

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 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 guarantees compile-time memory safety. Since the Rust compiler enforces these guarantees at static time, it attains high execution performance at run time. It also allows for “fearless concurrency”, where most of the bugs stemming from the use of concurrency concepts (e.g., threads) are detected at compile time. Being able to detect concurrency-related bugs before execution helps Rust developers trace bugs easier and fix 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*, Rust performed about slower on 6 out of 10 programs when comparing Rust to C [2]. On these 6 programs, the `rustc` was 11% slower on average than GCC.

Therefore, slow compile times open up opportunities for the “just-in-time” (or dynamic) compilation techniques that may speed up static compilation without hurting execution speed.

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); it manages the compilation process and decides whether to interpret or JIT-compile a block of code. The decision depends on the character of the 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, one could focus on optimizing those parts through JIT-compilation, and interpreting the rest.

Compiling language at runtime is not a novel idea. 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 [6]. 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. Each variable should have its type explicitly declared (or easily inferable) at compile time. However, type system alone is not enough to make the language more memory secure. Thus, a concept of *ownership*, which I will explain in this section, is what makes Rust truly a unique language.

Before understanding ownership, let us consider how programs manage memory. Some languages use a *garbage collector*; while program is running, it looks for unused

```

111 {                                     // s is not valid here, it s not yet declared
112
113
114     let s = "hello";    // s is valid from this point forward
115
116     // do stuff with s
117 }                               // this scope is now over, and s is no longer valid
118
119

```

Figure 1. Snippet of Rust lang [3]

memory locations and frees them constantly (i.e., at run time). Other languages completely delegate memory management to developers themselves; for example, if you are writing in C and want to allocate a variable at the heap, you have to allocate and free memory on your own using `malloc`.

Garbage collectors usually add runtime overhead because the collector has to find all (and there can be many at a time) allocated, but unused memory segments, and deallocate 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 detect and debug.

However, Rust introduces a third dimension of memory management that is enforced at compile time. According to Steve Klabnik and Carol Nichols, "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" [3]. Note that even though ownership features do not slow down the execution of a program, it slows down compilation 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 so-called *borrowing*. 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? 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).

Besides borrow checking, there are some other important (for compile-time safety) analyses that compiler runs; examples include *lifetime analysis*, *typestate analysis*, and so on. Because borrow checker is the only analyzer I am looking at in this paper, we are not going to consider the other ones.

3 Main ideas

In this section, I will introduce core ideas behind dynamic compilation of Rust lang. The section will present *benefits of dynamic compilation* over the static one; talk about *static borrow checker* and how it is integrated within the compilation system; and discuss the *modularity* achieved by separating concerns into their own "modules".

Despite the aforementioned ideas, I should mention the idea of decreased compilation time. I only note that idea here due to not having implemented it in the project, *yet*. The virtual machine (i.e., interpreter and compiler, for now) implementation should handle most of the language; its essential features such as closures, lifetimes, templates, and so on. Because there is no support for more advanced language constructs, there is no point in running full-fledged benchmarks to compare the static

```

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

```

Figure 2. Borrowing a value with references.

and dynamic compilers. Therefore, the focus of this paper is the challenge in separating static (dataflow) analysis such as borrow checking from the core compilation logic (i.e., code generation) so that the system can attempt to dynamically compile Rust.

3.1 Benefits of Dynamic Compilation

There are two ways to dynamically compile a language. First, the naive way, would be to only use *interpreter*. It takes the source code and directly *executes* it, without performing any optimization transformations. As expected, performance degrades rapidly; since source code on its own may not be efficient without optimizations, executing it directly will take much more time than to compile it *ahead-of-time* (AOT) and then run the executable.

On the other hand, just-in-time (JIT) compilation may combine the power of both interpreters and JIT compilers. Instead of directly executing code, we could compile it to some intermediate representation (IR), run optimization on that IR, and then compile and execute it.

