CS 131 Professor Eggert HW6 Report

Evaluating the Rust Programming Language

Justin Mealey

Introduction

Dudelsack is a large data analytics firm that processes millions of files, each containing potentially billions of lines. GNU's '`grep -r`' command is great for this, except for the fact that it is single-threaded. An application that could look at files/lines in parallel would be very beneficial, speeding up performance, improving efficiency and generating revenue. To research alternatives, I am looking into the Rust Programming Language, to evaluate its general strengths and weaknesses, as well as how it would fare in parallel file processing.

Rust Overview

Rust (version 1.85.0) is an imperative, general purpose programming language that takes innovative approaches to memory, concurrency and syntax to combine simplicity with high performance and safety. Programming in Rust is memory safe, thread safe, and fast due to its unique ownership paradigms and compile time checks. Learning Rust is an easy process compared to many other languages, as its system documentation is top notch. Firstly, *The Rust Programming Language*, also coined "The Book" is a fabulous text detailing essentially everything one needs to get a great grasp on the language. As well as this, Rust documents their standard library very well, so programmers can learn the ins and outs of the primitive types and standard functionality that comes with Rust. Companies all around the world use Rust for a wide variety of purposes, from systems programming to game development to security-critical applications.

The Rust Compiler

Rust is a strict, static-typed language that enforces borrowing/ownership rules to improve memory management. Rust also has type inferencing, making it easy for the programmer to be correct and even easier to fix incorrect programs with typing mismatches. Each value in a program has a single owner (ownership can be borrowed), and only the owner can reference that value. By combining this idea of ownership with lifetimes, or only letting values live under their appropriate scope, the compiler can eliminate dangling pointer and memory leak issues. This approach ensures memory safety while not requiring a garbage collector, allowing Rust to approach C/C++ levels of speed. As Rust catches many errors at compile time, programmers are also given detailed error messages and have an easier time debugging.

Syntax

Rust provides ease of use and simplicity through its elegant yet informative syntax. Firstly, Rust clearly distinguishes between mutable and immutable variables with the '`let`' and '`let mut`' keywords, avoiding the mutability confusion from languages like Python. Programmers can do explicit casting with the '`as`' keyword, which is both elegant and enforces safety and predictability (no implicit casting). Functions, control flow, and structs all have simple syntax reminiscent of C++ and OCaml, so many developers will naturally pick up the syntax. With a simple, powerful syntax, Rust creates a safe, predictable and robust language that programmers can enjoy writing in.

Security

Security within a Rust program is created with effective memory management, strict typing, and a strong compiler that rejects unsafe coding paradigms. Compared to C, Rust provides a much more secure environment via its compiler, safety checks, and less dangerous types. Rust has no '`null`', avoiding undefined behavior and crashes with dereferences. Out of bound errors with objects like arrays are instantly caught, preventing popular injection attacks and buffer overflows. Due to Rust's ownership paradigm, uninitialized memory and memory after freeing cannot be accessed, further improving security. Dangerous code is an issue in any application or language, but Rust allows developers to pay special attention to these areas by blocking them under the '`unsafe`' keyword. The concept of borrowing, when applied to two or more mutable borrows, ensures that we cannot have memory dangerous changes that could potentially lead to undefined behavior when another mutable borrower dereferences the object. Overall, Rust takes a strong stance on security, implementing a safe set of rules that force programmers to write secure code.

Parallelism

Safety and efficiency are the two key goals of concurrency in any programming language. Rust tackles undefined behavior and data races at compile time, accomplishing both of these goals swiftly. The '`thread`' object is a part of the std library. The ownership concept yet again provides safety here, as only one thread can use any specific data at once. This also allows Rust to process large files very efficiently, as threads can split up large files safely with fast execution. The Rust API provides a simple interface for creating threads ('`spawn`'), moving ownership between threads ('`move`'), borrowing data ('`&`' and '`&mut`'), sleeping ('`sleep`') and more. Classical concurrency objects such as Mutexes are also commonly used. Additionally, popular libraries such as Rayon can allow for further optimizations, and further speed up processing times on large files. Overall, Rust absorbs the advantages of classical concurrency ideas of other languages, with additional safety built-in due to strict ownership rules.

Strengths and Weaknesses

Rust contains more strengths than weaknesses, as it learned and took from many popular programming languages to create a hybrid model that exploits and combines other models' biggest strengths. Rust's innovative compiler allows for great memory safety without the need for a garbage collector, leading to blindingly fast execution speeds. Parallelism and security are priorities of the language, with built in rules and libraries that make these major programming concerns less of a headache. As well as this, the syntax is easily digestible and error catching at compile time with great debugging messages make development in Rust hassle-free.

However, everything comes at a cost. Although Rust's innovative ownership/borrowing rules draw many benefits, due to their unique nature it is difficult for programmers from other languages to grasp and master. While the compiler does do a ton of brilliant work aiding in the creation of robust and bug-free programs, this makes compilation costly and slow. As well as this, due to Rust's relatively new timescale, high level libraries have not been developed to the extent that they have in Python, for example, and interfacing with existing programs written in other languages may be challenging.

Is Rust a Good Fit For Dudelsack

Rust would be an amazing fit for the task at hand. Firstly, Rust's syntax is elegant and simple, so it would not be too treacherous to switch to Rust for the project, as developers could learn it fairly quickly. Dudelsack's application is performance intensive, something Rust excels at. Although '`grep`' is a highly optimized command line tool that efficiently searches files using regex, it has its limitations. It is single threaded, and requires manual memory management, often leading to bugs. Using the regex crate, Rust can effectively gain access to the benefits of '`grep`' while exploiting its great concurrency features. Dudelsack could effectively process large files in a safe, race condition free manner with memory safety guarantees. Getting the threads to cooperate and ensure ordered output is an inevitable bottleneck, but classical wait methods along with Rust's ownership guarantee correctness. Another inevitable bottleneck we would face is doing I/O in the files themselves, as they would need to be read in from external storage. These issues would be prevalent in any language, but with Rust we would have more correctness guarantees without sacrificing performance. Overall, Rust provides a unique combination of simplicity, performance, safety and parallelism, making it an excellent candidate for the application.

References

[1] GeeksforGeeks. *Introduction to Rust Programming Language*. November 14, 2024. Available from: https://www.geeksforgeeks.org/introduction-to-rust-programming-language/.

[2] Klabnik S, Nichols C, Krycho C. *The Rust Programming Language*. Available from: https://doc.rust-lang.org/book/.

[3] *The Rust Standard Library*. Version 1.85.0. Available from: https://doc.rust-lang.org/std/.

[4] *Overview of the Compiler*. *Rust Compiler Development Guide*. Available from: https://rustc-dev-guide.rust-lang.org/overview.html.

[5] *Rust by Example*. Available from: https://doc.rust-lang.org/rust-by-example/index.html.

[6] Sible J, Svoboda D. *Rust Software Security: A Current State Assessment*. December 12, 2022. Available from: https://insights.sei.cmu.edu/blog/rust-software-security-a-current-state-assessment/.

[7] The Rust Programming Language. Concurrency. MIT. Available at: https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/concurrency.html