Aeneas: Rust Verification by Functional Translation

Son Ho, Jonathan Protzenko, Aymeric Fromherz

AENEAS: Rust Verification by Functional Translation

Son Ho
Inria
son.ho@inria.fr

Jonathan Protzenko
Microsoft Research
protz@microsoft.com

Aymeric Fromherz
Inria
aymeric.fromherz@inria.fr

1 Introduction

We present the latest additions to the AENEAS framework, a verification pipeline for Rust programs based on a lightweight functional translation. AENEAS leverages Rust's rich region-based type system to eliminate memory reasoning for a large class of Rust programs, as long as they do not rely on interior mutability or unsafe code, by compiling Rust programs to pure, executable models. Doing so, we relieve the proof engineer of the burden of memory-based reasoning, allowing them to instead focus on *functional* properties of their code. AENEAS supports a limited but expressive subset of Rust, with shared and mutable borrows, recursive functions and data-structures, and non-nested loops; it currently has backends for F* and Coq. In a 30-min talk, after a quick overview of the core concepts of Aeneas, we propose to present the most recent additions to the framework, namely its support for loops and verification using Coq, as well as ongoing work on extending the supported Rust subset to traits.

2 Background

The key idea behind AENEAS' compilation is that, under the proper restrictions, a Rust function is fully characterized by a *forward* function, which computes its return value at call site, and a set of *backward* functions (one per lifetime), which propagate changes back into the environment upon ending lifetimes, thus accounting for side effects. Such forward and backward functions behave similarly to lenses. Relying on theorem provers to state and prove lemmas about those models, it is then possible to enforce guarantees about the original programs, such as panic freedom and functional correctness.

We first consider an example that, albeit small, showcases many of Rust's features. The `choose` function receives two mutable borrows `x` and `y` of lifetime `'a`, and returns one of the two depending on the value of the boolean `b`. `test_choose` then demonstrates a usage of `choose`.

```
1  fn choose<'a, T>(b: bool, x: &'a mut T, y: &'a mut T) -> &'a mut T {
2    if b { return x; } else { return y; } }
3
4  fn test_choose() {
5      let mut x = 0i32; let mut y = 0i32;
6      let z = choose(true, &mut x, &mut y);
7      *z = *z + 1;
8      assert!(*z == 1);
9      assert!(x == 1); assert!(y == 0); }
```

Properly modeling this program poses several challenges. Inside the function `test_choose`, we must compute a value for `z` where `choose` gets called (line 6). The in-place update (line 7) then indirectly modifies `x` and `y`, whose updated values we finally observe at line 9. We show the code generated by AENEAS in the snippet below, which uses F* syntax where `let*` stands for the monadic bind.

```
1  let choose_fwd (t : Type) (b : bool) (x : t) (y : t) : result t = if b then Return x else Return y
2
3  let choose_back (t : Type) (b : bool) (x : t) (y : t) (ret : t) : result (t & t) =
4    if b then Return (ret, y) else Return (x, ret)
5
```

This work received funding from the France 2030 program managed by the French National Research Agency under grant agreement No. ANR-22-PECY-0006.

```
6  let test_choose_fwd : result unit =
7    let* z = choose_fwd i32 true 0 0 in (* call to choose *)
8    let* z0 = i32_add z 1 in
9    massert (z0 = 1); (* monadic assert *)
10   let* (x0, y0) = choose_back i32 true 0 0 z0 in (* 'a terminates *)
11   massert (x0 = 1); massert (y0 = 0); Return ()
12 let _ = assert (test_choose_fwd = Return ()) (* unit test *)
```

Our solution is to use what we call *forward* and *backward* functions. We use a *forward* function to compute the value returned by a function at call site, i.e., the value `z` returned by `choose` at line 6 in the Rust code, which we translate to a call to `choose_fwd` at line 7 in the F* model. This function follows the normal computation flow from inputs to outputs, hence the name *forward*. We then notice that because `x` and `y` are borrowed for as long as `z` lives, they are only accessible *through* `z` until the `'a` lifetime ends. This allows us to temporarily pretend that only `z` gets incremented at line 7 in the Rust code, meaning we can limit ourselves to computing a new value for `z` in the pure model (`z0` introduced at line 8 in the F* snippet). Then, upon terminating `'a` (beginning of line 9 in the Rust code), we retrieve back access to `x` and `y`. At this point, we have to compute their new values: we do so by using a *backward* function, i.e., `choose_back` at line 10 in the F* model. *Backward* functions use the updated outputs (`z0` in the pure model) to compute new values (`x0` and `y0`) for the inputs that we borrowed at call site and give back upon lifetime termination. As a consequence they go in the direction opposite to the normal computation flow, hence the name *backward*. Finally, we give the possibility of generating unit tests for Rust functions with type `() -> ()` (line 12 in the F* snippet).

An interesting property of our translation is its modularity: in order to generate a model for `test_choose`, we need only look at the *signature* of `choose`, and can ignore its body. We leverage this property at every step of the translation an in particular when handling external dependencies, that we treat as opaque declarations. We believe it will also be useful to implement support for traits.

3 New Features

Aeneas supports a variety of Rust features which enable its application to existing Rust projects. Those features include recursive functions and data-structures, and as a recent addition non-nested loops. For instance, one might write the `nth` function below, which allows taking a mutable reference to the n-th element of a list, mutating it, and regaining ownership of the list, while panicking if n is out of bounds.

```
enum List<T> { Cons(T, Box<List<T>>), Nil }
pub fn nth<'a, T>(mut ls: &'a mut List<T>, mut i: u32) -> &'a mut T {
    while let List::Cons(x, tl) = ls { if i == 0 { return x; } else { ls = tl; i -= 1; } }
    panic!() }
```

A challenge is that the loop recursively dives into the list by manipulating a *mutable* borrow of the current list segment, which we can later use for in-place updates. Contrary to the function case, we can not rely either on a user-written signature to abstract away the loop body. Rather than asking the user for annotations, we compute a fixed-point for the loop, thus giving a memory predicate that is invariant through the loop body; this predicate amounts, in our framework, to a function signature, which allows us to reuse existing facilities to translate loops functionally. We show the complete translation below. In addition to forward and backward functions for `nth`, we also generate recursive forward and backward functions for the loop body (`nth_loop_fwd` and `nth_loop_back`).

```
type list_t (t : Type) = | ListCons : t -> list_t t -> list_t t | ListNil : list_t t

let rec nth_loop_fwd (t : Type)
  (ls : list_t t) (i : u32) : result t =
  begin match ls with
  | ListCons x tl ->
    if i = 0 then Return x
    else let* i0 = u32_sub i 1 in
         nth_loop_fwd t tl i0
  | ListNil -> Fail Failure end

let nth_fwd t ls i = nth_loop_fwd t ls i
```

```
let rec nth_loop_back (t : Type) (ls : list_t t)
  (i : u32) (ret : t) : result (list_t t) =
  begin match ls with
  | ListCons x tl ->
    if i = 0 then Return (ListCons ret tl)
    else let* i0 = u32_sub i 1 in
         let* tl0 = nth_loop_back t tl i0 ret in
         Return (ListCons x tl0)
  | ListNil -> Fail Failure end
let nth_back t ls i ret = nth_loop_back t ls i ret
```