

Bridging the gap between Tpestates and Rust in production code

José Duarte

António Ravara (Advisor)

March 2021

NOVA School of Science and Technology

2021-03-07

Bridging the gap between Tpestates and Rust in production code

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

Bridging the gap between Tpestates and Rust in production code

José Duarte
António Ravara (Advisor)
March 2021

NOVA School of Science and Technology

Outline

Introduction

Context

Problem

Objectives

State of the Art

Case Study

Plan

2021-03-07

Bridging the gap between Typestates and Rust in production code
└─ Introduction

A introdução segue a seguinte estrutura.

Outline

- Introduction
- Context
- Problem
- Objectives
- State of the Art
- Case Study
- Plan

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.

2021-03-07

Bridging the gap between Typestates and Rust in production code

- └ Introduction
- └ Context

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!

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 — in mainstream languages

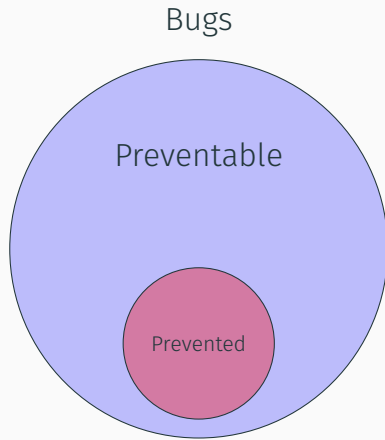
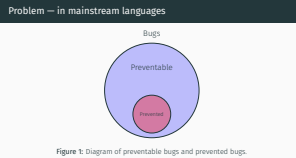


Figure 1: Diagram of preventable bugs and prevented bugs.

2021-03-07

Bridging the gap between Typestates and Rust in production code
└ Introduction
└ Problem



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.

Os que prevenimos são, por exemplo, type mismatch errors; os erros que não conseguimos produzir são, por exemplo, gestão de memória.

Problem — with Rust

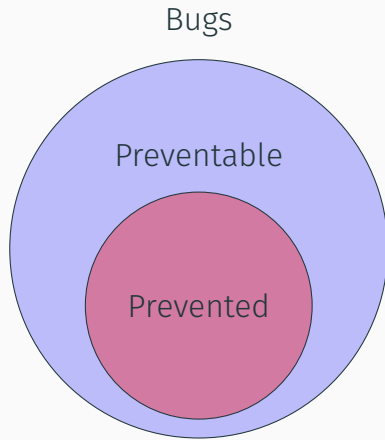
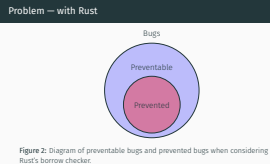


Figure 2: Diagram of preventable bugs and prevented bugs when considering Rust's borrow checker.

2021-03-07
code
└─ Introduction
 └─ Problem

Bridging the gap between Typestates and Rust in production



Rust permite-nos alargar o conjunto de bugs que conseguimos prevenir, incluindo assim os bugs de memória e de concorrência.
No entanto, continua a não conseguir cobrir todos os bugs.

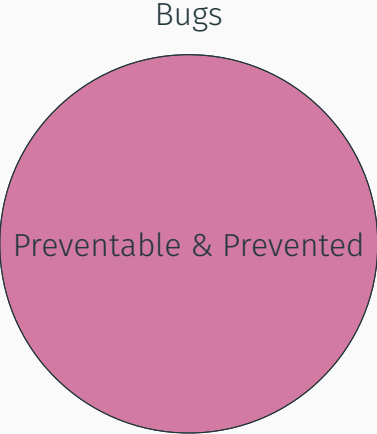


Figure 3: The ideal diagram of preventable bugs and prevented bugs, where all bugs are prevented.

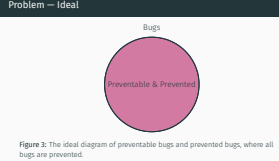


Figure 3: The ideal diagram of preventable bugs and prevented bugs, where all bugs are prevented.

