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

```rust
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}"),
      _ => panic!("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

```
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

```
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

Memory safety is critical for reliability and security

Empirical investigations have concluded that around 70% of the vulnerabilities found in large C/C++ codebases are due to memory handling errors. This high figure can be observed in projects such as:

- Android Open Source Project [1],
- the Bluetooth and media components of Android [2],
- the Chromium Projects behind the Chrome web browser [3],
- the CSS component of Firefox [4],
- iOS and macOS [5],
- Microsoft products [6, 7],
- Ubuntu [8]

Rust adoption is increasing fast

- The Android Open Source Project encourages the use of Rust for the SO components below the ART [9].
- The Linux kernel introduces in version 6.1 official tooling support for programming components in Rust [10, 11].
- At Mozilla, the Oxidation project was created in 2015 to increase the usage of Rust in Firefox and related projects. As of March 2023, the lines of code in Rust represent more than 10% of the total in Firefox Nightly [12].
- At Meta, the use of Rust as a development language server-side is approved and encouraged since July 2022 [13].
- At Cloudflare, a new HTTP proxy in Rust was built from scratch to overcome the architectural limitations of NGINX, reducing CPU usage by 70% and memory usage by 67% [14].
- At Discord, reimplementing a crucial service in Rust provided great benefits in performance and solved a performance penalty due to the garbage collection in Go [15].
- At npm Inc., the company behind the npm registry, Rust allowed scaling CPU-bound services to more than 1.3 billion downloads per day [16].
- A study of Rust-based code found it runs so efficiently that it uses half as much electricity as a similar program written in Java, a language commonly used at AWS [17].

The tooling is great and works out of the box

- cargo, the official package manager: Format, build, test, lint, and update packages.

The tooling is great and works out of the box

- cargo, the official package manager: Format, build, test, lint, and update packages.
- Test harness for unit tests, integration tests, and tests in documentation comments (doctests). No third-party libraries needed.

The tooling is great and works out of the box

- cargo, the official package manager: Format, build, test, lint, and update packages.
- Test harness for unit tests, integration tests, and tests in documentation comments (doctests). No third-party libraries needed.
- An official public registry for Rust packages (called “crates”): `https://crates.io/`.

The tooling is great and works out of the box

- cargo, the official package manager: Format, build, test, lint, and update packages.
- Test harness for unit tests, integration tests, and tests in documentation comments (doctests). No third-party libraries needed.
- An official public registry for Rust packages (called “crates”): `https://crates.io/`.
- Automatic generation of a static website from the doc comments of the project. It is published automatically to `https://docs.rs/`.

The tooling is great and works out of the box

- cargo, the official package manager: Format, build, test, lint, and update packages.
- Test harness for unit tests, integration tests, and tests in documentation comments (doctests). No third-party libraries needed.
- An official public registry for Rust packages (called “crates”): `https://crates.io/`.
- Automatic generation of a static website from the doc comments of the project. It is published automatically to `https://docs.rs/`.
- An official linter included with the default installation that catches even more errors and spots non-idiomatic code: clippy.

The tooling is great and works out of the box

- cargo, the official package manager: Format, build, test, lint, and update packages.
- Test harness for unit tests, integration tests, and tests in documentation comments (doctests). No third-party libraries needed.
- An official public registry for Rust packages (called “crates”): `https://crates.io/`.
- Automatic generation of a static website from the doc comments of the project. It is published automatically to `https://docs.rs/`.
- An official linter included with the default installation that catches even more errors and spots non-idiomatic code: clippy.
- Integration with git, GitHub, VSCode, IntelliJ is great and easy to use.

The tooling is great and works out of the box

- cargo, the official package manager: Format, build, test, lint, and update packages.
- Test harness for unit tests, integration tests, and tests in documentation comments (doctests). No third-party libraries needed.
- An official public registry for Rust packages (called “crates”): `https://crates.io/`.
- Automatic generation of a static website from the doc comments of the project. It is published automatically to `https://docs.rs/`.
