

Compile-time Deadlock Detection in Rust using Petri Nets

Horacio Lisdero Scaffino

Facultad de Ingeniería
Universidad de Buenos Aires

June 30, 2023

Agenda

1 Introduction

2 Rust

- What is Rust?
- How does it look like?
- Why Rust?

3 Petri nets

- Examples

4 Translation

- MIR
- Modelling threads
- Modelling mutexes
- Modelling condition variables

Agenda

1 Introduction

2 Rust

- What is Rust?
- How does it look like?
- Why Rust?

3 Petri nets

- Examples

4 Translation

- MIR
- Modelling threads
- Modelling mutexes
- Modelling condition variables

Agenda

1 Introduction

2 Rust

- What is Rust?
- How does it look like?
- Why Rust?

3 Petri nets

- Examples

4 Translation

- MIR
- Modelling threads
- Modelling mutexes
- Modelling condition variables

Agenda

1 Introduction

2 Rust

- What is Rust?
- How does it look like?
- Why Rust?

3 Petri nets

- Examples

4 Translation

- MIR
- Modelling threads
- Modelling mutexes
- Modelling condition variables

What is Rust?

Rust is a multi-paradigm, general-purpose programming language that aims to provide developers with a safe and efficient way to write low-level code.

What is Rust?

Rust is a multi-paradigm, general-purpose programming language that aims to provide developers with a safe and efficient way to write low-level code.

- Memory-safe
- Compiled to machine code, no runtime needed
- High-level simplicity
- Low-level performance (on the same level as C or C++)

Brief timeline of Rust

- 2007** Started as a side project by Graydon Hoare, a programmer at Mozilla
- 2009** Mozilla officially started sponsoring the project
- 2015** First stable version 1.0
- 2016** Mozilla releases Servo, a browser engine built with Rust
- 2019** `async/await` support stabilized
- 2021** The Rust Foundation is founded by AWS, Huawei, Google, Microsoft, and Mozilla
- 2021** The Android Open Source Project encourages the use of Rust for the SO components below the ART
- 2022** The Linux kernel adds support for Rust alongside C
- 2023** 8 years in a row the most loved programming language in the Stack Overflow Developer Survey

Memory safety

It achieves memory safety without using a garbage collector or reference counting. Instead, it uses the concept of **ownership** and **borrowing**.

Memory safety

It achieves memory safety without using a garbage collector or reference counting. Instead, it uses the concept of **ownership** and **borrowing**.

It prevents a wide variety of error classes at compile-time:

- Double free
- Use after free
- Dangling pointers
- Data races
- Passing non-thread-safe variables

If a violation of the compiler rules is found, the program will simply not compile.

Agenda

1 Introduction

2 Rust

- What is Rust?
- **How does it look like?**
- Why Rust?

3 Petri nets

- Examples

4 Translation

- MIR
- Modelling threads
- Modelling mutexes
- Modelling condition variables

Immutability by default

```
1 fn main() {
2     let x = 1;
3     x = x + 1;
4 }
```

error[E0384]: cannot assign twice to immutable variable `x`

--> src/main.rs:3:5

```
2 |     let x = 1;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
3 |     x = x + 1;
  |     ^^^^^^^^^ cannot assign twice to immutable variable
```

Move semantics by default

Each value has only one owner. If a variable is passed to another function or scope, the owner of the value changes.

```
1 fn main() {
2     let name = String::from("Alice");
3     print_name(name);
4     println!("The name is: {}", name); // Compilation error
5 }
6
7 fn print_name(name: String) {
8     println!("Name: {}", name);
9 }
```

Algebraic Data Types, aka enums with fields

```

1  enum Shape {
2      Circle { radius: f64 },
3      Rectangle { width: f64, height: f64 },
4      Triangle { base: f64, height: f64 },
5  }
6
7  fn main() {
8      let shapes = vec![
9          Shape::Circle { radius: 5.0 },
10         Shape::Rectangle { width: 10.0, height: 8.0 },
11         Shape::Triangle { base: 7.0, height: 4.0 },
12     ];
13
14     for shape in shapes {
15         match shape {
16             Shape::Circle { radius } => {
17                 let circle = Circle { radius };
18                 // Do something with the circle...
19             },
20             Shape::Rectangle { width, height } => {
21                 let rectangle = Rectangle { width, height };
22                 // Do something with the rectangle...
23             },
24             Shape::Triangle { base, height } => {
25                 let triangle = Triangle { base, height };
26                 // Do something with the triangle...
27             },
28         }
29     }
30 }

