Automatically Generating Memory Safety Certificates for Rust Programs

Master Thesis Project Description

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

I. Introduction

To this day, C and C++ are still the most widely used systems programming languages. While they are an excellent choice when it comes to resource usage, their memory accesses are not well defined which makes them prone to memory-related bugs. An investigation by Microsoft revealed that approximately 70% of their security vulnerabilities are caused by memory-safety issues [1]. *Rust* is a relatively new programming language which attempts to fix this problem by being both memory-efficient and memory-safe [2] .

Even though the *Rust* compiler guarantees memory safety, its guarantees can not be trusted because it is itself a non-trivial program with a large codebase and provides no formal correctness proofs [3]. From this, we have to conclude that using *Rust* is still not optimal to solve safety-critical problems. Having a certificate that ensures the correctness of a program's memory accesses would be preferable.

RustBelt [4] is a project in which a formal tool was created with which we believe we can create such certificates. The proofs of the *RustBelt* logic are mechanized using the Coq proof assistant which has a small codebase [3]. However, creating the correctness proofs for programs still has to be done manually. It would be more desirable to use the logic of *RustBelt* in a tool such as *Viper*, which is already able to check many properties of *Rust* programs automated through its front-end *Prusti*.

The main goal of this project is to find out if its feasible to automatically generate memory-safety certificates by modelling the *RustBelt* rules in *Viper*.

II. Approach

In this section we explain how we intend to implement a prototype to create certificates followed by a small example to illustrate how such a *Viper* program looks like. We then explain how the prototype will be evaluated and elaborate what kind of limitations and problems we are expecting.

To create memory-safety certificates with *Viper*, we require information about the program under compilation, most notably the reference creations (called *"borrows* in *Rust*), the lifetimes during which the references are valid, and the different types used in the program. The prototype *borrow checker* of the *Rust* compiler, *Polonius*, requires the same data to check memory-safety and provides all the relevant information needed to create the *Viper* certificates. Generating the certificates will be done in the front-end *Prusti*, where all other *Viper* code to check other properties of *Rust* programs is generated.

The certificates will be encoded as shown in the following example in which we verify memory-safety for the *Rust* program in Listing 1. This example illustrates how a borrow can be encoded in *Viper* and that our *Viper* certificate does not allow to have a second mutable borrow of the same value (which is not allowed in the *Rust* language).

```rust
fn assignment(){
    let mut a: i32 = 4;
    let x: &mut i32 = &mut a;
    ...
}
```

Listing 1: Variable `a` of type `i32` is instantiated on line 2 and the variable `x` of type `&mut i32` then borrows `a` with write-permission on line 3. We assume some code is present after those two lines such that the compiler can not remove them.

Listing 2 is a *Viper* certificate for the program in Listing 1. Note that the definitions of the domain `Lifetime`, the predicates `Owned`, `MutRef` and `DeadLifetimeToken` and the methods `borrow`, `newlft` and `endlft` are not shown.

At first, the value 4 is assigned to the field `a.val` before the predicate `Owned(a)` is folded to represent the ownership of that value. We then assign `a` to `x.ref` and as a preparation for the modelling of this borrow we create a lifetime for both the reference and the assigned object (`lft_x` and `lft_mut_a`) and a sufficiently small permission-amount `lft_perm` for read-access for the lifetimes. We then call the `borrow` method on line 20 which takes away our folded `Owned(a)` predicate and in return gives us a mutable reference in form of a `MutRef` predicate. Note that at this point we don't have ownership of `a` anymore and we can thus not borrow `a` again. A so called *magic wand* can later be applied to regain ownership of `a` as soon as the lifetime of the borrow has ended (as on line 33). After the borrow and the remaining function, we have reached the end of the *Rust* program and can therefore end the lifetimes, destroy the reference `x`, regain ownership of `a` using the magic wand and finally destroy `a`.

```
 1  method assignment() {
 2
 3      // let mut a = 4
 4      var a: Ref
 5      inhale acc(a.val)
 6      a.val := 4
 7      fold Owned(a)
 8
 9      // let x = &mut a
10      var x: Ref
11      inhale acc(x.ref)
12      x.ref := a
13
14      // model the borrow
15      var lft_perm: Perm := write / 3
16      var lft_mut_a: Lifetime
17      var lft_x: Lifetime
18      lft_mut_a := newlft()
19      lft_x := lft_mut_a
20      borrow(lft_mut_a, lft_perm, x.ref)
21      assert acc(MutRef(lft_mut_a, x.ref))
22
23      // encoding of remaining function
24      // ...
25
26      // end of scope
27      endlft(lft_x)
28
29      // destroy reference x
30      exhale acc(x.ref)
31
32      // regain ownership of a
33      apply acc(DeadLifetimeToken(lft_mut_a))
34            --* acc(Owned(a))
35
36      // destroy value a
37      unfold Owned(a)
38      exhale acc(a.val)
39  }
```