- An official linter included with the default installation that catches even more errors and spots non-idiomatic code: clippy.
- Integration with git, GitHub, VSCode, IntelliJ is great and easy to use.
- A new stable compiler release every 6 weeks [18].

Agenda

Agenda

Informal definition

A Petri net is a mathematical modeling tool used to describe and analyze the behavior of concurrent systems. It provides a graphical representation of the system's state and its transitions, allowing for visual and formal analysis of complex processes.

- Places: Represent states in the system (*circles*)
- Transitions: Represent usually events or actions that occur in the system (*rectangles*)
- Tokens: Marks inside of places that are created and consumed by transitions (*points inside of places*)

Mathematical definition

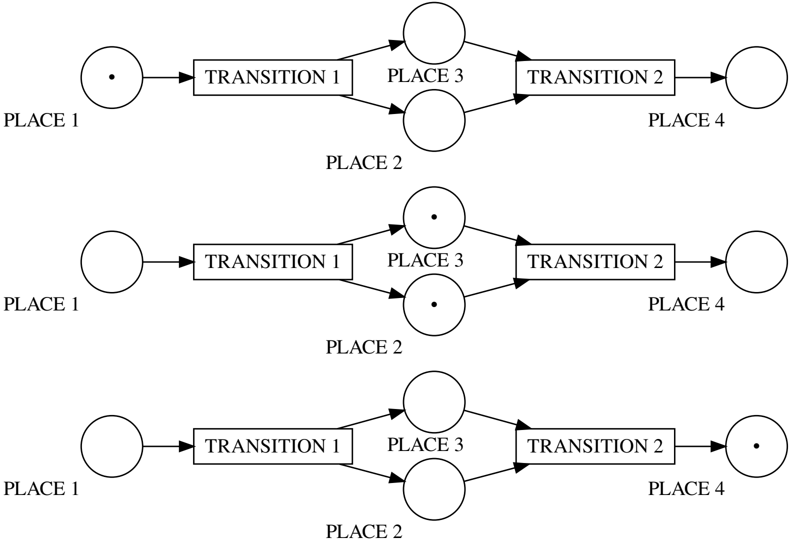
A Petri net is a 5-tuple, $PN = (P, T, F, W, M_0)$ where:

$P = \{p_1, p_2, \ldots, p_m\}$ *is a finite set of places,*
$T = \{t_1, t_2, \ldots, t_n\}$ *is a finite set of transitions,*
$F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (flow relation),*
$W : F \to \{1, 2, 3, ...\}$ *is a weight function for the arcs,*
$M_0 : P \to \{0, 1, 2, 3, ...\}$ *is the initial marking,*
$P \cap T = \varnothing$ *and* $P \cup T \neq \varnothing$

The graph is by definition *bipartite*. There can only be edges:

- from places to transitions or
- from transitions to places

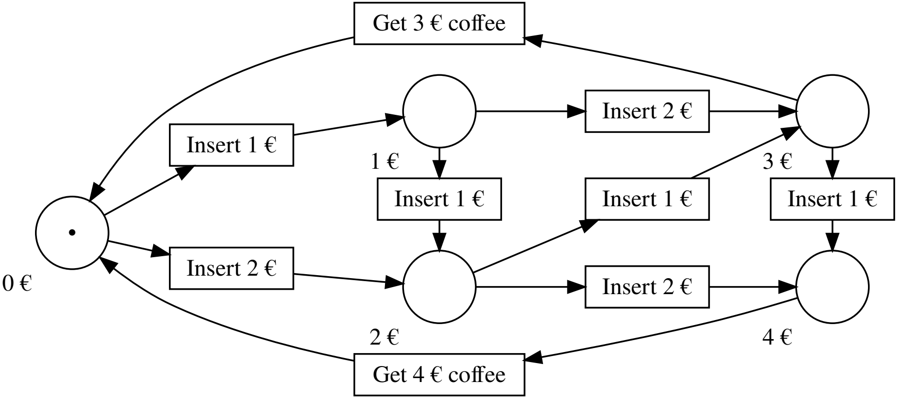
Transition firing rule



Agenda

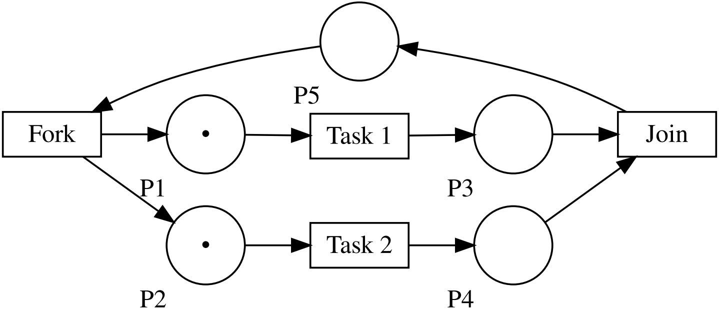
Vending machine

This is a finite-state machine (FSM), a subclass of Petri nets.



