Poster: Towards Refinement Types in Rust

Jasper Gräflich
Secure Systems
Agentur für Innovationen in der Cybersicherheit (Cyberagentur)
Halle, Germany
graeflich@cyberagentur.de
and
Potsdam University
Potsdam, Germany
graeflich@uni-potsdam.de

Abstract—The Rust programming language comes with memory safety by default due to its unique *Ownership and Borrowing System (OBS)*. But Rust provides only limited possibilities to restrict interfaces at compile time so that they only accept well-formed input. To increase Rust's capabilities to precisely specify valid inputs, we propose the introduction of refinement types to Rust which can be checked by SMT solvers.

Keywords—Rust, programming languages, refinement types, formal verification

I. Introduction

Many, if not most, common security vulnerabilities are due to different forms of memory corruption, like buffer overflow, use after free, and race conditions [1]. These are all addressed by Rust's unique *Ownership and Borrowing System (OBS)*. It ensures, at compile time, that resources are always freed correctly, and manages access to them.

However, OBS cannot prevent attackers from providing malformed inputs to a program. This can be resolved by theorem provers. They use elaborate type systems like dependent types to be able to specify and prove, again at compile time, arbitrary properties of values. This makes precise specifications for well-formed inputs possible and enables proving the correctness of programs. By combining OBS and dependent types, we get both memory safety and trusted interfaces.

Unfortunately, dependent types are undecidable and difficult to use. *Refinement types* are a subsystem of dependent types which can mostly be decided by SMT solvers and are easier to use since they resemble set-builder notation.

We propose to integrate a version of refinement types to Rust to enable programmers to formally specify programs with high security requirements and automatically check their correctness while keeping the complexity moderate.

II. The Rust Ownership and Borrowing System

A. Ownership

In most programming languages, the programmer can freely create resources, having then the responsibility to drop the resources after the last use. Rust instead has the concept of *ownership:* Each resource is bound to exactly one name, which owns the resource and has the responsibility to drop it at some point, usually when the name falls out of scope. This concept is also found in C++ as *Resource Acquisition Is Initialization (RAII).*

To prevent a drop, a resource may be *moved*, i.e., bound to a new name. Access rights and drop responsibilities are then transferred to the new name and the old one may not be used anymore. This is enforced at compile time.

B. References and Mutual Exclusion Principle

Rust provides *references* for temporary access to data without moving values. They have to follow *mutual exclusion,* i.e., mutations to a value are only allowed if an *exclusive reference* is held, of which only one may exist at any time. The owner of the resource is also locked out from access. On the other hand, there can be several *shared references*, but they are read-only to prevent race conditions. Also, as long as references exist, a value may not be dropped, preventing use-after-free.

These rules are enforced by a part of the compiler called the *borrow checker*. It uses lifetime analysis to determine if a reference or a value may be accessed in the future. Since lifetime analysis is undecidable, there are also dynamic versions of the references that enforce mutual exclusion at runtime.

III. Refinement Types

For some type T and some predicate P, a P-*refinement* of T is a type $\{t : T \mid P(t)\}$ consisting of all those values of type T that satisfy the predicate. Refinement types in general are not decidable. However, one can restrict the predicates to SMT formulas, which can be decided by corresponding solvers. Those formulas can involve real numbers, integers, lists, and sets, and some basic operations on them, as well as symbols for uninterpreted functions. Refinement types with SMT formulas as predicates are also called *liquid types* [2].

A. Advantages

Rust has support for algebraic data types, i.e., structs and enums carrying a payload. This is enough to model many behaviors, but it is impossible to specify with them which exact values are permissible to cross interface borders so that runtime checks are required. Using refinement types, a programmer could specify precisely which values are permissible as input and output for functions. Refinement type systems that already exist, as in Lean [3] or Liquid Haskell [4], demand relatively little overhead from the programmer, mainly writing down the specification in formal terms. These specifications would, as part of the source code and different from documentation, not fall out of sync. They would also be amenable to proof. Since refinement type checking happens at compile time, no runtime performance issues arise from using them.

B. Challenges

Even though refinement types are much easier to understand than full dependent types and familiar syntax can be used for logical connectors, there are some subtleties programmers have to be aware of. For example, SMT formulas may contain multiplication by a constant factor but not multiplication of two variables. Uninterpreted function symbols can be used, but only a decidable set of *measure functions* are allowed in formulas. Unfortunately, these intricacies have to be learned, but intuitive syntax and good compiler errors can guide new users and facilitate exploration.

Rust is notorious for its long compilation times, and refinement types would certainly add to that, reducing ergonomics, even though modern SMT solvers can already handle large inputs. Language servers, continuously watching changes and checking them in the background, can make programming with refinement types more practical. The Rust Analyzer is a widespread language server that could be extended to refinement type checking.