O ideal seria uma sobreposição total, como na figura. Os typestates ajudam-nos a dar um passo nesta direção.
Apesar da programação defensiva tentar maximizar os erros prevenidos, existe sempre o limite do erro humano.

Problem

Error happens at runtime, possibly crashing the program.

```
1 fn main() {  
2     let builder = XYBuilder::new();  
3     builder.setX(0.0);  
4     builder.build(); // runtime error: missing `y`  
5 }
```

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

2021-03-07

Bridging the gap between Typestates and Rust in production

code

└ Introduction

└ Problem

Problem

Error happens at runtime, possibly crashing the program.

```
1 fn main() {  
2     let builder = XYBuilder::new();  
3     builder.setX(0.0);  
4     builder.build(); // runtime error: missing `y`  
5 }
```

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

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 porque não inicializei o Y.

Enquanto programadores, queremos que a máquina trabalhe por nós, não ao contrário. Minimizando assim o tempo de debug, e a complexidade associada ao trabalho.

What are typestates?

- An approach to behavioral types.
- Allow the developer to cleanly and concisely express API constraints.

e.g. “Do not call `.next()` before `.hasNext()`.”

2021-03-07

Bridging the gap between Typestates and Rust in production code

- └ Introduction
- └ Objectives

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.

What are typestates?

- An approach to behavioral types.
- Allow the developer to cleanly and concisely express API constraints.

e.g. “Do not call `.next()` before `.hasNext()`.”

Objectives

A library which brings *practical* typestates to Rust.

- Minimal learning overhead.
- Scalable to large projects.

2021-03-07

Bridging the gap between Typestates and Rust in production code

- └ Introduction
- └ Objectives

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

- Devem ser fáceis de aprender e usar.
- Devem escalar a projetos grandes.

Objectives

A library which brings *practical* typestates to Rust.

- Minimal learning overhead.
- Scalable to large projects.

Introduction

State of the Art

Session Types

Typestates

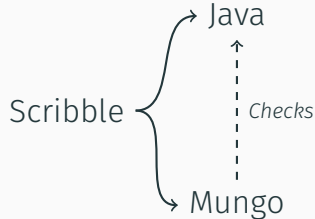
Case Study

Plan

2021-03-07

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

Devido a restrições de tempo vou cingir-me a implementações de tipos comportamentais para linguagens de alto nível em linha com a minha abordagem.



```
module example;

type <xsd> "...";
  from "..." as Hello;

global protocol P(role A, role B) {
  connect(Hello) from A to B;
}
```

```
typestate Protocol {
  Init =
    { /* ... */ }
  Close =
    { /* ... */ }
}
```

2021-03-07 Bridging the gap between Typestates and Rust in production code


- └ State of the Art
- └ 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 — StMungo

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



module example;
type <xsd> "...";
 from "..." as Hello;
global protocol P(role A, role B) {
 connect(Hello) from A to B;
}

typestate Protocol {
 Init =
 { /* ... */ }
 Close =
 { /* ... */ }
}

2021-03-07 Bridging the gap between Typestates and Rust in production code

- └ State of the Art
- └ Session Types

| | Kind | Compile-Time | Ext. Tool |
|---------------------|------------|--------------|-----------|
| Jespersen et al. | Binary | ✓ | |
| Lagaillardie et al. | Multiparty | ✓ | ✓ |

| | Kind | Compile-Time | Ext. Tool |
|---------------------|------------|--------------|-----------|
| Jespersen et al. | Binary | ✓ | |
| Lagaillardie et al. | Multiparty | ✓ | ✓ |

(Tanto quanto sei) O trabalho de Jespersen et al. foi o primeiro a introduzir session types em Rust, entre esse e o seu principal sucessor as diferenças residem no suporte de multiparty session types e o uso ferramentas externas. Ambos válidos para o seu sucessor Lagaillardie.

Os trabalhos apresentados providenciam tipos apenas para canais.

Plaid is a *typestate-oriented* language.