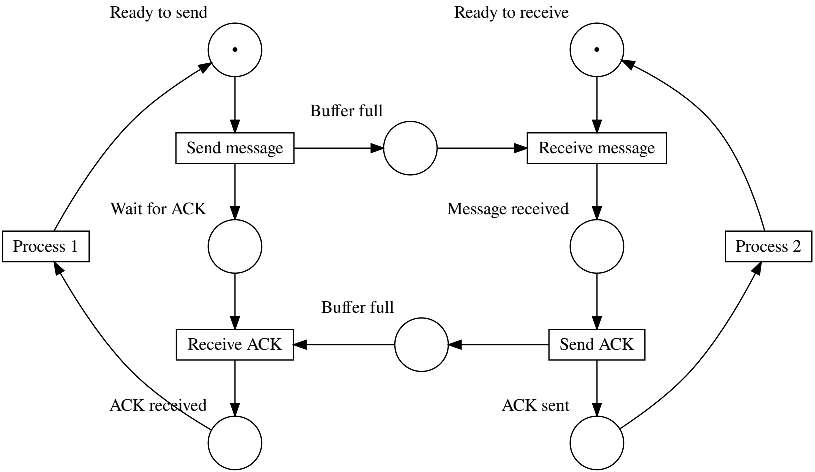
Parallel activities: Fork/Join

This is a marked graph (MG), a subclass of Petri nets. Observe the concurrency between Task 1 and Task 2. This cannot be modeled by a single finite-state machine.



Communication protocols: Send with ACK

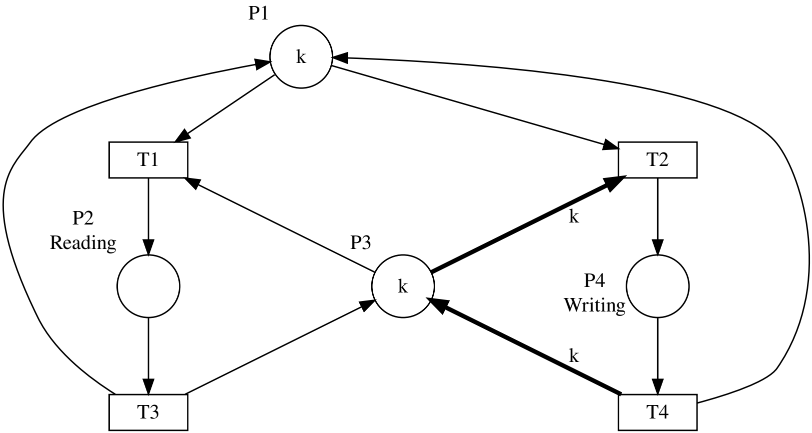
A simple protocol in which Process 1 sends messages to Process 2 and waits for an acknowledgment to be received before continuing. For simplicity, no timeout mechanism was included.



Synchronization control: Readers and writers

A Petri net system with k processes that either read or write a shared value.

- If one process writes, then no process may read.
- If a process is reading, then no process may write.
- There can only be zero or one process writing at any given time.