```

Error handling with Result

```
1  use std::fs::File;
2  use std::io::Read;
3  // This definition is part of the standard library
4  // It does not need to be imported
5  enum Result<T, E> {
6      Ok(T),
7      Err(E),
8  }
9
10 fn read_file_contents(path: &str) -> Result<String, std::io::Error> {
11     let mut file = File::open(path)?;
12     let mut contents = String::new();
13     file.read_to_string(&mut contents)?;
14     Ok(contents)
15 }
16
17 fn main() {
18     let file_path = "example.txt";
19     let result = read_file_contents(file_path);
20
21     match result {
22         Ok(contents) => {
23             println!("File contents:\n{}", contents);
24         }
25         Err(error) => {
26             eprintln!("Error reading file: {}", error);
27         }
28     }
29 }
```

The enum Option: No need for null pointers

```
1  // This definition is part of the standard library
2  // It does not need to be imported
3  pub enum Option<T> {
4      None,
5      Some(T),
6  }
7
8  fn main() {
9      let mut list = vec![1, 2, 3, 4, 5];
10
11     while let Some(element) = list.pop() {
12         println!("Popped element: {}", element);
13     }
14     // List::pop() returned `None`
15     println!("List is empty!");
16 }
```


No OOP: Just structs with methods

```
1  struct Rectangle {
2      width: u32,
3      height: u32,
4  }
5
6  impl Rectangle {
7      fn new(width: u32, height: u32) -> Rectangle {
8          Rectangle { width, height }
9      }
10
11     fn area(&self) -> u32 {
12         self.width * self.height
13     }
14
15     fn is_square(&self) -> bool {
16         self.width == self.height
17     }
18
19     fn double_size(&mut self) {
20         self.width *= 2;
21         self.height *= 2;
22     }
23 }
```

Define traits to share an interface

```
1  trait Container {
2      fn get_value(&self);
3  }
4
5  struct Storage {
6      value: i32,
7  }
8
9  impl Container for Storage {
10     fn get_value(&self) {
11         println!("Value: {}", self.value);
12     }
13 }
14
15 impl PartialEq for Storage {
16     fn eq(&self, other: &Self) -> bool {
17         self.value == other.value
18     }
19 }
20
21 impl std::fmt::Display for Storage {
22     fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
23         write!(f, "Value: {}", self.value)
24     }
25 }
```

Nearly everything is an expression as in Lisp

```
1  fn main() {
2      let numbers = vec![1, 2, 3, 4, 5];
3
4      let sum_of_squares: i32 = numbers
5          .iter()
6          .fold(0, |acc, x| acc + x * x);
7
8      let result = if sum_of_squares > 50 {
9          "Sum of squares is greater than 50"
10     } else {
11         "Sum of squares is not greater than 50"
12     };
13
14     println!("Result: {}", result);
15 }
```

Generics

```
1  struct Pair<T, U> {
2      first: T,
3      second: U,
4  }
5
6  impl<T, U> Pair<T, U> {
7      fn new(first: T, second: U) -> Self {
8          Pair { first, second }
9      }
10
11     fn get_first(&self) -> &T {
12         &self.first
13     }
14
15     fn get_second(&self) -> &U {
16         &self.second
17     }
18 }
```

Lifetimes

```
1 struct StringHolder<'a> {
2     value: &'a str,
3 }
4
5 impl<'a> StringHolder<'a> {
6     fn new(value: &'a str) -> Self {
7         StringHolder { value }
8     }
9
10    fn get_value(&self) -> &'a str {
11        self.value
12    }
13 }
14
15 fn main() {
16     let input_string = String::from("Hello, lifetimes!");
17
18     let holder;
19     {
20         let local_string = String::from("Local string");
21         holder = StringHolder::new(local_string.as_str());
22         println!("Holder value: {}", holder.get_value());
23     }
24
25     println!("Input string: {}", input_string);
26     println!("Holder value: {}", holder.get_value());
27 }
```

Lifetimes: Error message

```

error[E0597]: `local_string` does not live long enough
--> src/main.rs:21:36
|
20 |         let local_string = String::from("Local string");
|         ----- binding `local_string` declared here
21 |         holder = StringHolder::new(local_string.as_str());
|                                     ^^^^^^^^^^^^^^^^^^^^^ borrowed value does not
|                                                             live long enough
22 |         println!("Holder value: ", holder.get_value());
23 |     }
|     - `local_string` dropped here while still borrowed
...
26 |     println!("Holder value: ", holder.get_value());
|                                     ----- borrow later used here

```

For more information about this error, try ``rustc --explain E0597``.

