Proposal: Algorithmic Trading Application Based on Existing Algorithms in Rust

Live monitoring of stocks prices and executing trades on price targets

Amelia Milne
University of British Columbia
amelia.milne@gmail.com

Seungyup (Steven) Lee
University of British Columbia
syl24@live.com

Karan Grover
University of British Columbia
karan.kg.grover@gmail.com

Pahul Panesar
University of British Columbia
pahulpn@gmail.com

Josh Cheung
University of British Columbia
se77enn@gmail.com

ABSTRACT

This paper proposes the usage of Rust to implement a stock trading algorithm.

KEYWORDS

Rust, Stock, Algorithmic trading, Generic, Arrowhead, Liquidity, Detrending

1 INTRODUCTION

In 2015, Mozilla Research released the first stable version of a new programming language named Rust. Many features make this new language attractive for the scientific community, it is open source and it guarantees memory safety while offering zero-cost abstraction. We choose this topic because we were interested in how we would be able to use a functional language like Scheme, Rust or Haskell to create an algorithm that buys and sells stocks based on a researched algorithm. We choose Rust as our language of choice and will be exploring the language to create an application that will execute trades based off of a trading pattern we find through research. We have identified some potential algorithms below and further discuss why we choose Rust.

2 PROJECT TYPE

Prompt Four: Creating a substantial program in a potentially useful but esoteric (i.e., "weird" and ideally cutting-edge) language that you have not already studied. Your program should illustrate the particular value of the language.

3 WHY RUST?

We choose Rust to implement algorithmic trading because of its admirable qualities of speed and durability.

- Rust proves at compile time that certain types of errors are impossible and it figures out exactly when to free memory which means it doesn't need to insert time-consuming checks.
- Rust won't compile programs that attempt unsafe memory usage by ensuring that common memory problems such as null or dangling pointers and data races never make it into production. The compiler flags those issues and forces them to be fixed before the program ever runs. [6]
- Rust has strict memory management rules known as ownership. Any given value can be manipulated only by a single variable at a time.

These qualities are all very important to an automated trading environment. It is important to maximize performance while minimizing the over fitting bias for we don't want the algorithm to be too closely based on past data. Rust's memory management is beneficial to keep track of the many changing variables involved in an algorithmic trading algorithm. Overall, Rust is an ideal language to implement our goal.

4 RUST DOCUMENTATION

The following outlines useful resources about Rust documentation with links to helpful tutorials and more information. Link: Rust Documentation[4]

- An extensive Rust tutorial can be found at this link: The Rust Programming Language Book[4]. It contains a collection of concept chapters and project chapters covering the following: installation, common programming concepts (such as variables, mutability, data types and functions), Rust's ownership system, structs, enums and pattern matching, control flow, modules, privacy rules, common collection data structures, error handling, generic data types, traits, lifetimes, closures, iterators, Cargo, smart pointers and concurrent programming.
- Rust by Example (RBE) should be used in conjunction with this tutorial and can be found at this link: Rust by Example[2]. RBE is a collection of runnable examples which concretely illustrate Rust concepts and standard libraries.
- The Rust Standard Library can be found here: Rust Standard Library[5]. It is divided into a number of focused modules. Their documentation includes an overview of the module along with examples. Implicit methods on primitive types are also documented here. Also, although primitives are implemented by the compiler, the standard library includes and implements methods directly on the primitive types.
- Rustc is the compiler for the Rust programming language. Compilers take your source code and produce binary code, either as a library or executable. Information on how to use rustc is at this link: The rustc Book[6]. For the most part, Rust programmers do not invoke rustc directly and instead

mobilize Cargo. An index to Cargo will be included in the next point.
- Cargo is the Rust package manager. Cargo downloads the Rust project dependencies, compiles the project, makes packages, and uploads them to crates.io, the Rust community package registry. A guide to using Cargo can be found here: The Cargo Book[1].
- The standard Rust distribution ships with a tool called rustdoc, referenced at this link: rustdoc Index[7]. Its job is to generate documentation for Rust projects. Rustdoc takes either a crate root or a Markdown file as an argument and produces HTML, CSS, and JavaScript. Cargo also has integration with rustdoc to make it easier to generate docs.
- The following link is to the Rust error index which will be useful when debugging code: Rust Error Index[3].
- An additional tutorial for reference is included here: Rust Tutorial[8]. It covers basic Rust syntax and more in depth detail on pointers, data structures, concurrency, the standard library, and the rest of the language.

