RUSTY: Effective C to Rust Conversion via Unstructured Control Specialization

Xiangjun Han, Baojian Hua*, Yang Wang*, Ziyao Zhang, Qiliang Fan, and Zhizhong Pan
School of Software Engineering, University of Science and Technology of China, China
{pnext, zhangziyao21, sa613162, sg513127}@mail.ustc.edu.cn, {bjhua, angyan}@ustc.edu.cn*
* Corresponding authors.

***Abstract*—Rust is an emerging programming language designed for both performance and security, and thus many study efforts have been conducted to port legacy code bases in C/C++ to Rust to exploit Rust's safety benefits. Unfortunately, prior studies on C to Rust conversion still have three limitations: 1) complex structure; 2) code explosion; and 3) low performance. These limitations greatly affects the effectiveness and usefulness of such conversions. This paper presents RUSTY, the *first* system for effective C to Rust code conversion, via unstructured control specialization. The key technical insight in it is to implement C-oriented syntactic sugars on top of Rust, and thus eliminating the discrepancies between the two languages. We have conducted experiments to evaluate the effectiveness and testify the usefulness of RUSTY. We first applied RUSTY to micro-benchmarks, and experimental results demonstrated that RUSTY is effective in eliminating unstructured controls. We then applied RUSTY to 3 real-world C projects: 1) Vim; 2) cURL; and 3) the silver searcher. Experimental results showed that RUSTY successfully reduce the code size by 16% on average with acceptable overhead (less than 61 microseconds per line of C code).**

***Keywords*—Rust security; code conversion; control specialization**

I. INTRODUCTION

Rust [1] is an emerging programming language with design goals of memory safety, type safety, and thread safety. Due to its safety advantages, Rust has been successfully used to build a diverse range of low-level infrastructures. Recently, there have been a lot of academic studies as well as industry efforts on migrating legacy C/C++ code bases to Rust [2]. Such migrations bring two advantages: 1) safety; and 2) economy. First, it can significantly improve safety [3] in light of Rust's safety advantages. Second, it is often more economic to migrate legacy code to Rust than to rewrite every line of code from scratch, as most algorithm design and library implementation can be reused after the migration.

Two approaches can be utilized for the legacy code migration from C to Rust: 1) manual migration; and 2) automated conversion. First, manual migration can be employed to re-implement the existing codebase by (Rust) developers. Manual migration is promising for its convenience and flexibility. The developers can leverage programming idioms from the target language, as well as new programming frameworks or libraries without glue code while tune the generated code based on their domain knowledge and development experience. Unfortunately, its engineering efforts are considerable, especially for large C projects with huge code bases.

The second approach for legacy code migration is automated code conversion using well-designed transpilers. Recently, automated migration has been a hot research topic, and there have been a significant amount of academic studies as well as industry efforts, to investigate the theory underpinnings and practical transpiler development, respectively. With these progress, current transpilers are quite effective and successful.

Unfortunately, while prior studies on automated code migration have made considerable progress, prior studies and tools still have three limitations. First, the Rust code generated by the existing automated conversion technologies has more complex structures than the original C code. Second, prior technologies and tools for code conversion generate code of undesired sizes. Third, the generated Rust code has low performance.

To address these limitations, this paper presents RUSTY, the first system for effective C to Rust code migration via unstructured control specialization and in which the key technical insight is to implement C-oriented syntactic sugars on top of Rust. To eliminate the discrepancies between the two languages, RUSTY uses two main components: 1) a tree rewriter to introduce specialized unstructured code patterns; and 2) a library implementing the syntax extensions for Rust based on macros.

To evaluate RUSTY, we have applied it to 10 micro-benchmarks which include `goto` statement and 3 real-world C projects from different application domains. Experimental results showed that RUSTY successfully reduce the code size by 16% on average with acceptable overhead (less than 61 microseconds per line of C code). For further validation, we plan to apply RUSTY to more C projects.

II. APPROACH

This section presents our approach to design and implement RUSTY. Figure 1 presents the architecture of RUSTY, which consists of six key modules and these work in three stages respectively.

Pre-processing. In this stage, the C parser module takes as input the C source code, and parse it following the compilation process of C program, and generates C AST. The reasons we choose AST as the basis of RUSTY is that we need to eliminate the effect of `goto` on control flow. In our current implementation, RUSTY leverages Clang to parse C code.

