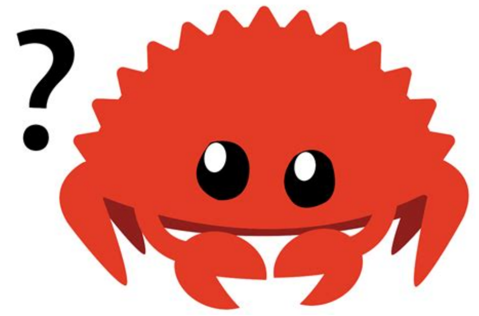
Unsafe Rust

Rust, In Theory and in Practice

CAS CS 392 M1

Motivation




```
fn swap<T>(a: &mut T, b: &mut T) {
    let temp = *a;
    *a = *b;
    *b = temp;
}
```

We sometimes need a way of *shifting* some of the analysis from the compiler to the programmer because:

- "static analysis is conservative"
- "Computer hardware is inherently unsafe"
- FFI is hard

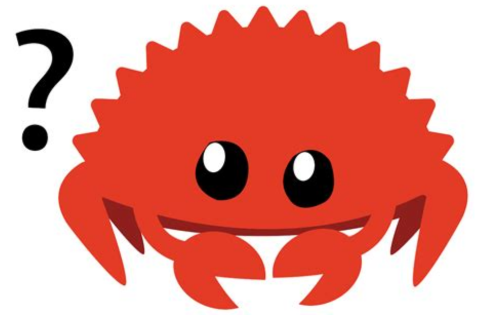
Disclaimer

Most people I've talked to don't use unsafe Rust

It matters a lot in some cases, but it also doesn't matter at all in most cases

Why care? The interface is fascinating at a PL/compilers level (and sometimes maybe you'll need it)

Common Misconceptions




```
fn swap<T>(a: &mut T, b: &mut T) {
    unsafe {
        let temp = *a;
        *a = *b;
        *b = temp;
    }
}
```

*unsafe Rust is when you **turn off** the borrow checker*

Using unsafe code in a function makes the whole function unsafe

(to be clear, these are *not* true)

Unsafe is Contractual

```rust
pub fn push_mut(&mut self, value: T) -> &mut T {
    // Inform codegen that the length does not change across grow_one().
    let len = self.len;
    // This will panic or abort if we would allocate > isize::MAX bytes
    // or if the length increment would overflow for zero-sized types.
    if len == self.buf.capacity() {
        self.buf.grow_one();
    }
    unsafe {
        let end = self.as_mut_ptr().add(len);
        ptr::write(end, value);
        self.len = len + 1;
        // SAFETY: We just wrote a value to the pointer that will live the lifetime of the reference.
        &mut *end
    }
}
```

safety claim

Writing unsafe Rust means holding up a contract

You'll be using operations that could break memory safety, and you're promising to verify using you're own brain (instead of the compiler) that it doesn't

"*trust me, I know what I'm doing*"

Undefined Behavior (UB)

> ⚠ **Warning**
>
> The following list is not exhaustive; it may grow or shrink. There is no formal model of Rust's semantics for what is and is not allowed in unsafe code, so there may be more behavior considered unsafe. We also reserve the right to make some of the behavior in that list defined in the future. In other words, this list does not say that anything will *definitely* always be undefined in all future Rust version (but we might make such commitments for some list items in the future).

When you write unsafe Rust, you promise not to cause *undefined behavior*

What is UB? *Who the hell knows...*

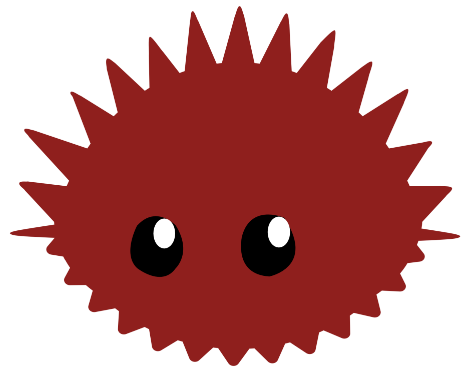
There's a list, but Rust is a bit cagey. There are a couple obvious ones:

» Data races

» accessing a dangling pointer

» aliasing mutable references

"Superpowers"



```rust
fn swap<T>(a: &mut T, b: &mut T) {
    let a_ptr = a as *mut T;
    let b_ptr = b as *mut T;
    unsafe {
        let temp = std::ptr::read(a_ptr);
        *a_ptr = std::ptr::read(b_ptr);
        *b_ptr = temp;
    }
}
```

Writing unsafe Rust means writing in an **unsafe** block. Within an unsafe block you get a couple "superpowers", the two primary ones:

1. Dereference a **raw pointer**

2. Call other unsafe functions

Raw pointers

```rust
let mut x = 1;
let y: *const i32 = &x; // coercion
let y: *mut i32 = &mut x;
let y = &raw const x; // raw borrow op
let y = &raw mut x;
let mut x = Box::new(1);
let y: *mut i32 = Box::into_raw(x);
unsafe { let _ = Box::from_raw(y); } // need to drop
```

Raw pointers are like references except that they:

» ignore borrow rules

» may be unaligned or out of bounds (may not point to valid memory)

» are allowed to be null

» do not have automatic clean-up

Example: Derefing a raw pointer

```
let mut x = 2;
let y = &raw mut x;
unsafe {
    *y += 1;
}
assert_eq!(x, 3);
```

Reading through a raw pointer requires an **unsafe** block

Mutating through a mutable raw pointer requires an **unsafe** block

(Note that creating a raw pointer does not require an **unsafe** block)

