2024 International Conference on Assured Autonomy (ICAA) | 979-8-3315-2101-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICAA64256.2024.00024

Leveraging RAG-LLM to Translate C++ to Rust

Ahmet Okutan
Leidos
Reston, VA, USA
ahmet.okutan@leidos.com

Samuel Merten
Leidos
Reston, VA, USA
samuel.a.merten@leidos.com

Christoph C. Michael
Leidos
Reston, VA, USA
christoph.c.michael@leidos.com

Ben Ryjikov
Leidos
Reston, VA, USA
benjamin.ryjikov@leidos.com

***Abstract*—Despite their similarities, translating C++ code into the Rust language is a challenging task on account of differences in syntax, ecosystem, philosophy, and idioms. Large Language Models (LLMs) using Retrieval Augmented Generation (RAG) are effective at solving challenging programming language tasks, including source code generation and program translation. We leveraged RAG LLMs for translating C++ code into Rust and created Cpp2Rust, an LLM-based C++-to-Rust transpiler grounding the OpenAI ChatGPT4 model with similar C++ and Rust code snippet pairs from LeetCode. Preliminary analysis results show that 84% of the solutions produced by Cpp2Rust that had no compile errors were accepted as correct solutions on LeetCode.**

***Index Terms*—Transpilation. C++ to Rust. Large Language Models.**

I. Introduction

Large Language Models (LLMs) enhanced with Retrieval Augmented Generation (RAG) have shown significant improvement across various tasks such as information retrieval, question answering, content generation, document analysis, and summarization [1] [2] [3]. RAG-LLMs leverage domain-specific knowledge to generate context-aware responses and often achieve better accuracies for the underlying downstream tasks. Recently, RAG-LLMs have shown to be effective while solving challenging programming language tasks as well, including source code generation [4] [5] and program translation [6].

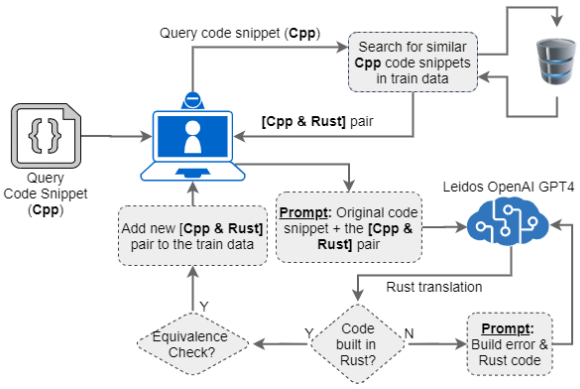

Fig. 1. An overview of the processes in Cpp2Rust, a RAG-LLM-based C++ to Rust transpiler.

In this paper, we present Leidos' experience in leveraging RAG-LLMs for translating C++ code into Rust. We created Cpp2Rust, an LLM-based C++ to Rust transpiler, by grounding the OpenAI ChatGPT4 model with similar C++ and Rust code snippet pairs. Our experiments focused on translating solutions for programming problems from the popular LeetCode platform. Data for constructing the system's knowledge graph and evaluation was sourced from a collection of publicly available LeetCode solutions found on GitHub [7]. We filtered this repository to only those problems that had matching solutions in C++ and Rust and removed any solutions from non-C++/Rust languages. This set of solutions was then partitioned into two sets, one used to build the system's knowledge base, and the other reserved for the system's evaluation. During experiments, Cpp2Rust was provided a piece of code in the form of a C++ solution to a LeetCode problem. The knowledge base was queried to obtain the most similar C++ code snippet(s) and their equivalent Rust implementations to ground the GPT4 LLM with relevant example(s) to translate the provided C++ code snippet. The code produced by the LLM is compiled to check for errors, and in case there is an error, the error is fed into the LLM together with the produced code to obtain a revised version of the translation. The process for feeding compile error(s) into the LLM could be repeated multiple times depending on the observed improvement at each step. If the generated code has no compile errors, it is checked against the original C++ code to make sure the produced Rust code snippet is equivalent to the provided one. Figure 1 shows the summary of the overall process flow for Cpp2Rust.