Conversion. In this stage, the CFG generator module takes as input the C AST, and converts non-control-flow C statements into their equivalents in Rust according to the correspondence between C and Rust syntax rules and replaces the `goto` and the label statements with the macros we implemented. This module generates the special CFG in which the `goto` statement can not cause a jump anymore. The

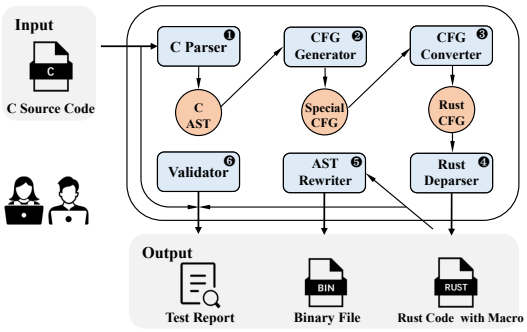

Figure 1. Overview of RUSTY Architecture

CFG converter module takes as input the special CFG, and converts control-flow structures from C into Rust to generate Rust CFG. The Rust deparser module takes as input the Rust CFG, and decompiles it to generate Rust AST. In our current implementation, RUSTY leverages C2Rust to convert, generate and decompile CFG. There are two reason to choose C2Rust: 1) high conversion rate and 2) support for large-scale transformations. The AST rewriter module takes as input the Rust code, and expands the syntactic sugar by overriding the AST. This module generates the Rust AST that conforms to the native Rust syntax for compiler. In our current implementation, RUSTY combines the loop and the break statement with label and uses procedure macro provided by Rust compiler to implement syntactic.

Validation. In this stage, the validator module is used to evaluate the effectiveness, complexity, correctness and cost of RUSTY. To evaluate the effectiveness and the complexity, the module takes as input the C program and Rust program, and compares the lines of code and the cognitive complexities of them. To evaluate the correctness, the module runs the programs, and compares the outputs to guarantee the functional effects (outputs) are identical. To evaluate the cost, the module compares the processing times of C2Rust before and after the introduction of RUSTY.

III. EVALUATION

The RUSTY is still under heavy development, and we have conducted some experiments with it. First, to evaluate the effectiveness of RUSTY, we applied it to 10 micro-benchmarks, and experimental results demonstrated that RUSTY is effective to reduce the code sizes of the generated Rust target code by 21.1% on average. Second, to evaluate the complexity of RUSTY, we applied it to 3 real-world C projects: 1) Vim; 2) cURL; and 3) the silver searcher, and experimental results demonstrated that RUSTY is effective to reduce the cognitive complexity of Rust target code by 65.3% on average. Third, to evaluate the correctness and the cost of RUSTY, we applied it to all the benchmarks, and experimental results demonstrate that RUSTY does not affect the functional correctness of Rust target code and the overhead RUSTY introduced is less than 61 microseconds per line of C. Finally, to evaluate the usefulness of RUSTY, we conducted a developer study, and the survey results demonstrated that RUSTY is helpful to end-users in converting C to Rust in a fully automated manner.

IV. RELATED WORK

Recently, the transpilers used for C to Rust automated conversion has begun to be studied.

Transpiler. There have been several transpilers for C to Rust. Bindgen [4] generates Rust FFI bindings to C libraries automatically. Corrode [5] is semantics-preserving transpiler which is intended for partial automation. Like Bindgen, but Cirtus [6] includes function bodies when doesn't try to preserve C semantics. As the successor of Corrode, C2Rust [7] supports large-scale automatic conversion while preserving semantics.

C to Rust automated conversion. All the existing studies on C to Rust automated conversion focus on safety. Emre et al. [8] first analyzed the sources of unsafety in Rust code generated by C2Rust , and proposed a technique to convert raw pointers into references in translated programs that hooks into the rustc compiler to extract type- and borrow-checker results. Hong and Bryan [9] proposed a approach to lift raw pointers to arrays that informed by a system of type constraints. Ling et al. [10] presented CRustS that eliminates non-mandatory unsafe keywords in function signatures, and refines unsafe block scopes inside safe functions by using code structure pattern matching and transformation.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work is partially supported by the National Natural Science Foundation of China (No.62072427, No.12227901), the Project of Stable Support for Youth Team in Basic Research Field, CAS (No.YSBR-005), Academic Leaders Cultivation Program, USTC.

REFERENCES

[1] "Rust programming language," https://www.rust-lang.org/.
[2] "Mitigating memory safety issues in open source software."
[3] P. Chifflier and G. Couprie, "Writing parsers like it is 2017," in *2017 IEEE Security and Privacy Workshops (SPW)*, 2017, pp. 80–92.
[4] "Bindgen," The Rust Programming Language, Aug. 2022.
[5] J. Sharp, "Corrode: Automatic semantics-preserving translation from c to rust," Aug. 2022.
[6] "Citrus / citrus · gitlab," https://gitlab.com/citrus-rs/citrus.
[7] "C2rust demonstration," https://c2rust.com/.
[8] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating c to safer rust," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 121:1–121:29, Oct. 2021.
[9] T. Y. Hong, "From c towards idiomatic & safer rust through constraints-guided refactoring," p. 85.
[10] M. Ling, Y. Yu, H. Wu, Y. Wang, J. R. Cordy, and A. E. Hassan, "In rust we trust – a transpiler from unsafe c to safer rust," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022, pp. 354–355.