Addison Goolsbee
CPSC 490 Proposal
Advisor: Dr. Lin Zhong (lin.zhong@yale.edu)

**Transforming Rust Code From Bad to Good Using Recursive LLMs and More**

The programming language C and its derivatives like C++ are some of, if not, the most popular programming languages in existence. An enormous portion of the world's code is written in C and its derivatives, but because of the general unsafeness of these languages, this means that an enormous portion of the world's code is often buggy, including even for high-security purposes. Many tools have been designed to help catch these bugs, but it's a losing battle. Rust is a relatively recent programming language designed in such a way that it prevents many of these C-style bugs from ever happening. To illustrate just how much better Rust code is from a correctness/safety perspective, last year, the US government issued a statement that they'd like to migrate all C code into Rust code.[1] Unfortunately, the process of translating C code into Rust is difficult. Because of logical differences in the two languages, most notably Rust's linear type system, there is no 1:1 conversion from C to good Rust code possible. Many tools have been created that convert C to Rust Code, such as C2Rust, but these only work via using Rust's unsafe code feature, and do not take full advantage of Rust's linear type system.

In the past few years, automatic translation to Rust code has gained popularity. But these tools all run into similar problems when it comes to utilizing Rust's linear type system. A 2020 study on approaches of programmatic c->rust translation highlights the types of unsafety that are hardest to resolve.[2] Using Rust's linear type system effectively as specified in this paper, we could theoretically translate most or all of C code into safe, effective Rust.[3] The only problem is, how to do it automatically. The process of translating C to good Rust is difficult even for experienced programmers, and a programmatic approach leaves many problems. With the recent rise in Large Language Models (LLMs), a new paradigm of code generation has become

[1] https://www.darkreading.com/application-security/darpa-aims-to-ditch-c-code-move-to-rust
[2] https://dl.acm.org/doi/pdf/10.1145/3485498
[3] https://www.yecl.org/publications/kisv2023ijaz.pdf

possible. The type and complexity of code generated by LLMs is much different than the type generated programmatically by code. According to this paper, we can harness LLMs to create Large Model Programming (LMP), and using LMP, we can hopefully make Rust translation a little (or a lot) better.[4]

To attempt to tackle this problem, I will be creating a recursively-prompting, execution-enabled LLM program that takes in bad Rust code as its input and outputs better Rust code. While the design may change dramatically as the project progresses, the initial design will be as follows:

1) The LLM will begin with an input of the bad Rust code, and a prompt efficiently designed to find one aspect of the Rust code to improve, usually via the linear type system.
2) Using the output of the above prompt, the program will begin it's evaluation loop. If at any stage of the loop, the evaluation step fails, the program will use the output, relevant context, and a custom prompt to reprompt itself, and it will keep doing this until the evaluation step succeeds. These are the evaluation steps planned:
    a) The code compiles
    b) The code passes a basic unit test (provided by the user)
    c) A set of compile-time static assertions that need to be true for the code
    d) For proof-of-concept purposes, some code similarity metric with a good version of the Rust code e.g. BLEU [optional]
    e) An LLM evaluation of whether the new change is actually better than the previous code
3) The code will run in a feedback loop where it keeps improving on the code

[4] https://arxiv.org/pdf/2212.06094

4) Safety guardrails: The LLM will not try to execute code directly (imagine if it runs rm -rf /). Instead, it will generate an interface file with limited actions possible, and the interface file will be run. Other safety guardrails may be necessary.

Since this is a proof of concept, I will be testing on an early, unsafe version of the 1 GbE driver of TheseusOS. TheseusOS is an operating system written entirely in Rust. By making use of Rust's linear type system, it is able to make guarantees few other operating systems can. Most drivers are written in C, so in order to support them, the TheseusOS developers would have to translate tens of thousands of drivers into Rust manually (which is a difficult process) in order to guarantee their safety. Thus, C drivers are a good use-case for good automated Rust translators. The 1 GbE TheseusOS driver will be my test case because we can compare what the LLM program generates against the most recent version of the driver, which makes better use of the linear type system and is safe. Being able to use this as a comparison will help with developing the program.

As the project progresses, I expect it to focus on optimizing prompts and figuring out how to evaluate programmatically that Rust code is better than other Rust code. Depending on what approaches work and don't, the project may significantly change in direction, so this is only an estimation. The same goes with the following timeline (check-ins every Wednesday):

- Up to 1/30 - Setup
    - Proposal finalized
    - TheseusOS set up to run on my computer
    - Ability to insert e1000 driver as networking device, and modify its code
    - Ability to send/receive messages to e1000 (the base unit test evaluation step)
    - Literature review
    - Test existing c->rust tools
- 2/5 - Begin the LLM project

    - Pass in a hard-coded transformation prompt, the bad code, and an execution script
    - Implement the first evaluation loop: ensuring the code compiles. If it doesn't, have the program reprompt itself in a feedback loop
        - Filter the output so it returns only the rust error message, and use this as the metric of success and as part of the prompt
    - Work on some prompt engineering to get the LLM to do what I want it to do
    - Some more literature review
- 2/12 - Finish the base evaluation loop
    - Extra week for above, mostly to be spent on evaluation loop code/prompting
- 2/19 - Unit test evaluation loop
    - Implement the second evaluation loop: the code passes a basic unit test (send/receive or ping)
    - Prompt engineering so the LLM inserts its own log messages (so it can tell if the unit test succeeded, and/or so it figures out what is going wrong)
    - Filter and clean the Theseus logs so they only contain relevant information, and use those as the feedback mechanism
- 2/26 - above
    - Second week for above; Prompt improvements to make it more efficient, more successful, and able to fix more unsafe code/apply linear type system
- 3/5 - Rust improver feedback loop
    - Add the outermost feedback loop, which is the mechanism of identifying a strategy for improving the bad rust code, and turning that into a prompt. This will replace the hard-coded transformation prompt
    - The program runs indefinitely, coming up with a new transformation prompt/strategy on each iteration

        - Mechanism to keep track of previous transformation strategies so they aren't repeated
- 3/7-3/23 - Spring Break
- 3/26 - Good Rust evaluator
    - Extra week for above, also modify it further now such that there is an evaluation at the end of each iteration to test if the new proposed Rust code is actually better than the old code. Also that it maintains the same functionality.
        - Evaluation will be rather simple (e.g. a prompt) instead of a programmatic approach of correctness
- 4/2 - Safety
    - Implementation of safety interface
    - Compile-time static assertions loo—parser that reads all assert_no_impl and various other invariants, puts them in a separate file, and makes sure they're still present in the newly generated code.
- 4/9 - Code similarity scorer and/or better 'good' Rust identifier
    - Play around with some things and see if anything helps:
    - Code similarity metric test combined with outermost evaluation loop (using BLEU?)
    - Look into more programmatic ways of marking some Rust code as better than other Rust code
- 4/16
    - Final report draft due 4/15
    - Code cleanup
- 4/23
    - Final report and poster

**Current References**

- https://www.yecl.org/publications/kisv2023ijaz.pdf
    - About using type system effectively within Rust
- https://dl.acm.org/doi/pdf/10.1145/3485498
    - Review of the problems with the programmatic approach of translate C to Rust
- https://arxiv.org/pdf/2212.06094
    - About using LLMs effectively to generate code. Specifically discusses developing a custom LLM, an LDP, to generate code more efficiently (which is probably above-scope for this project)