It also supports aliasing control through keyword usage:

| | Aliasing | Mutation |
|------------------------|----------|----------|
| <code>unique</code> | | ✓ |
| <code>immutable</code> | ✓ | |
| <code>shared</code> | ✓ | ✓ |

2021-03-07 Bridging the gap between Typestates and Rust in production code

- └ State of the Art
- └ Typestates

O Plaid é uma linguagem orientada aos typestates, fazendo uso dos mesmos como cidadãos de primeira classe.

A outra característica mais relevante da linguagem é o seu controle de aliasing, efetuado através de palavras-chave, como descrito na tabela.

Plaid is a typestate-oriented language.
It also supports aliasing control through keyword usage:

| | Aliasing | Mutation |
|------------------------|----------|----------|
| <code>unique</code> | | ✓ |
| <code>immutable</code> | ✓ | |
| <code>shared</code> | ✓ | ✓ |

```
1 [WithProtocol("open", "closed")]
2 class OuterSocket {
3     [InState("connected",
4         WhenEnclosingState="open"),
5         NotAliased(WhenEnclosingState="open")]
6     [Unavailable(WhenEnclosingState="closed")]
7     private Socket innerSocket;
8 }
```

2021-03-07 Bridging the gap between Typestates and Rust in production code

- └ State of the Art
- └ Typestates

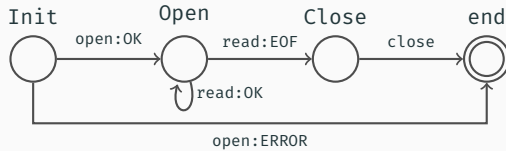
```
1 [WithProtocol("open", "closed")]
2 class OuterSocket {
3     [InState("connected",
4         WhenEnclosingState="open"),
5         NotAliased(WhenEnclosingState="open")]
6     [Unavailable(WhenEnclosingState="closed")]
7     private Socket innerSocket;
8 }
```

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 — Mungo



```
1 @Typestate("SocketProtocol")
2 class Socket { /* ... */ }
```

```
1 typestate Socket {
2   Init = { Status open(): <OK: Open, ERROR: end> }
3   Open = { Read read(): <OK: Open; EOF: Close> }
4   Close = { void close(): end }
5 }
```

15

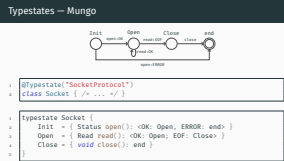
2021-03-07

Bridging the gap between Typestates and Rust in production

code

└ State of the Art

└ Typestates



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.

Introduction

State of the Art

Case Study

 Solution

 Approach

 Workflow

Plan

2021-03-07

Bridging the gap between Typestates and Rust in production code
└─ Case Study

Apresento agora o caso de estudo de acordo com a seguinte estrutura.

Introduction

State of the Art

Case Study

 Solution

 Approach

 Workflow

Plan

Problem

```
1 fn main() {  
2     let builder = XYBuilder::new();  
3     builder.setX(0.0);  
4     builder.build(); // runtime error: missing `y`  
5 }
```

2021-03-07

Bridging the gap between Tpestates and Rust in production

code

└─ Case Study

Problem

```
1 fn main() {  
2     let builder = XYBuilder::new();  
3     builder.setX(0.0);  
4     builder.build(); // runtime error: missing `y`  
5 }
```

Para lembrar, o problema é o compilador não avisar que aquela chamada está fora de ordem.

Solution

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

```
1 fn main() {  
2     let builder = XYBuilder::new();  
3     builder.setX(0.0);  
4     builder.build();  
5         // ^^^^^  
6         // | error: you cannot build without setting `y`  
7 }
```

How?

2021-03-07

Bridging the gap between Typestates and Rust in production

code

└ Case Study

└ Solution

Solution

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

```
1 fn main() {  
2     let builder = XYBuilder::new();  
3     builder.setX(0.0);  
4     builder.build();  
5         // ^^^^^  
6         // | error: you cannot build without setting `y`  
7 }
```

How?

