Rust: Powered by Ownership

May 26, 2018

Max Meldrum
KTH Royal Institute of Technology
mmeldrum@kth.se

Abstract

Rust is a systems programming language that has been gaining traction in the recent years. Rust supports features including, but not limited to, pattern matching, type inference and zero-cost abstractions. However, what makes Rust stand out is its ownership-based type system which allows it to guarantee memory safety and thread safety. Rust does not need to rely on a garbage collector, thus removing the runtime overhead often found in other languages, e.g, Java and Go. As it runs on bare-metal and is built on LLVM, it is able to deliver a similar runtime performance as C/C++. Although Rust is not the first language to introduce the concept of ownership, it is the first to have widespread use outside of academic research. In this report, we deep dive into Rust, a modern programming language powered by ownership.

General Terms Programming languages, type systems

Keywords Rust, ownership, safety

1. Introduction

Rust [1], is a modern systems programming language that focuses on memory safety, parallelism and speed. It is designed to allow developers to enjoy both low-level control and high-level safety. The Rust project was initially started by Graydon Hoare [2], who was working on it as a side project. Mozilla, where Hoare was working at the time, began sponsoring the project in 2009. The main reason for this is that Mozilla utilises Rust in Servo and some internal components of Firefox [3]. In languages such as C and C++, the programmer is given total control of the memory management, although it is a very attractive feature, it comes at the cost of safety. Languages that uses this approach are more vulnerable to exploits, e.g., buffer overflow. On the other hand, languages such as Java and Go rely on a garbage collector to constantly look for memory that is no longer used. It gives less control of low-level resources, however, developers become less prone to build exploitable binaries. Rust uses a third approach, memory is managed through a system of ownership [4], where the compiler enforces a set of rules that are checked at compile time. The compiler guarantees thread safety and memory safety, while also ensuring that no segfaults can occur. It has topped the most loved language category two years in a row (2017 and 2018) in Stack Overflow's yearly survey [5, 6]. Increasingly many open-source projects are being implemented in Rust. Redox [7] is an Unix like operating system that is being developed using Rust, it is a perfect use case for Rust and its properties. It is also being used by big industry companies. Dropbox, a cloud storage service provider started of by storing users files on AWS's (Amazon Web Services) object storage service called S3. Dropbox eventually decided to build its own computer network and implement a customized storage system, which was initially developed using Go. However, Go's memory footprint was too large which lead to the decision to migrate over to Rust [8].

The remainder of the report is outlined as following. Section 2 covers Rust with the main focus being on its ownership feature. Code examples are also provided in order to show the practical implications. In section 3, we shortly discuss some of Rust's current limitations. Section 4 covers attractive aspects of Rust that are not related to its Ownership concept. In section 5, we review related work and discuss the PLT[1] research that Rust is based on. Section 6 ends the report with the conclusion.

2. Rust

In this section, we will look at some core features of Rust to get an understanding of how the language is trying to overcome the tradeoff between the control of low-level languages and safety of high-level languages [9].

2.1 Ownership

In Rust's ownership model, once a variable has been bound to an object, it gains an exclusive ownership of the resource. Each object is uniquely referenced in the memory, thus making it possible for the compiler to track objects and make sure that they are released at the end of their lifetime, i.e., when the object goes out of scope. Ownership can also be transferred to another variable or so called alias. Heap memory management is one of the issues that ownership tackles, some examples are:

- Release memory that is tied to unused data
- Making sure no double free errors occur. i.e., same memory being freed twice.
- Reducing the amount of duplicate data

Ownership is a concept that is new to many developers. Although it has a learning curve, the idea is that once you have gotten used to it, you will be able to develop safe and efficient code. Rust's Ownership model employs the following rules [4]:

1. Each value in Rust has a variable that's called its owner
2. There can only be one owner at a time
3. When the owner goes out of scope, the value will be dropped.