C. *Alternatives*

Alternatives to refinement types are just using assertions, or contract programming [5]. Contracts make guarantees very similar to refinement types but do not have to be SMT solvable. They are checked at runtime and usually cause exceptions if they fail. They are valuable for their simplicity but create runtime overheads, limit optimizations and are a hindrance for systems with strong availability requirements. Existing frameworks for contract programming in Rust are *contracts* [6] and *adhesion* [7].

IV. Formalized Refined Rust

There are already several approaches to formal semantics for OBS. Most notable is Polonius [8], a Datalog definition which is part of the 2024 roadmap for Rust and is supposed to serve as the reference implementation for Rust's borrow checker in the future [9]. Polonius works on the so-called *mid-level intermediate representation* (MIR), a program's control-flow graph. Refinements can also be checked using control flow, which is why we base our research on Polonius.

Other semantics include the Polonius-inspired Oxide [10], which is a minimal, type-driven semantics for OBS, and RustSEM [11], a semantics covering large parts of Rust. The RustBelt project [12] specifically looked at so-called *unsafe* features of Rust, which must follow OBS but cannot be validated by the borrow checker.

Regarding formal semantics for refinement types, there are several examples of implementations of refinement types in functional programming languages, like Liquid Haskell [4], and ATS [13]. Imperative languages, like Rust, have not yet received much attention, one notable exception being Refined TypeScript [14]. To date, there is no semantics able to express both OBS and refinement types. *Linear types*, which are similar to OBS, are explored in Haskell and ATS, but interactions with refinement types are not yet systematically analyzed.

V. Future Steps

The immediate future work lies in finding a combined formal semantics that models both refinement types and OBS. For this, Polonius must be formalized and Refined TypeScript made compatible with MIR. Both will then be married into a common formal operational or denotational semantics. After developing the semantics, we can check basic properties, like consistency, and hopefully get insights into interactions between OBS and refinements.

The step after that will be the implementation of a refinement checker according to the specified semantics. Rust's annotation system makes it possible to seamlessly integrate refinements into the surface syntax, but external tool registration is not yet stable [15]. The control-flow checking requires hooking into the compiler after it generates the MIR. This is also still unstable [16].

Finally, we plan to extend the implementation of the checker with refinement type annotations, so that it can verify its own correctness.

Acknowledgment

I thank my advisor Prof. Dr. Christoph Kreitz (Potsdam University) for helping me to get on the path to research, and my coworkers at Cyberagentur, giving me time and support for my project, especially Dr. Sebastian Jester. Also, thanks to Stefanie Djohan for her help in my design-related questions.

References

[1] Gavin Thomas, "A proactive approach to more secure code," [Online]. Available: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

[2] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala, "Liquid Types," pp. 159–169, 2008, doi: 10.1145/1375581.1375602.

[3] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer, "The Lean theorem prover," 2015.

[4] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement types for Haskell," in *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, 2014, pp. 269–282.

[5] Richard J. Mitchell and James C. McKim, "Design by Contract, By Example," in *International Conference on Software Technology: Methods and Tools*, 2001.

[6] karroffel, *contracts.* [Online]. Available: https://gitlab.com/karroffel/contracts

[7] Erich Gubler, *adhesion-rs.* [Online]. Available: https://github.com/erichdongubler/adhesion-rs

[8] Niko Matsakis. "An alias-based formulation of the borrow checker." https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/ (accessed Apr. 6, 2023).

[9] N. M. Josh Triplett. "Rust Lang Roadmap for 2024." https://blog.rust-lang.org/inside-rust/2022/04/04/lang-roadmap-2024.html (accessed Apr. 6, 2023).

[10] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed, "Oxide: The Essence of Rust," *arXiv preprint arXiv:1903.00982*, 2021.

[11] Shuanglong Kan, Zhe Chen, Davin Saán, Shang-Wei Lin, and Yang Liu, "An Executable Operational Semantics for Rust with the Formalization of Ownership and Borrowing," *arXiv preprint arXiv:1804.07608*, 2020.

[12] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the foundations of the Rust programming language," *Proceedings of the ACM on Programming Languages*, vol. 2, POPL, pp. 1–34, 2017.

[13] H. Xi, *The ATS Programming Language*. Citeseer.

[14] P. Vekris, B. Cosman, and R. Jhala, "Refinement types for TypeScript," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 310–325.

[15] Rust Team. "Tracking issue for #![register_tool]." https://github.com/rust-lang/rust/issues/66079

[16] Rust Team. "The Rust Compiler Development Guide: rustc_driver and and rustc_interface." https://rustc-dev-guide.rust-lang.org/rustc-driver.html