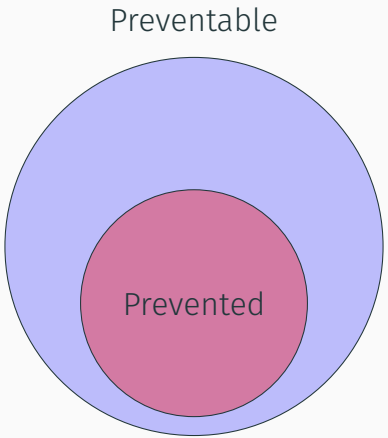


Figure 2: Diagram of preventable bugs and prevented bugs when considering Rust's borrow checker.

Problem — with Rust
Preventable
Prevented
Figure 2: Diagram of preventable bugs and prevented bugs when considering Rust's borrow checker.

Rust permite-nos alargar o conjunto de bugs que conseguimos prevenir, incluindo assim os bugs de memória e de concurrência.

No entanto, continua a não conseguir cobrir todos os bugs.

Problem — Ideal

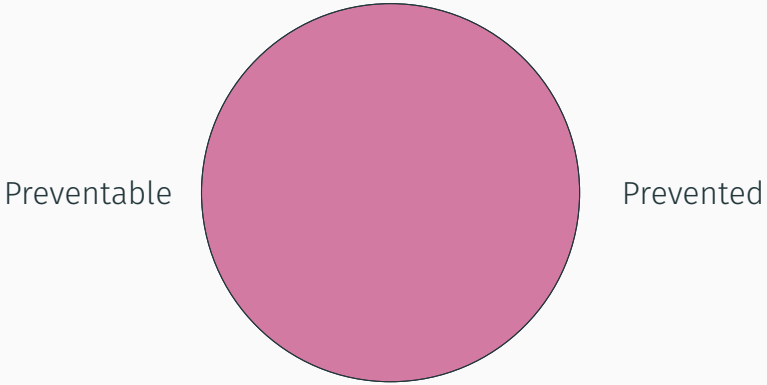


Figure 3: The ideal diagram of preventable bugs and prevented bugs, where all bugs are prevented.

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

O ideal seria uma sobreposição total, como na figura. Os typestates ajudam-nos a dar um passo nesta direção.

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.

Esta tese visa então providenciar typestates práticos na linguagem Rust.
Os mesmos:

- Devem ser fáceis de aprender e usar.
- Não devem impactar a performance durante o runtime.
- Devem escalar a projetos grandes.

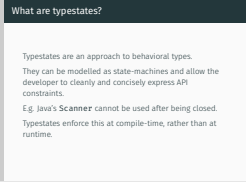
What are typestates?

Typestates are an approach to behavioral types.

They can be modelled as state-machines and allow the developer to cleanly and concisely express API constraints.

E.g. Java's `Scanner` cannot be used after being closed.

Typestates enforce this at compile-time, rather than at runtime.



Após esta apresentação de alto nível de typestates, o que afinal são typestates?

Typestates são uma abordagem a tipos comportamentais. Estes visam descrever o estado em runtime do programa através do sistema de tipos.

Podem ser modelados como máquinas de estados e permitem ao programador expressar de forma clara e consisa restrições de uso de uma API.

Por exemplo: o famoso `Scanner` de Java, não pode ser usado depois de ser fechado, no entanto, o compilador não avisa se tal acontecer. Só durante o runtime é que uma excepção será lançada.

Os typestates permitem que o compilador valide este tipo de restrições.

Outline

Outline

Introduction

State of the Art
Session Types
Typestates

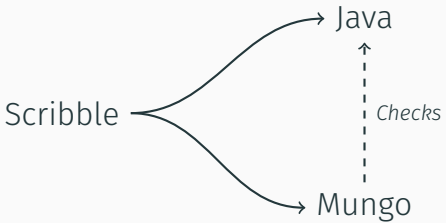
Case Study

Plan

Irei agora efectuar uma breve revisão do estado da arte e de outros projetos existentes. Começando por session types e seguindo para typestates.

Session Types — StMungo

StMungo is a tool that converts Scribble local protocols into a Mungo typestate specification and Java skeleton.