Example: Calling Unsafe Functions

Functions are labeled unsafe if they have the potential of causing undefined behavior if used incorrectly

It's a contract, you need to check that all the safety requirements are satisfied

Function from_raw_parts_mut

Since 1.0.0 (const: 1.83.0) · Source

```
pub const unsafe fn from_raw_parts_mut<'a, T>(
    data: *mut T,
    len: usize,
) -> &'a mut [T]
```

Performs the same functionality as `from_raw_parts`, except that a mutable slice is re

Safety

Behavior is undefined if any of the following conditions are violated:

- `data` must be non-null, valid for both reads and writes for `len * size_of::<T>` aligned. This means in particular:
  - The entire memory range of this slice must be contained within a single allocatio allocations.
  - `data` must be non-null and aligned even for zero-length slices or slices of ZSTs. optimizations may rely on references (including slices of any length) being align other data. You can obtain a pointer that is usable as `data` for zero-length slices
- `data` must point to `len` consecutive properly initialized values of type `T`.

demo

(basic examples)

Safe Abtractions

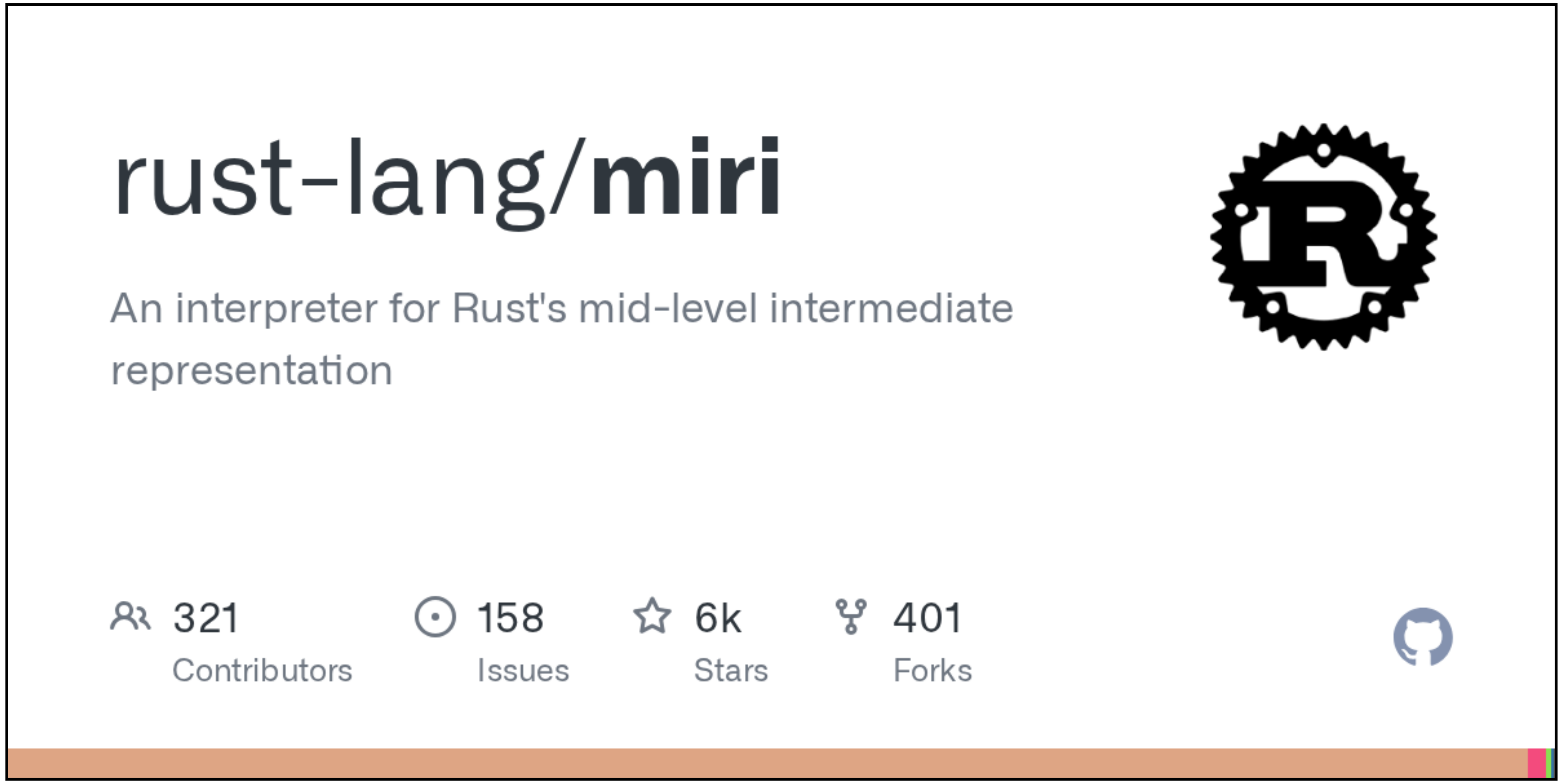
Again, using **unsafe** does not make the code you write unsafe

It just means it's *on you* to make sure it's safe

demo

(safe abstractions: example from the book)

Miri



`Miri is used to` **`detect undefined behavior`**

`(see, interpreters aren't useless)`

demo

(swap)

Workshop

Task

- Build your own version of **ChunkMut** (we'll talk a bit about this first) *Can we use **split_at_mut**? Can we do it without unsafe rust?*

- Read Aria Desires's chapter on Miri from **Learn Rust With Entirely Too Many Linked Lists**