In Rust We Trust

Revisiting Safety Implementation in Rust Compiler

Ben Chen
Computer Science and Engineering
Southern University of Science and Technology
Shenzhen, China
chenb2022@mail.sustech.edu.cn

Yicheng Xiao
Computer Science and Engineering
Southern University of Science and Technology
Shenzhen, China
xiaoyc2022@mail.sustech.edu.cn

Jiarun Zhu
Computer Science and Engineering
Southern University of Science and Technology
Shenzhen, China
12210212@mail.sustech.edu.cn

***Abstract*—Through the report, we have investigated some of the language features that Rust has. We focus on discussing how Rust compilers handle ownership of variables and lifetime parameters. Rust compilers have enabled static analysis to discern possible problems induced by ownership and lifetime.**

***Index terms*—Rust, Compiler Design, Memory Safety**

I. Introduction

Rust is a systems programming language that brings together the performance of languages like C and C++ with robust memory safety guarantees. This is achieved through its unique ownership model and strict borrowing rules, which help prevent common memory errors at compile-time without needing a garbage collector. Rust's design has made it a popular choice for projects that require both efficiency and safety, such as operating systems, embedded systems, and browser engines. This paper provides an overview of Rust's key features, including its memory safety, concurrency support, and expressiveness. We also explore the architecture of Rust's compiler, rustc, focusing on how it enforces memory safety through static analysis and the borrow checker. By understanding these aspects, we can appreciate Rust's growing adoption across the software industry and its role in advancing secure, performant systems programming.

II. Rust 101

A. What is Rust?

Rust is a systems programming language designed to provide the performance and control of low-level languages like C and C++ while eliminating common sources of programming errors, such as memory corruption, null pointer dereferences, and data races. Introduced by Mozilla in 2010, Rust quickly gained popularity for its unique approach to memory safety and concurrency, achieved through a strict system of ownership and borrowing rules checked at compile-time rather than runtime. This allows Rust to ensure memory safety without the need for a garbage collector, making it highly efficient for systems-level programming where resource control is critical.

Rust's features make it a powerful choice for projects that demand both safety and performance, such as operating systems, browser engines, game engines, and embedded systems. Its high-level abstractions and modern syntax make it expressive and developer-friendly, allowing for concise, readable code. Rust's focus on memory safety, efficient resource usage, and concurrency has led to its adoption in critical projects, including parts of the Linux kernel, where stability and security are paramount. Through its active open-source community, Rust continues to evolve, providing robust libraries and tools for a wide range of applications.

B. Why Rust?

What we talk when we talk about Rust? When we talk about Rust, we refer to words like

Memory safety The Linux kernel has been insisted in writing only in C, but in recent years the maintainers are becoming pleasant to embrace Rust programs in kernel functions. The main reason is that he biggest feature Rust brings to Linux is exactly the insurance of memory safety. Program compiled by `rustc` ensures no illegal access of memory will be conducted by a series of dynamic checks.

Efficiency With lots of modern feature, Rust still remains the speed close to C/C++, and sometimes event better, which will give credit to the state-of-the-art optimization of mighty Rust compiler.

Expressiveness Modern programming languages have plenty of syntax sugar integrated in compiler to free the programers from complex and repetitive source code development, like the essences of functional programming (high order function, currying and iterator). So that developers can express the algorithm clearly in codes.

Open-source It has been a long run that the open-source community is stirring up a spree of rewriting in Rust. Nearly

every tools are being re-implemented in Rust, with a huge performance boost and absence of memory leak problems.

Nowadays, when we talk about system safety, we can't avoid mentioning Rust. But what makes Rust different from others? Under our research, we propose that the reason can be summarized to two major aspects. One thing is that it has LLVM backed up as its powerful back-end. This eases its duty to adaption of specific platforms and directly gains the cross-platform compatibility. And therefore, developers of `rustc` can undisturbedly focus on the statical analysis and code optimization in front-end. The other thing is that `rustc` explicitly demands the programmers to notice the ownership management, otherwise aborts the compilation process. The explicit definition of ownership enables the compiler to perform static analysis on the code, resolving issues only using front-end. The statical check of source codes ensures that codes accepted by `rustc` is problem-free (but logical bugs remain).

