Internship proposal:
Formal verification of Rust programs

Advisor:
Son Ho
Team Prosecco – Inria
son.ho@inria.fr

Head of department:
Karthikeyan Bhargavan
Team Prosecco – Inria
karthikeyan.bhargavan@inria.fr

Location of the internship: Paris, France

Context One of the main challenges of program verification today consists in finding a way of efficiently reasoning about low-level, effectful programs. This problem has led to the development of a large collection of frameworks and logics [24, 11, 17, 25, 4, 8, 6] which make various compromises between ease of verification and expressiveness. Despite the intense research effort in this area, verifying such programs still represents a daunting task [29, 16, 13, 7].

Following the development of the Rust programming language [2], the recent years have seen the emergence of a new line of research [5, 21, 9, 26, 3, 15] which attempts to leverage the Rust type system to improve scalability when reasoning about low-level programs. Rust is indeed a perfect target for program verification, as its static ownership discipline strikes a good balance between a fine-grained, expressive control of memory, making it a language of choice for system programming, and strong memory guarantees, which can be leveraged to ease program verification. A key insight which drove the design of frameworks such as [9, 15] is that a large subset of Rust doesn't require to explicitly reason about memory [19], which typically involves tedious, low-level proofs. Instead, proof engineers can focus on the *functional behavior* of their programs.

Goals The topic of this internship is to work on the verification of Rust programs. In particular, we are currently developing two tools for this purpose: HACSPEC [21, 20] and AENEAS [15, 1].

HACSPEC is foremost a purely functional subset of Rust, which supports operations such as copies, moves and a certain form of loops, but forbids in-place updates and in particular mutable borrows. It allows regular engineers to write and test specifications directly in a mainstream programming language (i.e., Rust), rather than in the exotic language of an expert theorem prover. Once written, a HACSPEC specification can be compiled to equivalent specifications in backends such as F* or Coq, for proof engineers to use them in their proofs. In particular, they can serve as high-level specifications for low-level implementations written in frameworks such as Low* [24] or Steel [11]. As it generates *specifications*, the HACSPEC compiler attempts to generate code which looks as close as possible to the original user-written program. This methodology allows to bridge the gap between unverified application code and verified components.

AENEAS is a new verification pipeline for safe Rust programs. AENEAS leverages the Rust type system to compile Rust programs to pure, executable models (i.e., pure, functional versions of the original Rust programs). The key idea behind AENEAS' compilation is that, under the proper restrictions, a Rust function is fully characterized by a *forward* function, which computes its return value at call site, and a set of *backward* functions (one per lifetime), which propagate changes back into the environment upon ending lifetimes, thus accounting for side effects. Such forward and backward functions behave similarly to lenses [10]. Relying on theorem provers to state and prove lemmas about those models, it is then possible to enforce guarantees about the original programs. For instance, one can prove panic freedom and functional correctness [15], but also security guarantees like authentication and confidentiality in the case of cryptographic protocols as has been done in other frameworks [16], and potentially more.

As such, HACSPEC and AENEAS are complementary. HACSPEC attempts to preserve an almost one-to-one correspondance with the original Rust program, for the purpose of writing *specifications*. On its side, AENEAS plugs itself farther in the compilation process and attempts to support a larger subset of Rust. It tries to generate code which is readable and understandable for the purpose of *verifying* it.

Several topics are possible for this internship, including but not limited to the following. We are of course open to suggestions if the prospective intern has topics in mind, in particular with regards to potential applications.

We are interested in applying HACSPEC and AENEAS to specify and verify various large use cases, which include but are not limited to cryptographic applications [29, 16] or storage systems [13]. Verifying such applications would probably require to expand the expressiveness of those tools.

As of today, HACSPEC and AENEAS are independant projects, and establishing a link between them would be useful. One could then use HACSPEC to annotate a program with pure specifications in the form of preconditions and postconditions, before compiling this program with a combination of HACSPEC and AENEAS to generate both a pure, executable model, and a set of lemma statements left for the user to prove. This would allow us to bridge the gap between the pure model generated by AENEAS and the original Rust program. One difficulty when using a model is indeed that it introduces an indirection in the reasoning, meaning we have to pay even more attention to the way we write the specifications about the *model*, to make sure they properly reflect the properties we want to prove about the *original* program.

AENEAS currently supports a non-trivial subset of Rust which includes recursive datatypes and functions, and loops, but it still lacks several core features of Rust. One such restriction is the lack of support for function pointers, closures and traits. As in our translation a function is fully characterized by its forward and backward functions, we believe it is naturally amenable to supporting function pointers and traits: a function pointer would be characterized by a tuple of functions in the pure world, and a trait would be implemented as a type class. Closures are a special case of traits and shouldn't need any special treatment.

