Formalization of Rust Drop Elaboration

Bachelor Thesis

Viktor Fukala

August 5, 2024

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

Programming Language Foundations Lab
Department of Computer Science
ETH Zürich

Acknowledgement

I would like to thank my direct supervisor, Isaac van Bakel, for all his helpful advice and his time spent discussing this project with me.

I also thank Professor Jung for his guidance and for offering me the possibility to work on this project with his research group.

Contents

1 Introduction

1.1 Overview

Rust [1] is a modern systems programming language with strong safety guarantees. Rust programs are compiled by the Rust compiler, `rustc`. Among many other things, at one point in the compilation process, `rustc` automatically inserts destructor calls for places whenever they go out of scope. In Rust, the destructor calls are known as *drops*. Initially, they are inserted wherever a drop *might* be necessary. Only later, in a separate pass called *drop elaboration*, the individual drop statements are adjusted (*elaborated*) such that during the eventual execution of the compiled program, destructors will be called *exactly* where and when necessary.

If drop elaboration is done incorrectly, it can have disastrous consequences during program execution. Memory safety violations such as use-after-free or double-free can easily occur if we mistakenly run a destructor on an already moved value. On the other hand, if we mistakenly leave out a destructor call, we are likely to leak resources that would have been deallocated by the destructor.

Despite the great impact of potential bugs, the formal properties of the drop elaboration pass are poorly understood. Drop elaboration is a complex compiler pass partly because it itself operates in several distinct, yet logically connected, passes: To elaborate even a single drop statement, we need information about the potential states of the place being dropped, for which we first may need to analyze the entire function in which the drop appears. Then, if static analysis alone is not sufficient for us to determine how to elaborate a drop, we also need to modify various program locations to make sure we keep track of the required information dynamically. To store this information, we introduce additional local variables which are read by the code that some of the drops elaborated to.

To improve the formal understanding of the drop elaboration pass, we implement our own simplified version of it in the formal proof management system Coq [2], we formulate a formal specification of its correctness, and we provide the foundations for a proof of that specification.

1.1.1 Motivating Example 1

In `rustc`, drop elaboration operates on the Rust mid-level intermediate representation (MIR), which, as the name suggests, is one of several intermediate representations used by the compiler. It is around halfway between the initial Rust source code and the final assembly.

Functions in MIR have the form of control-flow graphs (CFGs). One Rust function is represented by what we call one MIR *body*[1]. In `rustc`, each MIR body carries with it a whole range of information, not all of which is interesting or useful in our discussions of drop elaboration. For our purposes, we reduce the representation of an MIR body to only

1. a CFG representing the executable part, and
2. a list of typed declarations of the represented function's local variables including the declarations of the function's arguments, which are marked as such.

Figure 1 shows an example. Let us first focus on the the body shown on the left side of the figure. There, local variables _0, _1, ..., _6 are declared. `_0` is (always) for the return value and `_1` and `_2` are for the function's arguments `n` and `b` respectively. The example has 4 basic blocks, numbered `0`, `1`, `4`, and `5`. In basic block `4`, we see an example of a drop. The _2 in `drop(_2)` is the place that is being dropped – in this case the local variable _2 corresponding to the function argument `b: Box<i32>`. Crucially, in the body on the left, basic block `4` will be run in all possible executions. Therefore, in that case, if the `drop` always actually called the destructor, the destructor of _2 would be called even if _2 had already been deinitialized by the `move _2` in basic block `1`.

That was the body before drop elaboration. On the right side of the figure, we see the body after drop elaboration. During drop elaboration, `rustc` introduced the boolean flag _7. Instead of the previous basic block `4`, we now have basic block `10`, where execution branches on the value of that flag. If the flag is `true` (`0x01`), the execution continues with the `drop` in basic block `9`, while if the flag is `false` (`0x00`), the

[1] From `rustc_middle::mir::Body`.

execution avoids the `drop` and returns. This is desirable because the flag _7 will be `true` if and only if basic block `1` has not executed, in other words, if and only if the local variable _2 has not been moved and thereby deinitialized.

Let us try to understand how the compiler came up with this elaboration. As part of the drop elaboration pass, `rustc` performed a static analysis of the given body. It concluded that the program location of the `drop(_2)` in basic block `4`,

- _2 might be uninitialized (if `n > 0` and basic block `1` has executed), and also
- _2 might still be initialized (if the `n > 0` branch has not been taken and basic block `1` has not executed).

In such a case, `rustc` cannot *statically* determine whether the `drop` should execute. Therefore, it resorts to introducing the boolean flag _7 whose purpose it is to *dynamically* (at execution time) track if _2 is initialized.

At the beginning of the function, _7 is set to `true` because _2 is a function argument and as such will always be initialized at the beginning. Then, if the `n > 0` condition holds, basic block `1` executes, _2 is moved and _7 is accordingly set to `false`. If `n > 0` does not hold, _2 is not moved and _7 remains set to `true`. In any case, when execution reaches basic block `10`, which the original drop was elaborated to, _2 is initialized if and only if _7 is set to true. That makes it sound to then execute the `drop(_2)` in basic block `9` conditionally on the value of _7, as is achieved by the `switchInt(_7)` that we see in the after-elaboration body.

1.1.2 Motivating Example 2

The previous examples illustrates several basic principles of `rustc` drop elaboration, but it does not show the intricacy that comes up when drop elaboration interacts with some the more complex types that Rust values and places can have.

Tuples and structs play a key role in Rust. At the level of MIR, they are represented in the same way, which is close to tuples in surface Rust: A tuple or struct *s* with, for example, 3 fields, is entirely determined by those 3 fields – `s.0`, `s.1`, and `s.2`.

Similarly to Figure 1, we show an MIR body before drop elaboration on the left of Figure 2. Again, it has only a single drop: `drop(_2)` in basic block `6`. When looking at the rest of the CFG (or at the original Rust source), we notice that some parts of _2 will be or might be deinitialized when basic block `6` is reached. It is the drop elaboration's task to make sure that only the initialized parts will be dropped.

To achieve that, it had to replace basic block `6` by basic blocks `23`, `34`, `35`, `37`, and `38`. These make use of the (also newly introduced) boolean flags `_12` and `_13`. `_12` tracks whether `_2.0.1` (`bxs.0.1`) is initialized and `_13` whether `_2.0.0` (`bxs.0.0`) is initialized. At first, both are set to `true` because `_2.0.0` and `_2.0.1` are initialized as being part of the argument _2. Later, the respective flag is set to `false` when either `_2.0.0` or `_2.0.1` is moved. At the end, basic blocks `38` and `37` conditionally drop `_2.0.0` based on the value of `_13` and `35` and `34` do the same for `_2.0.1` using the value of _12.

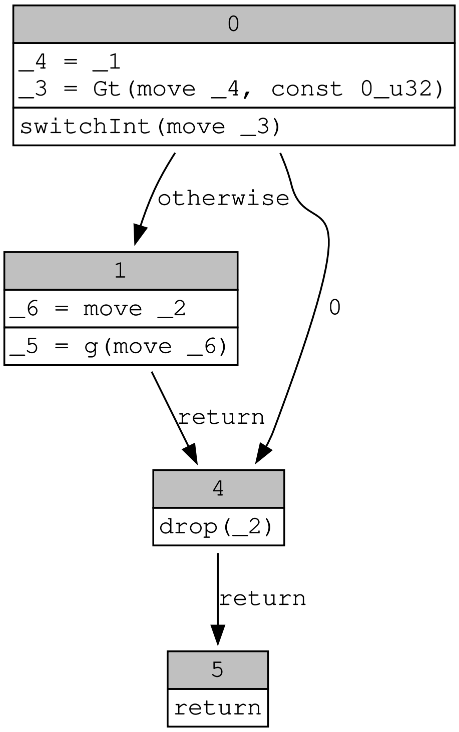
The compiler was able to statically deduce that `_2.1` (and its subfields) is always already deinitialized (due to the move in basic block `5`), so we see no drops of that place. Similarly, `_2.2` is always left initialized, so the drop `drop(_2.2)` in basic block `23` is unconditional.

1.1.3 Our Approach

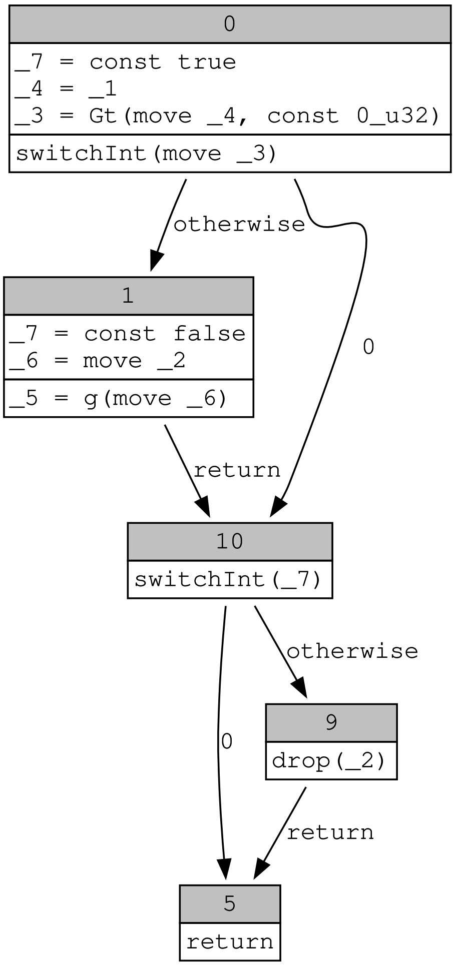
In this project, we *do not* verify the actual `rustc` drop elaboration implementation. At this time, we find it more important to focus on the more abstract ideas of drop elaboration and we believe that working with the real `rustc` implementation (itself written in Rust) would cause us to spend a considerable amount of effort on the less interesting technical details of that implementation.

