Miri

An interpreter for Rust's mid-level intermediate representation

Scott Olson
Supervisor: Christopher Dutchyn

CMPT 400
University of Saskatchewan

`https://www.rust-lang.org`

What is Rust? [review]

According to the website…

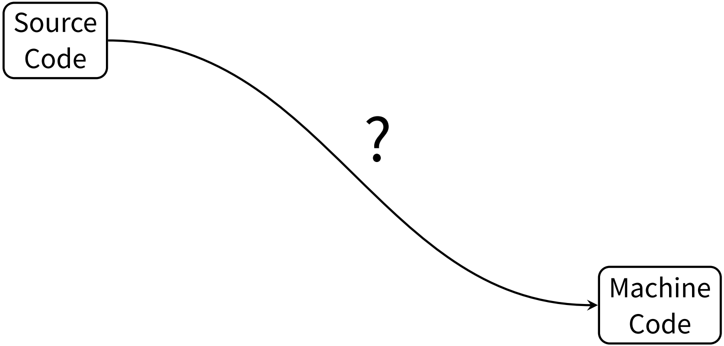
> ***Rust** is a systems programming language that runs blazingly fast, prevents nearly all segfaults, and guarantees thread safety.*

It's a new programming language from Mozilla, and it looks like this:

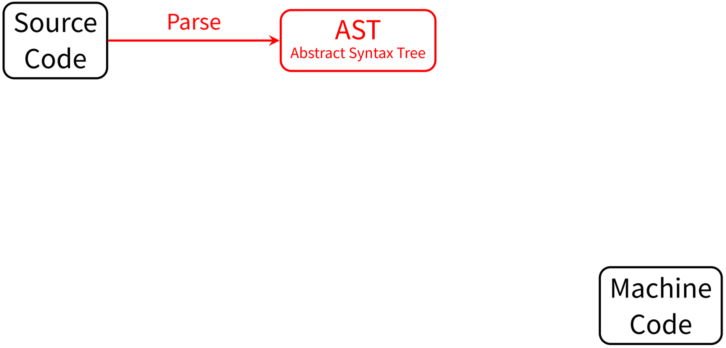
```
fn factorial(n: u64) -> u64 {
    (1..n).fold(1, |a, b| a * b)
}

fn main() {
    for x in 1..6 {
        println!("{}", factorial(x));
    }
    // ⇒ 1
    // ⇒ 1
    // ⇒ 2
    // ⇒ 6
    // ⇒ 24
}
```

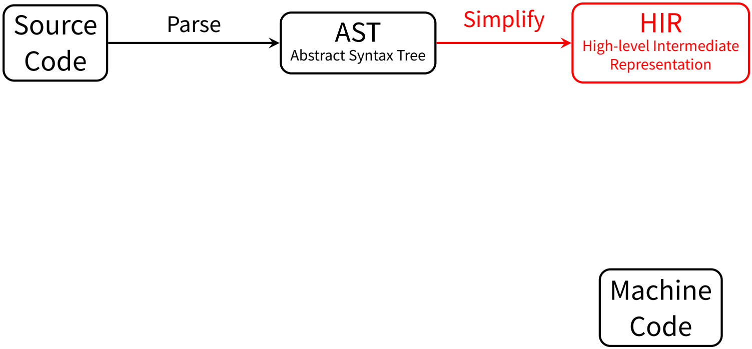
How does Rust compile code? [review]



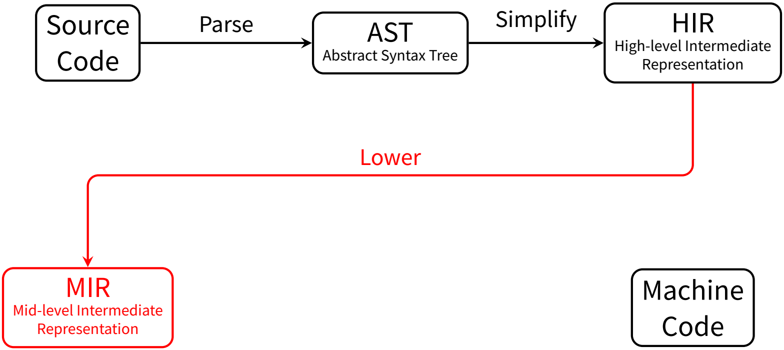
How does Rust compile code? [review]