C. *Who uses Rust?*

In recent years, Rust has gained significant traction in the tech industry due to its innovative approach to safety and performance. Major organizations like Microsoft have explored Rust's potential by using it to enhance the security of critical systems. For example, Microsoft has tested Rust to rewrite parts of the Windows kernel, aiming to reduce memory-related vulnerabilities that are common in low-level system code. Additionally, the Linux kernel community has approved Rust as a secondary language, allowing developers to integrate safe Rust code alongside traditional C for enhanced stability.

As more developers and companies recognize Rust's strengths, its adoption has expanded across various domains, from web development to embedded systems and game engines. This growth is further supported by a vibrant open-source community and a rapidly evolving ecosystem of libraries and tools, making Rust a modern, powerful language choice for both new and legacy systems.

III. Rust Compiler Design

A. *Structure of the Rust Compiler*

A textbook compiler will probably looks as the figures shown in Fig. 1. Inside the part of front-end, the compiler tokenizes the input of source code, forming symbol tables, constructing abstract syntax tree with CFG and checking the semantic errors (with context). And finally the front-end generates intermediate representation (IR, platform-free) and feeds it to the back-end for machine code generation.

Rust compiler dedicates to the semantic part, as the difference shown in Fig. 2. The semantic processing will continuously goes through two intermediate layers, which are HIR and MIR.

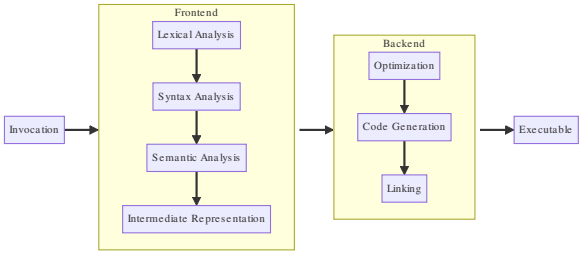

Fig. 1: Structure of general compilers

In the stage of HIR, the semantic analyzer fetches information from token stream and AST to further generate an improved representation friendlier to the analyzer. Meanwhile, the syntax is desugared at that time since the syntax sugar serves as a equivalent expression as the primitive grammar, like from `if let` to `match` and from `for` to `loop`. Once this kind of conversion and processing is done, the compiler performs type check and type inference to settle down the types of all variables and objects. Remind that Rust is static-type, meaning that the types of variables are finalized at compilation, and any changes in types detected during this stage results in a compile failure.

Next is the MIR stage. MIR majorly checks the lifetime and ownership. At an earlier stage without MIR, borrow checking is done by HIR stage when the granularity is too coarse to conduct a normal lifetime inference. Back then, the lifetime of variables were inferred by lexical field, as the C/C++ compilers. In 2018, Rust was introduced the concept of Non-Lexical Lifetime (NLL) to resolve the granularity problem.

The mighty MIR gives credit to the three key features. Firstly, the representation of MIR is based on control flow graph, which enables it to perform a comprehensive check of the lifetime throughout the program. Secondly, it has no recursive expression that would be most likely to increase the depth of analysis, making the analysis easier. Lastly, variables are fix-typed and everything is explicit, eliminating the ambiguity.

Thus, in the next section, we are going to deep in how MIR conducts lifetime and ownership checking.