Only one active mutable reference at any given time

```

1  struct Item {
2      value: i32,
3  }
4
5  fn main() {
6      let mut item = Item { value: 42 };
7
8      let reference1 = &mut item; // First mutable reference
9      let reference2 = &mut item; // Second mutable reference - COMPILATION ERROR
10
11     reference1.value += 1;
12     reference2.value += 1;
13
14     println!("Reference 1: {}", reference1.value);
15     println!("Reference 2: {}", reference2.value);
16 }

```

error[E0499]: cannot borrow `item` as mutable more than once at a time

--> src/main.rs:9:22

```

8 |     let reference1 = &mut item; // First mutable reference
  |                      ----- first mutable borrow occurs here
9 |     let reference2 = &mut item; // Second mutable reference - COMPILATION ERROR
  |                      ^^^^^^^^^ second mutable borrow occurs here
10 |
11 |     reference1.value += 1;
  |     ----- first borrow later used here

```

For more information about this error, try `rustc --explain E0499`.

A mutable reference is allowed only if no immutable references are present

```

1  fn main() {
2      let mut item = 42;
3
4      let reference1 = &item; // First mutable reference
5      let reference2 = &mut item; // Second mutable reference - COMPILATION ERROR
6
7      if *reference1 == 1 {
8          println!("Item is set to one");
9      }
10     *reference2 += 1;
11
12     println!("Reference 1: {}", reference1);
13     println!("Reference 2: {}", reference2);
14 }

```

error[E0502]: cannot borrow `item` as mutable because it is also borrowed as immutable

--> src/main.rs:5:22

```

4 |     let reference1 = &item; // First mutable reference
  |                       ----- immutable borrow occurs here
5 |     let reference2 = &mut item; // Second mutable reference - COMPILATION ERROR
  |                       ^^^^^^^^^ mutable borrow occurs here
6 |
7 |     if *reference1 == 1 {
  |           ----- immutable borrow later used here

```


Agenda

1 Introduction

2 Rust

- What is Rust?
- How does it look like?
- **Why Rust?**

3 Petri nets

- Examples

4 Translation

- MIR
- Modelling threads
- Modelling mutexes
- Modelling condition variables

Agenda

1 Introduction

2 Rust

- What is Rust?
- How does it look like?
- Why Rust?

3 Petri nets

■ Examples

4 Translation

- MIR
- Modelling threads
- Modelling mutexes
- Modelling condition variables

Agenda

1 Introduction

2 Rust

- What is Rust?
- How does it look like?
- Why Rust?

3 Petri nets

- Examples

4 Translation

- MIR
- Modelling threads
- Modelling mutexes
- Modelling condition variables

Agenda

1 Introduction

2 Rust

- What is Rust?
- How does it look like?
- Why Rust?

3 Petri nets

- Examples

4 Translation

- MIR
- Modelling threads
- Modelling mutexes
- Modelling condition variables

Agenda

1 Introduction

2 Rust

- What is Rust?
- How does it look like?
- Why Rust?

3 Petri nets

- Examples

4 Translation

- **MIR**
- Modelling threads
- Modelling mutexes
- Modelling condition variables

Agenda

1 Introduction

2 Rust

- What is Rust?
- How does it look like?
- Why Rust?

3 Petri nets

- Examples

4 Translation

- MIR
- **Modelling threads**
- Modelling mutexes
- Modelling condition variables

Agenda

1 Introduction

2 Rust

- What is Rust?
- How does it look like?
- Why Rust?

3 Petri nets

- Examples

4 Translation

- MIR
- Modelling threads
- **Modelling mutexes**
- Modelling condition variables

Agenda

1 Introduction

2 Rust

- What is Rust?
- How does it look like?
- Why Rust?

3 Petri nets

- Examples

4 Translation

- MIR
- Modelling threads
- Modelling mutexes
- **Modelling condition variables**

Bibliography