Compile-time Deadlock Detection in Rust using Petri Nets

Horacio Lisdero Scaffino

Facultad de Ingeniería
Universidad de Buenos Aires

June 30, 2023

Agenda

Agenda

Agenda

Agenda

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

Immutability by default

In other languages, immutability is the exception or an afterthought (e.g. `const`-ness in C/C++).

```rust
fn main() {
  let x = 1; // Immutable by default
  x = x + 1;
}
```

The Rust compiler points out exactly where the error is and provides help on how to fix it.

```
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 assigned to a different variable, the owner of the value changes.

```rust
fn main() {
  let name = String::from("Alice");
  print_name(name);
  println!("The name is: {}", name); // Compilation error
}

fn print_name(name: String) {
    println!("Name: {}", name);
}
```

Values have copy semantics only if they are marked as `Copy`. This is the case for numbers by default. Compare this with the default in C++ vs the best practices.

Algebraic Data Types, aka enums with fields

```
enum Shape {
  Circle { radius: f64 },
  Rectangle { width: f64, height: f64 },
  Triangle { base: f64, height: f64 },
}

fn main() {
    let shapes = vec![
        Shape::Circle { radius: 5.0 },
        Shape::Rectangle { width: 10.0, height: 8.0 },
        Shape::Triangle { base: 7.0, height: 4.0 },
    ];

    for shape in shapes {
        match shape {
            Shape::Circle { radius } => {
                let circle = Circle { radius };
                // Do something with the circle...
            },
            Shape::Rectangle { width, height } => {
                let rectangle = Rectangle { width, height };
                // Do something with the rectangle...
            },
            Shape::Triangle { base, height } => {
                let triangle = Triangle { base, height };
                // Do something with the triangle...
            },
        }
    }
}
```

Modeling data in Rust

- Leverage the type system in your favor
- Make invalid states unrepresentable
- Define new types for the entities in your domain
- Use enums when variables can take different values

```rust
struct FakeCat {
  alive: bool,
  hungry: bool,
}
```

```rust
enum RealCat {
  Alive { hungry: bool },
  Dead,
}
```

By directly modeling the business domain with Rust's expressive type system, the compiler is able to verify the business logic and we catch more errors at compile-time.

A more advanced match statement

A **match** statement works *with* the type system, while a mere **if** can do anything and it is not bound to the type system. Match statements are always *exhaustive*: They must handle all possibilities.

```rust
fn main() {
  let number = 42;

  match number {
      0 => println!("The number is zero"),
      1 | 2 | 3 => println!("The number is a small prime"),
      n @ 4..=9 => println!("The number is between 4 and 9: {}", n),
      n if is_even(n) => println!("The number is even: {}", n),
      n if is_odd(n) => println!("The number is odd: {}", n),
      _ => println!("The number doesn't match any specific case"),
  }
}
```

Python 3.10 introduced a similar feature (PEP 636). Java 17 has a limited version of this (JEP 406).

Error handling with Result

```rust
use std::fs::File;
use std::io::Read;
// This definition is part of the standard library
// It does not need to be imported
enum Result<T, E> {
  Ok(T),
  Err(E),
}

fn read_file_contents(path: &str) -> Result<String, std::io::Error> {
    let mut file = File::open(path)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents)
}

fn main() {
    let file_path = "example.txt";
    let result = read_file_contents(file_path);

    match result {
        Ok(contents) => {
            println!("File contents:\n{}", contents);
        }
        Err(error) => {
            eprintln!("Error reading file: {}", error);
        }
    }
}
```

The enum Option: No need for null pointers

```rust
// This definition is part of the standard library
// It does not need to be imported
pub enum Option<T> {
  None,
  Some(T),
}

fn main() {
  let mut list = vec![1, 2, 3, 4, 5];

  while let Some(element) = list.pop() {
      println!("Popped element: {}", element);
  }
  // List::pop() returned `None`
  println!("List is empty!");
}
```

No OOP: Just structs with methods

```rust
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn new(width: u32, height: u32) -> Rectangle {
        Rectangle { width, height }
    }

    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn is_square(&self) -> bool {
        self.width == self.height
    }

    fn double_size(&mut self) {
        self.width *= 2;
        self.height *= 2;
    }
}
```

Define traits to share an interface

```rust
trait Container {
  fn get_value(&self);
}

struct Storage {
    value: i32,
}

impl Container for Storage {
    fn get_value(&self) {
        println!("Value: {}", self.value);
    }
}

impl PartialEq for Storage {
    fn eq(&self, other: &Self) -> bool {
        self.value == other.value
    }
}

impl std::fmt::Display for Storage {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Value: {}", self.value)
    }
}
```

Nearly everything is an expression as in Lisp

```rust
fn main() {
  let numbers = vec![1, 2, 3, 4, 5];

  let sum_of_squares: i32 = numbers
      .iter()
      .fold(0, |acc, x| acc + x * x);

  let result = if sum_of_squares > 50 {
      "Sum of squares is greater than 50"
  } else {
      "Sum of squares is not greater than 50"
  };

  println!("Result: {}", result);
}
```

Generics

```rust
struct Pair<T, U> {
  first: T,
  second: U,
}

impl<T, U> Pair<T, U> {
    fn new(first: T, second: U) -> Self {
        Pair { first, second }
    }

    fn get_first(&self) -> &T {
        &self.first
    }

    fn get_second(&self) -> &U {
        &self.second
    }
}
```

Lifetimes

```rust
struct StringHolder<'a> {
  value: &'a str,
}

impl<'a> StringHolder<'a> {
    fn new(value: &'a str) -> Self {
        StringHolder { value }
    }

    fn get_value(&self) -> &'a str {
        self.value
    }
}

fn main() {
    let input_string = String::from("Hello, lifetimes!");

    let holder;
    {
        let local_string = String::from("Local string");
        holder = StringHolder::new(local_string.as_str());
        println!("Holder value: {}", holder.get_value());
    }

    println!("Input string: {}", input_string);
    println!("Holder value: {}", holder.get_value());
}
```

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

```rust
struct Item {
  value: i32,
}

fn main() {
    let mut item = Item { value: 42 };

    let reference1 = &mut item; // First mutable reference
    let reference2 = &mut item; // Second mutable reference - COMPILATION ERROR

    reference1.value += 1;
    reference2.value += 1;

    println!("Reference 1: {}", reference1.value);
    println!("Reference 2: {}", reference2.value);
}
```

```
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

```rust
fn main() {
  let mut item = 42;

  let reference1 = &item; // First mutable reference
  let reference2 = &mut item; // Second mutable reference - COMPILATION ERROR

  if *reference1 == 1 {
      println!("Item is set to one");
  }
  *reference2 += 1;

  println!("Reference 1: {}", reference1);
  println!("Reference 2: {}", reference2);
}
```

```
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

Agenda

Agenda

Agenda

Agenda

Agenda

Agenda

Agenda

Bibliography