We define our own, simplified implementation of drop elaboration in the formal proof management system Coq. This greatly lowers the proof engineering burden thanks to Coq's naturally built-in support for proofs. To further simplify our task, we include only a fraction of the features of the `rustc` drop elaboration in our implementation. Crucially, the input and output language (which is also defined by us in Coq) targeted by

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

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



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



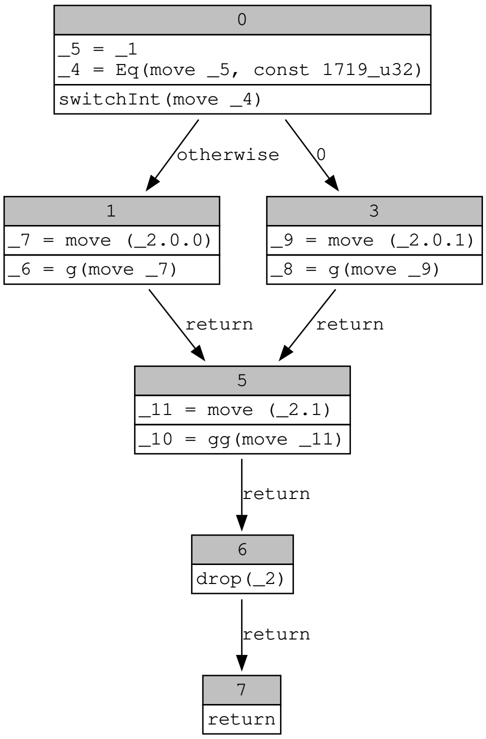
```
fn f(_1: u32, _2: Box<i32>) -> ()
let mut _3: bool;
let mut _4: u32;
let _5: ();
let mut _6: Box<i32>;
let mut _7: bool;
debug n => _1;
debug b => _2;
```

Figure 1: Rust source (top), MIR before drop elaboration (bottom left), and MIR after drop elaboration (bottom right) for an example function `f`. Note how drop elaboration introduced the boolean flag _7 and replaced basic block `4` by basic blocks `9` and `10`.

For readability, we made several simplifications to the MIR produced by the compiler before displaying it here. We removed some features that were not relevant to our discussion, such as basic blocks that handle unwinding, some type annotations, assignments to locals of the unit type, or basic blocks that consisted of only the `goto` terminator.

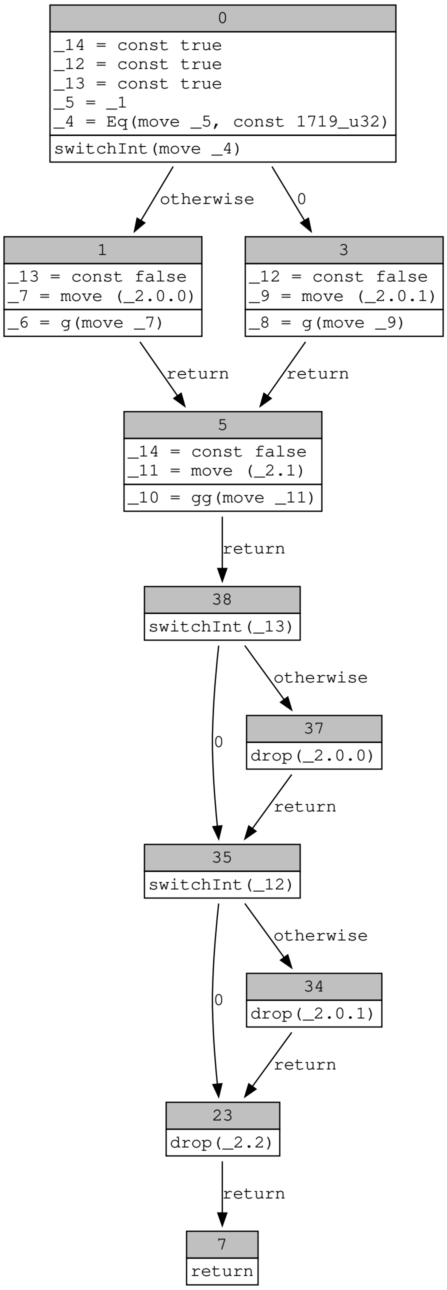
All examples were produced by `rustc 1.81.0-nightly`.

```
fn g(b: Box<i32>) { }
fn gg(bpair: (Box<i32>, Box<i32>)) { }

type Boxes =
  ((Box<i32>, Box<i32>),
   (Box<i32>, Box<i32>),
   (Box<i32>, Box<i32>));

fn f(n: u32, bxs: Boxes) {
    if n == 1719 {
        g(bxs.0.0);
    } else {
        g(bxs.0.1);
    }
    gg(bxs.1);
}
```



```
0
_5 = _1
_4 = Eq(move _5, const 1719_u32)
switchInt(move _4)

1
_7 = move (_2.0.0)
_6 = g(move _7)

3
_9 = move (_2.0.1)
_8 = g(move _9)

5
_11 = move (_2.1)
_10 = gg(move _11)

6
drop(_2)

7
return
```

```
fn f(_1: u32, _2: Boxes) -> ()
let _3: ();
let mut _4: bool;
let mut _5: u32;
let _6: ();
let mut _7: Box<i32>;
let _8: ();
let mut _9: Box<i32>;
let _10: ();
let mut _11: (Box<i32>, Box<i32>);
debug n => _1;
debug bxs => _2;
```



```
0
_14 = const true
_12 = const true
_13 = const true
_5 = _1
_4 = Eq(move _5, const 1719_u32)
switchInt(move _4)

1
_13 = const false
_7 = move (_2.0.0)
_6 = g(move _7)

3
_12 = const false
_9 = move (_2.0.1)
_8 = g(move _9)

5
_14 = const false
_11 = move (_2.1)
_10 = gg(move _11)

38
switchInt(_13)

37
drop(_2.0.0)

35
switchInt(_12)

34
drop(_2.0.1)

23
drop(_2.2)

7
return
```

```
fn f(_1: u32, _2: Boxes) -> ()
let mut _4: bool;
let mut _5: u32;
let _6: ();
let mut _7: Box<i32>;
let _8: ();
let mut _9: Box<i32>;
let _10: ();
let mut _11: (Box<i32>, Box<i32>);
let mut _12: bool;
let mut _13: bool;
let mut _14: bool;
debug n => _1;
debug bxs => _2;
```

Figure 2: Rust source (top left), MIR before drop elaboration (bottom left), and MIR after drop elaboration (right) for an example function `f`. As in Figure 1, the displayed MIR has been simplified. We see that flags _12, _13, and _14 have been introduced and the drop in basic block 6 elaborated to partially conditional drops in basic blocks 23, 34, 35, 37, and 38. Note that the flag _14 is never read in the displayed MIR, but it is read in one of the basic blocks that handle unwinding, all of which we have removed.

our drop elaboration implementation is neither Rust nor Rust MIR but our own, custom, simplified, Rust-like language (hereafter we usually refer to it as just “our language”). Despite all the limitations listed here, we believe that our project advances the formal understanding of the core features of Rust drop elaboration and prepares for the formal verification of their correctness.

As a whole, our project consists of three main parts: the definition of our Rust-like language (of its syntax and semantics), the definition of the drop elaboration function on programs in that language, and the formalization of a meaningful correctness statement for that elaboration. Together with the formal correctness statement, we also provide some partial proofs. All these parts are realized in Coq. The Coq formalization is a major part of this project, and appendix A details how it is available.

2 Background

2.1 Rust

In this section, we clarify some of the Rust concepts mentioned in the discussion of the examples above. We introduce other relevant concepts too.

2.1.1 Places

In the examples above, we sometimes referred to local variables and their fields as *places*. Intuitively, a place is a region of memory that can store a value.

In Rust, generally, a place is given by specifying a local variable and a list of projections. There exist several kinds of projections such as dereferencing, downcasting to a variant of an enum, accessing an array index, or accessing a tuple field. In this project, the language we work with is restricted such that only the field-of-a-tuple projection is relevant. For example, `_2.0.1` is a place specified by the local variable `_2` and a list of the two projections, `.0` and `.1`, i.e., accessing field `0` of `_2` and then accessing field `1` of `_2.0`. To give another simple example, any local variable is a place described by the local variable itself and an empty list of projections.

Each place has an associated type, which is the common type of all the values that the place might hold. In our simplified language, any type is either a tuple or an atomic type which is not built out of other constituent types. We will call any place with a non-tuple type a *leaf place* and any place with a tuple type a *non-leaf place*. Also, if p and p' are places derived from the same local variable such that the list of projections of p is a prefix of the list of projections of p', we say that p' is a *descendant* of p and p an *ancestor* of p'. Note that with this definition, each place is also both a descendant and an ancestor of itself.

2.1.2 Places Being Initialized

At any given point during the execution of an MIR body and for any given place declared in that body, we can define whether the place is initialized, uninitialized, or partially initialized. Each leaf place is always either initialized or uninitialized. At any point, an arbitrary place is initialized if all its descendant leaf places are initialized, uninitialized if all its descendant leaf places are uninitialized, and partially initialized if some of its descendant leaf places are initialized and some uninitialized.

