Formalization of Rust Drop Elaboration

Bachelor's Thesis Project Description

Viktor Fukala

supervised by Isaac van Bakel and Prof. Dr. Ralf Jung

Programming Language Foundations Lab, ETH Zürich

February 29, 2024

1 Introduction and Motivation

Rust [1] is an emerging low-level programming language which provides strong safety guarantees and as such aims to help programmers avoid common errors due to, for example, accessing unallocated or uninitialized memory, or data races.

In Rust, whenever an initialized local variable inside a function is overwritten or goes out of scope, it is *dropped* (its destructor is called). When compiling any function, the Rust compiler (`rustc`) initially inserts a drop statement for a variable whenever it goes out of scope regardless of whether it is initialized at that point or not. Later, in a `rustc` pass known as *drop elaboration*, some of the drop statements are removed, enclosed in conditional statements, or otherwise changed so that only the variables that are initialized at the time of the original drop are actually dropped. To be able to dynamically track which variables are initialized, `rustc` might also have to introduce additional statements throughout the entire function, not only near the original drop statements. These statements use auxiliary boolean flags to store which variables are initialized and they further contribute to the complexity of the drop elaboration pass.

As one step on a journey towards the ability to formally verify the Rust compiler, this project's goal is

- to define a simplified model programming language,
- to formally define the drop elaboration step operating on programs in this language, and
- to prove that this elaboration step has the desired properties we expect it to have.

2 Background

2.1 Rust Drop Elaboration

`rustc` doesn't perform drop elaboration directly on the Rust source code but instead on the generated Rust mid-level intermediate representation (MIR) which is in the form of a control-flow graph (CFG).

2.1.1 Conditional Drops

Figure 2.1.1 shows MIR with an already elaborated drop. We can see that `rustc` has introduced an auxiliary variable to keep track of whether `x` has been moved. We say that this drop is a *conditional drop*.

2.1.2 Partial Drops

In Rust, if the type of a variable is a struct that doesn't implement the `Drop` trait, the variable can be partially moved out of. In that case, drop elaboration must make sure that a drop of this variable is elaborated to only drop the fields that are haven't been moved out. We call this *partial dropping* and an example of it is shown in Figure 2.1.2.

```rust
fn g(b: Box<i32>) { }

fn f(n: u32, b: Box<i32>) {
    if n > 0 {
        g(b);
    }
}
```

```
fn f(_1: u32, _2: Box<i32>) -> () {
    debug n => _1;
    debug b => _2;
    let mut _0: ();
    let mut _3: bool;
    let _4: ();
    let mut _5: std::boxed::Box<i32>;
    let mut _6: bool;

    bb0: {
        _6 = const false;
        _6 = const true;
        _3 = Gt(_1, const 0_u32);
        switchInt(move _3)
          -> [0: bb2, otherwise: bb1];
    }

    bb1: {
        _6 = const false;
        _5 = move _2;
        _4 = g(move _5)
          -> [return: bb2, unwind continue];
    }

    bb2: {
        switchInt(_6)
          -> [0: bb3, otherwise: bb4];
    }

    bb3: {
        return;
    }

    bb4: {
        drop(_2)
          -> [return: bb3, unwind continue];
    }
}
```

Figure 1: Rust source on the left and the corresponding Rust MIR after drop elaboration on the right. `rustc` introduces the boolean flag `_6` which is set to `true` whenever variable `_2` is initialized. In `bb2` at the end of the function, `_2` is decided to be dropped if and only if the flag is set.

```rust
fn g(b: Box<i32>) { }

struct Boxes {
    b1: Box<i32>,
    b2: Box<i32>,
}

fn f(bxs: Boxes) {
    g(bxs.b1);
}
```

```
fn f(_1: Boxes) -> () {
   debug bxs => _1;
   let mut _0: ();
   let _2: ();
   let mut _3: std::boxed::Box<i32>;

   bb0: {
      _3 = move (_1.0: std::boxed::Box<i32>);
      _2 = g(move _3)
       -> [return: bb1, unwind: bb4];
   }

   bb1: {
      drop((_1.1: std::boxed::Box<i32>))
       -> [return: bb3, unwind continue];
   }

   bb2 (cleanup): {
      resume;
   }

   bb3: {
      return;
   }

   bb4 (cleanup): {
      drop((_1.1: std::boxed::Box<i32>))
       -> [return: bb2,
           unwind terminate(cleanup)];
   }
}
```

