Concurrency in Rust

Riccardo Sieve

November 5, 2025

University of Oslo

Outline

- Background on Types for Concurrent Systems
- Memory Management in Rust
- Threads in Rust
- Message Passing in Rust

Background

Types for Concurrent Systems

- So far: main ideas on theory of types
- Tool support external to language and standard library
  - Developers mixing libraries lose guarantees
  - Maintenance of libraries becomes a problem

Practical Types for Concurrent Systems

Rust, and to a smaller degree Go, have practical implementations

- Motivated by memory safety and concurrency
- Session types still external, but Rust library most sensible

Remark

Ownership types (introduced by David Clarke) are types for object oriented languages with similar basic ideas. Rust ownership system is not based on Clarke's ownership types.

Connecting Syntax and Semantics

There are several ideas and angles that connect syntax and semantics for memory and reference management.

- Linear types (deallocation after use)
- Aliasing (in OO)
- RAII (Resource Acquisition Is Initialization) and RBMM (Region-Based Memory Management)

Aliasing

Aliasing occurs if multiple references to one value/object exist

- Makes reasoning about the program more difficult
- Especially in concurrency: is there another *active* reference?
- Separates (semantic) value from (syntactic) variable

Connecting Syntax and Semantics

Region/Scope Based Memory Management (RBMM)

- RAII is mostly used to refer to OO, RBMM is more general
- Associate lexically-scoped part of the program with a *region* in the heap
- Region deallocated once scope exited
- Type checker ensures that no external pointers into region exist

Shared Themes

Linear types, uniqueness types, RBMM and alias analyses relate values, their lifetime at run-time and the syntactic structure of the program.

- Keep track of number of possible (types) references to reason about concurrent operations
- Prevent general errors from faulty memory management

Memory Management in Rust

Rust combines many ideas to guarantee memory and thread-safety, as well as static memory management without garbage collection

Ownership is how Rust manages memory

Ownership In Rust

- Each value (a String, i32, Vec, etc.) is owned by a single variable, called *owner*.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

Ownership in Rust

- Rust uses ownership to manage memory.
- This helps prevent data races at compile time.
- In other programming languages, memory is managed through garbage collection.
- In Rust, memory is managed through a set of ownership that enforces rules at compile time.
    - You can have only one owner for each value.
    - You can borrow a value through references, either mutably or immutably.
    - You can either have one mutable reference or many immutable references to a value at a time.
- An owner is usually a variable, but it can also be a data structure that contains other values.

Example in Rust

Does the following snippet of code compile?

Rust

```rust
fn calculate_length(s: String) -> usize {
    s.len()
}

fn main() {
    let s1 = String::from("Rust ownership");
    let s2 = s1;
    let len = calculate_length(s2);

    println!("The length of '{}' is {}", s1, len);
}
```

Ownership in Rust

- The previous code does not compile.
- We defined correctly the string `s1`
- However, when we assign `s1` to `s2`, the ownership of the string is moved to `s2`
- This means that `s1` is no longer valid and cannot be used
- If we try to print `s1` after the assignment, we will get a compile-time error
- How can we fix it?

Solution 1: using references

Rust

```rust
fn calculate_length(s: &String) -> usize {
    s.len()
}

fn main() {
    let s1 = String::from("Rust ownership");
    let s2 = &s1; // we only assign the reference
    let len = calculate_length(s2);

    println!("The length of '{}' is {}", s1, len);
}
```

Solution 2: using clone

Rust

```rust
fn calculate_length(s: String) -> usize {
    s.len()
}

fn main() {
    let s1 = String::from("Rust ownership");
    let s2 = s1.clone(); // we clone s1
    let len = calculate_length(s2);

    println!("The length of '{}' is {}", s1, len);
}
```

Mutability and Immutability in Rust

- Even if we pass the reference to the string, it is still immutable.
- Trying to append a string, we will get a compile-time error.
- If we want to change the value of `s1`, we need to make it mutable.
- This must be done declaring the variable with `mut`.

