Verification of Closures in Rust Programs

Master's Thesis Project Description

Fabian Wolff

Supervisors: Dr. Alexander J. Summers, Prof. Dr. Peter Müller

Start Date: March 30, 2020
End Date: September 29, 2020

1 Introduction

The Rust programming language prevents certain classes of errors commonly found in systems and C/C++ software—such as memory leaks, dangling pointers, and data races—statically, through the use of an advanced *ownership* type system, in which every value always has exactly one *owner*, a variable (and, by extension, a thread).

Sharing of data can happen via *moves* (ownership transfers), copies, and *borrowing*, which refers to the creation of non-owning references to an object, either immutable or mutable. The type system guarantees that only the owner may move or deallocate an object, and only if there are no live references to it. Furthermore, at most one mutable reference may exist at a time, and only if there are no other currently-usable references to the same object, to prevent race conditions and provide *framing* information: a referenced object is never modified unexpectedly by another actor (such as a called function or another thread).[1]

The strong static guarantees made by Rust make it an ideal platform for program verification efforts. Indeed, Astrauskas *et al.* [1] have demonstrated how Rust types can be utilized to simplify specification and verification of programs written in Rust. In this thesis, we will build on their work and expand their approach to include *closures*.

2 Closures

Closures, sometimes also called *lambdas*, are anonymous functions that can capture their lexical environment. They are available in many mainstream languages, including C++, Python and Java. What's special about Rust closures is that the aforementioned rules for

[1] An exception to this rule is the notion of *interior mutability* used by some libraries; this requires the use of `unsafe` code to circumvent some of the restrictions.

value sharing apply; in particular, Rust closures capture their environment by *borrowing* values immutably or mutably or by *moving* values into them. In the following example, `cl` borrows `x` immutably, `y` mutably, and `z` is moved into `cl`:

```
let mut x = "x".to_owned ();
let mut y = "y".to_owned ();
let mut z = "z".to_owned ();
let x_borrow = &x; // we can have overlapping immutable borrows

// || introduces a closure; arguments go between the bars:
let cl = || {
  let x_clone = x.clone (); // this only reads from x
  y.push_str (&x_clone); // modify y
  std::mem::drop (z); // this call requires ownership of z
};

cl ();
println! ("{} {}", x_borrow, y); // we can't use z anymore here
```

Apart from capturing, closures behave like regular functions, even to the extent that they can be coerced into the same (function pointer) type:

```
fn add (a: i32, b: i32) -> i32 { a + b }

let f: fn(i32, i32) -> i32 = add; // works
let g: fn(i32, i32) -> i32 = |a: i32, b: i32| -> i32
  { a + b }; // works, closure doesn't capture anything

let c = 42;
let h: fn(i32, i32) -> i32 = |a: i32, b: i32| -> i32
  { a + b + c }; // fails: closure captures c
```

This restriction to non-capturing closures is necessary: To be able to access the captured state, closures must be wrapped in `struct`-like objects by the compiler, containing fields for all the captured variables, as well as a function pointer to the actual closure code, and thus cannot simply be coerced into function pointers in the general case. Conversely, however, functions can always be automatically wrapped as "closure types":

```
fn inc (i: i32) -> i32 { i + 1 }

let c: i32 = 1;
let f: Box<dyn Fn(i32) -> i32> = Box::new (|i: i32| -> i32 { i + c });

let g: Box<dyn Fn(i32) -> i32> = Box::new (inc);
```

In this example, `inc` will be wrapped in a subtype of `Fn(i32) -> i32`. The `Fn`, `FnMut`, and `FnOnce` traits are common supertypes for closures; specifically, they describe how

closures access their captured state. The `Fn` trait, for instance, is only implemented by closures that neither modify nor move their captured state; since functions don't even have a captured state, they can always be used where a `Fn` trait is expected.

Closures have manifold applications in practice. For instance, closures can often be elegantly employed to control the behavior of certain "abstract" operations, such as filtering:[2]

```
let a = [0i32, 1, 2];
let mut iter = a.iter().filter(|x| x.is_positive());
assert_eq!(iter.next(), Some(&1));
assert_eq!(iter.next(), Some(&2));
assert_eq!(iter.next(), None);
```

or mapping, that is, applying a closure to every element of a container (and thereby modifying the container's contents according to the closure's behavior):[3]

```
let a = [1, 2, 3];
let mut iter = a.iter().map(|x| 2 * x);
assert_eq!(iter.next(), Some(2));
assert_eq!(iter.next(), Some(4));
assert_eq!(iter.next(), Some(6));
assert_eq!(iter.next(), None);
```

This can be particularly useful when working with specialized data structures, such as the `Option` type. In the following example, using a closure allows us to entirely avoid unwrapping or matching on the `Option`:[4]

```
let maybe_some_string = Some(String::from("Hello, World!"));
let maybe_some_len = maybe_some_string.map(|s| s.len());
assert_eq!(maybe_some_len, Some(13));
```

Another common use case for closures is as a callback for when certain events occur, such as when a button is pressed in a GUI framework:[5]

```
use gtk::{Button, ButtonExt};

let button = Button::new_with_label("Click me!");
button.connect_clicked(|but| {
    but.set_label("I've been clicked!");
});
```

In addition, closures can be used for code de-duplication, certain higher-order operations such as partial function application and function composition, as a decorator pattern to wrap other functions, and more.