There are several benefits from JIT compilation [1]. First, JIT compiler uses only a fraction of memory that AOT compiler would because we do not have to compile entire program at once; we simply compile whatever is necessary “now” (i.e., compiling a method being currently called). After obtaining machine code, we cache the function (and possibly its result for some state and arguments) so that we do not have to re-compile it again. Hence, we only focus on the necessary and frequently-called functions.

Unlike interpreters, JIT compilers can also gather statistics about called methods at runtime. Knowing which methods are the *hotspots* (places where a program spends most of the time executing) and/or are frequently called, JIT compiler may aim on optimizing these more aggressively. In addition, JIT compilers may leverage target-specific optimizations that interpreters ignore completely. Therefore, JIT compilation provides a better (“smarter”) way to turn high-level source code into native machine code.

3.2 Static borrow checker

`rustc` generates and uses (sequentially) three levels of IR. While performing static program analysis such as liveness or pointer analysis, Rust compiler refers to (i.e., copies

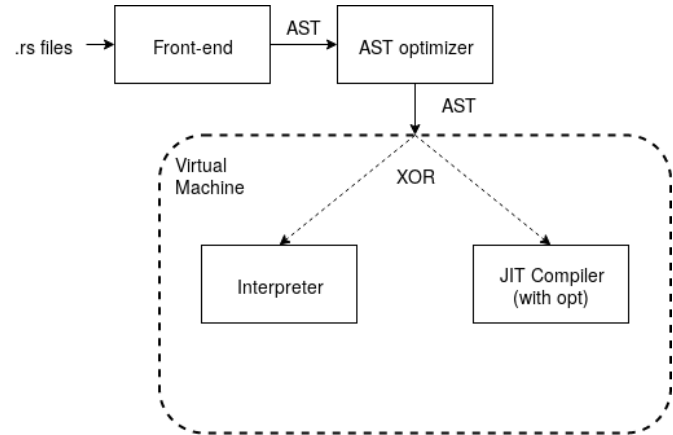


Figure 3. High-level architecture of the system.

and/or transforms) the mid-level IR (MIR), which is obtained from the high-level IR. The MIR is a control-flow graph-based IR which provides explicit type information [8]. Providing a program’s explicit execution flow, the MIR makes it possible to perform the *flow-sensitive* analysis such as borrow checking.

It is hardly possible to do analysis on the original source code. As an interpreter directly executes the code, it cannot perform any of these checks. Since I only implemented dynamic compilation for a subset of the language, it can handle borrow checking analysis *only*. However, as our project starts to support more of the language, interpreter will not be able to perform such checks *quick enough* (or, at least, as quick as the original static compiler). We could handle that case by taking an inspiration from the Java world, namely the HotSpot JVM [6]. At a high level, JVM is a system with two compilers: an AOT compiler, which reads source code, performs (if any useful) optimization transformations, and outputs the corresponding bytecode; and a JIT compiler, which then takes the bytecode, optimizes it, compiles and executes it. In a similar fashion, I present a system with two separate compilers (see Figure 3): the front-end compilers that outputs tree-based (abstract syntax tree, or AST) representation, optimizes it statically, and outputs a better AST; and the virtual machine that takes the AST and either interprets it, or JIT-compiles it into a machine code (which is then executed).

Due to the separation of concerns (i.e., *static* front-end and *dynamic* interpreter and JIT compiler), the system attains high modularity and less coupling. The modular

```

331  /**
332  * Class for variable declaration.
333  */
334  class VarDeclStmt: public Stmt {
335  public:
336      VarDeclStmt(Token name, Expr* initializer, bool _mutable)
337          : m_name{name}, m_initializer{initializer}, m_mutable{_mutable} { }
338
339      // ...
340
341      Token m_name;
342      Expr* m_initializer;
343      bool m_mutable;
344  };
345
346
347

```

Figure 4. AST.hpp maps each rule to its own class.

design allows for better unit and integration testing, as well as easier debugging via isolation of the components.