Let's start of by looking at an example where the above rules are applied.

```
fn main() {
    {
      // Ownership gained
      let str = String::from("kth");
      println!("{}", str); // Works!
```

[1] Programming Language Theory

```
    } // The str variable is deallocated


    {
      let str = String::from("kth");
      // only one owner is allowed!
      let str2 = str;
      println!("{}", str); // Error!
    }   // -- || --
}
```

Listing 1. Rust ownership example

In the second code block, **str** is stored on the heap. By assigning **str** to **str2**, both variables point to the same memory location. However, in order to guarantee memory safety, this is not allowed in Rust. So what happens is that the compiler considers the reference **str** to be invalidated. Don't worry though, Rust's clone method allow us to make a deep copy of the object.

```
fn main() {
    let str = String::from("kth");
    // Create a copy of str and assign it to str2
    let str2 = str.clone();
    println!("{}", str2);
}
```

Listing 2. Rust Clone Example

So far our examples have been very basic, next we will look at how ownership and functions work together:

```
fn main() {
    let v = vec![10, 20, 30];
    // v is moved into the calling function,
    // thus the ownership as well.
    performOperation(v);
    // Compilation stops at this point,
    // as v is invalidated
    printVec(v);
}

fn performOperation(v: Vec<i32>) {
    // Computation with v ongoing
    // Function exits and v is deallocated
}

fn printVec(v: Vec<i32>) {
    println!("{}", v);
}
```

Listing 3. Ownership and fuctions example

The above code shows how Rust differentiates from other languages, e.g., Java, where the following example would compile. We earlier mentioned how the ownership could be transferred. This is exactly what happens when vector **v** is applied as parameter to performOperation. The ownership now belongs to the scope of the called function. We could also have performOperation return the ownership. Let's take a look at how this changes our previous example:

```
fn main() {
    let v = vec![10, 20, 30];
     // Ownership is returned
    let v2 = performOperation(v);
    // Compiler does not complain!
    println!("{}", v2);
}

fn performOperation(v: Vec<i32>) -> Vec<i32> {
    // Computation with v ongoing
    v // Return ownership to the caller
}
```

Listing 4. Transfer ownership example

2.1.1 Borrowing

Instead of moving the object into the calling function and granting it ownership, we can let the function borrow it. This is done in Rust by using the & (ampersand) reference symbol. Note that in the following example, the & symbol is required both at the function call and parameter list.

```
fn main() {
    let test = String::from("kth");
    // Let print_string borrow test
    print_string(&test);
    // test is still valid, do a normal print
    println!("{}", test);
}

fn print_string(str: &String) {
    println!("{}", str);
}
```

Listing 5. Borrowing example

Dangling References A dangling reference is a reference to an object that no longer exists. Dangling references exist in languages with pointers, e.g., C/C++ and Rust. In C/C++, it is up to the developer to use best practices to make sure that a dangling reference doesn't create an irreversible runtime error. Rust's approach is more elegant, the compiler detects dangling references and issues a compile-time error to inform that the codes logic has to be revised.

```
fn main() {
    let uuid = uuid_gen();
}
// function returning a reference to
// an object that is just about to get deallocated
fn uuid_gen() -> &String {
    // Assuming function exists..
    let id = String::random(10);
    &id
}
```

Listing 6. Dangling reference example

Mutable References Mutable references won't be covered in this report. However, if you want to read more about it, then [10] is a good place to start.

2.1.2 Lifetimes

A lifetime in Rust is the scope of which a reference is valid. The intention of lifetimes is to hinder the use of references whose value has gone out of scope, i.e., dangling reference. In most cases, the lifetimes in Rust are implicit and inferred. However, in situations where lifetimes of different references are related, we must explicitly annotate it in the code. Let's take a look at two code examples to get an understanding of how lifetimes work.

```
{
    let r;
    {
        let x = 5;
        r = &x;
    }
    println!("r: {}", r);
}
```

Listing 7. Inferred lifetimes [11]

When the inner scope in listing 5 exits, the value that **r** has been assigned is a reference whose value has been deallocated. Rust's compiler has a borrow checker which examines scopes to verify that all borrows are valid. During compilation of the code, the borrow checker will notify us that **x** does not live long enough.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Listing 8. Explicit lifetimes [11]

Rust's lifetimes are annotated with an apostrophe. It is important to note that explicit annotations do not change the lifetimes, but only sets up rules for the compiler to check. In listing 8, the **longest** function has defined constraints, which is that both parameters and the function output must have the same lifetime 'a. If **x** and **y** have different lifetimes, then we could end up with a scenario where code is trying to utilise the result of **longest** when it is a dangling reference. This is why the borrow checker is essential to Rust.