The main contribution of Cpp2Rust is threefold: (1) the development of an automated and extensible RAG-LLM tool to translate C++ to Rust, (2) a code knowledge base of C++ and Rust code snippet pairs used for both LLM grounding and evaluation, and (3) the dynamic integration of Rust compiler errors to improve translation quality.

The organization of this paper is as follows. Section II presents an overview of prior related works. Section III provides an overview of Cpp2Rust and its sub-modules. Section IV evaluates Cpp2Rust on some real-life examples, and reviews the major contributions outlined above, and Section V provides concluding remarks with a discussion of future work, challenges, and intended enhancements.

II. Related Research

Our work was informed by two veins of related research. The first follows the uses of machine learning techniques

979-8-3315-2101-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ICAA64256.2024.00024

in code transpilation, and the second follows alternative approaches for transpilation from C to Rust.

1) Machine-learning-based Transpilation: Guess & Sketch is a framework that outperforms purely symbolic methods in ARMv8 to RISC-V translation by allowing a language model to influence translation [8]. Neural networks integrated with concolic execution have also been used to translate imperative Java and Python code into functional equivalents [9]. Unitrans is a framework that uses LLM-generated test cases to correct a symbolic translation of given code and has been evaluated in the transpilation of C++ to Java and Python [10].

2) C-to-Rust Transpilation: Galois' and Immunant's C2Rust [11] is possibly the best-known procedural C to Rust translator. C2Rust and its predecessors such as Corrode [12] and Citrus [13], focus on syntactic translation over program semantics or idiomatic translation. Researchers have used LLMs (without RAG), to improve aspects of C2Rust's output, specifically for macro generation [14]. We hope that approaches such as ours that directly integrate existing knowledge and notions of program semantics can be used to improve, rather than recreate, the capabilities of systems like C2Rust. Similar augmentations to C2Rust include Crusts [15], which reduces the number of unsafe sections in code transpiled by C2Rust, by using the TXL framework [16] to implement additional syntactic transformations.

III. System Overview

Retrieval Augmented Generation (RAG) systems combine retrieval mechanisms with generative models to improve the performance of LLMs on knowledge-intensive tasks. Grounding LLMs with retrieved knowledge enhances their ability to generate more accurate results and contextually relevant responses. The fact that an LLM is more strongly grounded in real factual knowledge makes it "hallucinate" less with generations that are more factual, and offer more control and interpretability [17].

Our methodology leverages the power of grounding the ChatGPT4 model with similar code snippets to translate C++ code into the Rust language. We scraped the LeetCode problems and their solutions in different languages and filtered C++ and Rust solutions for each problem to create a simple knowledge base for grounding. After obtaining 714 C++ and Rust solution pairs (across 3200 LeetCode problems), we produced vector embeddings for each C++ code snippet using Microsoft's 'codebert-base' pre-trained transformer, to create a database of C++ code snippets, their corresponding Rust implementations, and 'codebert-embeddings'. Figure 2 shows the distribution of the Cpp2Rust dataset (used for grounding and evaluation) across 3200 LeetCode problems.

Taking a new C++ code snippet as input, Cpp2Rust first generates its vector embedding (query vector), and then finds the largest 'dot product' of the query vector (and other vectors) in the knowledge base to identify similar C++ snippets. We apply the dot product on normalized embeddings to ensure comparability, therefore larger dot product values might imply greater similarity of associated code snippets. Once similar

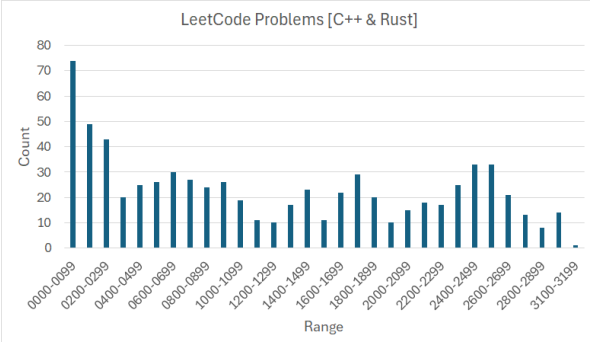

Fig. 2. The distribution of LeetCode solutions across problem ranges that have both a C++ and Rust solution.

