Place Capabilities Summaries for Rust Programs

Dylan Wolff
Supervisors: Alexander Summers, Federico Poli

November 10, 2019

1 Background and Introduction

The Rust programming language is primarily characterized by its unique approach to memory management. By enforcing ownership and borrowing restrictions for memory locations, the type-system of Rust precludes whole classes of memory saftey issues common in languages with manually managed memory.

The ascension of Rust has created new opportunities to build analysis and verification tools that take advantage of the information and guarantees provided by its type system, such as the Prusti project [2].

However, while the Rust compiler provides descriptive negative information for programs that do not compile, there is currently no way to extract positive information for a correct Rust program (i.e. why it type-checks in the first place). This kind of information is crucial for verification tools like Prusti [2] and may have other applications for visualization and analysis of Rust code.

We intend to present this information as a "Place Capability Summary" (PCS), similar to what was proposed in the Prusti paper [2]. Generally, this is a summary of the capabilities a Rust program has to memory locations ("places") at various points throughout its execution. This PCS, due to the nature of Rust's system of borrowing, necessarily also includes information about which variables are borrowing which places.

For example, initializing and assigning to a variable would result in adding the corresponding place to the PCS, as that variable now has permissions to access the memory location it refers to. While initially, it may exclusively

refer to that place (and thus be mutable[1]), if that place is later borrowed by another variable, its permissions may change based on the characteristics of the borrow.

So far, there have been several efforts towards formalizing the semantics of Rust (e.g. [7], [8], [3], [4]), but none have precisely outlined the notion of a PCS, nor implemented a way to extract this information from a Rust program. The goal of this project will be to expand on these prior works in this area.

2 Core Tasks

1. Formalize the notion of a "Place Capability Summary" (PCS) for a given line of a Rust program, as well as rules for how that summary evolves from one line of code to the next. Note that this may be done at the mid-level intermediate representation (MIR) level of abstraction, as this is the level at which the borrow checker operates.

2. Develop a plan for extracting this information from source code, MIR, and/or the Rust compiler/borrow checker

3. Implement a tool that can derive the PCS for a small subset of Rust code as a proof of concept for fully implementing something that provides a PCS API for the language as a whole

3 Future Work

Completing the implementation of a tool that construct a PCS for a wider swath of Rust code would be the logical continuation of this project.

Additionally, if the PCS is constructed from MIR code, then developing a way to map the summary back to the source level would also be helpful, as it would allow for source-level tools to utilize the API.

Beyond that, future work would involve building an analysis or visualization tool that uses the PCS API to provide information to developers or researchers. This could involve something like the lifetime visualization ideas discussed in [6], [1] or [5].

[1] As in [8], we treat mutability of a place as being determined by whether or not a single variable has exclusive access to it, not by the "mut" keyword. This keyword is just to help Rust developers be explicit about the exclusivity-ness of variables

References

[1] Borrow visualizer for the rust language service, Oct 2016.

[2] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. In *to appear in Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2019.

[3] Shuanglong Kan, David Sanan, Shang-Wei Lin, and Yang Liu. K-rust: An executable formal semantics for rust, 2018.

[4] Eric Reed. Patina: A formalization of the rust programming language. *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02*, 2015.

[5] Jeff Ruffwind. Graphical depiction of ownership and borrowing in rust, Feb 2017.

[6] Jeff Walker. Rust lifetime visualization ideas, Feb 2019.

[7] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. Krust: A formal executable semantics of rust. In *2018 International Symposium on Theoretical Aspects of Software Engineering, TASE 2018, Guangzhou, China, August 29-31, 2018*, pages 44–51, 2018.

[8] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The essence of rust. *CoRR*, abs/1903.00982, 2019.