4 Implementation

This section describes the high-level design and architecture of the project; the challenges I faced while realizing the project; as well as the details of the implementation of the borrow checker. Despite facing numerous challenges, I managed to implement the project *partially*; it includes the front-end compiler and interpreter. Integrating OMR’s JIT compiler remains a future goal.

The source code open sourced and is available at github.com/oneturkmen/rust-jit.

4.1 Design

Before I present the design of the project, I must note that the ideas for architecture as well as design patterns were taken from the *Crafting Interpreters* book by Bob Nyrddstrom [5].

The design of my implementation is similar to the one described on Figure 3. The entire system works as a *pipeline* that takes in raw Rust code, transforms it into an IR, runs optimizations on it, and finally, either directly executes it, or JIT-compiles the IR into machine code.

To make project’s components more reusable (namely, the AST serving as an IR), I resorted to a hybrid of two design patterns (namely, *visitor* and *interpreter*). The hybrid approach also makes the implementation code readable, testable, and extensible enough for future changes.

Each class in `ast.hpp` represents a corresponding grammar rule. For example, take a look at Figure 4.

`VarDeclStmt` inherits from abstract `Stmt` class, and contains `Token m_name`; and `Expr* m_initializer`; as l-value (variable name) and r-value (value to declare a variable with), respectively.

Visitor pattern allows us to add functionality on the AST classes without many changes. We only have to subclass a template class (as in Figure 5), pass a template argument that we want methods to return, and provide implementation for these. As of AST classes, we only need to add method definitions to “accept” a visitor class. This could be further improved by turning the code in `ast.hpp` into a more generic variant.

4.2 Challenges

The current design was not the kick-starting idea. Initially, to transform Rust code into the AST, I planned to use `syn` crate (package) from Rust’s ecosystem. The package turns Rust into an IR without any issues, with the exception that it generates too complex (to understand what I need from it, *exactly*) data structure that represents an MIR (or, an annotated control-flow graph). In addition to the MIR’s complexity, I immediately realized that I will not be able to pass the data structure to OMR as the latter currently needs C++ code to run.

As of a more fine-grained design, I initially thought to have a large “switch” statement that would branch off the control flow for each of the grammar rules. However, it is obviously a naive (and quite *bad* in many ways) approach. I found the hybrid approach to be better; besides, it was quite simple to understand that the AST classes would represent grammar rules, visitor classes would traverse the AST classes, and perform the necessary functionality without causing me any headache.


```

441 template <class T>
442 class ASTVisitor {
443     public:
444         virtual T visitVarDeclStmt (VarDeclStmt* varDeclStmt) = 0;
445         virtual T visitExprStmt (ExprStmt* exprStmt) = 0;
446         virtual T visitPrintStmt (PrintStmt* printStmt) = 0;
447         virtual T visitExpr (Expr* expr) = 0;
448         virtual T visitAssignExpr (AssignExpr* assign_expr) = 0;
449         virtual T visitIdentifier (Identifier* identifier) = 0;
450         virtual T visitLiteral (Literal* literal) = 0;
451         virtual T visitBinaryExpr (BinaryExpr* bin_expr) = 0;
452         virtual T visitUnaryExpr (UnaryExpr* unary_expr) = 0;
453         virtual T visitGroupingExpr (GroupingExpr* group_expr) = 0;
454     };

```

Figure 5. ASTVisitor for any class that wants to “visit” an AST.

4.3 Borrow checker

As previously explained, the borrow checker makes sure that ownership principles are not violated at any given statement. For the scope of this class, I implemented a simple borrow checker that relies on two hash maps (`std::unordered_map<>` in C++). One map (**env**) stores local variables in a certain environment (e.g., in a function). It maps variables to their values, as well as the information about mutability (whether a variable’s value can be changed or not) and ownership (whether a variable’s value is borrowed) of the variable.