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

Solution

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

```
1 fn main() {  
2     let builder = XYBuilder::new();  
3     builder.setX(0.0);  
4     builder.build();  
5         // ^^^^^  
6         // | error: you cannot build without setting `y`  
7 }
```

How? Declare the protocol.

2021-03-07

Bridging the gap between Typestates and Rust in production

code

└ Case Study

└ Solution

Solution

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

```
1 fn main() {  
2     let builder = XYBuilder::new();  
3     builder.setX(0.0);  
4     builder.build();  
5         // ^^^^^  
6         // | error: you cannot build without setting `y`  
7 }
```

How? Declare the protocol.

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

We can exploit the Rust typesystem to emulate tpestates.

2021-03-07

Bridging the gap between Tpestates and Rust in production code
└─ Case Study
 └─ Approach

Approach — Overview

We can exploit the Rust typesystem to emulate tpestates.

Usando o sistema de tipos de Rust, podemos emular tpestates. No entanto, esta abordagem requer muito boilerplate.

Esta abordagem não requer um esforço especial por parte do utilizador, visto que os macros são um elemento comum da linguagem.

Usando macros, podemos gerar o código por nós. Podemos ainda validar o código contra outras propriedades e lançar erros durante a compilação.

We can exploit the Rust typesystem to emulate tpestates.

However this approach requires boilerplate.

2021-03-07

Bridging the gap between Tpestates and Rust in production code

- └ Case Study
 - └ Approach

We can exploit the Rust typesystem to emulate tpestates.
However this approach requires boilerplate.

Usando o sistema de tipos de Rust, podemos emular tpestates. No entanto, esta abordagem requer muito boilerplate.

Esta abordagem não requer um esforço especial por parte do utilizador, visto que os macros são um elemento comum da linguagem.

Usando macros, podemos gerar o código por nós. Podemos ainda validar o código contra outras propriedades e lançar erros durante a compilação.

We can exploit the Rust typesystem to emulate tpestates.

However this approach requires boilerplate.

Use macros! Part of the language, requiring no new experience.

2021-03-07

Bridging the gap between Tpestates and Rust in production code
└─ Case Study
 └─ Approach

Usando o sistema de tipos de Rust, podemos emular tpestates. No entanto, esta abordagem requer muito boilerplate.

Esta abordagem não requer um esforço especial por parte do utilizador, visto que os macros são um elemento comum da linguagem.

Usando macros, podemos gerar o código por nós. Podemos ainda validar o código contra outras propriedades e lançar erros durante a compilação.

We can exploit the Rust typesystem to emulate tpestates.
However this approach requires boilerplate.
Use macros! Part of the language, requiring no new experience.

We can exploit the Rust typesystem to emulate tpestates.

However this approach requires boilerplate.

Use macros! Part of the language, requiring no new experience.

- Rewrite the annotated code, generating boilerplate for the user.

2021-03-07

Bridging the gap between Tpestates and Rust in production code

- └ Case Study
- └ Approach

Usando o sistema de tipos de Rust, podemos emular tpestates. No entanto, esta abordagem requer muito boilerplate.

Esta abordagem não requer um esforço especial por parte do utilizador, visto que os macros são um elemento comum da linguagem.

Usando macros, podemos gerar o código por nós. Podemos ainda validar o código contra outras propriedades e lançar erros durante a compilação.

We can exploit the Rust typesystem to emulate tpestates.
However this approach requires boilerplate.
Use macros! Part of the language, requiring no new experience.
• Rewrite the annotated code, generating boilerplate for the user.

We can exploit the Rust typesystem to emulate tpestates.

However this approach requires boilerplate.

Use macros! Part of the language, requiring no new experience.

- Rewrite the annotated code, generating boilerplate for the user.
- Throw errors during compile-time.

2021-03-07

Bridging the gap between Tpestates and Rust in production code

- └ Case Study
- └ Approach

Usando o sistema de tipos de Rust, podemos emular tpestates. No entanto, esta abordagem requer muito boilerplate.

