Compiling Rust at Runtime

Batyr Nuryyev
Department of Computing Science
University of Alberta
nuryyev@ualberta.ca

Abstract

Rust is statically-compiled programming language that focuses on memory and and type safety. The safety is ensured at static time through

Keywords compilers, rust, static-analysis

1 Introduction

1.1 Static vs. Runtime Execution Trade-Off

Rust is a multi-paradigm systems programming language focused on type and memory safety. It is syntactically similar to C++. Unlike C++, it has a stricter type system as well as built-in memory safety guarantees. The Rust compiler enforces these guarantees at static time, thereby maintaining high execution performance (at runtime). It also allows for "fearless concurrency", where most of the bugs stemming from the use of concurrency are detected at compile time. Being able to detect concurrency-related bugs before execution helps Rust developers trace bugs easier and fix their compile-time errors faster.

However, while the Rust compiler, `rustc`, performs these checks at compile time, its compilation time drastically increases. According to the *Computer Language Benchmarks Game* CITE, Rust performed about slower on 6 out of 10 programs when comparing Rust to C. On these 6 programs, the `rustc` was 11% slower on average than GCC. Add note that though Rust is getting better and benchmarks may have been anulled, there is still a need for runtime environment such as garbage collection

1.2 Potential solution

One way to alleviate the problem of slow compilation would be to compile Rust at runtime. Compiling code at run-time encompasses, in general, two forms of execution: *interpretation*, which immediately executes a block of code; and *just-in-time (JIT) compilation*, which first compiles a block of code (usually a method or function) to native machine code, and then executes it. Languages that are compiled at runtime usually require an environment that includes both interpreter and JIT compiler. Such environment is called a (process) *virtual machine* (VM), that manages compilation process and decides whether to interpret or JIT-compile a block of code. The decision depends on a block of code; for example, if the block of code is to known to be frequently called throughout a lifetime of a program, it would be more optimal to JIT-compile it and cache the result for future calls; but, if the code is rarely called and is trivial to execute (i.e., does not contain complex, resource-intensive operations to optimize), interpreting it would be faster.

Rust could leverage run-time techniques to speed up its compilation time. Knowing that very few parts in the code get executed the most (if the Pareto principle holds true cite pareto), one could focus on optimizing those parts through JIT-compilation, and interpreting the rest.

Move this paragraph under main ideasCompiling language at runtime is not a novel idea. JAVA cite Inspired by Smalltalk, Java is a strongly-typed static object-oriented programming language. Its execution model (Java Virtual Machine, or JVM) consists of two compilers. The front-end compiler performs necessary static checks (e.g., type checking) at compile time, and emits an intermediate representation, or *bytecode*. Then, the other compiler takes the bytecode, performs necessary optimizations, and JIT-compiles it (by translating it into machine code and executing it). JVM attempts to optimize more of so-called *hot spots*, the areas of code where a program spends most of its time. As Java is similar to Rust in its type system (strongly-typed and static), similar dynamic compilation techniques will help us implement similar runtime environment for Rust.

Before discussing core ideas and implementation of our runtime execution model, we will briefly describe Rust itself, and the features that make it unique among other languages.

2 Overview of Rust lang

As stated in the previous section, Rust is a strongly typed language compiled at static time. It has a strict type system where each variable's type should be known at compile time. Despite the strong type system, the concept of *ownership* is what makes Rust truly a unique language.

Before understanding ownership, let us consider how programs manage memory. Some languages use a *garbage collector*, which, as program is running, looks for unused memory locations and frees them constantly (i.e., at runtime). Other languages completely delegate memory

CMPUT500, Fall 2019,
2019.

```
{                       // s is not valid here, it  s not yet declared

    let s = "hello";    // s is valid from this point forward

    // do stuff with s
}                       // this scope is now over, and s is no longer valid
```

Figure 1. Snippet of Rust lang cite the book

management to developers themselves; for example, if you are writing in C and want to allocate a variable onto the heap segment, you have to allocate and free memory on your own.

Garbage collectors usually add overhead at runtime as the collector has to find all (and there can be many at a time) allocated, but unused memory segments, and de-allocate them. On the other hand, trusting developers to manage program memory manually is a *bad* and error-prone idea that may lead to problems such as dangling pointer (a pointer that points to deallocated memory location), memory leaks (forgetting to deallocate allocated memory), and others which are hard to debug.

However, Rust introduces a third dimension of memory management that is enforced at compile time. According to Steve Klabnik and Carol Nichols cite rust book, "memory is managed through a system of ownership with a set of rules that the compiler checks at compile time. None of the ownership features slow down your program while it's running." Note that even though ownership features do not slow down the execution of a program, it slows down a compilation process by having to perform additional analysis to verify the rules.

The following are the ownership rules that help Rust get rid of garbage collection or manual memory management:

- Each value has its *owner* (a.k.a., a variable).
- Each value has *a single owner* at a time.
- Value is dropped when an owner goes out of scope.

Take a look at Figure 1. Before the declaration statement, `s` is not valid in the nested scope. After it is declared, `s` can now be used, but only in the nested scope. As the scope is over, `s`'s value is dropped as owner goes out of the nested scope.

What makes ownership more exciting is *borrowing* owners. For comparison with Figure 1, take a look at Figure 2.

The `owner` is of type of struct `Foo`, which may contain some fields. Instead of either copying or *moving* the value of that struct into the variable `borrower`, we can instead borrow a value of `owner` by *referencing* it.

Several questions immediately raise - what about multiple references? Also, if I have multiple references that change (or, *mutate*) the value of `owner`, would not this cause *data races*? Before providing answers, we have to differentiate between *immutable* and *mutable* references. The immutable references do not modify the original value (i.e., read-only), and the mutable ones may change the original value (i.e., both read and write accesses). To borrow any object in a *mutable* way, you can use `&mut` instead of original `&` (immutable by default).

The **borrow checker**, that statically checks the validity of references (or borrows), also follows a system of rules. The following are some of its rules:

- You can have either one mutable reference, or any number of immutable references, but not both.
- References must always be valid (i.e., cannot reference values when they are out of their scope).

3 Main ideas

My paper mainly focuses on a subset of Rust language, particularly borrow checker. I have not yet implemented the support for lifetimes; concurrency and closures; templates and generic programming; and some others. However, as I will describe later, the challenge mostly lies in separating static (dataflow) analysis such as borrow checking from the core compilation logic (i.e., code generation).

provide a high-level architecture, i.e. the front-end, environment, and interpreter components.

describe how you implemented borrow checking with points-to sets.

4 Implementation

This is implementation!

5 Evaluation

This is evaluation!

6 Related Work

this is related work!

7 Conclusion

this is conclusion!!

```
let owner = Foo { ... };
let borrower = &owner; // borrow a value of `owner`
```

Figure 2. Borrowing a value with references.