Type invariants and ghost code for deductive verification of Rust code

Jacques-Henri Jourdan (jacques-henri.jourdan@cnrs.fr)

M2 internship proposal (2022-2023, second semester)

1 Context

The Creusot [1, 2] tool, developed in the LMF laboratory since 2020 aims at formally verifying Rust programs by letting the users manually write specifications in their programs, and letting a mostly automated tool chain try to check that they are correct. It does so by translating annotated Rust code into a variant of the WhyML language accepted by the Why3 tool, and then use Why3 to generate verification conditions which are discharged by various automated provers.

Type invariants and ghost code are tools provided by deductive verification tools such as Why3, which improve both the expressiveness and usability. Type invariants lets adding properties holding for all values of a types behind its abstraction barrier, and ghost code allows the user to write code and manipulate values that will not have any impact on the execution of the program, and thus are removed at compile time. Creusot lacks good support for them. This internship subject aims at fixing this issue.

2 Language integration and experiments

Creusot leverages the non-aliasing guarantees provided by Rust's type system in order to generate simpler WhyML code, which makes easier to discharge the verification conditions by provers: even though the code written in Rust usually contains side effects, the generated WhyML code is, in some sense, free of side effects. This ingenious translation comes at a cost: in order to translate *mutable borrows* (a certain kind of pointers in Rust), it is necessary to use the concept of *prophecy*, which makes possible to speculate in the proof over the future of the execution of the program [5].

It is unclear how invariants and ghost code should interact with the prophetic encoding of mutable borrows. An easy possibility would be to completely forbid ghost code to depend on prophecies, and make type invariants predicates that must be true for any value of a type. However, more expressive alternative might exist:

1. A variant of invariants, *historical invariants*, makes it possible to express properties of the *evolution* of values. Historical invariant could be used, for example, to relate the current and final (prophetic) value pointed to

by a mutable borrow. We have several examples where historical invariant could drastically simplify some specifications.

2. Ghost variables cannot depend in an unrestricted manner on prophecies without causing unsoundness. However, common uses of ghost variables in e.g., Why3, include *model fields* which attach a ghost field containing its logical model to a value in Creusot. This kind of use of ghost variable does require them to depend on prophecies. It is therefore interesting to tell what restrictions should we impose to preserve soundness.

There are other interesting questions to answer about the integration of these features in Creusot: how should these features be specified in Creusot code? When should invariant be enforced, exactly?

During the internship, we would like to explore the various possibilities, and do experiments by verifying simple examples in order to get a better idea of the various trade-offs. This will involve thinking about language design issues, writing Rust code in the implementation of Creusot itself, and proving simple Rust functions with this patched version of Creusot.

3 Metatheoretical study

While the prophetic encoding to a purely functional program simplifies writing specifications and reason about them, its soundness in far from trivial: we must ensure that these prophecies do not create *causality loops*, which would allow proving false. This proof of soundness is the goal of an existing formalization of this translation [4]. This formalization uses a logical relation for the type system of Rust, built using the Iris separation logic [3]. Depending on the kind of invariant and ghost code that we choose to add to Creusot, their soundness can be more or less easy to establish.

Therefore, an interesting task during the internship will be, if judged useful and if the intern is interested, to extend the existing formalization with ghost code and invariants. This will involve a theoretical study of these notions, and their formalization within the Iris separation logic in the Coq proof assistant.

4 Ghost ownership

While ghost variable are usually thought of as containing pure mathematical values that do not carry any ownership, we may ask ourselves whether it is useful in some cases to make non-duplicable the value they contain. If we do so, ghost variables will not only contain mathematical values, but they will also carry *ghost resources*. Such *ghost resources* have found useful for software verification: notably, they are the primitive used in Iris to specify a large variety of functions and libraries.

During the internship, we will explore this idea of making non-duplicable some ghost variables to specify the behavior of some libraries. In particular, ghost ownership could be useful to specify Rust libraries featuring *interior mutability*, where the non-aliasing guarantees of Rust's type system do not apply [6]. Because these libraries are particularly useful in a concurrent setting, ghost ownership may have interesting applications for the verification of multi-threaded

programs. Of course, the use of these techniques pose interesting metatheoretical questions that we may study as well.

5 Practical aspects

The internship will last a semester. The intern is expected to follow the second year of a research master program (or equivalent). The intern will be advised by Jacques-Henri Jourdan, CNRS researcher at the LMF laboratory, at Université Paris-Saclay. Xavier Denis, the main developer of Creusot, will participate to the discussions and answer questions whenever necessary. The internship may be followed by a PhD thesis if both the intern and the advisor are interested, and if founding permits.

References

[1] The creusot tool for rust deductive verification. `https://github.com/xldenis/creusot/`.

[2] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A foundry for the deductive verification of rust programs. In *International Conference on Formal Engineering Methods*, 2022.

[3] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.

[4] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. Rusthornbelt: a semantic foundation for functional verification of rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 841–856, 2022.

[5] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. Rusthorn: Chc-based verification for rust programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 43(4):1–54, 2021.

[6] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. Ghostcell: separating permissions from data in rust. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–30, 2021.