In Rust We Trust – A Transpiler from Unsafe C to Safer Rust

Michael Ling[1], Yijun Yu[1,2], Haitao Wu[1], Yuan Wang[1], James R. Cordy[3], Ahmed E. Hassan[3]
[1] Huawei Technologies, Canada; [2] The Open University, UK; [3]Queen's University, Canada

ABSTRACT

Rust is a type-safe system programming language with a compiler checking memory and concurrency safety. For a smooth transition from existing C projects, a source-to-source *transpiler* can auto-transform C programs into Rust using program transformation. However, existing C-to-Rust transformation tools (e.g. the open-source C2Rust transpiler[1] project) have the drawback of preserving the unsafe semantics of C, while rewriting them in Rust's syntax. The work by Emre et el. [2] acknowledged these drawbacks, and used `rustc` compiler feedback to refactor one certain type of raw pointers to Rust references to improve overall safety and idiomaticness of C2Rust output. Focusing on improving API-safeness (i.e. lowering `unsafe` keyword usage in function signatures), we apply source-to-source transformation technique to auto-refactor C2Rust output using code structure pattern matching and transformation, which does not rely on `rustc` compiler feedback. And by relaxing the semantics-preserving constraints of transformations, we present `CRustS`[2] a fully-automated source-to-source transformation approach that increases the ratio of the transformed code passing the safety checks of the `rustc` compiler. Our method uses 220 new `TXL` [1] source-to-source transformation rules, of which 198 are strictly semantics-preserving and 22 are semantics-approximating, thus reducing the scope of unsafe expressions and exposing more opportunities for safe Rust refactoring. Our method has been evaluated on both open-source and commercial C projects, and demonstrates significantly higher safe code ratios after the transformations, with function-level safe code ratios comparable to the average level of idiomatic Rust projects.

KEYWORDS

transpiler, safety, measurement, refactoring, code transformation

ACM Reference Format:
Michael Ling[1], Yijun Yu[1,2], Haitao Wu[1], Yuan Wang[1], James R. Cordy[3], Ahmed E. Hassan[3]. 2022. In Rust We Trust – A Transpiler from Unsafe C to Safer Rust. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3510454.3528640

[1]https://github.com/immunant/c2rust
[2]`CRustS` stands for C to Rust Safer transformation system

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9223-5/22/05.
https://doi.org/10.1145/3510454.3528640

1 INTRODUCTION

To benefit from the built-in memory and concurrency safety assurance [4] of `rustc` compiler, transpiled Rust program must avoid unnecessary usage of `unsafe` keywords, which reduces safety checks of the decorated function or code block. Popular manually-written idiomatic Rust projects typically have about 20 to 30% code marked with `unsafe` [5][3]. They are mostly used as a trade-off for higher low-level efficiency or hardware-specific operations [3]. However, current C-to-Rust transpilation tools, such as `c2rust`, in order to produce semantically equivalent Rust code, they produce non-idiomatic Rust code that utilize `unsafe` in almost all function signatures. This significantly limits the safety check that `rustc` compiler can perform on transpiled Rust code. Furthermore, semantically equivalent Rust code often fails to compile since it violates Rust design, e.g. initialization of static variable at runtime is legal in C but not in Rust. Although `c2rust` does provide command line refactoring tools [6] that take manually composed commands to improve the resulting Rust code, it requires much manual work and rather comprehensive knowledge of the source code.

In this work, we build our transpilation system on top of `c2rust`, and focus on producing compilable Rust program with higher ratio of compiler-checked code. Our approach aims to systematically eliminate non-mandatory `unsafe` keywords in function signagures, and refine `unsafe` block scopes inside safe functions, in a fully automated manner.

2 OUR APPROACH

According to The Rust Reference, `unsafe` can be used under four different contexts: Function qualifiers, `unsafe` blocks, `Traits`, and `Trait` Implementations.

In both pure Rust projects and C-to-Rust transpiled projects, the majority of `unsafe` Rust code involve the first two contexts, which are the focus in this work. We address the use of `unsafe` in function qualifiers first, because:

- It is the dominant use of `unsafe` in `c2rust` transpiled code;
- It makes all statements in the function body to be considered `unsafe`, hence a greater impact on the overall safe ratio;
- It has a domino effect – each use of `unsafe` function must either be wrapped in an `unsafe` block, or be declared as `unsafe` (e.g. when it is used as a function parameter);
- The majority of them can be removed by wrapping all `unsafe` statements in the function body in `unsafe` blocks.

In the case of `unsafe blocks`, the strategy is simple: break them into `unsafe` blocks, one per each statement, then remove each `unsafe` unless it's mandatory. Remaining adjacent `unsafe blocks` are merged and some statements in between might be included, considering name scopes.

[3]'Unsafe' code here refer to those marked `unsafe` and not fully checked by `rustc` compiler, they are not necessarily unsafe.