Example in Rust

Does the following snippet of code compile?

Rust

```rust
fn append_crab(s: &String) {
    s.push_str("Ferris was here");
}

fn main() {
    let mut s = String::from("Rust is cool! ");
    append_crab(&s);

    println!("{}", s);
}
```

Mutability and Immutability in Rust

- By default, variables are immutable in Rust.
- When we defined the string `s1`, it is immutable by default.
- If we want to change the value of `s1`, we need to make it mutable.
- This must be done at the time of declaration.

Solution: use mut

Using mut we make the string mutable, so we can change its value.

Rust

```rust
fn append_crab(s: &mut String) {
    s.push_str("Ferris was here");
}

fn main() {
    let mut s = String::from("Rust is cool! ");
    append_crab(&mut s);

    println!("{}", s);
}
```

Lifetime

- A reference/variable has a lifetime from until it goes out-of-scope.
- One cannot have a reference with a longer lifetime than the value it refers to

Lifetime not respected due to scoping

Rust

```rust
fn main() {
        let ref_vec;                                //---+
        {                                           //   |
                let vec = vec![1, 2, 3];            //-+ |
                ref_vec = &vec;                     // | |
        }                                           //-+ |
        println!("{}", (*ref_vec)[0]);              //   |
}                                                   //---+
```

Lifetime

- A reference/variable has a lifetime from until it goes out-of-scope.
- One cannot have a reference with a longer lifetime than the value it refers to

Lifetime respected

Rust

```
fn main() {
        let vec = vec![1, 2, 3];        //---+
        let refVec = &vec;              //-+ |
        println!("{}", (*refVec)[0]);   // | |
}                                       //-+-+
```

Referencing in Rust

A reference to a value cannot outlive the owner

Rust

```rust
let v = vec![1 , 2];
let x=&v[0] ;
let v2=v ;
let y = *x +1 // ERROR - x refers to v, but v is dead
```

A value can have one mutable reference or many immutable references

Rust

```rust
let mut  v = vec![1 , 2];
let x=&v[0];   //immutable borrow here
Vec :: push (&mut v , 3); // ERROR: mutable borrow here
let y = *x +1; // x still alive
```

Threads in Rust

- We can use multi-threading applications in Rust to improve performance.
- This can lead to some issues
    - Race conditions — multiple threads accessing shared data simultaneously
    - Deadlocks — two or more threads waiting for each other to release resources
    - Bugs that are hard to reproduce
- In Rust, we can create a new thread through the `std::thread::spawn` function

Example of thread in Rust

Rust

```rust
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("Reached {i} from the spawned thread!");
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("Reached {i} from the main thread!");
        thread::sleep(Duration::from_millis(1));
    }
}
```

Example of thread in Rust

Reached 1 from the main thread!
Reached 1 from the spawned thread!
Reached 2 from the main thread!
Reached 2 from the spawned thread!
Reached 3 from the main thread!
Reached 3 from the spawned thread!
Reached 4 from the spawned thread!
Reached 4 from the main thread!

How can we ensure that all threads are spawned

- So far only 5 threads are spawned.
- The main thread exits its execution after 5 iterations in the loop
- This cause the `main` function to end prematurely
- We can use `join` to ensure that all threads are spawned
- This is due to the fact that `thread::spawn` returns `JoinHandle<T>`
- We can call `join` on the `JoinHandle` to block the main thread until the thread completes

Ensure to spawn all threads

Rust

```rust
fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("Reached {i} from the spawned thread!");
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("Reached {i} from the main thread!");
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

Data Races and Race Conditions in Rust

Data Race

A data race occurs on a value in memory if

- two or more threads are concurrently accessing memory,
- one or more of them is a write, and
- one or more of them is unsynchronised.

Data races are prevented by the ownership system/borrow checker, since we are unable to alias a mutable reference.

However Rust does not prevent general race conditions

- Since, our hardware, OS and other programs might be Racy
- Still, a race condition cannot violate memory safety in a Rust program on its own
- Only in conjunction with some other unsafe code can a race condition actually violate memory safety

Race condition in Rust

Rust