C++ instances are identified by the dot product, Cpp2Rust grounds the LLM with these similar C++ examples and their Rust implementations and asks it to translate the provided C++ code snippet.

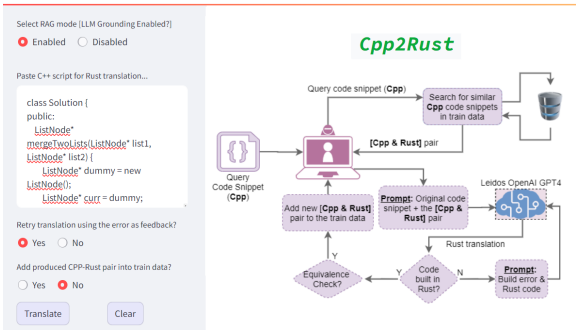

Fig. 3. An overview of the Cpp2Rust interface used to convert a given C++ code snippet to Rust through an interactive web application.

Recent works have shown that a user-guided iterative code generation approach incorporating user feedback and external verification tools such as grammar checkers, compilers, and SMV verifiers can guide the LLM code generation process and improve the overall model performance [18]. Cpp2Rust checks generated Rust code snippets for compile errors, and in case of any errors, feed these errors back to LLM to improve the translation process. Our experiment results confirm that feeding compiler errors to the LLM can fix some of the issues in the produced Rust code snippets and improve the model performance.

Figure 3 shows a screenshot from the interactive web application created to test Cpp2Rust. The user can select whether the RAG mode is enabled or disabled, or whether the translation should be tried again if the produced code is not compiled. There is also an option to add the code to the knowledge base (used for grounding the LLM) if the produced code is shown to be equivalent to the provided query code snippet.

IV. Evaluation

A. Experiment Design

To evaluate the performance of Cpp2Rust, we randomly selected 40 LeetCode C++ problems and asked the tool to translate them. The tool was configured to work in the RAG-enabled mode and try the translation once more if the produced Rust code had any compile errors. We recorded the identified similar C++ code snippets, generated prompts, and whether the translation was successful in the first or second iteration. As a practical way of checking for equivalence, we created a test account on the LeetCode platform and submitted generated Rust code snippets as solutions. If a solution was accepted by LeetCode, it was considered to be equal to the provided C++ code.

B. Results

The experiment process was automated to use the developed tool to produce the translations for 40 LeetCode problems. For 32 of the problems, generated Rust code snippets had no compile errors, whereas eight of the generated solutions did not compile successfully. However, when the compiler error was fed into the LLM, five of these eight problems were successfully translated without any further errors.

To check the equivalence of the provided C++ snippets and their generated Rust versions, we submitted generated code snippets into the LeetCode. We found that 27 of 32 examples (that had no compile errors) were accepted as correct submissions. Furthermore, one of the five solutions that were generated after the compile error was fed into the model was accepted as correct. Overall, 84% of the solutions that did not have a compile error were accepted as a correct solution on LeetCode. Included in these correct submissions are translations that included declarations of a `struct Solution`. When present, these declarations cause duplicate definition errors when submitted to LeetCode, but yield valid solutions after their removal.

For the solutions that were accepted correctly, we also recorded the stats providing info about the comparison of these solutions with previously submitted correct Rust solutions. We found that, on average Cpp2Rust solutions that were accepted by LeetCode were better than 78.5% of all previously submitted correct Rust solutions in terms of time efficiency. Similarly, the average score of beating older Rust solutions in terms of memory usage was found to be 65.3%. Figure 4 shows individual percentages for submitted Cpp2Rust solutions where automatically generated code snippets were better than previously submitted LeetCode Rust solutions in terms of time efficiency and memory use.

Listing 1 shows the first version of the Rust code generated by Cpp2Rust while translating the C++ solution of a LeetCode problem in our test set named *difference-between-element-sum-and-digit-sum-of-an-array* [19]. After the compile error description was fed to the LLM, the correct code shown in Listing 2 was produced, which had no errors and was accepted as a valid solution in LeetCode, beating 82% and 97% of the previous submissions in terms of time and memory, respectively. Feeding back the error description that the compiler produced for the first Rust program helped the model to fix the E0368 error, by dereferencing elements during iteration and using a temporary mutable variable.