Listing 2: Viper encoding of Listing 1

Such small programs will also be used to validate our implementation. Because of the limited size, we can easily check if the resulting *Viper* code successfully verifies memory-safety. While this will help to create a first prototype, it is not enough to know that the solution also works for real-world applications.

Crater [5] will be used to automatically evaluate our solution on many open source *Rust* programs and libraries. This will be done during the implementation as soon as the first version exists to ensure early detection of issues.

A. Extension Goals

There are some known limitations of the *RustBelt* rules which will consequently not work for us either. In addition we expect that some yet unknown language constructs will be difficult to implement with the rules. Analysing and solving all those problems are the extension goals of this project.

One of the known issues are *Two-phase Borrows* which require both a mutable reference (with write-permission) and a non-mutable reference (read-only) to the same item with overlapping lifetimes of the borrows. Rust only allows this in some special cases and *RustBelt* does not handle them. One example is shown in Listing 3 and expanded in Listing 4 where the vector `v` is needed both to read the length and to insert the value.

```rust
fn main(){
    let mut v = Vec::new();
    v.push(v.len());
}
```

Listing 3: Two-phase Borrow [6]

```rust
fn main(){
    let mut v = Vec::new();
    let temp1 = &two_phase v;
    let temp2 = v.len();
    Vec::push(temp1, temp2);
}
```

Listing 4: Expanded Two-phase Borrow of Listing 3 [6]

Another issue we expect are *Overlapping Shared References*. An example is given in Listing 5 where `t1` references only one of the variables of the object and `t2` references the entire object. Because both variables are used in the `println!` function, their lifetimes overlap.

```rust
fn main (){
  let t = T { x: 1, y: 2};
  let t1 = &t.x; // borrow of t.x
  let t2 = &t;   // borrow of t
  println!("{} {}" t1, t2);
}
```

Listing 5: Overlapping Shared References

Unsafe Rust will also be difficult to handle and will be ignored. This will exclude about 24% of all projects on crates.io [7]. We also expect to find more obstacles when evaluating our implementation on a larger scale.

III. Core Goals

C1	Create small Rust programs for initial tests.
C2	Extend Prusti to add borrow checking using the *RustBelt* rules.
C3	Evaluate the implementation with a large number of *Rust* programs.

IV. Extension Goals

E1	Find and implement a solution for *Two-phase Borrows.*
E2	Find and implement a solution for *Overlapping Shared References.*
E3	Analyze and implement solutions for yet unknown problems.

V. Schedule

6 weeks	Extend Prusti (C1, C2)
2 weeks	Testing and Evaluation (C3) concurrently with previous
8 weeks	Extension goals (E1 - E3)
4 weeks	Write project report

References

[1] S. Fernandez. We need a safer systems programming language. [Online]. Available: https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/

[2] Rust Programming Language. [Online]. Available: https://www.rust-lang.org/

[3] 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, no. POPL, p. 1–34, 2018.

[4] ERC Project "RustBelt". [Online]. Available: https://plv.mpi-sws.org/rustbelt

[5] Crater. [Online]. Available: https://github.com/rust-lang/crater

[6] Guide to Rustc Development - Two-phase borrows. [Online]. Available: https://rustc-dev-guide.rust-lang.org/borrow_check/two_phase_borrows.html

[7] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?" *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, p. 1–27, 2020.