When an MIR body starts executing, a place is initialized if and only if it is a descendant of a local variable which is a function argument in that body. All other places (i.e, those that are descendants of non-argument local variables) are uninitialized at the start. During execution, there are two operations that affect which places are initialized. An assignment to a place initializes all its descendants. Conversely, moving out of a places deinitializes all its descendants.

2.1.3 Destructors

Drops are closely tied to the notion of destructors. We think of every type as having an associated destructor, which is an MIR function with an argument of the given type and no return value. In our project, the only type construction we are concerned with is the tuple/struct. In such a simplified context, the destructor of a type

- first, calls that type's implementation of `Drop::drop`. If the type does not implement the `Drop` trait, there is no action in this part. Then,
- second, if the type is a tuple, the destructor sequentially calls the destructor of each of the tuple's fields in the order of their declaration.[2]

For many common types, the destructor has no effect. These include, for example, types implementing the `Copy` trait such as `bool` or integer types like `u64`, `i32`.

Destructors are important to us because the behavior of drop statements is defined in terms of calls to the appropriate destructors; see 2.1.4.

[2] See `https://doc.rust-lang.org/reference/destructors.html`.

2.1.4 Semantics of `drop` Before And After Drop Elaboration

As we will see later (3.2.5), to formalize the desired correctness property of the drop elaboration pass, we need to understand (and then formalize) the semantics of MIR drop statements.

Importantly, the semantics of `drop` is different before and after drop elaboration. We call it the *pre-elaboration semantics* and the *post-elaboration semantics* in the respective cases.

In the post-elaboration semantics, a drop of a place amounts to calling that place's destructor.

The pre-elaboration semantics is more complex. In it, the behavior of a drop of place depends on which descendants of the place are initialized. If the place itself is initialized, the drop simply calls its destructor. Otherwise, if it is a leaf place, then it must be uninitialized and the drop is a no-op. In the final case, the behavior of the drop can be defined recursively as that of the sequence of drop statements dropping the place's fields in order, starting with `.0`.

2.2 Coq

Coq is a formal proof management system based on the formal language of the calculus of inductive constructions. Coq has been used in many formal developments in programming language research, several of which specifically about Rust, such as RustBelt [3] using the Iris framework [4].

Discussions in this text are themselves independent of the technical details of Coq. They are however a somewhat informal treatment of what it otherwise precisely defined in our formal development, which is written entirely in Coq (see 5.4 and A).

3 Language

For the purposes of our project, we formally define a language on which our implementation drop elaboration operates. This is necessary to formulate formal statements about the properties of drop elaboration and to formulate formal proofs of those statements.

In the ideal case, this language that we formalize would be very close to Rust MIR because it is MIR on which the real drop elaboration in `rustc` operates. Unfortunately, we think that would be impractical because of the relatively high complexity of MIR. Formalizing a language very close to MIR would not only be tedious, it would also be unnecessary or even counterproductive because it would make it harder to focus on only the features that are relevant to drop elaboration, which, ultimately, is the object of our study in this project.

For these reasons, our language has simplifications with respect to MIR. Some simplifications replace an MIR mechanism with an alternative, simpler one: Instead of control-flow graphs, bodies are represented by inductively defined commands and expressions more similar to surface Rust (or some other high-level imperative programming language). Some simplifications abstract over a MIR mechanisms that are not directly relevant to drop elaboration: Memory which is not local to the current function, the heap, is represented by an opaque type and only manipulated in opaque ways through function calls or drops. Finally, some simplifications simply leave MIR features out: Our language does not support enum types.

Before we discuss the more formal technical details of our language, let us go back to the examples in Figure 1 and Figure 2 and demonstrate how they can be replicated in our language. For this, we turn to Figure 3 and Figure 4.

Expectably, the structure of the examples should resemble that of the original ones. However, there are also noticeable differences. Again, these differences primarily serve the purpose of simplifying the representation and focusing on drop elaboration. As motivation of a more formal discussion later, we informally mention a list of some of those observable differences here below:

- All types with a `trait Drop` implementation (such as `Box<T>`) and abstractly represented by a single type `Droppable`. Besides `Droppable`, the types that we support for leaf places `Num` (an unbounded integer) and `Bool`.
- In 4, we see that dropping the third field of `bxs` (`bxs.2`), which is always initialized, was elaborated to two drops – one for each of its two leaf descendants. The actual MIR drop elaboration produces to only a single drop of the entire `bxs.2`. This is because in our formalization of the post-elaboration semantics, we only allow dropping values of type `Droppable` and not tuples consisting of them.
- Due to the specifics of how we formalized the semantics, we need to introduce the `forget` command to make sure that the execution behavior of the pre-elaboration program is always the same as that of the post-elaboration program. See also 3.2.6.
- We introduced the local variable `tmp` because the function calls that we support always take one argument and return one value, which we must assign somewhere. As detailed later (3.2.4), function calls play an important role in the semantics of our language because they enable interaction with otherwise opaque values of the `Droppable` type.
- A read of any place is explicitly annotated with `copy` or `move` for clarity. (MIR leaves copy-accesses unannotated.)

```
b : Droppable
n : Num
tmp : Num

if (copy n) > 0 then
  tmp := g(move b)
end;
drop b
```

```
b : Droppable
n : Num
tmp : Num
flag : Bool

flag := true;
if (copy n) > 0 then
  tmp := g(move b);
  flag := false
end;
if (copy flag) then
  drop b;
  flag := false
end;
forget b;
flag := false
```

Figure 3: A reproduction of the example in Figure 1 in our language. Box on the left shows the body before drop elaboration; box on the right after drop elaboration. Each body has a list of local declarations (shown at the top) and an executable command (shown after the declarations).

```
bxs : ((Droppable, Droppable),
       (Droppable, Droppable),
       (Droppable, Droppable))
n : Num
tmp : Num

if (copy n) == 1719 then
  tmp := g(move bxs.0.0)
else
  tmp := g(move bxs.0.1)
end;
tmp := gg(move bxs.1);
drop bxs
```

```
bxs : ((Droppable, Droppable),
       (Droppable, Droppable),
       (Droppable, Droppable))
n : Num
tmp : Num
flag1 : Bool
flag2 : Bool

flag1 := true;
flag2 := true;
if (copy n) == 1719 then
  tmp := g(move bxs.0.0);
  flag1 := false
else
  tmp := g(move bxs.0.1);
  flag2 := false
end;
tmp := gg(move bxs.1);
if (copy flag1) then
  drop bxs.0.0;
  flag1 := false
end;
if (copy flag2) then
  drop bxs.0.1;
  flag2 := false
end;
drop bxs.2.0;
drop bxs.2.1;
forget bxs;
flag1 := false;
flag2 := false
```

Figure 4: A reproduction of the example in Figure 2 in our language. Body before and after drop elaboration on the left and on the right respectively.

3.1 Syntax

The complete syntax of our the executable snippets of our language can be concisely described as follows:

$$
\begin{array}{ll}
 & n \in \mathbb{Z} \\
 & d \in \mathcal{D} \\
 & x \in \mathit{VarId} \\
 & f \in \mathit{FuncId} \\
 & \mathit{field} \in \{\texttt{0},\texttt{1},\texttt{2},\dots\} \\
\mathit{Place} & p ::= x \mid p.\mathit{field} \\
\mathit{Exp} & e ::= \texttt{copy}\ p \mid \texttt{move}\ p \mid e \odot e \mid \texttt{true} \mid \texttt{false} \mid n \mid d \mid (e,\dots,e) \\
\mathit{Command} & c ::= \texttt{skip} \mid p := e \mid c;c \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c\ \texttt{end} \mid \texttt{while}\ e\ \texttt{do}\ c\ \texttt{end} \\
 & \qquad \mid \texttt{drop}\ p \mid \texttt{forget}\ p \mid p := f(e)
\end{array}
$$

As for the parts which warrant further comments:

- $\mathcal{D}$ is the set of all droppable values, which can be any arbitrary fixed set (see also 3.2.4).
- VarId and FuncId are the sets of variable and function identifiers respectively. We set them to be the sets of strings; although many other sets would be admissible too.[3]
- The $\odot$ symbol stands for any one of several binary operators that we support. These operators take integer or boolean operands and produce integer or boolean values. They include, for example, integer addition, integer equality, and the logical OR. They have little interaction with drop elaboration, so we don't discuss them much further.
- $(e,\dots,e)$ is the syntax for tuples where each field is given by some expression. The dots are not to be read verbatim – they stand for a comma-separated list of expressions.
- In contrast to surface Rust but more similarly to MIR, function do not appear in expressions, only as standalone commands. The syntax for a function call (which always also comes with assignment of the return value), $p := f(e)$, resembles the syntax for an assignment, $p := e$, but they are two definitely separate commands in the syntax.
- We usually abbreviate $\texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ \texttt{skip}\ \texttt{end}$ as just $\texttt{if}\ e\ \texttt{then}\ c\ \texttt{end}$.

Note that constant droppable values do not typically appear in the input or output programs. Droppable values are intended to be introduced primarily through the return values of functions. We do include the "d" variant for Exp because such constants appear in partially executed programs and we reuse the same syntactic representation for them.