Scribble further enables the conversion of multiparty session types into binary session types.

Session Types — StMungo
StMungo is a tool that converts Scribble local protocols into a Mungo typestate specification and Java skeleton.
Java
Scribble
Checks
Mungo
Scribble further enables the conversion of multiparty session types into binary session types.

O StMungo (ou Scribble to Mungo) é uma ferramenta que converte protocolos de Scribble para especificações de typestates em Mungo e esqueletos de API em Java. A implementação Java é depois verificada pelo Mungo.

O Scribble permite ainda a conversão de multiparty session types para binary session types.

Session Types — Session Types for Rust

The first work with session types and Rust.

Exploits the type system to provide binary session types checked at compile-time.

The library makes use of `unsafe` features.

Session Types — Session Types for Rust

The first work with session types and Rust.

Exploits the type system to provide binary session types checked at compile-time.

The library makes use of `unsafe` features.

(Tanto quanto sei) Esta biblioteca foi a primeira a trazer session types para Rust.

Tira partido do sistema de tipos para validar as especificações de session types.

Dado que usa `unsafe` pode não ser adequada para todos os casos de usos.

Session Types — Multiparty Session Types for Rust

This work is closer to StMungo in that it also makes use of Scribble to describe session types.

In comparison to the previous slide, this tool:

- Makes use of the Rust type system to provide compile-time safety.
- Allows for multiparty session types.
- Requires an external tool.

Session Types — Multiparty Session Types for Rust

This work is closer to StMungo in that it also makes use of Scribble to describe session types.

In comparison to the previous slide, this tool:

- Makes use of the Rust type system to provide compile-time safety.
- Allows for multiparty session types.
- Requires an external tool.

Esta ferramenta é um pouco como o StMungo para Rust. Tira partido do Scribble para descrever session types e em comparação com a biblioteca apresentada no slide anterior:

- Faz uso do sistema de tipos para garantir segurança em tempo de compilação.
- Permite o uso de multiparty session types.
- Requer, no entanto, uma ferramenta externa.

Typestates — Fugue

```
[WithProtocol("open", "closed")]
class OuterSocket {
    [InState("connected",
        WhenEnclosingState="open"),
     NotAliased(WhenEnclosingState="open")]
    [Unavailable(WhenEnclosingState="closed")]
    private Socket innerSocket;
}
```


Typestates — Fugue

Seguindo então para os typestates, começo com o Fugue. Um projeto vindo da Microsoft, onde o meu trabalho se inspira.
O Fugue permite a descrição de estados e transições através de anotações.
Permite ainda a ligação entre sub-estados para garantir o bom uso das APIs.

Typestates — Plaid

Plaid is a *typestate-oriented* language.

It also supports aliasing control through keyword usage:

	Aliasing	Mutation
`unique`		✓
`immutable`	✓	
`shared`	✓	✓

Typestates — Plaid

Plaid is a *typestate-oriented* language.

It also supports aliasing control through keyword usage:

	Aliasing	Mutation
`unique`		✓
`immutable`	✓	
`shared`	✓	✓

O Plaid é uma linguagem orientada aos typestates, fazendo uso dos mesmos como cidadãos de primeira classe.
A outra caracteristica mais relevante da linguagem é o seu controlo de aliasing, efetuado através de palavras-chave, como descrito na tabela.
(Descrever tabela.)

Typestates — Mungo

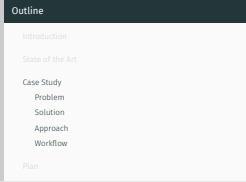
```
1 @Typestate("StateIteratorProtocol")
2 class StateIterator { /* ... */ }
```


Typestates — Mungo
1 @Typestate("StateIteratorProtocol")
2 class StateIterator { /* ... */ }

Por fim temos o Mungo, já falado antes mas não apresentado formalmente. O Mungo é um sistema de tipos / um verificador de typestates.

O seu uso em Java consiste no processamento de uma anotação que liga a descrição do typestate de uma classe, após o processamento da anotação, o Mungo verifica todos os usos da classe validando os mesmos contra a especificação.