Esta abordagem não requer um esforço especial por parte do utilizador, visto que os macros são um elemento comum da linguagem.

Usando macros, podemos gerar o código por nós. Podemos ainda validar o código contra outras propriedades e lançar erros durante a compilação.

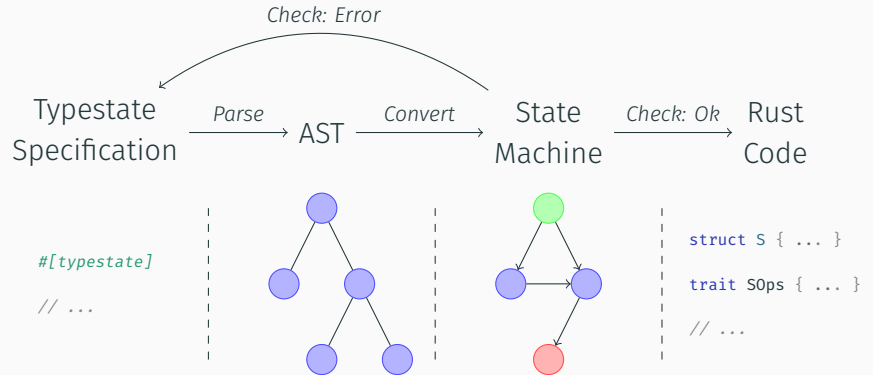
We can exploit the Rust typesystem to emulate tpestates.

However this approach requires boilerplate.

Use macros! Part of the language, requiring no new experience.

- Rewrite the annotated code, generating boilerplate for the user.
- Throw errors during compile-time.

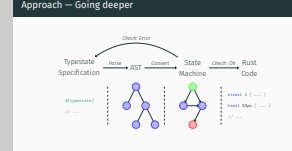
Approach — Going deeper



2021-03-07

Bridging the gap between Typestates and Rust in production code

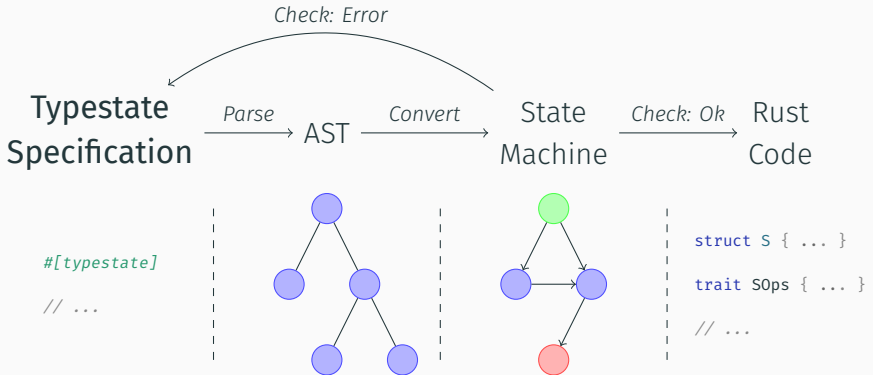
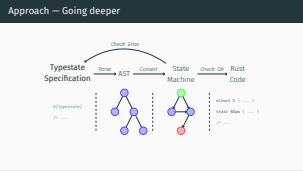
- Case Study
- Approach



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.

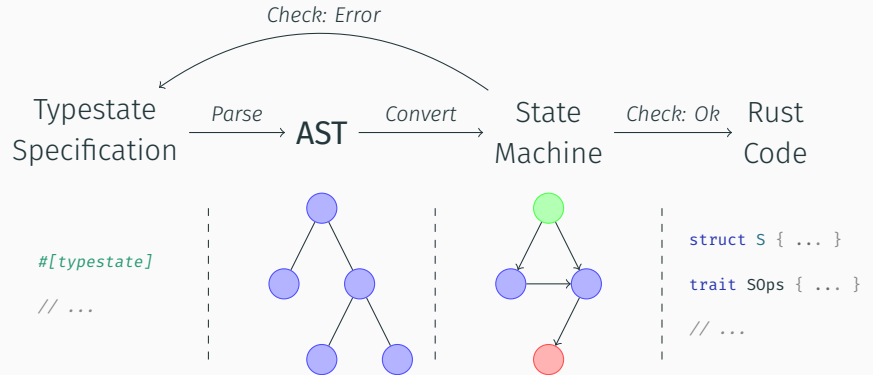
Approach — Going deeper

2021-03-07
Bridging the gap between Typestates and Rust in production code
└─ Case Study
└─ Approach



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.