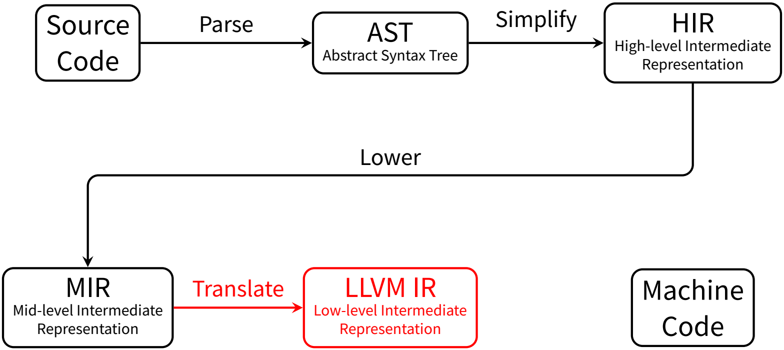
How does Rust compile code? [review]



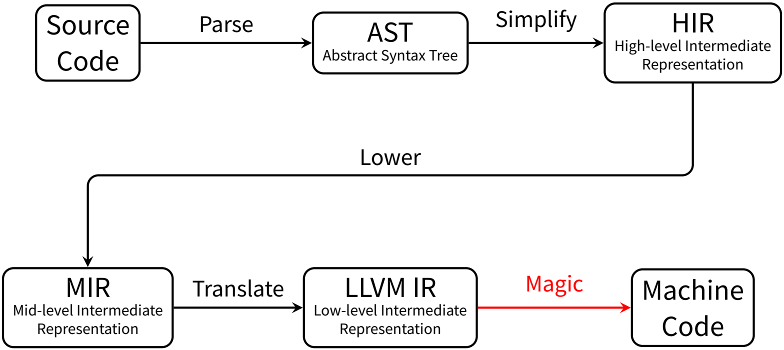
How does Rust compile code? [review]



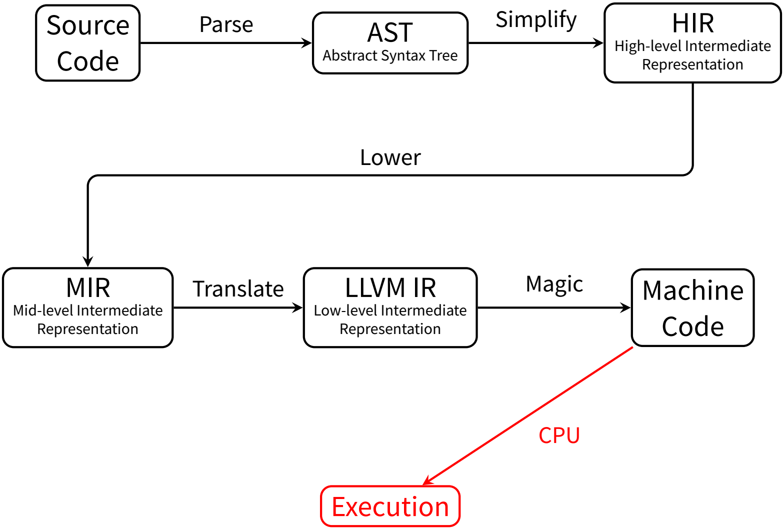
How does Rust compile code? [review]



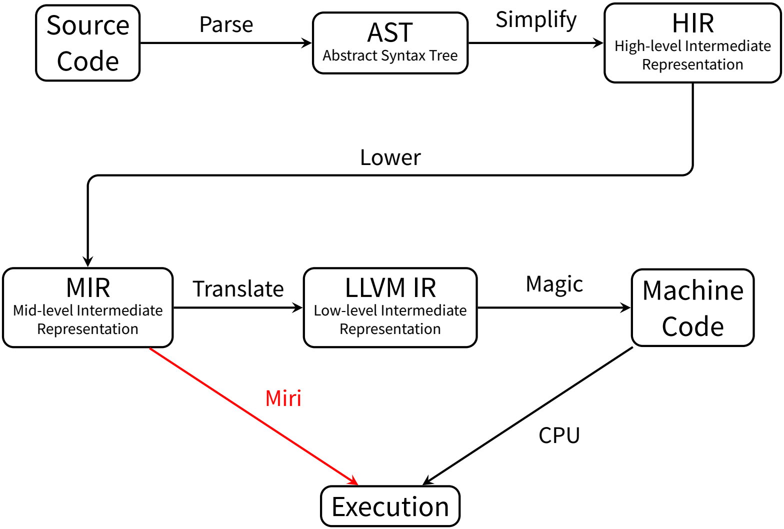
How does Rust compile code? [review]



How does Rust compile code?



How does Rust compile code?



Why build Miri?

- For fun and learning.
- I originally planned to use it for testing the compiler and execution of unsafe code, but shifted my goals along the way.

Why build Miri?