Lexcial vs Non-Lexical scopes Lifetimes are currently limited to a lexcial scope in Rust. Plans[2] are under way to introduce non-lexical scopes [3].

2.2 Concurrency

Dealing with threads in C++ or Java can be a nightmare as it is the developers responsibility to make sure there are no race conditions. One of Rust's prominent features is that it guarantees thread safety. Rather than experiencing a critical error at runtime, the compiler will let us know if we are up to no good [12]. So how is this achievable? Rust's ownership-based type system ensures that aliasing and mutation cannot occur at the same time, which prevents any form of race condition [9].

```
fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("vector:{:?}", v);
    });

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

Listing 9. Rust thread safety [12]

In listing 9, we have a piece of code that does three things. 1) Main thread creates vector **v** that holds 3 integers. 2) Main thread spawns a new thread and instructs it to print **v**. 3) The main thread waits for the spawned thread to finish. Although this code would compile in most other languages if we translated the code, we have to remember that Rust's ownership feature is involved in most aspects of the language. The **println!** command inside the closure (spawned thread) only needs a reference of **v** and even though the compiler infers **v**, it will still complain. The reason behind this is that it impossible to know exactly how long the spawned thread will live, which makes it hard to define a lifetime of the reference. In order to make the code work, we will have to use Rust's **move** keyword. Rather than having the compiler implicitly infer our vector to a reference, we use **move** to explicitly command the closure to take ownership of **v**. In listing 10, we can see how the syntax now looks.