3.2 Semantics

3.2.1 Expression Evaluation

In all situations, we use the following evaluation-context-based semantics of expression evaluation (reusing metavariables from 3.1):

$$
\begin{array}{ll}
\mathit{Value} & v ::= \texttt{true} \mid \texttt{false} \mid n \mid d \mid (v,\dots,v) \\
\mathit{ECtx} & E ::= - \mid E \odot e \mid v \odot E \mid (v,\dots,v,E,e,\dots,e)
\end{array}
$$

$$
\frac{v_1\ [\odot]\ v_2 = v}{\langle l, v_1 \odot v_2 \rangle \overset{exp}{\rightsquigarrow} \langle l, v \rangle}\ \text{EXP-BOP}
\qquad
\frac{p \in \mathrm{init}(l)}{\langle l, \texttt{copy}\ p \rangle \overset{exp}{\rightsquigarrow} \langle l, l(p) \rangle}\ \text{EXP-COPY}
\qquad
\frac{p \in \mathrm{init}(l)}{\langle l, \texttt{move}\ p \rangle \overset{exp}{\rightsquigarrow} \langle l \setminus p, l(p) \rangle}\ \text{EXP-MOVE}
$$

$$
\frac{\langle l, e \rangle \overset{exp}{\rightsquigarrow} \langle l', e' \rangle}{\langle l, E[e] \rangle \overset{exp}{\rightsquigarrow} \langle l', E[e'] \rangle}\ \text{EXP-CTX}
$$

[3] The only requirement would be that VarId is infinite so that it is always possible to pick an unused variable identifier when generating a flag.

Expressions are evaluated as part of configurations, $\langle l, e\rangle$, which are pairs consisting of the state of local variables, l, and the expression to evaluate, e. $\overset{exp}{\rightsquigarrow}$ is the relation of executing a basic evaluation step and $\overset{\overline{exp}}{\rightsquigarrow}$ extends it with the possibility of executing within an arbitrary evaluation context.

$\mathrm{init}(l)$ denotes the set of all places initialized in the local state l and $l \setminus p$ denotes the local state l but with the place p deinitialized. For more details, see 3.2.3 and 3.3.3.

3.2.2 Command Execution Besides `drop`

Here below, we describe the structural operational semantics of the most part of our language. These rules apply in all situations. We do not include the semantics of `drop` because it is different before and after drop elaboration – we discuss its semantics further below.

$$CCtx \qquad C ::= p := - \mid \texttt{if} \;\, - \;\, \texttt{then}\; c \;\texttt{else}\; c \;\texttt{end} \mid p := f(-)$$

$$\frac{p \in \mathrm{dom}(l)}{G \vdash \langle l, h, p := v\rangle \rightsquigarrow \langle l[p \mapsto v], h, \texttt{skip}\rangle}\;\textsc{exec-asgn} \qquad \frac{G \vdash \langle l, h, c_1\rangle \rightsquigarrow \langle l', h', c_1'\rangle}{G \vdash \langle l, h, c_1; c_2\rangle \rightsquigarrow \langle l', h', c_1'; c_2\rangle}\;\textsc{exec-seq}$$

$$\frac{}{G \vdash \langle l, h, \texttt{skip}; c\rangle \rightsquigarrow \langle l, h, c\rangle}\;\textsc{exec-seq-skip}$$

$$\frac{}{G \vdash \langle l, h, \texttt{if}\; \texttt{true}\; \texttt{then}\; c \;\texttt{else}\; c' \;\texttt{end}\rangle \rightsquigarrow \langle l, h, c\rangle}\;\textsc{exec-if-then}$$

$$\frac{}{G \vdash \langle l, h, \texttt{if}\; \texttt{false}\; \texttt{then}\; c' \;\texttt{else}\; c \;\texttt{end}\rangle \rightsquigarrow \langle l, h, c'\rangle}\;\textsc{exec-if-else}$$

$$\frac{}{G \vdash \langle l, h, \texttt{while}\; e\; \texttt{do}\; c\; \texttt{end}\rangle \rightsquigarrow \langle l, h, \texttt{if}\; e\; \texttt{then}\; c; \texttt{while}\; e\; \texttt{do}\; c\; \texttt{end}\; \texttt{else}\; \texttt{skip}\; \texttt{end}\rangle}\;\textsc{exec-while}$$

$$\frac{p \in \mathrm{dom}(l)}{G \vdash \langle l, h, \texttt{forget}\; p\rangle \rightsquigarrow \langle l \setminus p, h, \texttt{skip}\rangle}\;\textsc{exec-forget}$$

$$\frac{\mathrm{call}_G(f, v_{\mathrm{arg}}, h, h', v_{\mathrm{ret}})}{G \vdash \langle l, h, p := f(v_{\mathrm{arg}})\rangle \rightsquigarrow \langle l, h', p := v_{\mathrm{ret}}\rangle}\;\textsc{exec-call} \qquad \frac{\langle l, e\rangle \overset{\overline{exp}}{\rightsquigarrow} \langle l', e'\rangle}{G \vdash \langle l, h, C[e]\rangle \rightsquigarrow \langle l', h, C[e']\rangle}\;\textsc{exec-ctx}$$

We express command execution as a relation on the set of possible configurations, which is the set of triples of the form $\langle l, h, c\rangle$, where l is, again, the local state, h is the heap, and c is the command to execute. The execution relation is parametrized by a global context G; $G \vdash \langle l, h, c\rangle \rightsquigarrow \langle l', h', c'\rangle$ is the statement that the configuration $\langle l, h, c\rangle$ executes in one step to the configuration $\langle l', h', c'\rangle$ in the global context G.

We unified the semantics of evaluating expressions nested inside commands in a single rule, EXEC-CTX. It uses the definition of the $CCtx$ evaluation context, which captures the need to evaluate expressions in assignments, if-commands, and function calls.

3.2.3 Local State

The local state l can be viewed as a partial map from a set of places to values, mapping the fully initialized places to their values. As given by the EXP-COPY and the EXP-MOVE rule, only initialized places can be read. On the other other hand, to assign or "forget" a place, it has to exist but it needs not be initialized, which is expressed in EXEC-ASGN and EXEC-FORGET by $p \in \mathrm{dom}(l)$. $\mathrm{dom}(l)$ is the set of all places that exist in the local state l and, in general, $\mathrm{dom}(l) \supseteq \mathrm{init}(l)$. As an example, if l_0 is a local state at the beginning of the execution of a function, a place representing a local variable which is not a function argument will always be in the difference $\mathrm{dom}(l_0) \setminus \mathrm{init}(l_0)$ (the place exists but it is not initialized).

$l[p \mapsto v]$ stands for the local state l updated to map the place p to the value v. This also updates the images of all descendants of p such that they are either mapped to the corresponding subfield of v[4] or do not appear in $\mathrm{dom}(l[p \mapsto v])$ at all.

[4] Inductively on the structure of values, we define a *subfield* of a value v to be either v itself or, if v is a tuple, a subfield of one the fields of v.

The precise structure of the local state is defined in 3.3.3.

3.2.4 Global Context

The global context G carries the complete information about the execution behavior of functions that might be called and of drops (see 3.2.5). Correspondingly, it defines two relations: call_G and drop_G. call_G relates five objects: the function identifier f, the passed function argument v_{arg}, the heap before the call h, the heap after the call h', and the return value v_{ret}. $\mathrm{call}_G(f, v_{\mathrm{arg}}, h, h', v_{\mathrm{ret}})$ means that it is possible that starting with heap h and given the argument v_{arg}, the function f executes to heap h' and returns v_{ret}. Given any f, v_{arg}, and h, there is no restriction on the number of pairs of h' and v_{ret} for which $\mathrm{call}_G(f, v_{\mathrm{arg}}, h, h', v_{\mathrm{ret}})$ holds. In particular, the non-existence of any such h' and v_{ret} can signify that it is invalid to call f in h with the given v_{arg}. On the other hand, multiple such h' and v_{ret} encode non-deterministic behavior of f.

Analogously, drop_G is a relation on three sets instead of five. In particular, $\mathrm{drop}_G(d, h, h')$ means that it is possible that dropping the droppable value $d \in \mathcal{D}$ in the heap h executes to heap h'.

Note we have described neither the set of all droppable values $\mathcal{D}$ nor the set of all possible heap states in any way. They are intentionally left unspecified and the entire formalization is parametrized by what these sets are. This means that the statements that we prove about our language and our drop elaboration implementation hold for *any* choice of the set of heap states, $\mathcal{D}$, and G.

3.2.5 Drop

The semantics of `drop` is the only part that differs between the pre-elaboration and post-elaboration semantics. Post elaboration, we have just

$$\frac{p \in \mathrm{init}(l) \wedge l(p) \in \mathcal{D} \wedge \mathrm{drop}_G(l(p), h, h')}{G \vdash \langle l, h, \texttt{drop}\ p\rangle \rightsquigarrow \langle l, h', \texttt{forget}\ p\rangle}\ \textsc{exec-drop-post}\quad ,$$

which means that only places initialized with a droppable value can be dropped and that has the effect of transforming the heap precisely as given by the drop_G relation. The `drop` executes to a `forget`, which will just deinitialize the place p in the local state as per EXEC-FORGET.

On the other hand, in the pre-elaboration semantics, we have

$$\frac{p \in \mathrm{dom}(l) \wedge \mathrm{smartdrop}_G(l(p), h, h')}{G \vdash \langle l, h, \texttt{drop}\ p\rangle \rightsquigarrow \langle l, h', \texttt{forget}\ p\rangle}\ \textsc{exec-drop-pre}\quad .$$

In contrast to EXEC-DROP-POST, the EXEC-DROP-PRE rule allows to also execute drops of uninitialized or partially-initialized places and places which are initialized with values other than droppable values. Intuitively, the $\mathrm{smartdrop}_G$ relation represents the combined effect of dropping all values $d \in \mathcal{D}$ to which subfields of the dropped v are initialized.