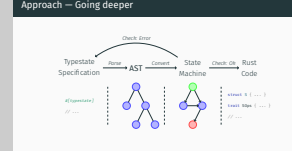
Approach — Going deeper



2021-03-07

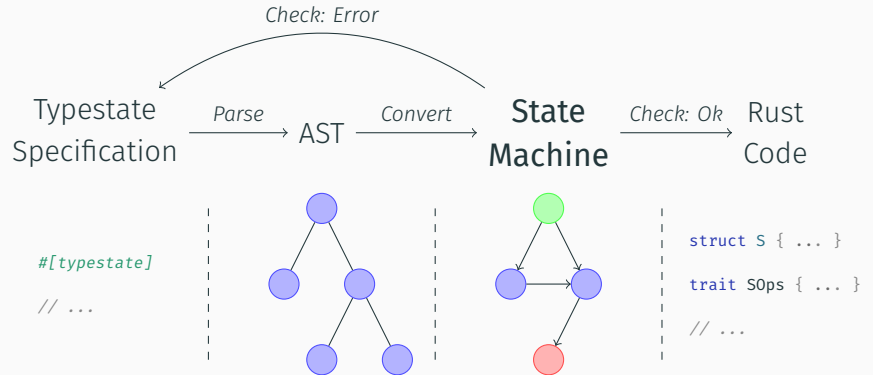
Bridging the gap between Typestates and Rust in production code

- Case Study
- Approach



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.

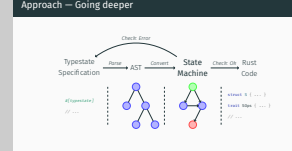
Approach — Going deeper



2021-03-07

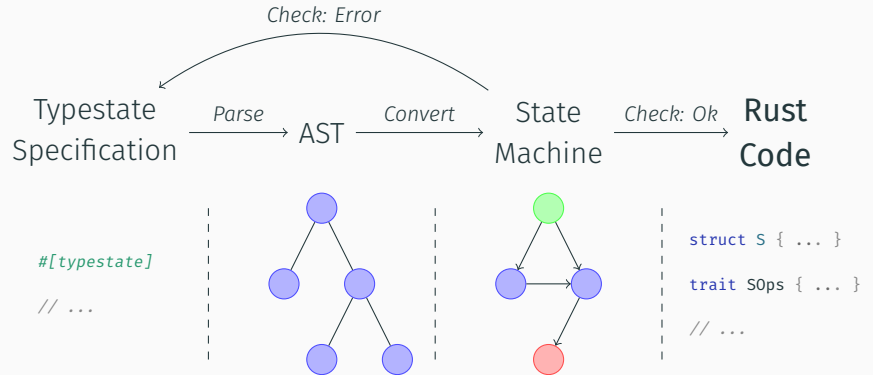
Bridging the gap between Typestates and Rust in production code

- Case Study
- Approach



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.