Agenda

Reachability analysis

Petri nets can be analyzed using formal methods to conclude whether the net can reach a deadlock or not. There is a notion of liveness analogous to the one found in computer systems.

Reachability analysis

Petri nets can be analyzed using formal methods to conclude whether the net can reach a deadlock or not. There is a notion of liveness analogous to the one found in computer systems.

Several model checkers are being developed and there is even a Model Checking Contest that takes place every year. State-of-the-art tools can handle Petri net models with more than **70 000 transitions** and **one million places**.

Reachability analysis

Petri nets can be analyzed using formal methods to conclude whether the net can reach a deadlock or not. There is a notion of liveness analogous to the one found in computer systems.

Several model checkers are being developed and there is even a Model Checking Contest that takes place every year. State-of-the-art tools can handle Petri net models with more than **70 000 transitions** and **one million places**.

Translating source code to a Petri net has been done before for other programming languages [19, 20] and also for Rust [21, 22]. The difficulty lies in supporting more synchronization primitives than simple mutexes and translating code from real-world applications.

Agenda

Compilation stages in *rustc*

- Lexing: The source text is turned into a stream of atomic source code units known as tokens.

Compilation stages in *rustc*

- Lexing: The source text is turned into a stream of atomic source code units known as tokens.
- Parsing: The stream of tokens is converted into an **Abstract Syntax Tree**.

Compilation stages in *rustc*

- Lexing: The source text is turned into a stream of atomic source code units known as tokens.
- Parsing: The stream of tokens is converted into an **Abstract Syntax Tree**.
- High-level Intermediate Representation (HIR):
  - Desugar loops: `while` and `for` to simple `loop`.
  - Type inference: The automatic detection of a type of an expression.
  - Trait solving: Ensuring that each implementation block (`impl`) points to a valid trait.
  - Type checking.

Compilation stages in *rustc*

- Lexing: The source text is turned into a stream of atomic source code units known as tokens.
- Parsing: The stream of tokens is converted into an **Abstract Syntax Tree**.
- High-level Intermediate Representation (HIR):
  - Desugar loops: `while` and `for` to simple `loop`.
  - Type inference: The automatic detection of a type of an expression.
  - Trait solving: Ensuring that each implementation block (`impl`) points to a valid trait.
  - Type checking.
- Mid-level Intermediate Representation (MIR):
  - Checking of exhaustiveness of pattern matching.
  - Desugar method calls to function calls (`x.method(y)` becomes `Type::method(&x, y)`).
  - Add implicit dereferencing operations.
  - Borrow checking.

Compilation stages in *rustc*

- Lexing: The source text is turned into a stream of atomic source code units known as tokens.
- Parsing: The stream of tokens is converted into an **Abstract Syntax Tree**.
- High-level Intermediate Representation (HIR):
  - Desugar loops: `while` and `for` to simple `loop`.
  - Type inference: The automatic detection of a type of an expression.
  - Trait solving: Ensuring that each implementation block (`impl`) points to a valid trait.
  - Type checking.
- Mid-level Intermediate Representation (MIR):
  - Checking of exhaustiveness of pattern matching.
  - Desugar method calls to function calls (`x.method(y)` becomes `Type::method(&x, y)`).
  - Add implicit dereferencing operations.
  - Borrow checking.
- Code generation:
  - Rust uses LLVM for the backend.
  - It leverages many optimizations of the LLVM intermediate representation.
  - LLVM takes over from this point on.
  - At the end object files are linked to create an executable.

Agenda

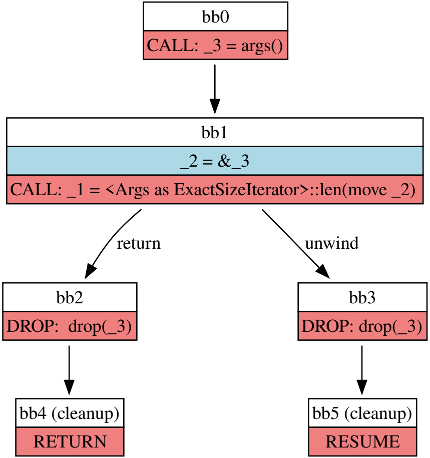
Hello World in MIR

BB means “basic block”. Each one is formed by statements and one terminator statement. The terminator statement is the only place where the control flow can jump to another basic block.