B. *The Borrow Checker*

The borrow checker works on MIR to enforce dozens of restrictions [1]:

- all variables are initialized before being used
- you can't move the same value twice
- you can't move a value while it is borrowed
- you can't access a place while it is mutably borrowed
- you can't mutate a place while it is immutably borrowed
- …

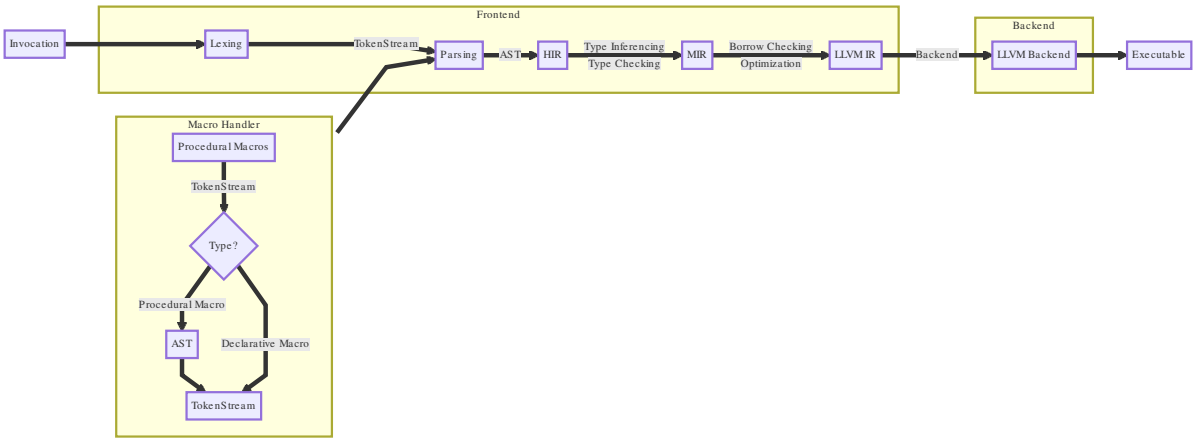

Fig. 2: Structure of the rust compiler, rustc

The main entry of borrow checker is the `mir_borrowck` function. During its execution, borrow checker will create a local copy of MIR and substitute all regions in MIR to inference variable[1]. Next, the checker will perform a number of dataflow analyses to track the data, and a duplicate type check is done to determine all of the constraints between different regions. Then, the checker does region inference to compute the scope of variables and their reference/borrow from the constraints collected previously. Finally, the checker will secondly walk over the MIR, to check if every statements is legal. For example, when seeing a assignment `*a = *b + 1`, the checker will check (a) if `a` is a a variable itself, mutably borrowed, (b) if `b` is initialized, mutably borrowed, not moved and not dropped[2].

The example is from the `rustc` development guide [2]. The checker tracks the context when walking through the codes.

```
fn foo() {
    let a: Vec<u32>;
    // a is not initialized yet

    a = vec![22];
    // a is initialized here

    std::mem::drop(a); // a is moved here
    // a is no longer initialized
    let l = a.len(); //~ ERROR
}
```

And for move checking, the checker will computes the move paths of variables, for example.

```
fn foo() {
    let a: (Vec<u32>, Vec<u32>) = (vec![22], vec![44]);
    // a.0 and a.1 are both initialized

    let b = a.0; // moves a.0
    // a.0 is not initialized, but a.1 still is

    let c = a.0; // ERROR
    let d = a.1; // OK
}
```

To implement it, the checker records the location of variables to a `MovePath` data struct and it's indexed by `MovePathIndex`. `MovePath` is basically the same in `Place` in MIR. Also the move checker will search for illegal moves, like move of static variables, move of array element(s) and move of borrowed reference.

IV. Lifetime & Ownership

[1]A inference variable represents a variable of unknown type, like a place holder, to present as its original variable. The exact type of the variable will be determined once initialized.

[2]The states of initialized, moved and dropped can be summerized to two states: available or not. When being moved or dropped, b no longer owns the value and thus changed from initialized to uninitialized

Good programmers select language, while great language selects programmers. This is exactly the case with Rust. **Lifetime** and **Ownership** are core concept used in Rust. In fact, these concepts exist not only in Rust but also in C/C++. Nearly all memory safety issues stem from the incorrect use of ownership and lifetimes. Any programming language that does not use garbage collection to manage memory faces these challenges.

However, Rust explicitly defines these concepts at the language level and provides language features that allow users to explicitly control ownership transfer and declare lifetimes. Additionally, the compiler checks for various misuse errors, enhancing the program's memory safety.