Concretely, for $x \in \{\texttt{true}, \texttt{false}\} \cup \mathbb{Z} \cup \{\bot\}$, $\mathrm{smartdrop}_G(x, h, h') \iff h = h'$, where $\bot$ represents an uninitialized value. For $d \in \mathcal{D}$, $\mathrm{smartdrop}_G(d, h, h') \iff \mathrm{drop}_G(d, h, h')$. And for structs,

$$\begin{gathered}\mathrm{smartdrop}_G((v_1, \ldots, v_k), h, h') \\ \iff \\ \exists h_0, h_1, \ldots, h_k.\ h_0 = h \wedge \mathrm{smartdrop}_G(v_1, h_0, h_1) \wedge \cdots \wedge \mathrm{smartdrop}_G(v_k, h_{k-1}, h_k),\end{gathered}$$

for any, even partially-initialized, values $v_1, \ldots, v_k$. Note that this respects the rule that dropping a tuple drops its fields in the order of their declaration, which is followed in `rustc` (refer to 2.1.3).

Note the differences between EXEC-DROP-PRE and EXEC-DROP-POST. In the pre-elaboration semantics (EXEC-DROP-PRE), `drop` is relatively complex and powerful because it be used on a wide variety of values. It accesses information which actually cannot be directly accessed during execution: whether a place is initialized or not. On the other hand, in the post-elaboration semantics (EXEC-DROP-POST), `drop` is restricted to the simple case of always only dropping initialized droppable values from $\mathcal{D}$.

3.2.6 Forget

The `forget` command does not have a direct equivalent in the Rust MIR and it might be unclear why we introduce it in our language.

We could remove `forget` from our syntax, remove the corresponding EXEC-FORGET rule, and make our `drop` rules directly deinitialize the dropped place, like so:

$$\frac{p \in \mathrm{dom}(l) \land \mathrm{smartdrop}_G(l(p), h, h')}{G \vdash \langle l, h, \texttt{drop}\ p\rangle \rightsquigarrow \langle l \setminus p, h', \texttt{skip}\rangle}\ \textsc{exec-drop-pre-no-forget}$$

$$\frac{p \in \mathrm{init}(l) \land l(p) \in \mathcal{D} \land \mathrm{drop}_G(l(p), h, h')}{G \vdash \langle l, h, \texttt{drop}\ p\rangle \rightsquigarrow \langle l \setminus p, h', \texttt{skip}\rangle}\ \textsc{exec-drop-post-no-forget}\quad .$$

Unfortunately, in such a language, we could not properly elaborate drops of non-leaf places. In the correctness statement of our elaboration, we assert that a particular execution of a function in the pre-elaboration semantics is possible if and only if a similar execution of the function after drop elaboration is possible in the post-elaboration semantics (see 5.3 and 5.1 for details). This could not be reasonably be in this language without `forget`:

For example, suppose we have a local variable `x: (Droppable, Num)`, suppose that it is fully initialized, and also suppose G is such that $\mathrm{drop}_G(d, h, h')$ and $\mathrm{call}_G(f, v_{\mathrm{arg}}, h, h', v_{\mathrm{ret}})$ hold for any $d, h, h', v_{\mathrm{arg}}$ and v_{ret}. Then, in pre-elaboration semantics, `drop x; x.1 := 3` cannot execute because after the `drop`, `x` has been removed from the local state, so the place `x.1` does not exist in the local state. However, in the post-elaboration semantics, `drop x.0; x.1 := 3`, which the original command naturally elaborates to, can always execute because the place `x.1` still exists.

In our actual language, we can elaborate to `drop x.0; forget x; x.1 := 3`, which, correctly, will not be able to execute.

3.3 T-Trees And T-Forests

Let us clarify how we represent the local state. The descendants of a places of a local variable are naturally organized in a tree structure such that places are descendants of one another in the tree if and only if they are descendants as places. Since a tuple can have an arbitrary number of fields, nodes in such a tree can have an arbitrary number of children.

3.3.1 T-Trees

We call such trees *T-trees*[5]. A T-tree is defined over some base set of objects that can be stored in its leaves. Specifically, given a set S the set $\mathit{ttree}(S)$ can be defined inductively as containing all the elements S and all the lists (tuples) of elements of $\mathit{ttree}(S)$.

1. $S \subseteq \mathit{ttree}(S)$
2. $k \geq 0 \land t_1, \ldots, t_k \in \mathit{ttree}(S) \implies (t_1, \ldots, t_k) \in \mathit{ttree}(S)$

We can define the *domain* of a T-tree as the set of all lists of indexes that represent one of its subtrees. For $t \in \mathit{ttree}(S)$,

1. if $t \in S$, then $\mathrm{dom}(t) = \{()\}$ (only the empty list)
2. if $t = (t_1, \ldots, t_k)$, then $\mathrm{dom}(t) = \{()\} \cup \bigcup_{i=1}^{k} \{I \text{ prepend } i \mid I \in \mathrm{dom}(t_i)\}$.

Given a T-tree $t \in \mathit{ttree}(S)$ and $I \in \mathrm{dom}(t)$, we use $t(I)$ to denote the subtree of t corresponding to the path of indices l. $t(I)$ is itself also a T-tree; $t(I) \in \mathit{ttree}(S)$. Given some other $r \in \mathit{ttree}(S)$, we also write $t[I \mapsto r]$ to denote a new T-tree where the subtree $t(I)$ is replaced by t'.

[5] T for tuple.

3.3.2 T-Forests

Since there can be multiple different variables, we define a notion of *T-forests*. A T-forest over S is just a partial map $VarId \rightharpoonup ttree(S)$. We write $tforest(S) = \{f : VarId \rightharpoonup ttree(S)\}$. For any $f \in tforest(S)$, we define $\mathrm{dom}(f)$ as a set of places

$$\mathrm{dom}(f) = \{p \in Place \mid \exists var, I, t.\ f : var \mapsto t \wedge I \in \mathrm{dom}(t) \wedge p = var \text{ followed by } I\}.$$

For any $f \in tforest(S)$, $p \in \mathrm{dom}(f)$, and $r \in ttree(S)$, with $p = (var \text{ followed by } I)$, we define $f(p) = f(var)(I)$ and $f[p \mapsto r] = f[var \mapsto f(var)[I \mapsto r]]$. Additionally, we define we set of all variables to which a forest assigns some tree: $\mathrm{vars}(f) = \{var \in VarId \mid var \in \mathrm{dom}(f)\}$.

3.3.3 Local State As a T-Forest

We can define the local state as a T-forest over the base set $\mathcal{V}'_{\text{base}} := \mathbb{Z} \cup \{\texttt{true}, \texttt{false}\} \cup \mathcal{D} \cup \{\bot\}$. The notations that we introduced specifically for the local state in 3.2.1 and 3.2.3 for reading and assigning values and for the domain of a local state then coincide with those for T-forests. Additionally, for $p \in \mathrm{dom}(l)$, we define $l \setminus p = l[p \mapsto \bot]$ and $\mathrm{init}(l) = \{p \in \mathrm{dom}(l) \mid \forall I \in \mathrm{dom}(l(p)).\ l(p)(I) \neq \bot\}$.

3.3.4 Types As T-Trees

We also represent types using T-trees: A type is a T-tree over the base set $\mathcal{T}_{\text{base}} = \{\texttt{Num}, \texttt{Bool}, \texttt{Droppable}\}$. The typing context of all local variables is then represented by a T-forest over the same set, which generally consider to be arbitrary but fixed and denote by $\Gamma \in tforest(\mathcal{T}_{\text{base}})$.

We need to work with types because the elaboration of a place naturally depends on the place's type (for example, `drop x.3` might be elaborated to `skip` if `x.3` has type `Num`, `drop x.3` if `x.3` has type `Droppable`, and `drop x.3.0; drop x.3.1.1` if `x.3` has type `(Droppable, (Num, Droppable))`). As such, a particular elaboration of drops is correct only in the given type context. Therefore, we also need to restrict the values in the local state with which we execute the function to the types prescribed by Γ and we need to require that the body that we elaborate only accesses places consistent with Γ. See also 5.2.

For the above purpose, we want to be able to express that a T-tree follows the structure of another ttree. For T-trees t_1 and t_2, possibly over different base sets, we write $t_1 \preccurlyeq t_2$ if either

- t_1 is just a leaf (an element from its base set), or
- t_1 and t_2 are both lists of T-trees, $t_1 = (t_{1,i})_i$ and $t_2 = (t_{2,i})_i$, with the same length, and for all i, $t_{1,i} \preccurlyeq t_{2,i}$.

Note that $t_1 \preccurlyeq t_2$ implies $\mathrm{dom}(t_1) \subseteq \mathrm{dom}(t_2)$ but not the other way around.

For two T-forests f_1 and f_2, we define that $f_1 \preccurlyeq f_2$ if for all $var \in VarId$,

$$(var \in \mathrm{dom}(f_1) \iff var \in \mathrm{dom}(f_2)) \wedge (var \in \mathrm{dom}(f_1) \implies f_1(var) \preccurlyeq f_2(var))$$

For example, for any local state l that we attain, it should hold that $l \preccurlyeq \Gamma$. (This is a statement that we prove given some assumptions.)

4 Drop Elaboration

There many possible ways to correctly elaborate a particular function. In both rustc and in our implementation, the general idea is to introduce a set of boolean flags which track whether individual places are initialized. Two kinds of accesses to these flags are introduced:

- A flag is assigned when the corresponding place is initialized or deinitialized (assigned true when the place becomes initialized and false when it becomes uninitialized).
- A corresponding flag is read when it appears as a condition in an if-command around a drop statement to determine whether a particular place should be dropped.

In contrast to some potential alternatives, we do not need to move drop-commands around the elaborated function in this approach. We only replace each drop command by some elaborated version of it (e.g. by enclosing it in an if-command).