```
fn main() -> () {
    let mut _0: ();
    let _1: ();
    let mut _2: std::fmt::Arguments<'_>;
    let mut _3: &[&str];
    let mut _4: &[&str; 1];

    bb0: {
        _4 = const _;
        _3 = _4 as &[&str] (Pointer(Unsize));
        _2 = Arguments::<'_>::new_const(move _3) -> bb1;
    }

    bb1: {
        _1 = _print(move _2) -> bb2;
    }

    bb2: {
        return;
    }
}
```

MIR as a graph that shows the flow of execution

The MIR is a form of control flow graph (CFG) used in compilers. In this form, the translation to a Petri net becomes evident.



Agenda

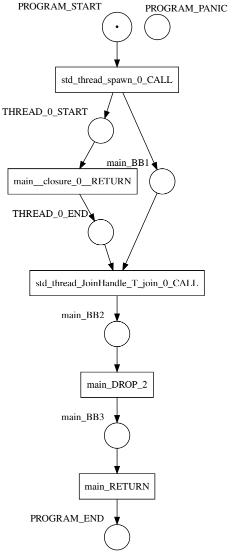
Example program

Let’s consider a trivial program that spawns a thread that does nothing and immediately joins it.

```
fn main() {
  let thread_join_handle = std::thread::spawn(move || {
      // some work here
  });
  // some work here
  let _res = thread_join_handle.join();
}
```

- `std::thread::spawn` should create an additional token that models the program counter of the second thread.
- The joining thread should wait until the spawned thread finishes.

Petri net model for a thread



Agenda

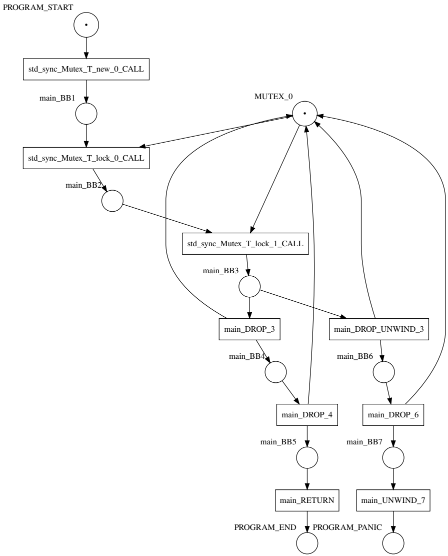
Example program

Consider a simple program that locks a mutex twice. The second lock operation will deadlock because the lock handle returned by the first call to `std::sync::Mutex::lock` is not dropped until it falls out of scope.

```
fn main() {
  let data = std::sync::Mutex::new(0);
  let _d1 = data.lock();
  let _d2 = data.lock(); // cannot lock, since d1 is still active
}
```

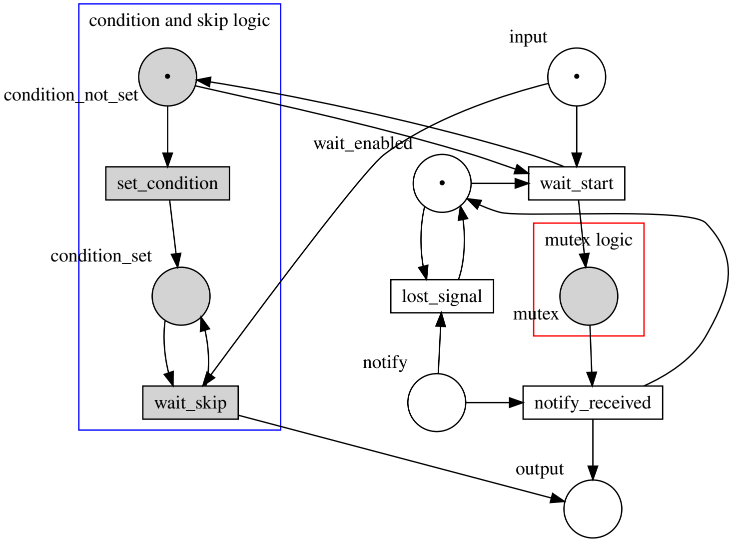
- There should be a single place that models the mutex.