The following examples used are taken from [3].

A. *What is Ownership?*

Ownership means control over a memory region associated with a variable. This region can exist in various memory locations (like heap, stack, or code segment). In high-level languages, accessing these memory regions typically requires associating them with variables, unlike in low-level languages where direct access is possible. If not dealing with Ownership properly, the following five errors might be produced:

- Buffer Overflow
- Segmentation Fault
- De-referencing null pointer
- Dangling pointer
- Invalid memory de-allocation

```
// Buffer overflow
#include <iostream>
using namespace std;
int main() {
    int values[3]= { 1,2,3 };
    cout<<values[0]<<","<<values[3]<<endl;
    return 0;
}

// Segmentation fault
struct P { int x; int y; };
P* newP(int x,int y) {
    P p { .x=x, .y=y };
    return &p;
}

int main() {
    P *p = (P*)malloc(sizeof(P));
    cout<<p->x<<","<<p->y<<endl;
    return 0;
}

// Dereferencing null pointer
int main() {
    P p = NULL;
    cout<<p->x<<endl;
    return 0;
}

// Dangling pointer and Invalid memory
deallocation
int main() {
    P *p = newP(10,10);
    delete p;
    return 0;
}
```

B. *Ownership Binding*

From the examples above, we can see that writing C will produce this errors easily as the compilers will not warn you about these misusages. However, in Rust the **ownership** concept comes with binding instead of assigning likewise in C/C++. In this way, Rust compiler can track the ownership of each variable and thus ensure according issues will not be manifested.

```
fn main() {
    let a = 1; // Creating an immutable
binding
    println!("a:{}",a);      /* 1 */
    println!("&a:{:p}",&a); /* 0x9cf974 */
    a = 2;                  /* Error: cannot
assign twice to immutable variable `a` */
    let a = 2;              /* Rebinding */
    println!("&a:{:p}",&a); /* 0x9cf9a4 */
}
// In this case, the address of `a` changes.
// An immutable binding means that a
variable is bound to a memory address and
given ownership.

fn main() {
    let mut b = 1; // Creating a mutable
binding
    println!("b:{}",b);      /* 1 */
    println!("&b:{:p}",&b); /* 0x9cfa6c */
    b = 2;
    println!("b:{}",b);      /* 2 */
    println!("&b:{:p}",&b); /* 0x9cfa6c */
    let b = 2;              /* Rebinding */
    println!("&b:{:p}",&b); /* 0x9cfba4 */
}
// In this case, the address of `b` remains
the same.
// A mutable binding allows the variable to
modify the data in the associated memory
region.
```

From the example provided, **assignment** is the act of writing a value into the memory region associated with a variable, while **binding** is the process of establishing the relationship between a variable and a memory region. In Rust, this also involves transferring ownership of that memory region to the variable.

C. Ownership Transfer and Borrowing

With binding concepts, Rust transfers ownership of a memory region to a variable. This transfer is called **ownership transfer**. Transfer of ownership avoids copying the data, which will be more efficient. If programmers just want to use the data without transferring ownership, borrowing the data can do. This is called **ownership borrowing**.

```
fn main() {
    let a = 1;
    let b = a; // Ownership transfer
    println!("a:{}",a); /* Error: use of
moved value: `a` */
    println!("b:{}",b); /* 1 */
}
```

```
fn main() {
    let mut a = 1;
    let im_ref = &a;         // Unmutable
borrowing
     let mut_ref = & mut a;   // Mutable
borrowing
    println!("{}",im_ref);
     //[At line 4] error[E0502]: cannot
borrow `a` as mutable because it is also
borrowed as immutable
}
```

```
fn main() {
    let mut a = 1;
     let mut_ref = & mut a;   // Mutable
borrowing
    println!("{}",a);
    println!("{}",mut_ref);
    //[At line 5] error[E0507]: cannot move
out of `a` because it is borrowed
}
```

In the first example, the ownership of `a` is transferred to `b`. Therefore, `a` cannot be used after the transfer. In the second example, `im_ref` and `mut_ref` are borrowed from `a`. The ownership of `a` is not transferred, so `a` can still be used. However, `mut_ref` cannot be used before `im_ref` is released.

