Extended Support for Borrowing and Lifetimes in Prusti
Master Thesis Proposal

Lorenz Gorse

April 28, 2020

1 Rust

Rust is a new systems language aiming to achieve safety, concurrency, and speed. While low-level languages like C are very fast, they allow the programmer to arbitrarily mutate the entire program memory using pointers, which can (and often does) introduce subtle bugs. For example, buffer overflows silently overwrite unrelated memory regions and dangling pointers access memory that has already been repurposed in the meantime. Incorrect use of pointers is the leading cause of security vulnerabilities in software [8], one of the most famous examples probably being OpenSSL's Heartbleed [9].

Rust solves a wide range of commonly occurring problems arising from incorrect use of pointers by imposing a strict memory management discipline on the programmer, enforced at compile time through the type system. At the heart of this discipline are the concepts of ownership [3], borrowing [4], and lifetime [2]:

Ownership. Every memory location is exclusively owned by a variable. When the variable goes out of scope, the memory location is deallocated.

Borrowing. A program part can operate on memory it does not own by borrowing a reference to it. A reference is nothing more than, using C lingo, a pointer, but with additional compile time checks to guarantee memory safety.

Lifetime. Every reference has a lifetime, which is the time during which the reference can be used to access the pointed-to memory location. The Rust compiler enforces two properties about lifetimes. First, every usage of the reference must happen within its lifetime (this imposes a minimal valid lifetime). Second, the variable owning the pointed-to memory location does not go out of scope during the lifetime (this imposes a maximal valid lifetime).

Besides memory safety, the second major application of these Rust concepts is to enforce safe concurrency. Specifically, Rust programs, at least when restricted to the *safe* subset of the language, cannot have data races[1]. A data race happens if one thread reads memory that another thread is mutating at the same time. In Rust, where memory is accessed through references, this can only happen when two threads hold references to the same memory simultaneously. To ensure the absence of data races, the language distinguishes between shared and mutable references. A shared reference allows one to read the pointed-to memory, while a mutable reference additionally allows the mutation of the pointed-to memory. The compiler checks that there is always either exactly one mutable reference or an arbitrary number of shared references to every variable.

2 Prusti

Rust's type system goes a long way towards verifying various memory safety properties about a program. But to statically check complex functional properties of a program, further instruments are necessary. Prusti [7] is a tool that extends Rust with a way to provide functional specifications of functions, i.e., descriptions of what a function does phrased in a formal logic, and also verify their validity. It works by encoding Rust programs

[1] This is true as long as you discount soundness bugs in the compiler [1] that allow you to make tons of money [6].

and their specifications to Viper [10], an intermediate verification language, which ultimately proves their validity. Prusti already supports some of the most essential features of Rust, including certain aspects of the references and borrowing system. In this context, a pattern termed *re-borrowing* is of particular interest. In a re-borrowing situation, a function receives a reference to some data x and returns a reference to data reachable from x. Consider the following simple example:

```
struct Point { x: u32, y: u32 }
fn get_x<'a>(p: &'a mut Point) -> &'a mut u32 { &mut p.x }
```

The function `get_x` receives a mutable reference to a point and returns a mutable reference to data contained within the point, namely the `x` field. We say it re-borrows `p.x` from the `p` reference. Because the input reference and output reference alias and Rust forbids multiple aliasing mutable references, it must block access to one of them while the other is still in use. Indeed, the input reference is unusable as long as the output reference is still in use. Only once the output reference expires can the client use the input reference again.

This temporary restriction of access to the input reference is reflected in the specification. A special environment, called `after_expiry`, contains the facts that become available once the output reference expires. The reason for introducing this environment is twofold. For one thing, because the input reference is not accessible at the time the postcondition is evaluated, it allows the specifications and ultimately the Viper encoding to mirror Rust's behavior more closely. Maybe even more importantly, specifications may wish to put the state of the input reference after the re-borrow expires in relation to the final state of the re-borrow (because modifications of the re-borrow are usually reflected in some way in the state of the original reference). Consequently, such facts cannot be introduced in the postcondition, but only after the final state of the re-borrow is known, i.e., immediately before it's expiry. Prusti's specification language uses the `before_expiry` construct to refer to a re-borrow's pre-expiry state. The specification may then look as follows:

```
#[ensures="after_expiry<result>(
    p.x == before_expiry(*result) &&
    p.y == old(p.y)
)"]
fn get_x<'a>(p: &'a mut Point) -> &'a mut u32 { ... }
```