```
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

let data = vec![1, 2, 3, 4];
let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

thread::spawn(move || {
        other_idx.fetch_add(10, Ordering::SeqCst);
}); // we can mutate idx since its atomic it cannot cause a Data Race.

if idx.load(Ordering::SeqCst) < data.len() {
        unsafe { // Incorrectly loading the idx after we did the bounds check.
                //  This is an unsafe race condition
                println!("{}", data.get_unchecked(idx.load(Ordering::SeqCst)));
        }
}
```

Message Passing in Rust

- The main idea comes from Go, where it says "Do not communicate by sharing memory; instead, share memory by communicating"
- Channels are a way to communicate between threads in Rust
- Rust provides synchronous channels in the standard library through `std::sync::mpsc` module
- Rust uses transmitter(tx) and receiver(rx) for channel communication to send and receive over channel respectively.
- Channel can have multiple sending ends producing values but only one receiving end consuming values

Message passing to transfer data between Threads

- First, we create a new channel using the `mpsc::channel` function (`mpsc` stands for multiple producer, single consumer).
- The `mpsc::channel` function returns a tuple, the first element of which is the sending end —- the transmitter -— and the second element of which is the receiving end -— the receiver
- Using move moves ownership of tx to new thread
- Thread must own transmitter to send messages on channel
- The *send* method returns a Result<T, E> type and *unwrap* to panic in case of an error (send has nowhere to send).
- The *recv* method is used to receive messages from the channel and will block the current thread until a message is available.

Example of Message Passing in Rust

Rust

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
        let (tx, rx) = mpsc::channel();
        thread::spawn(move || {
                let val = String::from("Sending data");
                tx.send(val).unwrap();
        });
        let received = rx.recv().unwrap();

        println!("Got: {}", received);
}
```

Channels and Ownership Transference

- Ownership rules play a vital role in message sending because they help you write safe, concurrent code
- Let's consider, for example, to try to print val after we sent it down the channel via tx.send
- This results in error, since once the value has been sent to another thread, that thread (i.e., function recv) takes the ownership
- This means we cannot use val in the original thread anymore

Creating Multiple Producers by Cloning

- Before creating the first spawned thread, we call clone on the transmitter.
- Gives us a new transmitter we can pass to the first spawned thread.
- We pass the original transmitter to a second spawned thread which gives us two threads, each sending different messages to the one receiver.

Creating Multiple Producers by Cloning

Rust

```rust
let tx1 = tx.clone();
thread::spawn(move || {
        let vals = vec![ ... ];
        for val in vals {
                tx1.send(val).unwrap();
                thread::sleep(Duration::from_secs(1));}
});
thread::spawn(move || {
        let vals = vec![ ... ];
        for val in vals {
                tx.send(val).unwrap();
                thread::sleep(Duration::from_secs(1)); }
});
for received in rx {
        println!("Got: {}", received);
}
```

Mutex in Rust

- A `Mutex<T>` is a mutual exclusion primitive useful for protecting shared data.
- It is a type that provides interior mutability, meaning that it allows you to mutate data even when the `Mutex<T>` itself is immutable.
- A `Mutex<T>` has two main methods: `lock` and `unlock`.
- The `lock` method locks the mutex, preventing other threads from accessing the data until the mutex is unlocked.
- The `unlock` method unlocks the mutex, allowing other threads to access the data.

Summary

- Rust ownership system is a practical implementation of ideas from linear types, RAII/RBMM, and aliasing
- Ownership system guarantees memory safety and thread safety at compile time
- Message passing and concurrency supported in standard library
- Session types supported through external library
- We can use threads and message passing to build concurrent applications in Rust
- In the next lecture, we will look more in detail in the type system elements to support concurrency in Rust