Figure 2: Rust source on the left and the corresponding Rust MIR after drop elaboration on the right. At the end of the function in `bb1`, a drop of `bxs` (represented by `_1`) has been elaborated to only drop `bxs.b2` (represented by `_1.1`) because `bxs.b1` has already been moved. Note that if we disregard the possibility of unwinding (which we will in this project), only basic blocks `bb0`, `bb1`, and `bb3` are relevant in this code listing.

```
fn f(n: int,
     r: ref int,
     r2: (ref int, ref int))
{
    let mut a = 0;
    if n > 0 {
        a = f(move r);
        a = g(move r2.0)
    };
    drop r;
    drop r2
}
```

```
fn f(n: int,
     r: ref int,
     r2: (ref int, ref int))
{
    let mut a = 0;
    let mut flag = true;
    if n > 0 {
        flag = false;
        a = f(move r);
        a = g(move r2.0)
    };
    if flag {
        drop r
    };
    if flag {
        drop r2.0
    };
    drop r2.1
}
```

Figure 3: A possible representation of a program in the model language that we will define, before (left) and after (right) drop elaboration. Note that the elaborated program is slightly optimized in that it uses only one boolean flag (`flag`) for both `r` and `r2.0`.

2.2 Coq

Coq [2] is an interactive theorem prover based on the calculus of inductive constructions. It is being used to formalize logical frameworks (such as Iris [3]) and (parts of) programming languages. We plan to use Coq to formalize our definitions and theorems, and to check our proofs.

3 Goals

3.1 Program Representation

Although choosing the exact representation of programs in our model language is still a future task for us, we already expect our representation to have some key differences from the representation in `rustc`. For the sake of simplicity, we plan to define a recursive type of statements or expressions to represent the programs as opposed to the CFG-based MIR used by `rustc`. Therefore, the drop elaboration step that we formalize will most likely be more similar to what is shown in Figure 3.1 than what Figures 2.1.1 and 2.1.2 display.

3.2 Core Goals

- In Coq, define the syntax of the model language. The features of the model language should include local variables (some with and some without drop obligations), conditional statements, loops, calls to external functions (where values can be moved from local variables into the function's arguments and from the function's return value into a local variable), and drop statements.
- In Coq, define the semantics of the model language before and after drop elaboration.
- In Coq, define the elaboration function which elaborates drops in programs in the model language. This function should support conditional dropping, but by using information from a data-flow analysis that we implement, the elaboration function should refrain from introducing unnecessary boolean flags for variables which are not involved in any non-linear control flow (such as conditionals or loops).

3.3 Extension Goals

- Optimize the elaboration function to avoid unnecessary boolean flags even for variables which are involved in non-linear control flow (if possible).
- In Coq, prove that our elaboration function preserves whole-program semantics.

- Add support for tuples to the model language so that we also model partial dropping.
- Add support for enums to the model language and adjust the data-flow analysis accordingly. The data-flow analysis should take into account that the different variants of an enum might have different drop behavior and that at drop time, it might be impossible for the enum value to be of some of the variants.
- Prove that not only does elaboration preserve the semantics of each individual function but also that in a program consisting of multiple (potentially mutually recursive) functions, elaborating all the functions preserves the execution behavior of the entire program.
- Add support for owned pointers (so that we can, for example, model moving out the inner `Box` in a `Box<Box<T>>`).
- Add support for types with custom destructors to the model language.
- In the Iris framework, prove that a pre-elaboration term contextually refines its corresponding post-elaboration term.

4 Estimated Timeline

Task	Duration
defining model language syntax and semantics	3 weeks
defining an unoptimized elaboration function	2 weeks
proving that the elaboration function preserves whole-program semantics	3 weeks
adding support for tuples and partial dropping	2 weeks
adding support for enums	2 weeks
optimizing the elaboration function	2 weeks
adding support for owned pointers	1 week
adding support for custom destructors	2 weeks
writing the final report	3 weeks

References

[1] "Rust programming language." [Online]. Available: https://www.rust-lang.org/

[2] *Coq Reference Manual*, 2024. [Online]. Available: https://coq.inria.fr/doc/V8.19.0/refman/

[3] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkeda, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *Journal of Functional Programming*, vol. 28, 2018.