- For fun and learning.
- I originally planned to use it for testing the compiler and execution of unsafe code, but shifted my goals along the way.
- Now it serves as an experimental implementation of the upcoming compile-time function evaluation feature in Rust.

Why build Miri?

- For fun and learning.
- I originally planned to use it for testing the compiler and execution of unsafe code, but shifted my goals along the way.
- Now it serves as an experimental implementation of the upcoming compile-time function evaluation feature in Rust.
  - Similar to C++14's `constexpr` feature.
  - You can do complicated calculations at compile time and compile their *results* into the executable.

Why build Miri?

- For fun and learning.
- I originally planned to use it for testing the compiler and execution of unsafe code, but shifted my goals along the way.
- Now it serves as an experimental implementation of the upcoming compile-time function evaluation feature in Rust.
  - Similar to C++14's `constexpr` feature.
  - You can do complicated calculations at compile time and compile their *results* into the executable.
  - For example, you can compute a "perfect hash function" for a statically-known map at compile-time and have guaranteed no-collision lookup at runtime.

Why build Miri?

- For fun and learning.
- I originally planned to use it for testing the compiler and execution of unsafe code, but shifted my goals along the way.
- Now it serves as an experimental implementation of the upcoming compile-time function evaluation feature in Rust.
  - Similar to C++14's `constexpr` feature.
  - You can do complicated calculations at compile time and compile their *results* into the executable.
  - For example, you can compute a "perfect hash function" for a statically-known map at compile-time and have guaranteed no-collision lookup at runtime.
  - Miri actually supports far more of Rust than C++14's `constexpr` does of C++ — even heap allocation and unsafe code.

How was it built?

At first I wrote a naive version with a number of downsides:

- represented values in a traditional dynamic language format, where every value was the same size.
- didn't work well for aggregates (structs, enums, arrays, etc.).
- made unsafe programming tricks that make assumptions about low-level value layout essentially impossible.

How was it built?

- Later, a Rust compiler team member proposed a "Rust abstract machine" with specialized value layout which solved my previous problems.

How was it built?

- Later, a Rust compiler team member proposed a “Rust abstract machine” with specialized value layout which solved my previous problems.
- His proposal was intended for a compile-time function evaluator in the Rust compiler, so I effectively implemented an experimental version of that.

How was it built?

- Later, a Rust compiler team member proposed a “Rust abstract machine” with specialized value layout which solved my previous problems.
- His proposal was intended for a compile-time function evaluator in the Rust compiler, so I effectively implemented an experimental version of that.
- After this point, making Miri work well was primarily a software engineering problem.

Data layout

- Memory in Miri is literally a HashMap from “allocation IDs” to “abstract allocations”.
- Allocations are represented by:

Data layout

- Memory in Miri is literally a HashMap from “allocation IDs” to “abstract allocations”.
- Allocations are represented by:
    1. An array of **raw bytes** with a size based on the type of the value

Data layout

- Memory in Miri is literally a HashMap from “allocation IDs” to “abstract allocations”.
- Allocations are represented by:
  1. An array of **raw bytes** with a size based on the type of the value
  2. A set of **relocations** — pointers into other abstract allocations

Data layout

- Memory in Miri is literally a HashMap from “allocation IDs” to “abstract allocations”.
- Allocations are represented by:
  1. An array of **raw bytes** with a size based on the type of the value
  2. A set of **relocations** — pointers into other abstract allocations
  3. A mask determining which bytes are **undefined**

square example

```
// Rust
fn square(n: u64) -> u64 {
    n * n
}

// Generated MIR
fn square(arg0: u64) -> u64 {
    let var0: u64; // n          // On function entry, Miri creates
                                 // virtual allocations for all the
                                 // arguments, variables, and
                                 // temporaries.

    bb0: {
        var0 = arg0;             // Copy the argument into `n`.
        return = Mul(var0, var0); // Multiply `n` with itself.
        goto -> bb1;             // Jump to basic block `bb1`.
    }

    bb1: {
        return;                  // Return from the current fn.
    }
}
```

sum example

```
// Rust
fn sum() -> u64 {
    let mut sum = 0; let mut i = 0;
    while i < 10 { sum += i; i += 1; }
    sum
}

// Generated MIR
fn sum() -> u64 {
    let mut var0: u64; // sum
    let mut var1: u64; // i
    let mut tmp0: bool;

    bb0: {
        // sum = 0; i = 0;
        var0 = const 0u64; var1 = const 0u64; goto -> bb1;
    }
    bb1: {
        // if i < 10 { goto bb2; } else { goto bb3; }
        tmp0 = Lt(var1, const 10u64);
        if(tmp0) -> [true: bb2, false: bb3];
    }
    bb2: {
        var0 = Add(var0, var1);        // sum = sum + i;
        var1 = Add(var1, const 1u64); // i = i + 1;
        goto -> bb1;
    }
    bb3: {
        return = var0; goto -> bb4;
    }
    bb4: { return; }
}
```