5 MILESTONES

- Project Proposal (Oct. 26) - Proposes the project intentions, what language we are going to use, what are we interested in changing, why does it need to be changed, and projections.
- Internal Research (Oct. 27 - Nov.8) - Gather all necessary research regarding Rust and it's benefits it brings to existing algorithm based trading applications. Afterwards, start on draft of research report.
- Background Research Report (Nov. 9) - Clearly state the importance of our topic: implementing a trading application in Rust. While relating to our topic, research the technical benefits, impacts it may have, and value that may be generated in everyday programming practice. This may include analyses, semantic forms, other projects/idea, or similar languages. Last, describe a project that can satisfy a 100 percent goal.
- Internal Development (Nov. 10 - 22) - Algorithm/API selection and creation of initial commit: As a group, we shall start implementing the structure of the trading application. With the decision made on which API and Algorithm would be suitable, an initial commit will be made. At the very least, create a blueprint of how the application is going to be created, with all components.
- Proof of Concept (Nov. 23) - Provide a clear plan for how you would complete your Final Project. In particular, lay out the minimal core goals you want to achieve for the final project and describe a very achievable "simplest suitable" approach that will help you achieve those goals. Then describe a grander, full set of goals and a more ambitious approach that would help you achieve your full goals.
- Internal Creation (Nov 24 - 29) - Gather all the necessary pieces for the poster, and spend a day or so together to make the poster.
- Poster (Nov. 30) - A poster will be made outlining our work, including the inception, reasoning, decision to use Rust, the added benefits, what is being changed, and what added values are present.
- Internal Polishing (Nov. 31 - Dec. 2) - Polish our work, collect all the necessary things, get ready to present.
- Final (Dec. 3) - Illustrate that you have completed the project plan. Report on what you did, how you did it, and why it's important. Show case our work about the changes we've made, and the benefits of Rust.

6 GOALS

6.1 80 Percent:

For this goal we will demonstrate how an algorithm can be represented as a program and be executed on chart data. We will demonstrate what the process would look like if an algorithm were monitoring a live stock price and show example buy and sell signals. Furthermore, we will demonstrate a selected algorithm in pseudo code.

Steps:

1) Identify an algorithm that we believe would be implementable in Rust, and one that rust would have execute with high performance.

2) Write pseudo code that shows how the algorithm we selected would look like if implemented.

3) Create a flow diagram that shows how a program and APIs would interact with this newly coded pseudo function (i.e. call API for current stock price then call new function to determine if buy signal or not.)

4) Discuss the next steps we would have taken to further build this out.

6.2 90 Percent:

For this goal we will run an algorithm we find through research successfully on a historic chart for any given stock and the program will output all buy and sell signals where the algorithm conditions were met.

Steps:

1) Complete the 80 percent goals

2) Write the main function that is passed in hard coded historic stock data and returns whether to buy or not. This will show how this could be implemented in Rust.

6.3 100 Percent:

For this goal we will have an algorithm that is coded into a program that can pull live stock price information and determine whether or not to send a buy or sell signal.

Steps:

1) Complete the 90 percent goals

2) Call an API to bring in live stock data into the program instead of hard coded data.

This could look something like: Have an API updating a data set which a program runs on. When more data is passed into this dataset the core algorithm is run again and determines if there should be a trade based on certain criteria. If so a buy signal is sent out.

7 HOW WE WILL CONDUCT THE BACKGROUND REPORT

We intend on conducting the background report by separating the process into two distinct parts: 1) Research and choose a suitable trading algorithm that is elegant and practical, such that it could potentially be used in industry to solve problems dealing with stock trading. Focus on how the algorithm will solve business problems and potentially choose an API to monitor live stock market data. List of relevant articles:

- Basics of algorithmic trading. [10]
- K-means clustering algorithm. [9]
- Using genetic algorithms to forecast financial markets. [11]

2) Research the common programming concepts in Rust (mutability, data types, control flow etc.) along with its unique functional language features such as iterators and closures. The core design principle of Rust is that memory safety should be the default for all programming languages, especially low-level systems languages such as C/C++, Scheme, etc. Rust proves that memory safety can be kept without sacrificing the speed of the language. We will explore how Rust safely manages memory and how this memory management makes concurrent systems programming safe, even with manual memory management and threading.

List of relevant articles:

- Rust 2018 book.
- Rust Documentation.
- Fearless concurrency with Rust.
- Overview of memory management with Rust.
- Static garbage collection in Rust.

REFERENCES

[1] [n. d.]. The Cargo Book. ([n. d.]). https://doc.rust-lang.org/cargo/index.html

[2] [n. d.]. Rust By Example. ([n. d.]). https://doc.rust-lang.org/rust-by-example/index.html

[3] [n. d.]. Rust Compiler Error Index. ([n. d.]). https://doc.rust-lang.org/error-index.html

[4] [n. d.]. The Rust Programming Language. ([n. d.]). https://doc.rust-lang.org/book/2018-edition/index.html

[5] [n. d.]. Rust Standard Library. ([n. d.]). https://doc.rust-lang.org/std/index.html"

[6] [n. d.]. The rustc Book. https://doc.rust-lang.org/rustc/index.html

[7] [n. d.]. What is rustdoc? ([n. d.]). https://doc.rust-lang.org/rustdoc/index.html

[8] Wil Thomason Alex Lamana, Rob Michaels and David Evans. 2018. *Welcome to Rust!* University of Virginia.

[9] J. A. HARTIGAN and M. A. WONG. 1979. A K-Means Clustering Algorithm. Blackwell Publishing for the Royal Statistical Society. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28, 1 (1979). http://www.labri.fr/perso/bpinaud/userfiles/downloads/hartigan_1979_kmeans.pdf

[10] Jing Yu Hiroshi Moriyasua, Marvin Wee. 2018. *The role of algorithmic trading in stock liquidity and commonality in electronic limit order markets.* Pacific-Basin Finance Journal, Nagasaki, Japan.

[11] Szczepkowski M. Markowska-Kaczmar U., Kwasnicka H. 2008. *Genetic Algorithm as a Tool for Stock Market Modelling.* Springer, Berlin, Heidelberg, Berlin, Heidelberg.