Outline



Irei agora apresentar o caso de estudo para apresentar a biblioteca. Começo com o problema geral e a sua solução. Irei rever ainda a abordagem ao problema e por fim, o workflow, onde irei apresentar um problema concreto e apresentar passo a passo como usar a biblioteca para o resolver.

Problem

Error happens at runtime, possibly crashing the program.

```
1  fn main() {
2      let protocol = Protocol::new();
3      protocol.step1();
4      protocol.step3(); // runtime error
5      protocol.step2();
6  }
```

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

Como discutido na introdução, queremos diminuir os bugs. No entanto, os bugs que queremos endereçar são principalmente maus usos das APIs.

Como na figura, o programa iria lançar um panico. Enquanto programadores, queremos que a máquina trabalhe por nós, não ao contrário.

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();
}
```

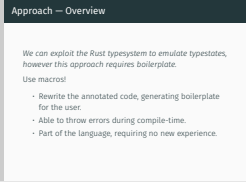

Nós queremos que o compilador nos diga ”hey, não podes chamar este método nesta altura”. Como é que o podemos fazer?

Approach — Overview

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

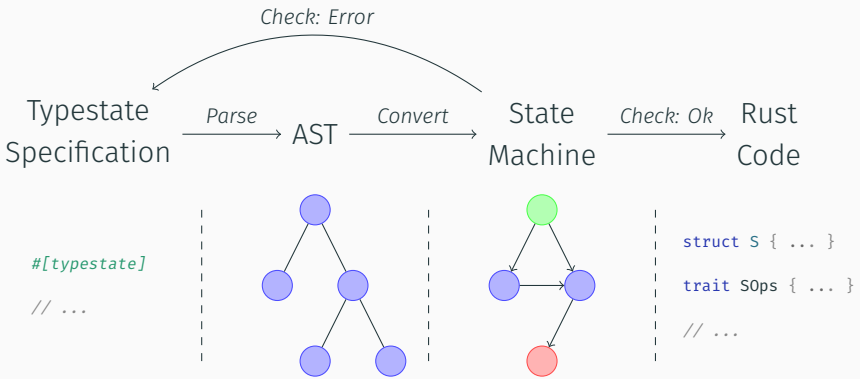
Use macros!

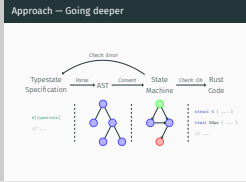
- Rewrite the annotated code, generating boilerplate for the user.
- Able to throw errors during compile-time.
- Part of the language, requiring no new experience.



Usando o sistema de tipos de Rust, podemos emular typestates. No entanto, esta abordagem requer muito boilerplate.
Usando macros, podemos gerar o código por nós. Podemos ainda validar o código contra outras propriedades e lançar erros durante a compilação. Esta abordagem não requer um esforço especial por parte do utilizador, visto que os macros são um elemento comum da linguagem.

Approach — Going deeper





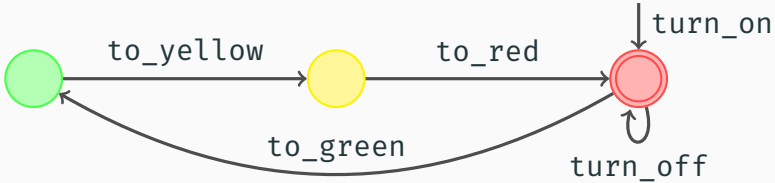
Aprofundando, para fazer com que tal aconteça a arquitetura do macro será como na figura. A especificação é código Rust normal, recorrendo a macros.

Daí, o sistema de macros extrai a AST, a qual precisamos apenas de processar para extrair uma máquina de estados e gerar o código final.

A máquina de estados extraída é verificada para uma série de propriedades e se tudo estiver OK, podemos gerar o código final. Caso contrário, podemos lançar um erro de compilação.

Workflow — Design the state machine

Consider a traffic light as a state machine.



Workflow — Design the state machine
Consider a traffic light as a state machine.
to_yellow
to_red
turn_on
to_green
turn_off

Seguindo para o modo de trabalho, consideremos um semáforo.
Ele percorre um ciclo, verde, amarelo, vermelho e o mesmo pode ser ligado e desligado.
Vamos então modelá-lo com a biblioteca.

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!

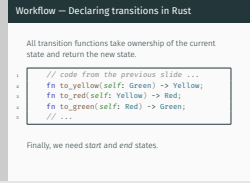
Começamos por declarar o autómato e os estados do mesmo.

Workflow — Declaring transitions in Rust

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

