Secure Compilation Project Abstract

Ari Primak

November 12, 2022

1 Problem

I don't have much experience with C (I've mostly used C++, Python, and Java), but for a long time I've been interested in getting more practice with it due to how commonly its found in real software development. I've also seen a lot of hype about Rust the last few years, and while I'm sure the enthusiasm will wane somewhat as the next fad takes off it has had enough engagement from industry that it's clear it would be a useful skill to have. Because of how similar Rust's syntax is to C's, it seems more worthwhile to become familiar with the current-gen tool and know I can fall back into somewhat-familiar waters if I find myself needing C proficiency.

2 Relevance

After doing some research, I have learned that Rust doesn't require manual memory management and it achieves this not through use of a garbage collecting algorithm but instead with a new idea of memory "ownership". I don't feel confident I can explain it precisely without further reading, but to summarize: there can only be one mutable reference to a value at any point (plus an arbitrary amount of read-only references), as tracked by the compiler during compilation, and this ensures memory safety (and type safety by implication, I think). Additionally, during compilation the compiler keeps track of at which point each variable will never again be referenced, and once it determines that it automatically adds instructions to the target code to de-allocate associated memory. The compiler will not compile any code that is unsafe unless it is inside an unsafe block, and the only alternative paradigm through which variable lifetime is tracked is with specific RC() and Arc() pointers ("(Atomically) Reference Counted") which can be used to implement values with multiple owners. I'm not sure on the exact behavior of these pointers but they seem to be common tools so I'm sure I'll know more once I've worked with them.
Another compelling benefit to using Rust is that it has actual (literal, even?) zero-cost abstractions, which means that regardless of how the source code is implemented (through functional or imperative styling, any logically equivalent data structures, etc) the target code produced will be the same.

Here are some of the pages I found information on, and although I didn't look at it this time the official documentation seems well fleshed out:

- `https://discord.com/blog/why-discord-is-switching-from-go-to-rust`
- `https://deepu.tech/my-second-impression-of-rust/`
- `https://stackoverflow.com/a/32678736`

3 Goals

I'd like to write a (very) simple programming language with very generic syntax to use as a source language for a homebrew compiler coded in Rust that compiles that language into something typical, like Python (I'm not going to set in stone what language to target until I decide how broad a range of functionality I aim to include in the simple language, as I will lock that spec in before moving on to the compiler). I think this kind of task that requires the development of diffuse tools that must in conjuction is a good exercise towards gaining mastery over a new language. In the past I've found that the broader a base of study I begin learning a language with the easier it is to pick up new techniques as needed later.

4 Deliverables

The simple, generic programming language (I expect this to make up only a tiny part of the total complexity of the project) and a compiler that compiles that language into functioning code in a commonly-used language. If I decide to target a language like C or Java, I will require an additional normal compiler to compile the result from my compiler into executable code, but relying on a Python (or whatever language) interpreter is also an extra step even if its not as visible to the user, so I don't see a meaningful difference and will decide on a target language once I've formalized my plans for the simple source language.

5 Intermediate Milestones

1. Finish the spec of the simple language
2. Decide on a target language

It's really more like 1 and 1.5 though, haha.