Comparison Analysis on Rust and C Through the Lens of GNU grep Implementation

Angela Yang, *CS 131 Winter 2023 Discussion 1C*

Abstract

Rust is a fairly new and extremely versatile language, developed with the intention of creating a language with the speed and low level capability of C/C++, without the error-prone memory management system. I will be going over five arenas and analyzing how Rust performs in each of them. These arenas are reliability, performance, ease of use, flexibility, and security. Each of these are vital when evaluating programming languages as they determine how the language will likely be used. After assessing Rust's features, I will introduce grep as a case study for the parallelizability of Rust. I chose grep and parallelizing as the topics of analysis as grep is a widely used unix tool, and multithreading involves all of the five arenas. Both of these have widespread implications, and thus are the motivation of this analysis.

1. Rust Assessment

1.1. Reliability

Reliability is arguably, the most important aspect of a programming language. Software must be reliable above all else. If a language is easy, fast, secure, and flexible, yet doesn't produce correct results, this will render it useless. As such a vital field, reliability has several determining factors: reliable memory management and reliable error checking for common errors, such as type errors, scope errors, etc. As stated above in the abstract, memory management was a huge priority in the development of Rust, as C/C++ are valued for their performance yet disliked for their very unrestricted memory management, which causes the majority of the unreliability in C/C++ written software. Rust's solution to this, was an ownership and borrow-checker system. In layman terms, the principle of this relied on the idea that each value belonged to only one owner. Ownership could change only when assignment occured and in this way as soon as a value was assigned to something else, the old one was automatically unuseable and freed. This eliminated need for use side memory management. [1, Ch 4.1-4.2]

Secondly, is Rust's frontloading error checking to compile time. Types were static so could be checked during compile time. While C/C++ are both statically typed--so this isn't a comparison to those languages--many other languages that hoped to achieve this reliability utilized runtime checking that cost the user performance as well as made errors harder to find as programs now needed to have cases that would hit the bugs in order to ever find them. Rust had compile time checking to eliminate such invisible bugs, and thus increased reliability. [1, Ch 3.1-3.2] In conclusion for this arena, Rust is extremely reliable.

1.2. Performance

Rust is also incredibly fast, due to the aforementioned compile time frontloading. Though compilation may take longer, runtime was incredibly fast as no checks had to be done: the program could just be run.

Rust is also faster thanks to the unique ownership memory management method. This made garbage collection obsolete and thus there was no overhead from creating more objects from the garbage collector. [1, Ch 4.1-4.2]

Rust is also modeled after C/C++ and thus can be very low level. This makes for faster compilation as it is closer to machine language. In conclusion for this arena, Rust is very fast.

1.3. Ease of Use

Though ease of use is subjective and varies from developer to developer, in general the learning curve for Rust comes from the ownership model yet again. This is because it is quite unique and harder to learn than other memory management systems. For beginner developers, this may feel unncessary as they don't understand the costs of not having this system. For experienced developers coming from other languages, knowledge of previous programming language may make it harder to learn Rust as it is so vastly different in this aspect.

Rust also supports a lot of customizability which makes it difficult to begin for those who are new to the language, and may actually get in the way, as there's less of a conventional method. In conclusion for this arena, Rust is not very easy to learn.

1.4. Flexibility & Generality

As mentioned in the previous section, Rust allows for much customizability. It allows for the option of high level or low level programming, so Rust can be customized to the project or software being written [1, Ch 19.1]. It can also support different programming paradigms, such as procedural, object oriented, functional, and reactive [1, Ch. 6.3, 13]. This allows for many different styles of programming, as well as ability to utilize Rust for many different kinds of software. In conclusion for this arena, Rust is very flexible and generalizable, and allows many customizations.

1.5. Security

Security is somewhat difficult to analyze, as it refers to the limitation of access to data structures, so that clients or other code can't mess with other parts of the codebase. This can be realized in a few different ways.

