Reoxidizer: Idiomizing Rust Code Using Recursive LLMs

Addison Goolsbee, advised by Lin Zhong and Ramla Ijaz
Department of Computer Science, Yale University

INTRODUCTION

Rust is a systems programming language focused on safety, speed, and concurrency, with a unique ownership model that eliminates data races and many classes of bugs at compile time. It offers performance comparable to C/C++ while preventing null pointer dereferencing, buffer overflows, and other memory errors without a garbage collector. Increasingly useful for all sorts of critical systems applications like Operating Systems.

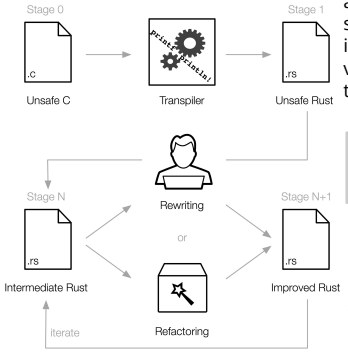
Rust's guarantees are especially powerful when it comes to its linear type system. In order to make full use of it, though, we need to code in a very idiomatic way, such as by encoding invariants into types. This would improve correctness, maintainability, and compiler-enforced guarantees, reducing the surface for bugs and undefined behavior. Unfortunately, it's hard—certainly not something one could programmatically achieve. It requires thinking.

With the emergence of LLMs, suddenly have the potential for a script to do this thinking for us, and if we make a whole system out of it, we can create an **autonomous Rust idiomizer**.

C2RUST

C2Rust is an existing project that programmatically translates C code into semantically equivalent Rust (very unsafe and unidiomatic), then refactors it into safer rust, until finally it relies on a human to make the Rust code idiomatic

An autonomous Rust idiomizer has the potential to improve on this by replacing the bottom half of the diagram, which would allow for relatively easy direct translation of C codebases into idiomatic Rust



Reoxidizer: the Autonomous Rust Idiomizer

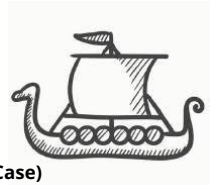
The aim of Reoxidizer is for it to take in a Rust file, a build command, and a test script as inputs, run on its own for a few minutes to an hour, and spit out cleaner, safer, more idiomatic Rust code.

TEST CASES

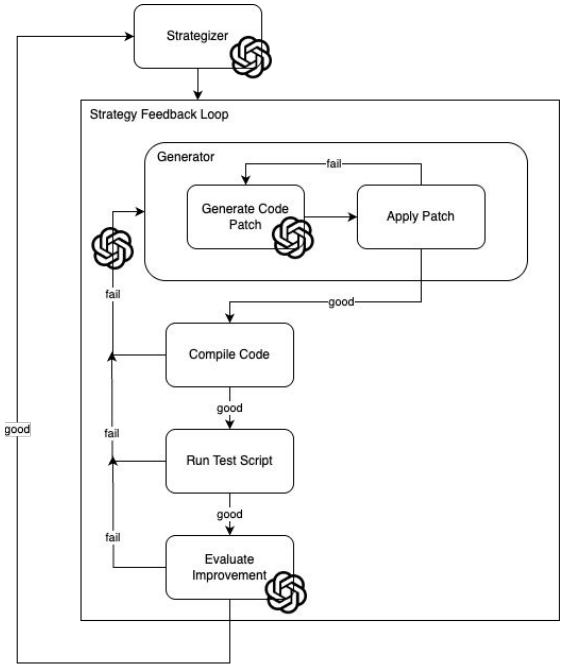
To test Reoxidizer I used three Rust files, all of which have lots of unidiomatic Rust, and are varying levels of length and complexity.

Quicksort
The quicksort algorithm in Rust. Directly translated from a C file using C2Rust, so the entire file is unsafe and poorly written. 60 lines of code, low complexity

RobotFindsKitten
A simple CLI game (top right) where you move around and bump into objects until you find the kitten. Again, a C file translated directly to Rust using C2Rust. 1500 lines of code, medium-high complexity



TheseusOS e1000 Driver (Primary Test Case)
TheseusOS is a novel Operating System written entirely in Rust. By being written only in Rust, not only does it have basic Rust safety guarantees, it also allows it to enforce clear runtime-persistent bounds, better memory safety, and many more benefits. The e1000 driver is a simple network interface card (NIC) driver, which allows Theseus to network. I used a version of the driver from 7 years ago, back when it was unidiomatic. The test script is simply "ping 8.8.8.8". 700 lines of code, high complexity.



Strategy Generation	Code Change Generation + Application Loop	Evaluation Feedback Loop

Reoxidizer can be split into three components, each with their own challenges. Strategy generation is the part where Reoxidizer reads the code, looks at the previous strategies that didn't work, and comes up with what method of idiomization it will try. The code generation/application loop is the part that actually generates the new code. New code is generated in a replacement file format instead of directly replacing the code, which speeds up generation 10x for the Theseus case. Finally, the evaluation feedback loop runs the newly generated code through a series of tests to ensure its validity. If any test step fails, it will return to the generation step using the new replacement file and a new prompt.

Results, Conclusions and Future Work

In the end, Reoxidizer performed underwhelmingly
Quicksort - Perfect: outputs a fully safe, idiomatic file within 2 minutes consistently
RobotFindsKitten - Despite being 1500 lines, code generation/application is relatively fast. Strategy generation works well, but code generation is horribly ineffective. Struggles to finish strategy loops before aborting
TheseusOS e1000 - Strategy generation is moderately effective. Code generation/application works fine enough to get past the generator quickly, but struggles to incorporate the many confusing build errors and tends to get stuck at the compilation stage.

In conclusion, creating an autonomous Rust idiomizer is **infeasible** (at least for a single college student). It may be possible to create, but each of the three components of Reoxidizer are uniquely difficult problems.