Heap allocations!

```
fn make_vec() -> Vec<u8> {
    // Empty array with space for 4 bytes - allocated on the heap!
    let mut vec = Vec::with_capacity(4);
    // Initialize the first two slots.
    vec.push(1);
    vec.push(2);
    vec
}

// For reference:
//   struct Vec<T> { capacity: usize, data: *mut T, length: usize }

// Resulting allocations (on 32-bit little-endian architectures):
//   Region A:
//     04 00 00 00  00 00 00 00  02 00 00 00
//                  └───(B)───┘
//
//   Region B:
//     01 02 __ __ (underscores denote undefined bytes)
```

Evaluating the above involves a number of compiler built-ins, “unsafe” code blocks, and more inside the standard library, but Miri handles it all.

Unsafe code!

```
fn out_of_bounds() -> u8 {
    let mut vec = vec![1, 2]
    unsafe { *vec.get_unchecked(5) }
}

// test.rs:3: error: pointer offset outside bounds of allocation
// test.rs:3:     unsafe { *vec.get_unchecked(5) }
//                         ^~~~~~~~~~~~~~~~~~~~~

fn undefined_bytes() -> u8 {
    let mut vec = Vec::with_capacity(10);
    unsafe { *vec.get_unchecked(5) }
}

// test.rs:3: error: attempted to read undefined bytes
// test.rs:3:     unsafe { *vec.get_unchecked(5) }
//                         ^~~~~~~~~~~~~~~~~~~~~
```

What can’t Miri do?

- Miri can’t do all the stuff I didn’t implement yet. :)
  - non-trivial casts
  - function pointers
  - calling destructors and freeing memory
  - taking target architecture endianess and alignment information into account when computing data layout
  - handling all constants properly (but, well, Miri might be replacing the old constants system)

What can't Miri do?

- Miri can't do all the stuff I didn't implement yet. :)
  - non-trivial casts
  - function pointers
  - calling destructors and freeing memory
  - taking target architecture endianess and alignment information into account when computing data layout
  - handling all constants properly (but, well, Miri might be replacing the old constants system)
- Miri can't do foreign function calls (e.g. calling functions defined in C or C++), but there is a reasonable way it could be done with libffi.
  - On the other hand, for constant evaluation in the compiler, you want the evaluator to be deterministic and safe, so FFI calls might be banned anyway.

What can't Miri do?

- Miri can't do all the stuff I didn't implement yet. :)
  - non-trivial casts
  - function pointers
  - calling destructors and freeing memory
  - taking target architecture endianess and alignment information into account when computing data layout
  - handling all constants properly (but, well, Miri might be replacing the old constants system)
- Miri can't do foreign function calls (e.g. calling functions defined in C or C++), but there is a reasonable way it could be done with libffi.
  - On the other hand, for constant evaluation in the compiler, you want the evaluator to be deterministic and safe, so FFI calls might be banned anyway.
- Without quite some effort, Miri will probably never handle inline assembly...

Questions?

varN vs. argN

```
// Rust
type Pair = (u64, u64);
fn swap((a, b): Pair) -> Pair {
    (b, a)
}

// Generated MIR
fn swap(arg0: (u64, u64)) -> (u64, u64) {
    let var0: u64; // a
    let var1: u64; // b

    bb0: {
        var0 = arg0.0;          // get the 1st part of the pair
        var1 = arg0.1;          // get the 2nd part of the pair
        return = (var0, var1); // build a new pair in the result
        goto -> bb1;
    }

    bb1: {
        return;
    }
}
```

factorial example

```
// Rust
fn factorial(n: u64) -> u64 {
    (1..n).fold(1, |a, b| a * b)
}

// Generated MIR
fn factorial(arg0: u64) -> u64 {
    let var0: u64; // n
    let mut tmp0: Range<u64>; // Miri calculates sizes for generics like Range<u64>.
    let mut tmp1: [closure];

    bb0: {
        var0 = arg0;

        // tmp0 = 1..n
        tmp0 = Range<u64> { start: const 1u64, end: var0 };

        // tmp1 = |a, b| a * b
        tmp1 = [closure];

        // This loads the MIR for the `fold` fn from the standard library.
        // In general, MIR for any function from any library can be loaded.
        // return tmp0.fold(1, tmp1)
        return = Range<u64>::fold(tmp0, const 1u64, tmp1) -> bb1;
    }

    bb1: {
        return;
    }
}
```