- Locking the mutex is taking the token from the mutex place.
- Unlocking the mutex is setting the token back in the mutex place.

Petri net model for a mutex



Agenda

How to model a condition variable



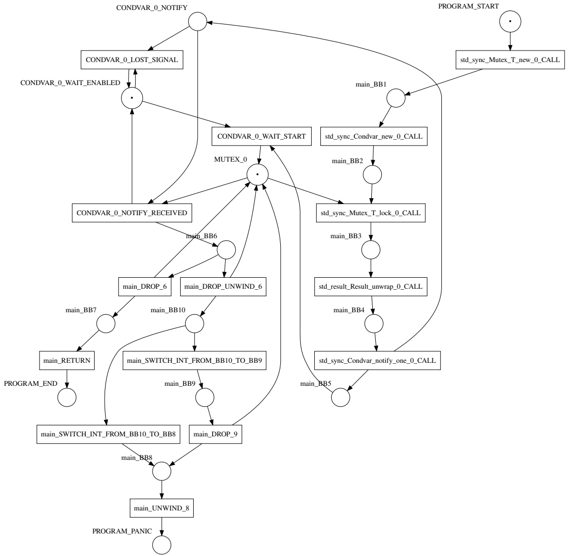
Example program

We have to use a very simple example program to keep the net small. In this case, the thread is trying to notify itself, which leads to a lost signal.

```
fn main() {
  let mutex = std::sync::Mutex::new(false);
  let cvar = std::sync::Condvar::new();
  let mutex_guard = mutex.lock().unwrap();
  cvar.notify_one();
  let _result = cvar.wait(mutex_guard);
}
```

- The model for the condition variable should appear in the Petri net.
- The notify place should be set.
- But the signal gets consumed because `std::sync::Condvar::wait` was not called.

Petri net model for the example program



Agenda

Bibliography I

E. Stepanov, “Detecting Memory Corruption Bugs With HWASan.” https://android-developers.googleblog.com/2020/02/detecting-memory-corruption-bugs-with-hwasan.html, 2020. Accessed on 2023-02-24.

J. V. Stoep and C. Zhang, “Queue the Hardening Enhancements.” https://android-developers.googleblog.com/2020/02/detecting-memory-corruption-bugs-with-hwasan.html, 2020. Accessed on 2023-02-24.

The Chromium Projects, “Memory safety.” https://www.chromium.org/Home/chromium-security/memory-safety/, 2015. Accessed on 2023-02-24.

D. Hosfelt, “Implications of Rewriting a Browser Component in Rust.” https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/, 2019. Accessed on 2023-02-24.

P. Kehrer, “Memory Unsafety in Apple’s Operating Systems.” https://langui.sh/2019/07/23/apple-memory-safety/, 2019. Accessed on 2023-02-24.

M. Miller, “Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape.” https://www.youtube.com/watch?v=PjbGojjnBZQ, 2019. Accessed on 2023-02-24.

S. Fernandez, “A proactive approach to more secure code.” https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/, 2019. Accessed on 2023-02-24.

Bibliography II

A. Gaynor, “What science can tell us about C and C++’s security.”
https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/, 2020.
Accessed on 2023-02-24.

J. V. Stoep and S. Hines, “Rust in the Android platform.”
https://security.googleblog.com/2021/04/rust-in-android-platform.html, 2021.
Accessed on 2023-02-22.