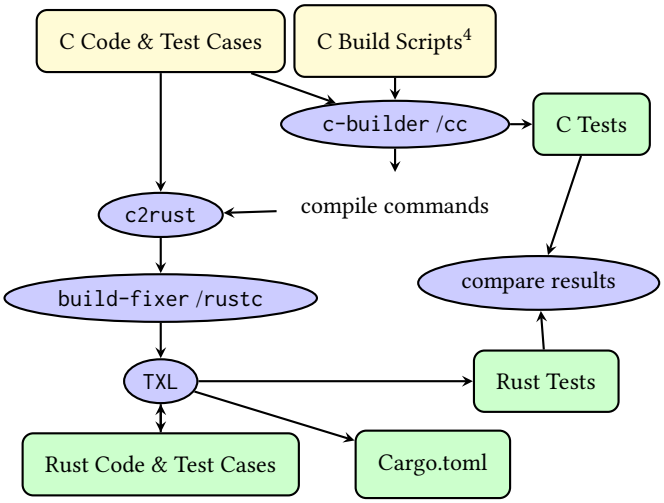

Figure 1: An overview of the process of CRustS

Our main processing steps in migrating unsafe C to safe Rust are shown in Figure 1. The process extends the `c2rust` tool by:

- `c-builder` that automates the building of source C project (Make, CMake, or Bazel), and the generation of intermediate artifacts needed by `c2rust` (e.g. `compile_commands.json`);
- Invocation of `c2rust` to transpile to a Rust project;
- `build-fixer` to fix build errors based on `rustc` error hints;
- `TXL` rules to auto-refactor Rust code to reduce `unsafe` usage and scope.

A `TXL` program is typically organized as many separate transformation rules. Each of them performs one specific code transformation, e.g. depending on hardware spec, transforming the C library type *libc* :: *c_short* to the Rust primitive type *i*16. The main steps in the application of `TXL` transformation rules are:

- Parse the Rust source file into a parse tree;
- Apply user-defined transformation rules to rewrite the intended parts of the parse tree, e.g. each node of grammatical type `TypeNoBounds`[5] with value *libc* :: *c_short* to be replaced by a node of the same type but of value *i*16[6];
- Unparse and pretty print the parse tree as Rust source code back to a file.

3 EVALUATION

RQ1. How much can the safe function ratio be enhanced using our source-to-source transformation approaches?

We evaluated `CRustS` by a comparison of open-source and commercially developed C projects, before and after the transpilation.

For each project in the dataset, we considered the size of the project in terms of lines of code (LOC), and its safe/unsafe function ratio. We compared these statistics with several projects transformed by `c2rust` and `CRustS` to measure the improvement in the safe function ratio and processing efficiency. The results are listed in table 1.

[5]Rust grammar definitions as specified in The Rust Reference at commit e1abb17.
[6]Such C to Rust primitive type mapping is configurable in `CRustS`

Table 1: Rust safe function ratios after applying `c2rust` and `CRustS` to unsafe C projects

Project name	LOC	# func	`c2rust`	`CRustS`
TLVcodec[7]	136092	73	0.014	**0.959**
tinycc v0.9.27	72394	527	0.002	**0.991**
RosettaCode*	7328	381	0.223	**0.995**
Checked_C	724402	3535	0.002	**0.997**
ptrdist-1.1 [8]	9857	236	0.021	**0.983**
BusyBox	303275	4589	0.003	**0.647**

RosettaCode*: summed and averaged over 85 tasks in RosettaCode

As shown in the results, the safe function ratios have been enhanced substantially from the unsafe C projects (which are at 0% by definition). Note that most of the projects reach a safe function ratio higher than 95%. Though the safe function ratio of BusyBox is far less than 95%, the improvement over the previously transformed RustyBox project is significant. Further analysis reveals that the relatively low safe function ratio in this case is caused by its OS-related structures, which is natural in OS-type projects. A solution to this case is one of our future research directions.

RQ2. How are the safe-enhancing Rust project transformations accepted by product teams?

Product teams from a large telecom company have tried the `CRustS` approach. One of such teams is from the product TLVcodec, whose feedback is positive. The most prominent comments are that "it overcomes the steep learning curve of a new programming language", and "[it] reduces the workload to transform legacy C code to Rust". Another product team tried `CRustS` in a low-level hardware control product to create an initial Rust version from 2K lines of core C code module, in order to help the team to jump start the transition of this module from C to Rust. In this effort, the initial deployment and minor adaptation of the tool in the production environment took about two days, then the transformed Rust project was produced after another day. The lead time of this approach is significantly shorter than the manual rewrite effort, which in this specific case would have taken more than two weeks according to experience project managers. Our tool is highly appraised by the team as it "produced an initial Rust implementation very quickly, and provided a solid stepping stone for the coming work". In addition, they also pointed out several drawbacks of the tool mainly concerning C macros and `enum` handling, which is on our radar for further improvement.

REFERENCES

[1] James R. Cordy. 2006. The TXL source transformation language. *Science of Computer Programming* 61, 3 (2006), 190–210.
[2] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating C to Safer Rust. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 121 (oct 2021).
[3] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe Systems Programming in Rust. *Commun. ACM* 64, 4 (March 2021), 144–152.
[4] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust language. *ACM SIGAda Ada Letters* 34, 3 (Oct. 2014), 103–104.
[5] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN Conf. on PLDI*. London, UK.
[6] Garming Sam, Nick Cameron, and Alex Potanin. 2017. Automated refactoring of Rust programs. In *Proceedings of the Australasian Computer Science Week Multiconference (ACSW '17)*. ACM, Geelong, Australia, 1–9.