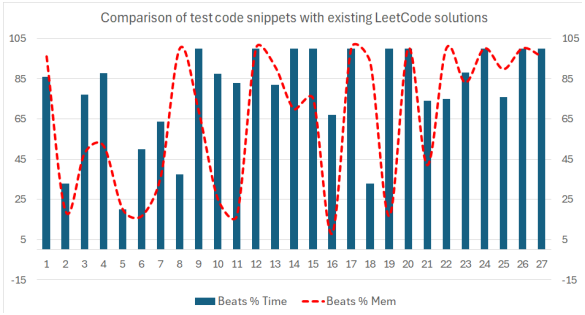

Fig. 4. Comparison of test code snippets with existing LeetCode solutions in terms of time efficiency and memory use.

```rust
impl Solution {
    pub fn difference_of_sum(nums: Vec<i32>)
        -> i32 {
        let mut a = 0;
        let mut b = 0;
        for mut x in nums.iter() {
            a += x;
            while *x != 0 {
                b += x % 10;
                x /= 10;
            }
        }
        (a - b).abs()
    }
}
```

Listing 1. The first solution produced by Cpp2Rust that had error E0368: binary assignment operation ‘/=‘ cannot be applied to type ‘&i32‘

```rust
impl Solution {
    pub fn difference_of_sum(nums: Vec<i32>)
        -> i32 {
        let mut a = 0;
        let mut b = 0;
        for &x in nums.iter() {
            a += x;
            let mut temp_x = x;
            while temp_x != 0 {
                b += temp_x % 10;
                temp_x /= 10;
            }
        }
        (a - b).abs()
    }
}
```

Listing 2. The second (correct) solution produced by Cpp2Rust after the compile error from the first wrong solution was given to the LLM

V. Conclusion & Future Work

In this paper, we developed Cpp2Rust and described progress towards RAG-LLM-based translation of C++ to Rust. The main advantage of an LLM-based approach is that it

translates C++ to idiomatic Rust rather than wrapping the C++ code in Rust syntax as procedural transpilation approaches do. We have evaluated the accuracy of our approach on a suite of Leetcode solutions, using C++ solutions as inputs and the corresponding Rust solutions as ground truth. By using Leetcode examples, we were able to check the correctness of the RAG-LLM's translations by submitting them to Leetcode for testing. However, the evaluation of our approach against existing tools that translate from C (rather than C++) is still ongoing.

In addition to evaluating our approach on C, our future work also places a greater emphasis on software semantics, as opposed to software syntax, since prior work by others has a focus on syntax. Our immediate next steps are:

Alternative methods for selecting RAG examples: When selecting examples to augment an LLM query, we plan to use knowledge graphs extracted from LLVM IR [20] as the basis of the embedding, in order to shift the focus of the augmented examples towards software semantics instead of software syntax. Using graph kernels [21], [22] to select similar examples, instead of using the cosines of neural embeddings, might also improve the relevance of the selected examples, since an approach based on graph kernels loses less information.

Evaluation on C code: In this paper, we evaluated our approach on C++ examples, although the approach should be extensible to other language pairs. In contrast to existing work in this domain, our work here shows the ability for RAG-LLMs to quickly produce idiomatic, domain-specific translations from C++ to Rust. To compare the translations provided by our approach against existing C to Rust transpilers, we have attempted to lift the C++ solutions from our evaluation set into C, using ChatGPT, and then apply tools such as C-to-Rust. However, there a number of remaining difficulties to be resolved. First, as mentioned above, Leetcode solutions are not submitted as complete programs, and without the use of a RAG component to ground the translation, ChatGPT produces compilable, but unsubmittable solutions. In the future, we intend to develop a programmatic approach that translates C into Leetcode format.

Scaling: Our approach has focused on improving the translation of small C++ snippets to Rust. To translate programs at scale using this approach will require the ability to reliably decompose and recompose translated code regions. While the results of this study indicate the ability to translate decomposed code with appropriate grounding, composing those individual translations back into a coherent Rust program may require creating and solving semantic constraints on the data flow and control flow of the translated Rust.