J. Corbet, “The 6.1 kernel is out.” https://lwn.net/Articles/917504/, 2022.
Accessed on 2023-02-24.

S. D. Simone, “Linux 6.1 Officially Adds Support for Rust in the Kernel.”
https://www.infoq.com/news/2022/12/linux-6-1-rust/, 2022.
Accessed on 2023-02-24.

Mozilla Wiki, “Oxidation Project.” https://wiki.mozilla.org/Oxidation, 2015.
Accessed on 2023-03-20.

E. Garcia, “Programming languages endorsed for server-side use at Meta.” https://engineering.fb.com/2022/07/27/developer-tools/programming-languages-endorsed-for-server-side-use-at-meta/, 2022.
Accessed on 2023-02-24.

Y. Wu and A. Hauck, “How we built Pingora, the proxy that connects Cloudflare to the Internet.” https://blog.cloudflare.com/how-we-built-pingora-the-proxy-that-connects-cloudflare-to-the-internet/, 2022.
Accessed on 2023-03-20.

Bibliography III

J. Howarth, "Why Discord is switching from Go to Rust."
https://discord.com/blog/why-discord-is-switching-from-go-to-rust, 2020.
Accessed on 2023-03-20.

The Rust Project Developers, "Rust case study: Community makes rust an easy choice for npm," 2019.
https://www.rust-lang.org/static/pdfs/Rust-npm-Whitepaper.pdf.

R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Energy efficiency across programming languages: how do energy, time, and memory relate?," in *Proceedings of the 10th ACM SIGPLAN international conference on software language engineering*, pp. 256–267, 2017.

P. Albini, "RustFest Barcelona - Shipping a stable compiler every six weeks."
https://www.youtube.com/watch?v=As1gXp5kX1M, 2019.
Accessed on 2023-02-24.

K. M. Kavi, A. Moshtaghi, and D.-J. Chen, "Modeling multithreaded applications using petri nets," *International Journal of Parallel Programming*, vol. 30, pp. 353–371, 2002.

A. Moshtaghi, "Modeling Multithreaded Applications Using Petri Nets," Master's thesis, The University of Alabama in Huntsville, 2001.

T. Meyer, "A Petri Net semantics for Rust," Master's thesis, Universität Rostock — Fakultät für Informatik und Elektrotechnik, 2020.
https://github.com/Skasselbard/Granite/blob/master/doc/MasterThesis/main.pdf.

K. Zhang and G. Liua, "Automatically transform rust source to petri nets for checking deadlocks," *arXiv preprint arXiv:2212.02754*, 2022.

Agenda

Resources to learn Rust

- The Rust Book: Available online and locally with the default Rust installation.
- Rust by Example: Another official book with a more practical approach.
- Rustlings: Small exercises to get you used to reading and writing Rust code!
- Comprehensive Rust: A three-day Rust course developed by the Android team.
- Take your first steps with Rust: A simple course on Microsoft Learn.
- Rust Programming Course for Beginners by freeCodeCamp.org.
- No Boilerplate: A Youtube channel mainly dedicated to topics connected with Rust. Some ideas were used for this presentation.

Online Petri net simulators

- A simple simulator by Igor Kim can be found on https://petri.hp102.ru/. A tutorial video on Youtube and example nets are included in the tool.
- A complement to this is a series of interactive tutorials by Prof. Wil van der Aalst at the University of Hamburg. These tutorials are Adobe Flash Player files (with extension `.swf`) that modern web browsers cannot execute. Luckily, an online Flash emulator like the one found on https://flashplayer.fullstacks.net/?kind=Flash_Emulator can be used to upload the files and execute them.
- Another online Petri net editor and simulator is http://www.biregal.com/. The user can draw the net, add the tokens, and then manually fire transitions.

Questions?

Links

Thesis	`https://github.com/hlisdero/thesis`
Tool	`https://github.com/hlisdero/cargo-check-deadlock`
Prasentation	`https://github.com/hlisdero/thesis/tree/main/presentation`
Published crate	`https://crates.io/crates/cargo-check-deadlock`