```
    let handle = thread::spawn(move || {
        println!("vector:{:?}", v);
    });
```

Listing 10. Rust move command [12]

3. Rust Limitations

Rust's ecosystem is small compared to languages such as Java or Python. As pointed out in section 2.1, the concepts of ownership, borrowing and lifetimes are new to most developers and takes some time to fully understand. Rust is not a good language for quick and efficient prototyping. Although the borrow checker is there to help, you will most likely end up battling with it when you get started. Rust does not support HKT's (Higher-Kinded Types). The Rust team states that other improvements have been prioritized, but also that implementing HKT's is a major change, thus requiring a careful approach to it. [2].

4. Rust Strengths

In this section, we present some other aspects of Rust which make it very appealing to developers.

4.1 Cargo

If you are tired of dealing with build tools such as Maven, Make or Ant, then you will love what Rust has in store. Cargo is a build system and package manager built using Rust. It takes care of downloading library dependencies and compiling your project. In Rust terminology, a crate is equivalent to a library or package. With Cargo, it is possible for Rust developers to share crates by publishing them to a central registry[4].

4.2 LLVM Backend

Rust's compiler rustc uses LLVM as its backend. Despite suffering from the compile-time incurred by the large LLVM framework, it enables Rust to have a blazingly fast runtime speed [2]. Its performance can be seen in various benchmarks, with one being [13].

4.3 Active and Growing Community

Rust's active open source community is growing each year [14] and the Rust team is actively working to make the language even better.

5. Related Work

Linear types [15] together with modern PLT research is what lead to the development of Rust [16]. Cyclone [17] is a type-safe programming language whose goal was to add memory safety to C. The aim

[2] https://github.com/rust-lang/rust/issues/43234

[3] https://github.com/nikomatsakis/nll-rfc/blob/master/0000-nonlexical-lifetimes.md

[4] https://www.crates.io

behind Rust and Cyclone are very similar, however, they use different approaches. Cyclone uses Region-Based Memory Management [18]. Data is placed into regions and references must be explicitly annotated to identify which region their data resides in. Rust's lifetimes are similar to regions in Cyclone. Vault [19] is a safe programming language designed by Microsoft Research. It employs a linear type system to allow it to track object usages at compile time [20]. [21] introduces a static type system for multi-threaded programs. It uses ownership types in order to prevent deadlocks and data races. [20] discusses the need for safe systems programming languages and studies how languages such as SafeC [22], CCured [23], Vault and Cyclone are attempting to accomplish it. [24] presents a type system for safe parallelism through the use of reference immutability. Wadler et. al. introduced Type classes [25], which have found its way into Rust's traits, which are comparable to Typeclasses in Haskell.

As pointed out earlier in this report, Rust is not the first language to adopt ownership, however, preceding languages have been either to restrictive or too expressive [9]. More related work can be found in the Rust bibliography [26]. It includes papers that have influenced the design of Rust and publications that are specifically about Rust.

6. Conclusion

Rust is a state-of-the-art programming language with distinctive concepts such as ownership, borrowing and lifetimes, which are derived from academic research. Rust is to a great degree influenced by multiple languages, e.g., Cyclone, C++ and Standard ML. The project is constantly evolving, and with an active open-source community, it is very likely to continue to grow in popularity. It is also imaginable that future languages will be influenced by Rust's approach of memory management through ownership.

Acknowledgments

Credit goes out to the Rust team for producing a well written Rust book (The Rust Programming Language). Also, the paper of Jung et al. RustBelt: securing the foundations of the Rust programming language for providing a more theoretical background of Rust.

References

[1] Nicholas D. Matsakis and Felix S. Klock, II. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, 2014. ACM.

[2] Mozilla Research. The rust project, faq. `https://www.rust-lang.org/sv-SE/faq.html#project`, 2018. Accessed: 2018-05-19.

[3] Mozilla Research. Rust programming language. `https://research.mozilla.org/rust/`, 2018. Accessed: 2018-05-19.

[4] The Rust Project Developers. The rust project, ownership. `https://doc.rust-lang.org/book/second-edition/ch04-01-what-is-ownership.html`, 2018. Accessed: 2018-05-19.

[5] Stack Overflow. Developer survey results. `https://insights.stackoverflow.com/survey/2017/#most-loved-dreaded-and-wanted`, 2017. Accessed: 2018-05-24.

[6] Stack Overflow. Developer survey results. `https://insights.stackoverflow.com/survey/2018/#most-loved-dreaded-and-wanted`, 2018. Accessed: 2018-05-24.

[7] Redox Developers. Redox, unix like operating system built using rust. `https://www.redox-os.org/`, 2018. Accessed: 2018-05-19.

[8] Cade Metz. The epic story of dropbox's exodus from the amazon cloud empire. `https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/`, 2016. Accessed: 2018-05-25.

[9] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, December 2017.

[10] The Rust Project Developers. The rust project, borrowing. `https://doc.rust-lang.org/book/second-edition/ch04-02-references-and-borrowing.html`, 2018. Accessed: 2018-05-22.

[11] The Rust Project Developers. The rust project, lifetimes. `https://doc.rust-lang.org/book/second-edition/ch10-03-lifetime-syntax.html`, 2018. Accessed: 2018-05-23.

[12] The Rust Project Developers. The rust project, fearless concurrency. `https://doc.rust-lang.org/book/second-edition/ch16-00-concurrency.html`, 2018. Accessed: 2018-05-24.

[13] Benchmarks Game Team. Benchmarks game. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/rust-java.html`, 2018. Accessed: 2018-05-25.

[14] Judy DeMocker. Wheres rust headed in 2018? ask the community. `https://hacks.mozilla.org/2018/01/rust-community-roadmap-2018/`, 2018. Accessed: 2018-05-25.

[15] Philip Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

[16] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. System programming in rust: Beyond safety. *SIGOPS Oper. Syst. Rev.*, 51(1):94–99, September 2017.

[17] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[18] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM.

[19] Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 13–24, New York, NY, USA, 2002. ACM.

[20] Li P. Safe systems programming languages. Google Scholar, 2004. Accessed: 2018-05-24.

[21] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM.

[22] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 290–301, New York, NY, USA, 1994. ACM.

[23] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005.

[24] Colin S. Gordon, Matthew Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. Technical report, October 2012.

[25] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

[26] The Rust Project Developers. Rust bibliography. `https://forge.rust-lang.org/bibliography.html`, 2018. Accessed: 2018-05-24.