Improving the feedback loop: In addition to using compiler errors, we are planning to improve the feedback loop further by leveraging analysis results from other software testing and analysis tools.

REFERENCES

[1] G. Izacard and E. Grave, "Leveraging passage retrieval with generative models for open domain question answering," 2021. [Online]. Available: https://arxiv.org/abs/2007.01282

[2] V. Karpukhin, B. Oğuz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W. tau Yih, "Dense passage retrieval for open-domain question answering," 2020. [Online]. Available: https://arxiv.org/abs/2004.04906

[3] K. Guu, K. Lee, Z. Tung, P. Pasupat, and M.-W. Chang, "Realm: Retrieval-augmented language model pre-training," 2020. [Online]. Available: https://arxiv.org/abs/2002.08909

[4] M. R. Parvez, W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Retrieval augmented code generation and summarization," in *Findings of the Association for Computational Linguistics: EMNLP 2021*, M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, Eds. Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 2719–2734. [Online]. Available: https://aclanthology.org/2021.findings-emnlp.232

[5] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *NAACL*, 2021.

[6] W. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang, "Avatar: A parallel corpus for java-python program translation," in *ACL-Finding (short)*, 2023.

[7] "Leetcode solutions in any programming language," https://github.com/doocs/leetcode/tree/main, 2024, (Accessed on 08/02/2024).

[8] C. Lee, A. Mahmoud, M. Kurek, S. Campanoni, D. Brooks, S. Chong, G.-Y. Wei, and A. M. Rush, "Guess & sketch: Language model guided transpilation," 2024. [Online]. Available: https://arxiv.org/abs/2309.14396

[9] B. Mariano, Y. Chen, Y. Feng, G. Durrett, and I. Dillig, "Automated transpilation of imperative to functional code using neural-guided program synthesis," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, apr 2022. [Online]. Available: https://doi.org/10.1145/3527315

[10] Z. Yang, F. Liu, Z. Yu, J. W. Keung, J. Li, S. Liu, Y. Hong, X. Ma, Z. Jin, and G. Li, "Exploring and unleashing the power of large language models in automated code translation," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, jul 2024. [Online]. Available: https://doi.org/10.1145/3660778

[11] Immunant, "Introduction to c2rust," https://immunant.com/blog/2019/08/introduction-to-c2rust/, 8 2019, (Accessed on 08/02/2024).

[12] J. Sharp, "Corrode," https://github.com/jameysharp/corrode, 4 2017, (Accessed on 08/02/2024).

[13] K. Lesiński, "Citrus," https://gitlab.com/citrus-rs/citrus, 8 2018, (Accessed on 08/02/2024).

[14] A. Karvonen, "Using gpt-4 to assist in c to rust translation," https://galois.com/blog/2023/09/using-gpt-4-to-assist-in-c-to-rust-translation/, 9 2023, (Accessed on 08/06/2024).

[15] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating c to safer rust," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485498

[16] J. R. Cordy, "Source transformation, analysis and generation in txl," ser. PEPM '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 1–11. [Online]. Available: https://doi.org/10.1145/1111542.1111544

[17] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," 2021. [Online]. Available: https://arxiv.org/abs/2005.11401

[18] M. Fakih, R. Dharmaji, Y. Moghaddas, G. Q. Araya, O. Ogundare, and M. A. Al Faruque, "Llm4plc: Harnessing large language models for verifiable programming of plcs in industrial control systems," in *2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2024, pp. 192–203.

[19] "2535. difference between element sum and digit sum of an array," https://leetcode.com/problems/difference-between-element-sum-and-digit-sum-of-an-array/description/, 2024, (Accessed on 08/02/2024).

[20] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," San Jose, CA, USA, Mar 2004, pp. 75–88.

[21] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels." *Journal of Machine Learning Research*, vol. 12, no. 9, 2011.

[22] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, "Graph kernels," *The Journal of Machine Learning Research*, vol. 11, pp. 1201–1242, 2010.