Verifying Real-World Rust Programs

Bachelor Thesis Project Description

Jonas Maier
Supervised by Prof. Dr. Peter Müller, Vytautas Astrauskas
Department of Computer Science
ETH Zürich
Zürich, Switzerland

I. Introduction

Rust is a systems programming language that puts emphasis on safety, speed, and concurrency. Its strong type system prevents programming errors of other systems languages, like buffer overruns or use-after-free errors that are common in C. This greatly simplifies reasoning about program behavior, and makes program verification easier, as basic properties are already given, and don't need to be verified first, e.g. ensuring references don't alias.

Sometimes the properties guaranteed by the Rust compiler aren't satisfactory. This is where Prusti [1], a program verifier, comes in – it can be used to verify the absence of exceptions (or panics, in Rust jargon), correctness of the implementation in regards to specifications, and termination. Since Rust's type system already gives a lot of guarantees less effort needs to be spent on proving basic properties, such as references not interfering with each other. Instead, more time can be spent on *functional* verification of programs, i.e. proving that an algorithm or a datastructure adheres to its specification. Prusti utilizes the Viper [2] verification infrastructure, which is the common backend of multiple program verifiers.

This thesis aims to explore real-world practical applications of the Rust verifier Prusti by verifying existing code, and assessing to what degree this is possible, and what aspects of Prusti need to be adjusted or improved upon. A comparison to a different program verifier by contrasting the different implementations and verifications is a central goal.

II. Approach

To assess how practical Prusti is for verifying real-world code, some real-world code needs to be verified. It should be a code snippet that is largely self-contained, so that only that piece of code needs to be verified, and not a whole code base. A suitable type of code for this task would be a basic *datastructure* that does not use other datastructures. In the ideal case, the piece of code to be verified is already verified with a different verification tool, such that the practicality of Prusti can be assessed, and compared to the other tool.

One concrete codebase that fulfills the two properties which are mentioned above, is the implementation of Verified BetrFS [3]. Verified BetrFS, or VeriBetrFS, is a verified file system with the design of BetrFS [4]. Its source contains mathematical proofs of the correctness of the implementation in the programming language Dafny [5] with the extension for linear types [6]. In particular, it contains some data structure implementations that are largely self-contained and could be interesting to verify in a new environment.

For illustrative purposes, the Linear Mutable Map [7] will be taken as the datastructure to be implemented, but the choice of the code snippet may be different in the thesis. The Linear Mutable Map is already implemented and verified in Dafny, so it first needs to be translated to Rust, to verify it in this language. In particular, it needs to be translated to the subset of Rust that Prusti is able to verify.

As for the verification, the question stands what exactly to verify. The plan is to start with proving simple properties and then move on to prove more complex ones. One first class of properties to verify would be that the program execution can't go wrong in unexpected ways. Rust does not feature *undefined behavior* as prominently as other programming languages, but there are still a few things that could happen. Especially due to the absence of undefined behavior, many things that could induce that instead make the program *panic* and abort. Since a program that aborts in the middle of its execution is not very useful, the first property to verify is the absence of panics.

One example of a programming error that causes panics is an out of bounds access on an array. To verify absence of panics in such a program, it is therefore essential to guarantee that indexing into an array stays within the arrays bounds. *Integer overflows* can unexpectedly cause invalid index accesses. Thus, verifying the absence of

integer overflows is a second property to verify.

A map that is guaranteed not to fail when functions are called can still have major bugs. Therefore, a further property to verify is that the map's implementation is indeed a map. Since Prusti doesn't yet have a builtin definition of a map, one needs to be added to the Prusti codebase. This is best done by specifying a mathematical model of a map, what actions or functions there are on it, and how they change the map. Further, the model of the map helps in verifying any code that makes use of such a datastructure by making it possible to reason about the stored values.

Using this mathematical model of a map, it is possible to verify that the implementation adheres to this model, and doesn't just return random values. If the verification isn't possible, it must be that the model of the map was wrong, or, more likely, that the implementation contains a bug. To finish verification of the map this needs to be fixed.

Coming back to the main goal of this thesis: The practicality and ease of use of Prusti on real-world code needs to be assessed. Since the map is already implemented and verified in Linear Dafny, a direct comparison can be performed. Points to compare include, but are not limited to:

- Lines of Code
- Lines of Specifications or Annotations
- Missing features of Rust, or the subset thereof that Prusti can verify
- Features of Rust not present in Dafny that simplify the work

III. Core Goals

This summarizes the core milestones of the thesis and notes how long each part is estimated to take, in weeks.

- Implementing the Datastructure (1)
- Verifying Absence of Panics and Integer Overflows (1)
- Implementing the mathematical model (2)
- Verifying Correctness (2)
- Comparison to Dafny and Analysis (2)

IV. Extension Goals

A. Liveness

Verify the liveness of the datastructure. Every function on the datastructure should return in finite time. In other words, one needs to prove *termination* of all operations.

B. Verifying another Datastructure

Verify a different datastructure the same way as laid out in the core goals. The same steps may be applied: implementing the datastructure in Rust, implementing the mathematical model, then verifying correctness and absence of crashes.

C. Addressing Issues of Prusti

Address shortcomings of Prusti which may or may not be discovered in the analysis core goal. What exactly these shortcomings are is not known in advance, and therefore quite open ended.

D. Verifying Algorithms using the Map

Verify one or multiple algorithms making use of the map. This might be a simple function modifying each stored value, adding or removing values to the map depending on the existing stored values, or something else. Some examples are the classic $filter$ or *map* functions. Another example would be finding the key to a specific value.

E. Verifying different Algorithms

Verify algorithms unrelated to the map to aid in assessing to what degree Prusti can be used on real world code. A sorting algorithm would be one example.

V. Schedule

- Core Goals: 8 weeks
- Extension Goals: 8 weeks
- Writing Report: 4 weeks

References

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification," in *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, vol. 3, no. OOPSLA. ACM, 2019, pp. 147:1–147:30.

[2] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62.

[3] T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, and B. Parno, "Storage systems are distributed systems (so verify them that Way!)," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 99–115. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/hance

[4] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter, "BetrFS: A Right-Optimized Write-Optimized file system," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 301–315. [Online]. Available: https://www.usenix.org/conference/fast15/technical-sessions/presentation/jannen

[5] K. R. M. Leino, "Accessible software verification with dafny," *IEEE Software*, vol. 34, no. 6, pp. 94–97, 2017.

[6] "Linear types for large-scale systems verification," under review.

[7] A. Lattuada, T. Hance, R. Johnson, J. Li, C. Hawblitzel, G. Zellweger, and J. Howell, "Linear mutable map," https://github.com/vmware-labs/verified-betrfs/blob/master/lib/DataStructures/LinearMutableMap.i.dfy, [Online; accessed 25-February-2022].