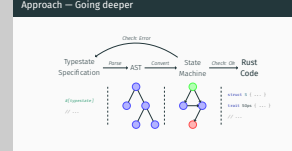
Approach — Going deeper



2021-03-07

Bridging the gap between Typestates and Rust in production code

- Case Study
- Approach

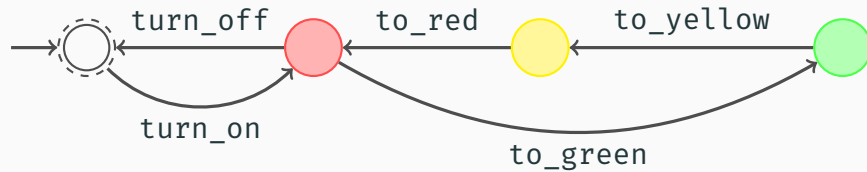


Aprofundando, para fazer com que tal aconteça a arquitetura do macro será como na figura. A especificação é código Rust normal, recorrendo a macros. Daí, o sistema de macros extrai a AST, a qual precisamos apenas de processar para extrair uma máquina de estados e gerar o código final.

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.



2021-03-07 Bridging the gap between Typestates and Rust in production code

- └ Case Study
- └ Workflow

Workflow — Design the state machine



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:

```
1  #[typestate] mod traffic_light {  
2      #[automata] struct TrafficLight;  
3      #[state] struct Green;  
4      #[state] struct Yellow;  
5      #[state] struct Red;  
6      // ...
```

We still need transitions!

2021-03-07

Bridging the gap between Typestates and Rust in production

code

└─ Case Study

└─ Workflow

Workflow — Declaring state in Rust

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

```
1  #[typestate] mod traffic_light {  
2      #[automata] struct TrafficLight;  
3      #[state] struct Green;  
4      #[state] struct Yellow;  
5      #[state] struct Red;  
6      // ...
```

We still need transitions!

Começamos por declarar o autômato 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 // ...
```

De seguida, as transições entre estados.

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

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 // ...
```


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

```
1 // code from the previous slides ...  
2 fn turn_on() -> Red;
```

code

└─ Case Study

└─ Workflow

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.

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

```
1 // code from the previous slides ...  
2 fn turn_on() -> Red;
```

Workflow — Declaring *end* states in Rust

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

```
1 fn turn_off(self: Red);  
2 }
```

Our traffic light is ready!

2021-03-07

Bridging the gap between Typestates and Rust in production

code

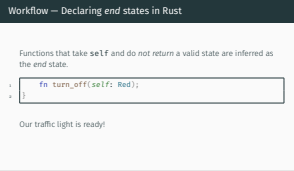
└ Case Study

└ Workflow

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.



For maintenance purposes, 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 autômato, tornando-o disponível em todos os estados.

For maintenance purposes, 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:

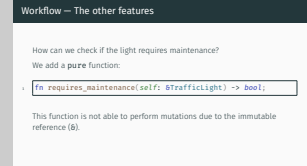
```
1 fn requires_maintenance(self: &TrafficLight) -> bool;
```

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

2021-03-07
code
└─ Case Study
 └─ Workflow

Bridging the gap between Tpestates and Rust in production

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:

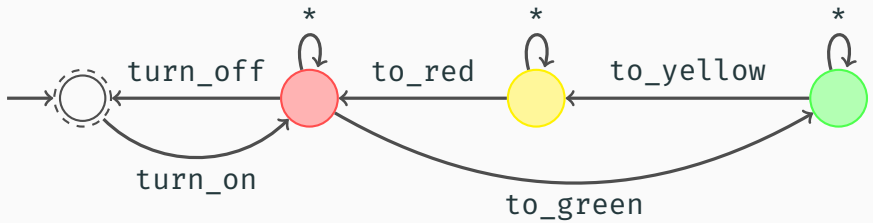
```
1 fn reset_counter(self: &mut TrafficLight);
```

The function is able to perform mutations over fields, 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 — The new state machine

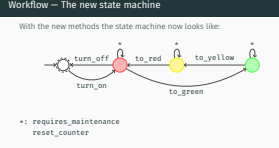
With the new methods the state machine now looks like:



`*: requires_maintenance
reset_counter`

2021-03-07

Bridging the gap between Typestates and Rust in production
code
└─ Case Study
└─ Workflow



Após a adição das novas funções, ficamos com a seguinte máquina de estados.

After the code is checked and expanded, all that is left is for the user to implement the state transitions.

