Verification of Unsafe Code in Rust Programs by Abstract Interpretation

Master 2 research internship proposal, 2024–2025

Supervisor:	Antoine Miné (antoine.mine@lip6.fr)
Internship location:	APR team, LIP6 Sorbonne Université Jussieu Campus, Paris, France
Duration:	4 to 6 months
Related projects:	PEPR Secureval, MOPSA analyzer
Relevant courses:	MPRI 2.6: Abstract Interpretation: Application to verification and static analysis Master 2 STL: Typage et analyse statique (Sorbonne Université)

The goal of this internship is to develop static analysis techniques to verify Rust code [5] that employs unsafe constructs. On the theoretical level, the methods will be developed and proved formally correct within the framework of abstract interpretation [1]. On the practical level, the techniques will be implemented within the MOPSA [3] open-source static analysis platform written in OCaml, and experimented on representative fragments of unsafe Rust code.

Context and Motivation

Rust [5] is a fairly recent programming language which emphasizes on performance and safety. In particular, Rust features a unique ownership type system that is checked at compile time by the *borrow checker* to ensure memory safety without resorting to a garbage collector. Rust enjoyed a rapid adoption and is used for instance by Mozilla and now in the Linux kernel.

Nevertheless, Rust features *unsafe* constructs, that allow the programmer to introduce code that is not checked, such as dereferencing raw pointers, accessing and modifying mutable variables. Unsafe code is used when the borrow checker is too conservative, in order to allow code that is too low-level, that interfaces with C, or for performance reasons. It is used notably in the standard Rust library. The presence of unsafe code is cause for concern: executing unsafe code can not only produce an error, it could also invalidate the safety guarantees offered by the language in the following "safe" code. Informal guidelines [6] have been proposed to ensure the safety of unsafe code. However, there is a strong incentive to use verification methods, beyond the borrow checker, to verify unsafe code.

Several efforts have tailored formal methods to Rust, some handling unsafe code as well. The RustBelt project provided a formal semantics and safety proof for a realistic subset of Rust using Coq [7]; operational semantics for Rust have been presented in [8] and [9] using the K framework; while [11] proposes a type system to verify Rust programs. The Master Thesis [10] presents a good coverage of the efforts to formalize Rust. Practical verification tools, including the symbolic execution tool Klee [12] and the bounded model-checker CBMC [13], have been applied to Rust verification. Finally, the MIRAI tool [14] combines static analysis by abstract interpretation with dynamic analysis. The goal of the internship is to study the application of fully static analysis by abstract interpretation to the verification of Rust programs, with an emphasis on verifying errors caused by careless uses of unsafe constructs.

The APR team at Sorbonne University has been developing a multi-language static analysis platform, MOPSA [3, 2], that can be extended to support new abstractions in a modular way, as well as new languages. It has been notably applied to the analysis of run-time errors in C programs [4] and typing and uncaught exceptions in Python programs, through a fully flow-sensitive, context-sensitive, and relational value analysis. It is also able to analyze programs mixing several languages. Due to its extensibility capabilities, the presence of readily-available abstractions, and the ability to analyze low-level code as demonstrated by its C analysis, it is a good fit to develop an analysis for Rust.

Expected Work

The expected work includes both theoretical aspects (semantic development, abstract domain design and proof of correctness) and practical aspects (implementation, experimentation and evaluation on representative program fragments). A first step will be the design of a concrete collecting reachability operational semantics for (a relevant fragment of) Rust. It should be able to express safety requirements on unsafe code, e.g. based on [6]. This step can rely and previous efforts [7, 8, 9, 10], but must express the semantics in a way that is amenable to abstract interpretation. A second step is the design of suitable abstract domains to infer automatically these requirements. Existing domains may be leveraged, either serving at the basis of new domains, or being reused directly in a reduced product with new domains. This includes notably numeric domains as well as memory domains designed for (low-level) C analysis [3]. In a third step, the abstractions will be implemented within the MOPSA platform [2] and evaluated through the analysis of small but representative Rust codes that use unsafe constructs. This can include, for instance, code from the Rust standard library.

The internship will take place in the APR team, in the LIP6 laboratory, on the Jussieu Campus at Sorbonne Université, Paris. If successful, the internship could lead to a fully-funded PhD on a follow-up subject.

Required Skills

- Strong knowledge of abstract interpretation (having followed a Master-level course, e.g., at MPRI or at the STL Master at Sorbonne Université).
- Knowledge of Rust.
- Experience with OCaml (the language MOPSA is implemented in).
- Willingness to both formalize the semantics on paper and to implement it in a computer, and to conduct experiments.

References

[1] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proc. of POPL'77, pages 238-252, 1977.

[2] MOPSA - A Modular Open Platform for Static Analysis. https://mopsa.lip6.fr

[3] M. Journault, A. Miné, M. Monat, and A. Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyzer. In VSTTE 2019, pages 1–17, Jul. 2019.

[4] A. Ouadjaout and A. Miné. A library modeling language for the static analysis of C programs. In SAS 2020, LNCS 12389, pages 223–246. Springer, Nov. 2020.

[5] Rust Programming Language. https://www.rust-lang.org

[6] Rust's Unsafe Code Guidelines.
https://github.com/rust-lang/unsafe-code-guidelines

[7] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the Foundations of the Rust Programming Language. In POPL 2018.

[8] Shuanglong Kan, Zhe Chen, David Sanan, Shang-Wei Lin, and Yang Liu. An Executable Operational Semantics for Rust with the Formalization of Ownership and Borrowing.

[9] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, Jun Zhang. KRust: A Formal Executable Semantics of Rust In TASE 2018.

[10] Alexa White. Towards a Complete Formal Semantics of Rust. Master's Thesis, California Polytechnic State University. March 2021.

[11] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer RefinedRust: A Type System for High-Assurance Verification of Rust Programs. In PLDI 2024.

[12] KLEE Symbolic Execution Engine. https://klee-se.org/

[13] CBMC. https://www.cprover.org/cbmc/

[14] MIRAI. https://github.com/facebookexperimental/MIRAI