Following the above description, one correct implementation of drop elaboration could create a boolean flag for every droppable leaf place that appears in the function, elaborate any drop of a place p to the sequential composition of dropping all droppable leaf places descendant of p, and then for each drop of a droppable leaf place q, drop q, to a conditional which only executes the drop if the flag corresponding to q is true, if $flag(q)$ then drop q end.

While this is a correct and feasible solution, we make an effort to minimize the footprint that drop elaboration leaves in the compiled program. Otherwise, the performance of the resulting program is likely to be negatively affected. Specifically, our main goal is to reduce the number of added boolean flags as much as practically possible.

In the above-outline approach which adds a flag for every droppable places, some flags are unnecessary. For some drops, we can statically determine that the place they are dropping will always be initialized when execution reaches the drop and for some other drops, we can determine that they will always be uninitialized. For places that only appear in such drops, we do not need a flag.

To that end, we first perform an analysis on the function we are about to elaborate. The goal of the analysis is to determine, at the each program location of a drop in the function, which places might be initialized and which might uninitialized.

4.1 Initialization State

Let us specify how we represent the initialization state our analysis computes at each program location. By *initialization state*, we mean the statically-computed finite representation of a possibly infinite set of dynamic program execution states that respect some property about which places are initialized. Specifically, given a particular local state l and a particular initialization σ, we can always decide whether l has places initialized in a way consistent with σ or not. If it does, we say that l belongs to σ and write $l \in \sigma$. Otherwise, $l \notin \sigma$.

One simple initialization state representation would be a map from the set of all droppable leaf places to the set $\mathcal{I}_{\text{base}} = \{\texttt{IS\_INIT}, \texttt{IS\_UNINIT}, \texttt{UNKNOWN}\}$. While this would also be sensible choice, it does not provide quite as much information as we would hope for. Specifically, consider a tuple s : (Droppable, Droppable). Suppose that at some program location, we know that s is either initialized or uninitialized but is not partially initialized. Such a state would be represented by the map

$$\{\texttt{s.0} \mapsto \texttt{UNKNOWN}, \texttt{s.1} \mapsto \texttt{UNKNOWN}\}.$$

However, this same map also represents the initialization state where we know nothing at all about how s is initialized. In other words, such a simple map representation does not capture information about correlations between which leaf places are initialized. Since this information is useful in limiting the number of number of generated flags (see 4.2), we choose a more sophisticated representation.

Luckily, we can reuse the notions of T-trees and T-forests. We represent the initialization state of a single variable of type τ as a T-tree over the base set $\mathcal{I}_{\text{base}}$ such that $\text{dom}(\iota) \subseteq \text{dom}(\tau)$. Going back to the example when we know that the variable s is not partially initialized, that would be represented just by a zero-depth T-tree $\iota = \texttt{UNKNOWN}$. We can think of each UNKNOWN as being (independently from all other UNKNOWNs) substituable by either an `IS_INIT` or a `IS_UNINIT`. So if we have only one `UNKNOWN` for the entirety of `s`, then `s` is either wholly `IS_INIT` or wholly `IS_UNINIT` but not a mixture of the two.

In particular, this is in contrast to the T-tree $\iota = (\texttt{UNKNOWN}, \texttt{UNKNOWN})$. This represents the initialization state where each of the two fields is *independently* `UNKNOWN`. As such, it allows for the possibility that, for example, `s.0` is initialized but `s.1` is not.

Naturally, when a T-tree over $\mathcal{I}_{\text{base}}$ represents an initialization state of a single variable, a T-forest σ over $\mathcal{I}_{\text{base}}$ with $\sigma \preccurlyeq \Gamma$ represents an initialization state of the entire local variable context.

4.2 Drop Elaboration Given Initialization State

Now we describe how, given the initialization state that we have computed (see 4.3 for how), we elaborate any particular `drop`.

Suppose we want to elaborate `drop` p at a location for which we have computed the initialization state $\sigma \in tforest(\mathcal{I}_{\text{base}})$. We start by looking up the initialization state of the particular place p, which will be a T-tree $\iota = \sigma(p)$. However, it is possible that $p \notin \text{dom}(\sigma)$ if the initialization state of p is only summarily described by the initialization state of one of its ancestors. In that case, we define $\iota \coloneqq \sigma(p')$, where p' is the nearest ancestor of p for which $p' \in \text{dom}(\sigma)$.

We wish to compare ι with the type of the place p, which we denote $\tau \coloneqq \Gamma(p)$. Since $\sigma \preccurlyeq \Gamma$, we necessarily have $\iota \preccurlyeq \tau$ and, hence, $\text{dom}(\iota) \subseteq \text{dom}(\tau)$. Therefore, for each leaf of the T-tree ι at some path I, $\iota(I) \in \mathcal{I}_{\text{base}}$, there is a corresponding subtree of τ, $\tau' \coloneqq \tau(I) \in ttree(\mathcal{T}_{\text{base}})$. During elaboration, for each leaf of ι, we consider the corresponding subtree τ' and we emit a particular command:

- If the leaf is `IS_UNINIT`, we only emit `skip`.
- If the leaf is `IS_INIT`, we visit all droppable leaf places in τ' and emit a `drop` of each such place and sequentially compose all of these.
- If the leaf is `UNKNOWN`, we generate a flag for the place that corresponds to this place (which is some descendant of p but not necessarily a leaf place; see also 4.4 for details on flag management). We emit an `if`-command with a `copy` read of the generated flag, the command that would be generate in the `IS_INIT`case above in the `then` branch, and `skip` in the `else` branch.

Whenever we iterate through all leaves of a T-tree, we visit them in depth-first order, prioritizing the first fields in each tuple. When we emit a command for each visited leaf, the command resulting from the whole iteration is the sequential composition of the commands for leaves in that order. This ordering corresponds to the the semantics defined for smartdrop_G (3.2.5).

To avoid generating any code or boolean flags when there is nothing to drop, we additionally check each of the types τ' to see it any of its leaves is `Droppable` and if not, we ignore it altogether instead of following the procedure outlined above. Then, finally, for each `drop` p in the original program, we append `forget` p to its elaboration to ensure that the elaborated program matches the semantics of the original one (see 3.2.6).

4.3 Initialization Effect and Initialization State

To be able to compute the initialization at various program locations, we need to consider how the initialization state can be changed by executing various commands. For this purpose, we introduce the notion of an initialization effect. Any command can be associated with some initialization effect. Each initialization effect is a representation of a particular map from and to the set of initialization states $(tforest(\mathcal{I}_{\text{base}}) \to tforest(\mathcal{I}_{\text{base}}))$.

We want to construct initialization effects in such a way that whenever a command c can execute from a local state l to a local state l' and a l belongs to an initialization state σ, then l' also belongs to the initialization state obtained by applying the initialization effect of c on σ. This property will not immediately follow from our definition; we instead prove it as a lemma in our formal development.

For the concrete representation of initialization effects, we again turn to T-forests. We represent initialization effects as T-forests ϵ over the base set $\mathcal{E}_{\text{base}} = \mathcal{P}(\{\texttt{INIT}, \texttt{DEINIT}, \texttt{KEEP}\}) \setminus \{\emptyset\}$, where $\mathcal{P}$ denotes the set of all subsets, with the requirement that $\epsilon \preccurlyeq \Gamma$. INIT, DEINIT, and KEEP represent actions that a command can have on a place in the local state. It can initialize the place (INIT), it can deinitialize the place (DEINIT), and it can leave it unaffected as to whether it is initialized (KEEP). For certain commands, it is possible that they can have one or the other of these three effects (or any of them) on a certain place. That's why $\mathcal{E}_{\text{base}}$ contains arbitrary subsets of the three actions.

To define the initialization effects of commands, we introduce several basic operations on initialization effects. For two initialization effects ϵ_1 and ϵ_2, we can define their join, $\epsilon_1 \vee \epsilon_2$, which satisfies

$$\forall l, \sigma.\ (l \in \epsilon_1(\sigma) \vee l \in \epsilon_2(\sigma)) \implies l \in (\epsilon_1 \vee \epsilon_2)(\sigma),$$

and their sequential composition $\epsilon_1; \epsilon_2$, which satisfies

$$\forall l, \sigma.\ l \in \epsilon_2(\epsilon_1(\sigma)) \implies l \in (\epsilon_1; \epsilon_2)(\sigma).$$

These can both be given appropriate constructive definitions by recursion on the T-tree structures. For any place $p \in \text{dom}(\Gamma)$, we also define *initialize*(p) (or *deinitialize*(p)), which is represented by a T-forest which has only an INIT (or DEINIT) at p and KEEP elsewhere. Given these building blocks, we can provide our definition of the maps $\phi_{\text{exp}} : \mathit{Exp} \to \mathit{tforest}(\mathcal{E}_{\text{base}})$ and $\phi : \mathit{Command} \to \mathit{tforest}(\mathcal{E}_{\text{base}})$, which to every expression and command assign its initialization effect.