In Rust, the compiler checks for ownership transfer and borrowing. If the code violates these rules, the compiler will throw an error. This is called **static analysis**.

D. Lifetime

The lifecycle of a variable is mainly related to its scope, and in most programming languages, it is implicitly defined. In Rust, however, one should explicitly declare the lifetime parameters of variables, which is a very unique design. This syntactic feature is something that is rarely seen in other languages.

```
struct V{v:i32}

fn bad_fn() -> &V{
    let a = V{v:10};
    &a
}

fn main(){
    let res = bad_fn();
}
// [At line 3] error[E0106]: missing
lifetime specifier
```

```
struct V{v:i32}

fn bad_fn<'a>() -> &'a V{
    let a = V{v:10};
    let ref_a = &a;
    ref_a
}

fn main(){
    let res = bad_fn();
}
// [At line 6] error[E0515]: cannot return
reference to local variable `a` -> Dangling
pointer
```

The `'a` lifetime parameter in Rust specifies the required lifespan of a reference returned by a function, ensuring that the returned reference is valid within the caller's context.

`'a` imposes a requirement on the reference returned within the function body: the data referred to by the returned reference must have a lifetime at least as long as `'a`, meaning it must last at least as long as the variable in the caller context that receives the return value.

a) *Lifetime in function:*

```
fn echo<'a, 'b: 'a>(content: &'b str) ->
&'a str {
    content
}
fn longer<'a, 'b: 'a>(s1: &'a str, s2: &'b
str) -> &'a str {
    if s1.len() > s2.len()
        { s1 }
    else
        { s2 }
}
// Mark that 'b is at least as long as 'a
```

b) *Lifetime in `struct`:*

```
struct G<'a>{ m:&'a str}

fn get_g() -> () {
    let g: G;
```

```
    {
        let  s0 = "Hi".to_string();
        let  s1 = s0.as_str();
        g = G{ m: s1 };
    }
    println!("{}", g.m);
}
```

The lifetime definition of a `struct` ensures that, within an instance of the `struct`, the lifetime of its reference members is at least as long as the lifetime of the `struct` instance itself.

E. Static Analysis

With the notion of ownership and representation of lifetime, Rust compilers can conduct static analysis to the code to ensure zero tolerance to ownership and lifetime issues. This is mainly conducted by the borrow checker and lifetime checker. The borrow checker is located in the `rustc_borrowck` and the lifetime is regarded as a variable in AST and is mainly analyzed in `rustc_hir`, `rustc_hir_analysis` and `rustc_resolve`.

V. Conclusion

Rust represents a significant advancement in systems programming, offering a unique balance between low-level control and high-level safety guarantees. Through its ownership and borrowing model, Rust achieves memory safety without relying on a garbage collector, which makes it suitable for performance-critical applications in fields like operating systems, embedded systems, and network programming. The Rust compiler (`rustc`) plays a central role in enforcing these safety standards, with tools such as the borrow checker, non-lexical lifetimes, and LLVM-based optimizations ensuring that Rust programs are both efficient and reliable.

As this report has discussed, Rust's adoption continues to grow as companies and developers recognize its value for building secure, robust software. From its integration into the Linux kernel to experimental use in Windows kernel components, Rust is proving to be a practical alternative to traditional systems languages like C and C++. With its active open-source community and evolving ecosystem, Rust is poised to remain a leading choice for developers focused on safety, concurrency, and performance.

References

[1] "MIR borrow check - Rust Compiler Development Guide." [Online]. Available: https://rustc-dev-guide.rust-lang.org/borrow_check.html

[2] "Tracking moves and initialization." [Online]. Available: https://rustc-dev-guide.rust-lang.org/borrow_check/moves_and_initialization.html

[3] M. Xiao, "Rust 精选." [Online]. Available: https://rustmagazine.github.io/rust_magazine_2021/chapter_1/rust_ownership.html