```
1    // code from the previous slide ...
2    fn to_yellow(self: Green) -> Yellow;
3    fn to_red(self: Yellow) -> Red;
4    fn to_green(self: Red) -> Green;
5    // ...
```

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



De seguida, as transições entre estados.

Note-se o consumo do estado atual e o retorno do novo estado.

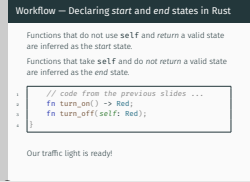
Workflow — Declaring *start* and *end* states in Rust

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.

```
1     // code from the previous slides ...
2     fn turn_on() -> Red;
3     fn turn_off(self: Red);
4 }
```

Our traffic light is ready!



Por fim, o estado inicial e final, que neste caso é o mesmo.

Funções que não consomem o estado atual e retornam um estado são consideradas declarações de estados iniciais.

Funções que consomem o estado atual e não retornam um estado são consideradas declarações de estados finais.

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:

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


Existem ainda mais alguns detalhes a apresentar.

Imaginemos que queremos saber o número de ciclos atravessados pelo semáforo, para questões de manutenção.

Adicionamos então um campo novo ao autómato, tornando-o disponível em todos os estados.

Workflow — The other features

How can we check if the light requires maintenance?

We add a **pure** function:

```
1 fn requires_maintenance(
2     &self: TrafficLight
3 ) -> 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 (&).

Para confirmar se o semáforo precisa de manutenção criamos um predicado. Neste caso, a função só tem acesso imutável a uma referência de qualquer estado. Sendo assim uma função pura.

Workflow — The other features

After maintenance, how can we reset the counter?

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

```
1  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:
1 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.

Após a manutenção, é prático poder dar reset ao semáforo. Para tal usamos uma função impura, capaz de mutar o estado atual, mas não de fazer transições.

Workflow — Final steps

Finally, the developer is only required to implement the state transitions.

```
1 impl GreenState for TrafficLight<Green> {
2     fn to_yellow(self) -> TrafficLight<Yellow> { /*
3 }
```


É possível adicionar a implementação diretamente às declarações das funções, no entanto, tal poderia tornar o código díficil de ler devido às transformações necessárias ao código.

Workflow — Usage

The final user API would look like the following:

```
let tl = TrafficLight<Red>::turn_on();
let green = tl.to_green();
if green.requires_maintenance() {
    // ...
    green.reset();
}
green.to_red(); // compile-time error
```


A API apresentada ao utilizador final seria a seguinte.

O erro é apanhado pelo compilador, devido a um ”type mismatch”.

Outline

Outline

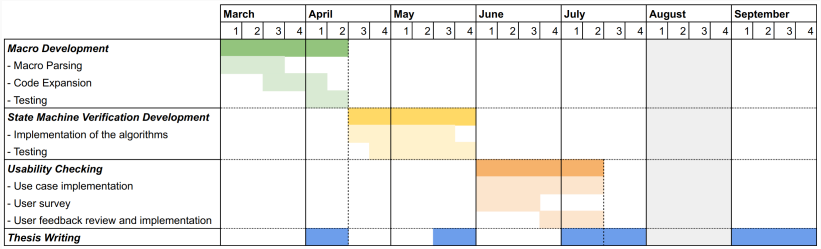
Introduction

State of the Art

Case Study

Plan

Plan Overview



	March				April				May				June				July				August				September			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Macro Development																												
- Macro Parsing																												
- Code Expansion																												
- Testing																												
State Machine Verification Development																												
- Implementation of the algorithms																												
- Testing																												
Usability Checking																												
- Use case implementation																												
- User survey																												
- User feedback review and implementation																												
Thesis Writing																												

Figure 4: Work plan Gantt chart

Plan Overview

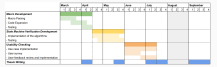


Figure 4: Work plan Gantt chart

Por fim, apresento o plano de trabalho.
(Apresentar e explicar a tabela.)