$e \in \mathit{Exp}$	$\phi_{\text{exp}}(e) \in \mathit{tforest}(\mathcal{E}_{\text{base}})$
copy p	KEEP
move p	*deinitialize*(p)
$e_1 \odot e_2$	$\phi_{\text{exp}}(e_1); \phi_{\text{exp}}(e_2)$
$(e_1, \ldots, e_k)$	$\phi_{\text{exp}}(e_1); \ldots; \phi_{\text{exp}}(e_k)$
true, false, n, d	KEEP

$c \in \mathit{Command}$	$\phi(c) \in \mathit{tforest}(\mathcal{E}_{\text{base}})$
skip	KEEP
$p := e$	$\phi_{\text{exp}}(e); \mathit{initialize}(p)$
$c_1; c_2$	$\phi(c_1); \phi(c_2)$
if e then c_1 else c_2 end	$\phi_{\text{exp}}(e); (\phi(c_1) \vee \phi(c_2))$
while e do c end	$\phi_{\text{exp}}(e); ((\phi(c); \phi_{\text{exp}}(e)) \vee \text{KEEP})$
drop p	*deinitialize*(p)
forget p	*deinitialize*(p)
$p := f(e)$	$\phi_{\text{exp}}(e); \mathit{initialize}(p)$

Note that in the initialization effect of a while, we do not need to consider more than one loop iteration because it holds for any initialization effect ϵ that $\epsilon; \epsilon = \epsilon$.

At last, we outline the definition of the application of initialization effects on initialization states. The definition of $\epsilon(\sigma)$ is recursive on the T-tree structures:

- If ϵ is a leaf and KEEP $\notin \epsilon$, that is, $\epsilon \in \mathcal{E}_{\text{base}}$, then
  - $\epsilon = \{\text{INIT}\} \implies \epsilon(\sigma) = \text{IS\_INIT}$
  - $\epsilon = \{\text{DEINIT}\} \implies \epsilon(\sigma) = \text{IS\_UNINIT}$
  - $\epsilon = \{\text{INIT}, \text{DEINIT}\} \implies \epsilon(\sigma) = \text{UNKNOWN}$.
- If ϵ is a leaf, KEEP $\in \epsilon$, and σ is also a leaf (i.e., $\sigma \in \mathcal{I}_{\text{base}}$), we define $b_{\text{i}} = (\text{INIT} \in \epsilon) \vee (\sigma \in \{\text{IS\_INIT}, \text{UNKNOWN}\})$ and $b_{\text{u}} = (\text{DEINIT} \in \epsilon) \vee (\sigma \in \{\text{IS\_UNINIT}, \text{UNKNOWN}\})$ and
  - if $b_{\text{i}} \wedge b_{\text{u}}$, then $\epsilon(\sigma) = \text{UNKNOWN}$,

  - if $b_\mathrm{i} \wedge \neg b_\mathrm{u}$, then $\epsilon(\sigma) = \texttt{IS\_INIT}$,
  - if $\neg b_\mathrm{i} \wedge b_\mathrm{u}$, then $\epsilon(\sigma) = \texttt{IS\_UNINIT}$,
  - ($\neg b_\mathrm{i} \wedge \neg b_\mathrm{u}$ is impossible).

- If ϵ is a leaf, $\texttt{KEEP} \in \epsilon$, and σ is a list $(\sigma_1, \dots, \sigma_k)$, then $\epsilon(\sigma) = (\epsilon(\sigma_1), \dots, \epsilon(\sigma_k))$.
- If ϵ and σ are both lists, then they necessarily have the same length, so we can write $\epsilon = (\epsilon_1, \dots, \epsilon_k)$ and $\sigma = (\sigma_1, \dots, \sigma_k)$ and define $\epsilon(\sigma) = (\epsilon_1(\sigma_1), \dots, \epsilon_k(\sigma_k))$.

4.4 Flag Management and Flag Assignments

In 4.2, we at one point say that we generate a flag for a particular place p. That should be a boolean flag which, throughout the program, tracks whether p is initialized. However, while a place can be fully initialized or fully deinitialized (information that can be stored in one boolean), it can also be partially initialized. Therefore, for the purposes of generating a flag for p, we choose one fixed leaf place p' among its descendants, which has the advantage that it cannot be partially initialized. For consistency, we choose p' to be the *first* leaf descendant place of p (if p is a leaf place, we choose p itself, otherwise, we pick the first field of p and recurse).

We can afford to track initializations of p through initializations of p' because the associated flag will only be used when p is either entirely initialized or deinitialized, which means that it is initialized if and only if any one of its descendants is initialized.

For a leaf place, we can introduce a flag that tracks whether it is initialized. To create a name for the flag, we choose an arbitrary variable identifier which does not appear Γ nor has it been used for another flag already. During elaboration, we maintain a map of all places and flags already created so that we can avoid the flags names already used and we can reuse the same flag for the same place. After drop elaboration, for every place p' with flag $flag$, we sequentially compose all atomic commands which initialize (or deinitialize) p' with the assignment $flag \coloneqq \texttt{true}$ (or $flag \coloneqq \texttt{false}$).

5 Drop Elaboration Correctness Proof

Before we can prove that our drop elaboration implementation is correct, we need to formalize what that means.

In general terms, we say that a drop elaboration of a body is correct if the set of possible executions of the body in the pre-elaboration semantics has an appropriate one-to-one correspondence to the set of possible executions of the elaborated body in the post-elaboration semantics. In other words, we want to show that by preserving the execution behavior, the drop elaboration correctly lowers the program from a language with a high-level description of drops (pre-elaboration sem.) to one with a low-level description (post-elaboration sem.).

To formalize this statement, we need to define what it means to execute a body (5.1) and we need to appropriately constrain the sets of executions that we consider (5.2).

5.1 Body Execution

Similarly to MIR (see 1.1.1), the inputs and outputs of our elaboration function are not just commands but entire bodies. In our language, a body B consists of a command $c \in \mathit{Command}$, the local typing context $\Gamma \in \mathit{tforest}(\mathcal{T}_{\text{base}})$, and a set of variable identifiers $A \subseteq \text{vars}(\Gamma)$, which specifies which of the local variables are arguments to the function that the body represents.

In the same way as plain command execution, body execution is parametrized by the global context G (see 3.2.4). The execution behavior of a particular body is given a relation between an assignment of arguments represented by $l_{\text{args}} \in \mathit{tforest}(\mathcal{V}_{\text{base}})$ with $\text{vars}(l_{\text{args}}) = A$, an initial heap h, and a final heap h'. We say a body can execute from l_{args} and h to h' if after completing l_{args} to a local state $l_0 \in \mathit{tforest}(\mathcal{V}'_{\text{base}})$ with $\text{vars}(l_0) = \text{vars}(\Gamma)$ by assigning $\perp$ to every non-argument local variable, there exists a final local state $l' \in \mathit{tforest}(\mathcal{V}'_{\text{base}})$ such that $G \vdash \langle l_0, h, c \rangle \rightsquigarrow \langle l', h', \texttt{skip} \rangle$. We intentionally do not include the final local state l' in the body execution relation because since drop elaboration adds boolean flags to the local state, it would make comparing the pre- and post-elaboration execution behavior more complicated.

5.2 Well-Typedness

We call only can guarantee that there is a meaningful relationship between the execution behavior of the original and the elaborated body if the types given in Γ are respected. One, the argument assignment l_{args} must have the types declared in Γ, and, two, the command c in the body must only make assignments consistent with Γ.

To give an example: We might have a variable `x : (Droppable, Num)` and might have computed an initialization state that at the location of some `drop x`, the variable `x` will always be (fully) initialized. We thus elaborate to `drop x.0; forget x`. However, to be able to guarantee that our elaboration is correct, we must be able to exclude executions such as one in which `x` stores the value $(37, -2)$ because we could execute `drop x` in the pre-elaboration semantics but could not execute `drop x.0; forget x` in the post-elaboration semantics. The exclusion of such malformed executions is the main purpose of our simple type system.

We define that a (potentially partially initialized) value $v \in \mathit{ttree}(\mathcal{V}'_{\text{base}})$ has type $\tau \in \mathit{ttree}(\mathcal{T}_{\text{base}})$ if (recursively) one of the following holds:

- $v = \perp$ (v is an uninitialized leaf);
- v and τ are both leaves and $v \in \mathcal{D}$ and $\tau = \texttt{Droppable}$;
- v and τ are both leaves and $v \in \mathbb{Z}$ and $\tau = \texttt{Num}$;
- v and τ are both leaves and $v \in \{\texttt{true}, \texttt{false}\}$ and $\tau = \texttt{Bool}$;
- v and τ are both lists of the same length k, $v = (v_1, \ldots, v_k)$ and $\tau = (\tau_1, \ldots, \tau_k)$, for all $i \in \{1, \ldots, k\}$, v_i has type τ_i.

Now, we can say that argument assignments l_{args} are well typed with respect to Γ if for all arguments $\mathit{arg} \in A$, the assigned value $l_{\text{args}}(\mathit{arg})$ has type $\Gamma(\mathit{arg})$.