[2]Example taken from `https://doc.rust-lang.org/std/iter/trait.Iterator.html#method.filter`
[3]Example taken from `https://doc.rust-lang.org/std/iter/trait.Iterator.html#method.map`
[4]Example taken from `https://doc.rust-lang.org/std/option/enum.Option.html#method.map`
[5]Example taken from `https://gtk-rs.org/docs-src/tutorial/closures`

3 Challenges

Closures are useful but make verification trickier for several reasons, including:

- When calling a closure, it is, in general, statically unknown where and how the closure was defined, what internal state it has, which side effects may occur, etc.

  Indeed, not even the precise pre- and postcondition will usually be known to the caller. They could depend on captured memory locations invisible to the caller, so the caller has to rely on abstract guarantees.

- Closure invocation may change the program state, including that of the captured variables (if any) of the closure. Preserving information about the state across closure invocations is thus necessary but difficult; and in particular, verification of repeated closure invocations, such as in the mapping and filtering examples above, becomes complicated.

- To specify the behavior of closures as function arguments and return values, it is necessary to nest function specifications; for instance, the precondition of a function that takes a closure as an argument may have to mention the pre- and postcondition of that closure. The current language and semantics of function specifications therefore have to be extended to support a kind of "higher-order" specification.

  In fact, closures can even be used to implement various higher-order functionalities not supported by the first-order logic of SMT solvers, such as statically-unbounded iterated functions ($f^n = f \circ \ldots \circ f$, for unknown n). This may or may not be a relevant problem for this project, depending on which, if any, of these higher-order features arrive in practical use cases.

Note that many of the same challenges arise in the treatment of regular functions; as described above, closures and functions can often even be used interchangeably. For this reason, we expect this project to benefit not only the verification of code using closures, but also function pointers, wrapped functions in the place of `Fn*` traits, etc. We have (nominally) focused on closures here because they pose the more general challenge.

4 Core Goals

1. **Explore** existing codebases (such as popular crates, Rosetta Code, ...) to identify common classes of closure use cases—such as combinators (filter, map, ...)—or more specialized, but important, use cases, such as AST transformations.

2. Choose one of these classes and collect a set of **examples** representative for this class, both to justify the practical relevance of this project as well as to help guide its path, although the aim remains an approach as generally applicable as possible.

   This includes examples for pure and non-pure, capturing and non-capturing closures, as well as for functions in the place of `Fn*` traits.

3. Develop a **methodology** to handle these examples in the context of the Prusti project.

4. **Implement** this methodology in Prusti.

5. **Evaluate** the implementation on a set of representative and challenging example programs.

In the course of his research internship at ETH, Thomas Hader has already made an initial exploration of the topic of Rust closure verification, based on an earlier, more general inquiry into the issue of closure verification by Kassios and Müller [3]. In his report [2], he gives several examples for closure use cases and outlines a potential verification methodology for some of them. We plan to perform a more in-depth exploration of closure use cases in practice and develop a methodology for a wider variety of such use cases, ideally with a smaller number of core cases (i.e., with one general approach encompassing many use cases).

5 Extension Goals

1. Evaluate the implementation on an existing code base, such as a library or small application, to demonstrate the applicability and strength of the chosen approach.

2. Design and implement techniques to handle more of the closure use cases discovered in the first Core Goal. This could include, for instance, different modes of capturing the environment (compare the `Fn`/`FnMut`/`FnOnce` traits), or boxed closures.

3. Explore simplifications and/or alternatives in the chosen language of specification primitives to simplify closure verification for Prusti end users.

References

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification," *Proceedings of the ACM on Programming Languages*, vol. 3, Oct. 2019. DOI: `https://doi.org/10.1145/3360573`.

[2] T. Hader, "Proposal for supporting closures in Prusti," 2019. [Online]. Available: `https://bit.ly/2Ry7ZGj` (visited on 04/13/2020).

[3] I. T. Kassios and P. Müller, "Specification and verification of closures," ETH Zurich, Department of Computer Science, Tech. Rep., 2010. DOI: `https://doi.org/10.3929/ethz-a-006843251`.