The other map (**pointer**) stores a pointer as a key, and the referenced variable (i.e., where it points to) as a value. Unlike **env**, **pointer** stores the mutability of the references.

In other words, we store the information about **ownership** and **mutability** as boolean types. They help us answer questions as “Is object borrowed?”, “Is object *mutably* borrowed?”. These “flags” help us determine whether some statement (assignment expression or variable declaration) violates the ownership rules.

Despite its logical simplicity, the graph representation would have been better suited for tracking the ownership and mutability of objects. First, it is easier to track references, especially nested ones, and visualize them (as a graph). Second, a garbage collection framework, which will highly likely be introduced in future to manage heaps, uses a graph representation to get rid of unused memory locations (e.g., mark-and-sweep algorithm).

5 Evaluation

I did not perform any benchmarking or evaluation program because the dynamic compilation infrastructure does not support “enough” of the language yet. There

are, however, official benchmarks that Rust compiler is evaluated against. These are the programs that I plan to verify my main ideas (hypothesis) against, to see how well a dynamic compilation system works.

As such, there is currently only support for unary, binary expressions; variable declaration statements; assignment statement expressions; as well as references (accessing an address, or equivalently, dereferencing an element). The next step would be to introduce a notion of *scope* (as well as functions) into the program. After scopes, functions, lifetimes, and (at least) closures I could perform benchmarking against the set of programs provided by Rust community [7].

6 Related Work

There is currently no dynamic compiler (whether an interpreter or JIT) that entirely supports Rust. For example, there is HolyJit, a “generic purpose just-in-time compiler for Rust” [4]. It extends static Rust compiler by obtaining its intermediate representation (MIR) and converts it into a representation for the JIT compiler from HolyJit library. Despite supporting much more language features than my system, there is currently no one working actively on the project. As it has been inactive since mid-2018, not much can be done given that Rust itself evolves quite rapidly (e.g., it has recently introduced **async/await** into the language).

Nevertheless, the Rust compiler, **rustc**, is also getting faster “with every commit”. Section 1 briefly describes the performance of the default Rust compiler.

7 Conclusion

In this paper, we took a look at a potential idea of dynamic compilation of Rust. We saw that the static

analysis checks such as borrow checker can be separated into their own stage in the pipeline. The pipeline, thus, contains both static and dynamic aspects: on one side, we are validating the source for ownership (and other) rules; on the other, we are dynamically compiling, optimizing (in a *smarter* way) and generating machine code at runtime.

In future, one could look into branching off directly from the static Rust compiler. That way, we do not have to think about implementing front-end (performing lexical, syntactic and semantic analysis) because the work has already been done by the Rust compiler itself. In addition, using the HIR/MIR representation would give us more semantic information about the source that I currently generate and store. The same information could be used to perform additional analyses (e.g., taint analysis).

References

- [1] Louis Croce. 2014. Just in Time Compilation. (2014).
- [2] The Computer Language Benchmarks Game. [n. d.]. C gcc versus Rust fastest programs. Retrieved November 29, 2019 from <https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/gcc-rust.html>
- [3] Steve Klabnik and Carol Nichols. 2019. *The Rust Programming Language*. No Starch Press.
- [4] Nicholas B. Pierron (nbp). [n. d.]. HolyJit: Generic purpose Just-In-time compiler for Rust. <https://github.com/nbp/holyjit>
- [5] Bob Nystrom. 2015-2018. *Crafting Interpreters*.
- [6] Kenneth Russell and Lars Bak. 2001. The HotSpot Serviceability Agent: An Out-of-Process High-Level Debugger for a Java Virtual Machine. 117–126.
- [7] rust-lang nursery. [n. d.]. The Benchmark Suite. <https://github.com/rust-lang-nursery/rustc-perf/tree/master/collector/benchmarks>
- [8] Guide to Rustc Development. [n. d.]. The MIR (Mid-Level IR). Retrieved November 29, 2019 from <https://rust-lang.github.io/rustc-guide/mir/index.html>