AENEAS currently has limited support for interior mutability, by which one can introduce aliasing in safe Rust programs, and which programmers notably leverage to implement synchronization primitives such as mutexes for concurrent programs. The key challenge would be to design a lightweight, flexible effects system, which would allow to reason about effectful behavior while leveraging the pure translation whenever possible. We are leaning towards using interaction trees [27, 28] to implement such an effects system.

We are also interested in proving soundness properties of our translation mechanism, with the end goal of having fully mechanized proofs. Such proofs would be carried in several steps. AENEAS' translation mechanism relies at its core on the semantics of a language called LLBC (Low-Level Borrow Calculus), which captures a large subset of Rust. In order to account for Rust's borrow mechanism, LLBC's memory model is *structured*, in the sense that it doesn't rely on a notion of map from memory addresses to values. A first step would be to prove that such semantics preserves an alternative version of the semantics built on top of an *unstructured* memory model, in the spirit of previous work modeling stateful programming languages [18]. Then, we would prove that the translation generates a pure program which is refined by the original Rust program, whose semantics is given by LLBC. The soundness proof for the semantics of LLBC could be formalized once and for all, first with pen and paper, then in theorem prover. On the other hand, a more realistic approach for the soundness of the translation might be to implement a proof-producing mechanism which, given a Rust program, generates a proof of refinement alongside the pure translation, as has been done in other projects [22, 14, 23, 12].

Qualifications This internship of 6 months or less would be hosted by the team Prosecco at Inria Paris. The preferred qualifications for the student at the beginning of the internship would be:

- familiarity with the Rust programming language;
- fluency with a functional programming language like OCaml;
- some experience with at least one theorem prover (Coq, F*, LEAN, HOL4, Isabelle/HOL, ...);

References

[1] Aeneas. `https://github.com/AeneasVerif/aeneas`.

[2] Rust. `https://www.rust-lang.org/`.

[3] Verus. `https://github.com/verus-lang/verus`.

[4] Andrew W Appel. Verified Software Toolchain. page 17.

[5] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. Leveraging rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[6] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K Rustan M Leino, Jacob R Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying High-Performance Cryptographic Assembly Code. page 18.

[7] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. Provably-Safe Multilingual Software Sandboxing using WebAssembly. page 19.

[8] Adam Chlipala. The Bedrock Structured Programming System. page 12.

[9] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. The CREUSOT Environment for the Deductive Verification of Rust Programs. page 32.

[10] J Nathan Foster, Bell Labs, Jonathan T Moore, and Alan Schmitt. Combinators for Bi-Directional Tree Transformations: A Linguistic Approach to the View Update Problem. *ACM Transactions on Programming Languages and Systems*, page 64.

[11] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. Steel: proof-oriented programming in a dependently typed concurrent separation logic. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–30, August 2021.

[12] David Greenaway, June Andronick, and Gerwin Klein. Bridging the Gap: Automatic Verified Abstraction of C. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Lennart Beringer, and Amy Felty, editors, *Interactive Theorem Proving*, volume 7406, pages 99–115. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. Series Title: Lecture Notes in Computer Science.

[13] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage Systems are Distributed Systems (So Verify Them That Way!). page 17.

[14] Son Ho, Oskar Abrahamsson, Ramana Kumar, Magnus O. Myreen, Yong Kiam Tan, and Michael Norrish. Proof-producing synthesis of CakeML with I/O and local state from monadic HOL functions. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference (IJCAR)*, volume 10900 of *Lecture Notes in Computer Science*, pages 646–662. Springer, 2018.

[15] Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages*, 6(ICFP):711–741, August 2022.

[16] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations (Long Version). page 29.

[17] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *European Symposium on Programming*, pages 696–723. Springer, 2017.

[18] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[19] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. Rusthorn: Chc-based verification for rust programs (full version).

[20] Denis Merigoux, Franziskus Kiefer, and Karthikeyan Bhargavan. Hacspec. `https://github.com/hacspec/hacspec`.

[21] Denis Merigoux, Franziskus Kiefer, and Karthikeyan Bhargavan. *Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust.* PhD thesis, Inria, 2021.

[22] Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ML from higher-order logic. In *International Conference on Functional Programming (ICFP)*, pages 115–126. ACM Press, September 2012.

[23] Liam O'Connor, Christine Rizkallah, Zilin Chen, Sidney Amani, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Alex Hixon, Gabriele Keller, Toby C. Murray, and Gerwin Klein. Cogent: Certified compilation for a functional systems language. *ArXiv*, abs/1601.05520, 2016.

[24] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, et al. Verified low-level programming embedded in f. *Proc. ACM program. lang.*, 1(ICFP):17–1, 2017.

[25] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. Refinedc: automating the foundational verification of c code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 158–174, 2021.

[26] Sebastian Ullrich. Electrolysis. `https://github.com/Kha/electrolysis`.

[27] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, January 2020. arXiv: 1906.00046.

[28] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for LLVM IR. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–30, August 2021.

[29] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, Dallas Texas USA, October 2017. ACM.