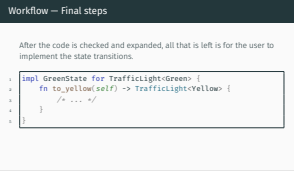
```
1 impl GreenState for TrafficLight<Green> {  
2     fn to_yellow(self) -> TrafficLight<Yellow> {  
3         /* ... */  
4     }  
5 }
```

code

└─ Case Study

└─ Workflow

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



The final user API would look like the following:

```
1 let tl = TrafficLight<Red>::turn_on();
2 let green = tl.to_green();
3 if green.requires_maintenance() {
4     // ...
5     green.reset();
6 }
7 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".

Workflow — Usage

The final user API would look like the following:

```
1 let tl = TrafficLight<Red>::turn_on();
2 let green = tl.to_green();
3 if green.requires_maintenance() {
4     // ...
5     green.reset();
6 }
7 green.to_red(); // compile-time error
```


The final user API would look like the following:

```
1 let tl = TrafficLight<Red>::turn_on();
2 let green = tl.to_green();
3 if green.requires_maintenance() {
4     // ...
5     green.reset();
6 }
7 green.to_red(); // compile-time error
```

```
1 let tl = TrafficLight<Red>::turn_on();
2 let green = tl.to_green();
3 if green.requires_maintenance() {
4     // ...
5     green.reset();
6 }
7 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".

The final user API would look like the following:

```
1 let tl = TrafficLight<Red>::turn_on();
2 let green = tl.to_green();
3 if green.requires_maintenance() {
4     // ...
5     green.reset();
6 }
7 green.to_red(); // compile-time error
```

code

└─ Case Study

└─ Workflow

The final user API would look like the following:

```
1 let tl = TrafficLight<Red>::turn_on();
2 let green = tl.to_green();
3 if green.requires_maintenance() {
4     // ...
5     green.reset();
6 }
7 green.to_red(); // compile-time error
```

A API apresentada ao utilizador final seria a seguinte.

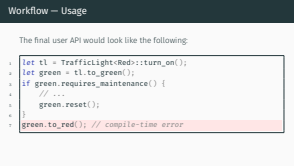
O erro é apanhado pelo compilador, devido a um "type mismatch".

The final user API would look like the following:

```
1 let tl = TrafficLight<Red>::turn_on();
2 let green = tl.to_green();
3 if green.requires_maintenance() {
4     // ...
5     green.reset();
6 }
7 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".



Introduction

State of the Art

Case Study

Plan

2021-03-07

Bridging the gap between Typestates and Rust in production
code
└─ Plan

Introduction

State of the Art

Case Study

Plan

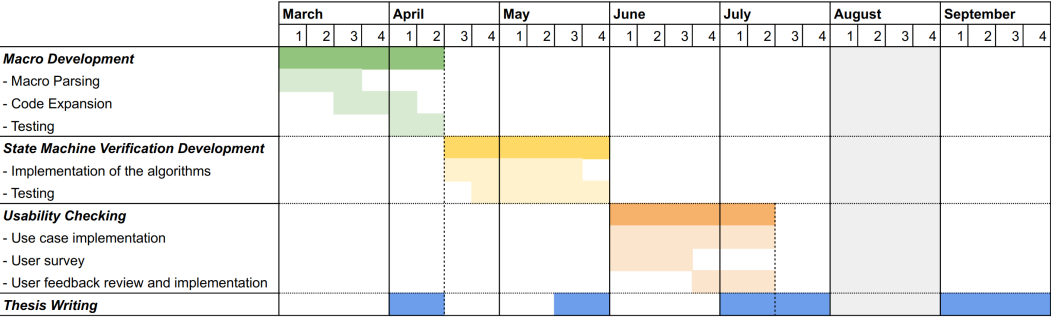
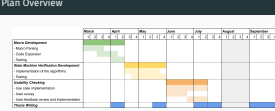


Figure 4: Work plan Gantt chart

2021-03-07

Bridging the gap between Typestates and Rust in production
code
└ Plan



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