On the Viper level, the main tool enabling this kind of specification are magic wands, which allow the exchange of one resource for another. In this example, we can use a magic wand to trade the permission to access the output reference for the permission to access the input reference we originally passed into `get_x`. Additionally, the magic wand can also provide us with facts about the memory we just gained the permission to access, like this:

```
RefMutU32(res) --*
    RefMutPoint(p) &&
    p.val_ref.x.val_u32 == old[lhs](res.val_ref.val_u32) &&
    p.val_ref.y.val_u32 == old(p.val_ref.y.val_u32)
```

The current re-borrowing support in Prusti is limited to functions that receive a single reference and return another reference with the same lifetime. However, many re-borrowing functions are more complicated than that. Functions may take out multiple re-borrows, in some cases originating from several different input references with different but possibly related lifetimes. Structures with reference-typed fields can be the source of another re-borrowing scenario that does not involve returned references at all. Certain methods like `iter_mut` can produce a statically unbounded number of re-borrows of a single data structure. All of these can be found out in the wild and would as such constitute a useful extension of Prusti's re-borrowing support. The remaining sections will discuss them in greater detail.

3 Core Goals

A function is not limited to re-borrowing a single reference from the parameter. Continuing with the example from before, a function `get_xy` could return a pair of references to the two coordinates of the point given as parameter:

```
fn get_xy<'a>(p: &'a mut Point) -> (&'a mut u32, &'a mut u32) {
    (&mut p.x, &mut p.y)
}
```

Here, the input reference is blocked until both returned references expire.

Some functions receive multiple references as parameters. In turn, they can return re-borrows from any of the input parameters. There are two variations of this. First, function signatures can mention multiple independent lifetimes:

```
fn foo<'a, 'b>(p: &'a mut Point, q: &'b mut Point)
    -> (&'a mut u32, &'b mut u32)
```

In this case, the first element of the returned pair must be a re-borrow from the `p` parameter, since the lifetime of the `q` parameter is not guaranteed to exceed the lifetime `'a`. A similar observation can be made about the second element of the returned pair.

Second, lifetimes may have inclusion constraints imposed by the function signature. In the following example, the lifetimes `'a` and `'b` must both outlive `'c`:

```
fn foo<'a: 'c, 'b: 'c, 'c>(p: &'a mut Point, q: &'b mut Point)
    -> &'c mut u32
```

Since both `'a` and `'b` are known to outlive `'c`, the returned reference could be re-borrowed from both `p` with lifetime `'a` and `q` with lifetime `'b`. Consequently, both input references are blocked until the output reference expires. This gets even more interesting when the function re-borrows not just one reference with lifetime `'c`, but also another reference with an input lifetime:

```
fn foo<'a: 'c, 'b: 'c, 'c>(p: &'a mut Point, q: &'b mut Point)
    -> (&'c mut u32, &'a mut u32)
```

As before, both input references are blocked at least until the first output reference expires. However, the input reference `p` is also blocked at least until the second output reference expires. Whether or not the two input references are unblocked at the same time therefore depends on the order in which the output references expire.

The core goal of this thesis is to develop a strategy to support the re-borrowing scenarios described above in Prusti. Broadly speaking, this involves the following steps:

1. Determine the exact rules used by the compiler for blocking and unblocking references in re-borrowing situations. This gets particularly interesting for the later scenarios with multiple output references and lifetime constraints.
2. Extend Prusti's specification language for re-borrows to support multiple input and output references. This language must be very flexible, because output references can expire and input references can get unblocked at different times.
3. Derive a Viper encoding of Prusti's extended re-borrowing specification language. This likely means adapting the current use of magic wands, since the permission management gets vastly more complex once there is more than one input or output reference.
4. Implement the solution in Prusti.
5. Evaluate the final implemented solution. The evaluation should show that the encoding works for the presented re-borrowing scenarios and identify potential shortcomings.

4 Extension Goals

4.1 Structures With Lifetime Parameters

Rust's structures can contain reference-typed fields:

```
struct S<'a> { x: &'a mut u32 }
```

The presence of a reference-typed field must be encoded in the type, to inform the type checker that the rules for references also apply to this structure. A structure with reference-typed fields can give rise to another kind of re-borrowing situation, one that looks slightly different that the ones we've seen so far:

```
fn foo<'a, 'b>(s: &'a mut S<'b>, x: &'b mut u32) { ... }
```

Since the function could copy the `x` reference into `s`, the compiler must assume that `x` and a field of `s` alias and therefore blocks `x` until `s` expires. There are a multitude of different variations of this. Structures can have multiple lifetime parameters, a function can take multiple structures with (possibly multiple) lifetime parameters, a function can take structures with lifetime parameters and return re-borrows at the same time, and so on.

