Toward LLM-based Large-scale C-to-Rust Code Translation

Momoko Shiraishi
The University of Tokyo
Tokyo, Japan

Takahiro Shinagawa
The University of Tokyo
Tokyo, Japan

Extended Abstract

While C language has been widely used in existing applications, C programs often suffer from memory safety vulnerabilities. In recent years, Rust has been gaining increasing attention as a memory-safe language suitable for system software, triggering a motivation to migrate existing C code to Rust. There are two main approaches to the automatic conversion of C code to Rust. The first is a rule-based method, which relies on handcrafted predefined transformation rules [1, 3]. The second is an LLM-based conversion method, which utilizes Large Language Models (LLM) to perform the translation [2, 5]. While rule-based approaches can transform large programs relatively accurately, LLM-based approaches are known to produce less unsafe blocks and more idiomatic expressions [4], and therefore, LLMs have the promissing potential to produce better code more easily.

Unfortunately, LLM-based conversion is still difficult to convert large codes. For example, several studies indicate that LLM-based approaches can only convert less than 100 lines of C programs into Rust code that can be compiled [2, 5]. A study analyzing conversion bugs with LLMs showed that 50% of translation bugs were due to compilation errors in GPT-4 for C to Rust conversion, and that compilation errors were also a major factor in unsuccessful conversions between other programming languages [4]. One reason for the compilation errors is obviously the difference in syntax between C and Rust, but another reason is that LLM has a negative correlation between prompt length and task performance, leading to poor performance in large code translation. Therefore, it is worth considering breaking the original C code into appropriate smaller units for sequential conversion in order to reduce translation bugs. However, it is an open question how to partition the source C code and what prompts to use to improve the compilation success rate of the converted Rust code.

We propose an LLM-based C-to-Rust conversion scheme that leverages context-supplement prompts between split conversion to improve compilation success rates in large codes. Our scheme first splits the original C program into appropriate sizes for each LLM, since too large code degrades LLM performance and too small code loses information from other code. Our scheme then pre-parses the original C program and adds a small database representing its structure to the prompt as supplemental information to compensate for the context lost between split conversions. For example, this database stores sorted function names and function call graphs in JSON format. This database also stores information about the correspondence between the original C code fragments and converted Rust code fragments, as well as their data types. Furthermore, our scheme also analyzes dependencies between functions and files in advance, and performs transformations in the order in which functions are called. The addition of such compressed and localized supplementary context information to the prompts facilitates accurate split conversion with short prompts.

Once the code has been converted to Rust, we request the LLM to make corrections until the Rust code successfully compiles. Each code fragment is separately compiled and the referenced information is also included in the prompt of this repair process. Moreover, we ask the LLM to provide corrections in logical units that can be parsed, rather than on a file-by-file basis. In this way, we aim to automate the correction of conversion bugs and improve conversion accuracy through feedback for the next conversion.

In summary, we are trying to answer the following questions.

RQ1 Does dividing the code into smaller parts increase the compile success rate?

RQ2 Does summarizing the referencing data (rather than directly showing the referencing parts (files) to the LLM) affect the compile success rate?

RQ3 What is the appropriate scope for corrections by an LLM to avoid compilation errors? Should it be at the file level or the logical unit level?

We target 10 C programs, with lines of code ranging from 154 to 2,410. We tested them using Claude Anthropic (Sonnet 3.5) so far and the code up to line 2,410 has successfully compiled. Then, we observed that (1) parsing the original C code and strictly converting it from the referenced elements, (2) summarizing and including the information of referenced elements (such as called function signatures and definitions of used data types) in the prompt during both the conversion and compile repair phases, and (3) keeping a record of the compilation correction process, improve the compilation success rate. We plan to test with other LLMs such as GPT-4 and Gemini Pro. Additionally, we plan to analyze the equivalence between the original C code and the converted Rust code. The equivalence in the conversion is determined by comparing the results for the testcases of the programs.

References

[1] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating C to safer Rust. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–29.

[2] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2024. Towards Translating Real-World Code with LLMs: A Study of Translating to Rust. *arXiv preprint arXiv:2405.11514* (2024).

[3] Immunant. 2022. C2Rust. https://github.com/immunant/c2rust.

[4] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[5] Aidan ZH Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. 2024. Vert: Verified equivalent rust transpilation with few-shot learning. *arXiv preprint arXiv:2404.18852* (2024).