Extending Support for Borrows in a Rust Verifier

Master Thesis Project Description

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

Start Date: March 4, 2019
End Date: September 2, 2019

Introduction

Rust is an emerging systems programming language, aiming to provide a memory-safe and type-safe alternative to C and C++. It features an ownership type system, where each value has at most one owner (a variable or a function argument) at all times. The owner is responsible for deallocating the memory occupied by the owned value once it goes out of scope, allowing Rust to provide memory safety guarantees without requiring a garbage collector. Ownership can be transferred by *moving* the value, which is illustrated in Listing 1.

Since moving values all the time is both tedious and potentially expensive, Rust allows one to create references, which are typically called *borrows*. Rust distinguishes between two types of borrows, *mutable* and *immutable* ones. Often times, immutable borrows are also called *shared* borrows. While there is no upper bound on how often a value can be immutably borrowed, only one mutable borrow can be active at all times. Also, no immutable borrow may exist while a mutable borrow is active. This design allows Rust to prevent data races at compile time.

Recently, Astrauskas et al. published a working paper on a novel verification technique for Rust programs [1] which exploits Rust's ownership type system for verification. While the approach currently supports ownership and mutable borrows, support for shared borrows is very limited; currently, shared borrows may only be used as arguments to so-called `pure` functions. A pure function is a function that is both deterministic and side-effect free. This allows pure functions to be used in specifications.

```
1 let x: String = "Hello".to_string();
2 let y = x; // Transfer ownership to y
3 let z = x; // Compile error: x was already moved
```

Listing 1: Transferring ownership of a value in Rust.

This thesis aims to extend the support for shared borrows in the aforementioned technique. The central goal is to allow pure functions to return shared borrows.

Problem

The ownership type system is what enables modular verification of Rust programs. Through ownership information one can determine which memory locations can be accessed by each function, allowing verifiers to not only reason about which memory locations may be changed, but also about which of them are guaranteed to be preserved. This is called *framing* and is one of the main challenges of modular verification.

However, static types and ownership alone do not provide enough information to determine whether a memory access is allowed, as illustrated in Listing 2. The technique from [1] therefore uses *capabilities* to keep track of the access rights to memory locations and to model the Rust compiler's internal representation of this information. Capabilities are computed for each program point and are subsequently used for framing. They may be *packed* and *unpacked*: the former is needed when an entire `struct` is passed as an argument or a return value, the latter when members of a packed `struct` are accessed.

```
1  struct Pair {
2    fst: i32,
3    snd: i32,
4  }
5
6  fn foo() {
7    let mut p = Pair { fst: 0, snd: 1 };
8    let x = &mut p;
9    p.fst = 2;
10   x.snd = 1;
11 }
```

Listing 2: An example which demonstrates that ownership does not imply being allowed to access memory locations. While `p` owns the created `Point`, the statement on line 9 causes a compile error. Since `x` mutably borrows `p`, the read and write permissions are temporarily transferred to `x`. As long as `x` is active, `p` cannot be used to access the underlying `Point`.

Prusti — an implementation of the technique described in [1] — translates Rust programs to Viper [2], an intermediate verification language. Viper's logic is based on implicit dynamic frames [4] and uses so-called *permissions* to solve the framing problem. Similar to Rust, where a memory location can be accessed if a reference has the necessary capability, memory locations in Viper can be accessed if the required *permission* is held to do so. Unlike Rust, Viper's permission accounting is much more precise; for example, the implicit packing and unpacking operations for capabilities are explicit operations in Viper. Moreover, while a shared reference capability in Rust can be duplicated, Viper's permissions are non-duplicable; instead, they can be split and recombined. Therefore, figuring out which Viper operations to emit is an extremely challenging task in some cases (for instance when a borrow expires).

This mismatch between the precise permission accounting in Viper and the more relaxed model used in Rust both complicates and significantly slows down the verification process. It is also the reason why pure functions cannot return shared borrows.

Approach

Our plan is to change or extend Viper's permission model to make it better suited for modeling Rust code. As a starting point, we plan to explore a new mechanism which eliminates the need to perform explicit unpacking by allowing Viper code to peek arbitrarily deep into a packed capability. One of Viper's verification backends — Silicon — recently gained support for custom resources [3], which we might want to leverage for the evaluation of our methodology.

As mentioned before, the primary goal is to allow pure functions to return shared borrows. Our approach might be suitable to also support other types of shared borrows, for instance shared references as arguments to impure functions. However, supporting internal mutability (for instance Rust's `Cell<T>`), which allows writing to memory to which only a shared borrow exists, is beyond the scope of this thesis.

Goals

Core Goals

1 Collect Samples

In a first step, we will collect a set of Rust programs, that we want to be able to verify, but are currently not verifiable.

2 Design Methodology

The second goal is to develop a methodology as an extension of Viper that enables the verification of the collected sample set. A starting point was sketched in the previous section. The potential interactions of this extension with the Viper features used by Prusti have to be considered.

3 Extending Viper and Prusti

To determine whether the designed methodology actually works, it should be implemented in both Viper and Prusti.

4 Evaluation

Finally, the implementation from the previous goal should be evaluated against both the chosen sample set and functions from the top 500 Rust crates on crates.io that fall into the supported subset of Rust. Furthermore, the interaction with the Viper features Prusti uses has to be explicitly tested.

Extensions

1 Interaction with Other Viper Features

Since Prusti does not use all language features Viper offers, the interaction between those unused features (quantified permissions for example) and the designed extension could be tested as an extension of this thesis.

2 Soundness

Another possible extension is to provide a formal proof why the chosen methodology is sound.

3 Support Shared Borrows in Impure Functions

Since shared borrows often occur as arguments to impure functions, extending the methodology to support such arguments may be considered.

4 Support References as Struct Members

Rust allows structs to have members that are shared references. As a fourth extension, support for such struct members may be implemented.

Schedule

Task	Finished by	Estimate
Project Description	24.03.2019	3 weeks
Collecting Sample Programs	31.03.2019	1 week
Methodology Design	28.04.2019	4 weeks
Implementation	12.05.2019	2 weeks
Evaluation and Bugfixing	02.06.2019	3 weeks
Extensions and Buffer	21.07.2019	8 weeks
Thesis Writing	01.09.2019	7 weeks

References

[1] Vytautas Astrauskas et al. "Leveraging rust types for modular specification and verification". en. 2018-12.

[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)*. Ed. by B. Jobstmann and K. R. M. Leino. Vol. 9583. LNCS. Springer-Verlag, 2016, pp. 41–62.

[3] Robin Sierra. "Towards Customizability of a Symbolic-Execution-Based Program Verifier". Bachelor's Thesis. 2017. URL: https://www.ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Robin_Sierra_BA_report.pdf (visited on 03/14/2019).

[4] Jan Smans, Bart Jacobs, and Frank Piessens. "Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic". In: *ECOOP 2009 – Object-Oriented Programming*. Ed. by Sophia Drossopoulou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 148–172. ISBN: 978-3-642-03013-0.