This extension goal seeks to add support for structures with lifetime parameters to Prusti. This involves the following steps:

1. Determine the exact rules used by the compiler for blocking and unblocking references in re-borrowing situations that involve structures with lifetime parameters. Due to the high number of different combinations, this is very demanding in itself.
2. Extend or adapt Prusti's specification language once again to support structures with lifetime parameters in re-borrowing situations.
3. Derive a Viper encoding of the new re-borrowing specification language.
4. Implement the solution in Prusti.
5. Evaluate the final implemented solution. The evaluation should show that the encoding works for common re-borrowing scenarios involving structures with lifetime parameters and potentially identify situations where the approach falls short.

4.2 Statically Unbounded Number of Re-Borrows via Iterators

A common pattern seen in both the Rust standard library and the overall ecosystem is the iteration over mutable references to a container's elements. One example of this can be found in slices, where the `iter_mut` function [5] returns an iterator that provides mutable references to all elements of the slice. While the implementation is typically littered with unsafe (the `iter_mut` for slices generously wraps the whole method body in an unsafe block), the function itself is safe to use. And even though the unsafe implementation itself is for that reason not verifiable at the moment, we can still try to provide a trusted specification for `iter_mut` to allow the verification of clients.

The use of `iter_mut` poses several challenges during verification. First of all, there is the permission management. As opposed to earlier re-borrowing scenarios, it is not clear statically how many references the iterator will hand out. However, Viper's permission-based reasoning requires every permission taken from the slice (i.e., every reference taken from the iterator) to be returned back when the permission for the whole slice is assembled back together. Second, there is the functional specification. Previously, modifying a re-borrowed reference had a very clear effect on the original data structure. Now, the element of the original slice that a reference taken from the iterator points to depends on how many references the iterator has already handed out before.

The goal is to support a single data structure with `iter_mut` functionality. This involves the following steps:

1. Determine how `iter_mut` is most commonly used in real code.

2. Find a specification of `iter_mut`'s behavior that is precise enough to support as many of the most common use cases as possible.
3. Develop an encoding of this specification in Viper.
4. Implement the solution in Prusti.
5. Evaluate the final implemented solution. The evaluation should show that the encoding works for the `iter_mut` of one data structure and identify potential shortcomings.

4.3 Generalized Support for a Statically Unbounded Number of Re-Borrows

Iterators are not the only example of APIs that can produce a statically unbounded number of re-borrows. Other cases include reference-counted pointers and manual memory-management tools. Instead of solving the problem of permission management and functional specification for every one of these instances anew, a universal approach that supports this whole class of data structures would clearly be preferable. This requires new specification syntax that is flexible enough to formulate the different behaviors, and it requires developing encodings at the Viper level that are powerful enough to support these specifications.

References

[1] Rust bug tracker: Borrowed referent of a &T sometimes incorrectly allowed. URL: `https://github.com/rust-lang/rust/issues/38899`.

[2] The Rust programming language: Lifetimes. URL: `https://doc.rust-lang.org/1.9.0/book/lifetimes.html`.

[3] The Rust programming language: Ownership. URL: `https://doc.rust-lang.org/1.9.0/book/ownership.html`.

[4] The Rust programming language: References and Borrowing. URL: `https://doc.rust-lang.org/1.9.0/book/references-and-borrowing.html`.

[5] The Rust standard library: Slices. URL: `https://doc.rust-lang.org/stable/std/primitive.slice.html#method.iter_mut`.

[6] Underhanded Rust Contest: Results. URL: `https://web.archive.org/web/20190511070359/http://blog.community.rs/underhanded/2017/09/27/underhanded-results.html`.

[7] 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)*, volume 3, pages 147:1–147:30. ACM, 2019. `doi:10.1145/3360573`.

[8] Catalin Cimpanu. Microsoft: 70 percent of all security bugs are memory safety issues. URL: `https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/`.

[9] Synopsys Inc. The Heartbleed bug. URL: `https://heartbleed.com/`.

[10] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.