Firstly, the ownership model once again lends itself to safer access to objects, eliminating access to objects that don't exist anymore. Objects will only be used or accessed by one owner or borrower at any given time. This is good for memory management in general but also for helping prevent buffer overflows and double free errors, as all of these are predicted during compile time and Rust can predict such vulnerabilities and refuse to run. Memory not belonging to the current program would never be accessed in this way. This also has implications in concurrency which allows threads to access memory only when it borrows it from the owner, or owns it. This effectively creates the effect of locks, and prevents collision. [1, Ch 4.1-4.2]

A few other features exist to prevent integer overflow and null pointer dereferences, which are all checked at compile time. Rust ensures predictable behavior in its program in this way.

2. GNU grep

1.1. Brief Overview of grep

grep is a command line tool that is used to conditionally select and return lines from files that satisfy a given regular expression. grep allows for flexibility by having flag options that allow for customizability in how grep conducts its search. GNU grep is written in C so it runs fairly quickly. GNU grep is fast because it avoids looking at every input byte. Instead, it looks first for the final letter of the target string, and skips forward based on the Boyer-Moore algorithm. However, even with these optimizations, grep is still single threaded. If these benefits were available but grep was also multithreaded, this would significantly improve performance.

1.2. Concurrency in grep

grep is used many times a day, and scans oftentimes, very large files, so performance and speed is very important. However, reliability and correctness is also highly important as in any highly used software. grep is parallelizeable, as the file can be divided into chunks that may be searched in parallel and then results can be merged at the end. Other implementations of grep have utilized this, but the mainly used GNU grep is single threaded regardless. We will now discuss the possibility of Rust-implemented parallelizable grep.

3. Rust and C comparison

Rust's fearless concurrency allows multiple threads to access data safely, without data races. This is because it utilizes ownership and borrowing restrictions to act like locks, ensuring that threads can't access the same piece of memory at the same time as only one can own it at a time. [2] Rust is able to utilize the same algorithm to avoid looking at each byte, and because it is also mostly compile time checking, with no garbage collection, the speed is comparable to C. A version of grep has been written in Rust called ripgrep, which utilizes the Chase-Lev work-stealing queue for distributing work. It is incredibly fast but lacks mobility as it's not POSIX compliant and in this way, GNU grep is the better option. However, in an analysis on parallelizability, reliability, security, and ease of access, it becomes claer that Rust is the better option for writing grep. [2]

References

1. Klabnik, S, Nichols, C. The Rust Programming Language. No Starch Press. 2022;Ch0-21. https://doc.rust-lang.org/book/title-page.html

2. Gallant, A. ripgrep is faster than {grep, ag, git grep, ucg, pt, sift}. Andrew Gallant's Blog. 2016. https://blog.burntsushi.net/ripgrep/

3. Haertel, M. [Email] why GNU grep is fast. freebsd-current Mailing List. 2010. https://lists.freebsd.org/pipermail/freebsd-current/2010-August/019310.html

ChatLog

Me: can you elaborate on the part about supporting multiple programming paradigms

Chat:

Sure! Rust is a language that supports multiple programming paradigms, meaning that it allows developers to use different approaches to solving programming problems. Here are some examples of the programming paradigms that Rust supports:

1. Procedural programming: This is the most basic programming paradigm, where the program is organized around a sequence of statements that modify the program state. Rust provides support for procedural programming with its functions and control flow constructs like loops and conditionals.

2. Object-oriented programming (OOP): OOP is a programming paradigm that is based on the concept of objects, which encapsulate data and behavior. Rust supports OOP by providing structs, which allow developers to define custom data types with associated methods.

3. Functional programming (FP): FP is a programming paradigm that emphasizes immutability, pure functions, and higher-order functions. Rust provides support for FP with features like closures, iterators, and the Option and Result enums, which are used for error handling and returning values from functions.

4. Reactive programming: Reactive programming is a programming paradigm that is based on the concept of streams, which represent a sequence of events over time. Rust provides support for reactive programming with libraries like Tokio, which allows developers to write asynchronous, event-driven programs.

By supporting multiple programming paradigms, Rust allows developers to choose the programming approach that best fits their use case. This flexibility can be especially helpful for developers who are coming from other languages or who are working on complex projects that require a mix of different programming approaches.