To define which commands are well typed, we first define a typing relation on expressions (elements of the syntactic category *Exp*).

$$\frac{p \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \texttt{copy}\ p : \Gamma(p)}\ \text{TYPE-EXP-COPY} \qquad \frac{p \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \texttt{move}\ p : \Gamma(p)}\ \text{TYPE-EXP-MOVE}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \wedge \Gamma \vdash e_2 : \tau_2 \wedge \mathit{boptype}(\odot, \tau_1, \tau_2, \tau)}{\Gamma \vdash e_1 \odot e_2 : \tau}\ \text{TYPE-EXP-BOP}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}}\ \text{TYPE-EXP-TRUE} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}}\ \text{TYPE-EXP-FALSE} \qquad \frac{n \in \mathbb{Z}}{\Gamma \vdash n : \texttt{Num}}\ \text{TYPE-EXP-NUM}$$

$$\frac{d \in \mathcal{D}}{\Gamma \vdash d : \texttt{Droppable}}\ \text{TYPE-EXP-DROPPABLE} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \wedge \cdots \wedge \Gamma \vdash e_k : \tau_k}{\Gamma \vdash (e_1, \ldots, e_k) : (\tau_1, \ldots, \tau_k)}\ \text{TYPE-EXP-TUPLE}$$

To type entire commands, we need to additionally introduce the typing context for functions that might be called, which we denote Ψ, and for $f \in \mathit{FuncId}$, we use $\Psi \vdash f : \tau_{\text{arg}} \to \tau_{\text{ret}}$ to denote that the function f takes values of type τ_{arg} as argument and returns values of type τ_{ret}. We say that the global context G is well typed if for all f, v_{arg}, h, h', and v_{ret},

$$\mathrm{call}_G(f, v_{\text{arg}}, h, h', v_{\text{ret}}) \wedge v_{\text{arg}} : \tau_{\text{arg}} \implies v_{\text{ret}} : \tau_{\text{ret}}.$$

With that, we can now define when a command c is well typed with respect to some local typing context Γ and global function typing context Ψ, which we denote simply by $\Gamma, \Psi \vdash c$.

$$\frac{}{\Gamma, \Psi \vdash \texttt{skip}}\ \text{TYPE-COM-SKIP} \qquad \frac{p \in \mathrm{dom}(\Gamma) \wedge \Gamma \vdash e : \Gamma(p)}{\Gamma, \Psi \vdash p := e}\ \text{TYPE-COM-ASGN}$$

$$\frac{\Gamma, \Psi \vdash c_1 \wedge \Gamma, \Psi \vdash c_2}{\Gamma, \Psi \vdash c_1; c_2}\ \text{TYPE-COM-SEQ} \qquad \frac{\Gamma \vdash e : \texttt{Bool} \wedge \Gamma, \Psi \vdash c_1 \wedge \Gamma, \Psi \vdash c_2}{\Gamma, \Psi \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end}}\ \text{TYPE-COM-IF}$$

$$\frac{\Gamma \vdash e : \texttt{Bool} \wedge \Gamma, \Psi \vdash c}{\Gamma, \Psi \vdash \texttt{while}\ e\ \texttt{do}\ c\ \texttt{end}}\ \text{TYPE-COM-WHILE} \qquad \frac{p \in \mathrm{dom}(\Gamma)}{\Gamma, \Psi \vdash \texttt{drop}\ p}\ \text{TYPE-COM-DROP}$$

$$\frac{p \in \mathrm{dom}(\Gamma)}{\Gamma, \Psi \vdash \texttt{drop}\ p}\ \text{TYPE-COM-FORGET} \qquad \frac{p \in \mathrm{dom}(\Gamma) \wedge \Gamma \vdash e : \tau_{\text{arg}} \wedge \Psi \vdash f : \tau_{\text{arg}} \to \Gamma(p)}{\Gamma, \Psi \vdash p := f(e)}\ \text{TYPE-COM-CALL}$$

Note that this type system does *not* guarantee that execution will not get stuck (neither in pre-elaboration nor in post-elaboration) because, for example, the program could try to read an uninitialized place.

5.3 Correctness Statement

The notions of body execution and well-typedness defined in 5.1 and 5.2 allows us to formulate the full correctness statement for our drop elaboration implementation. Our elaboration takes a body $B = \langle \Gamma, c, A \rangle$ as input and produces an elaborated body $B' = \langle \Gamma', c', A \rangle$ with the same arguments as output (the command c' is constructed as described in 4 and Γ' by extending Γ with types (simply `Bool`) of the introduced flags). We state the elaboration of our correctness as that for any

- global context G,
- global function typing context Ψ,
- argument assignment $l_{\text{args}} \in \mathit{tforest}(\mathcal{V}_{\text{base}})$ with $\mathrm{vars}(l_{\text{args}}) = A$,
- initial heap h, and
- final heap h',

if

- G is well typed with respect to Ψ,
- c is well typed with respect to Γ and Ψ, $\Gamma, \Psi \vdash c$,
- l_{args} is well typed with respect to Γ,

then B executes from h and l_{args} to h' in the pre-elaboration semantics if and only if B' executes from h and l_{args} to h' in the post-elaboration semantics.

5.4 Proof Formalization in Coq

At the time of writing, we have a Coq implementation of the drop elaboration described in this text and we have a work-in-progress Coq proof of the correctness statement outlined in 5.3. In our Coq development, we do provide a proof of the stated correctness statement, but it relies on many unproved lemmas about the various features of the language and, in some cases, of the drop elaboration itself.

From an earlier iteration of our development, we have a drop elaboration implementation together with a complete proof of its correctness. In this version, however, the language does not support tuples, so the elaboration is only concerned with scalar variables. Therefore, many of the concepts introduced in this text, such as T-trees or the `forget` command, are not relevant and do not appear there.

Information about the access to our Coq development can be found in A.

5.5 Proof Strategies

Details about how we perform specific parts of the proof can be found in the Coq development. Here, let us just note that the core of the correctness theorem is proved by structural induction on the syntax of the executed command c, where the induction hypothesis is that there is some correspondence between the pre-elaboration semantics executions of c and the post-elaboration semantics executions of elaborated c. In that proof, in each constructor case of c, we construct a post-elaboration execution given a pre-elaboration execution and vice versa. To be able to do this, we include, in the induction hypothesis, an invariant that the actual local state during execution belongs to the initialization state computed by our analysis. In the `while` case, we also need to reason by strong induction on the length of the execution trace.

6 Conclusion

In this project, we have used the proof management system Coq to provide a formal definition of a programming language and, on top of it, a formally-defined implementation of drop elaboration to model the drop elaboration pass in the Rust compiler. To reduce its footprint on the elaborated programs, our drop elaboration performs non-trivial optimizations such as avoiding generation of superfluous flags when we can statically determine that whether a place will be initialized or when we can show that entire tuples are always initialized or uninitialized entirely.

We have formally formulated an adequate correctness theorem for the drop elaboration implementation and provided a skeleton for its proof. We do provide a full correctness proof for a simpler language with a simpler drop elaboration, where all local variables are only scalar.

6.1 Future Work

There are many directions in which this project can be extended. The first natural step is the completion of the correctness proof of the drop elaboration we have defined.

One meaningful way to build on top of this project is to add support for more features that exists in the Rust MIR and in the actual `rustc` drop elaboration. One can implement the drop elaboration of enums and of owned pointers and correspondingly extends the elaboration correctness proof. There is also non-trivial logic that is required to correctly elaborate drops in the presence of unwinding, which is not currently present in our formalization. We think that unwinding support would be more meaningful after changing our language to represent functions as control-flow graphs, which in itself would be an important step towards more closely reproducing the modelled `rustc` behavior. Yet another improvement would be to prove elaboration correctness for not just individual MIR bodies as we do now, but for entire programs that entire programs consisting of multiple functions that can potentially call each other.

We hope that in future, this work might help serve as a basic for the verification of the actual drop elaboration implementation in `rustc`. Although, this goal seems relatively distant at this time and it would require a formal verifier that can reason about an implementation in Rust as opposed to the Coq specification language.

Lastly, there are several minor points that could be improved in our current formal development. Here are a few examples:

- For reasons of technical convenience, our drop elaboration currently inserts a number of `skip` commands into the elaborated body (we removed them in the examples 3 and 4 for brevity and readability). While they do not affect the execution behavior, they make the output harder to read and they could be removed.
- As can be seen in the examples 3 and 4, we insert assignments to the generated boolean flags even at locations after which the flags will never be read (often around `drop` and `forget` commands at the end of the body). These assignments could be optimized away.
- When elaborating a `drop` p, where p is a place of type `Droppable`, we do not need to append a `forget` p as we currently do. That is, we could simplify our elaboration from, for example,

  $$\texttt{if}\ \mathit{flag}\ \texttt{then}\ \texttt{drop}\ p\ \texttt{end};\ \texttt{forget}\ p$$

  to just

  $$\texttt{if}\ \mathit{flag}\ \texttt{then}\ \texttt{drop}\ p\ \texttt{end}$$

  because the `forget` is only necessary when elaborating drops of tuples or of scalar types other than `Droppable`.
- To help simplify our proofs, we might consider making more use of dependent types to avoid defining behavior for cases we intend to be unreachable. For example, we sometimes have a representation of a place p and a T-forest representation of a initialization state σ, from which we need to retrieve the initialization information for just the particular place p. Currently, the retrieve operation returns an option type because p (and all its ancestors) might be outside of $\mathrm{dom}(\sigma)$. However, we are always only interested in the result of this operation if both p and σ have a particular relation to the typing context Γ ($p \in \mathrm{dom}(\Gamma)$ and $\sigma \preccurlyeq \Gamma$), which guarantees there is an ancestor of p in $\mathrm{dom}(\sigma)$. If we used

a dependent type that includes the information $p \in \mathrm{dom}(\Gamma)$ for p and $\sigma \preccurlyeq \Gamma$ for σ, we could avoid the option type for the result of the retrieve operation, which might help reduce the complexity of our code.

References

[1] Rust programming language. `https://www.rust-lang.org/`.

[2] Coq contributors. Coq reference manual. `https://coq.inria.fr/doc/v8.19/refman/`.

[3] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[4] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

A Appendix: Coq Development Location

The repository with our Coq development is located at

`https://gitlab.inf.ethz.ch/ou-plf/rust-drop-elaboration/`.

For the purposes of the thesis, the final version of our Coq development is located on the `thesis-final` branch, tagged as `to-be-graded` and with commit hash

`120d9b3d1e868ddd00ce9163b2bb53d099154247`.

This version contains two variants of our formalization: In the `tuple-support-incomplete-proof` directory, we define a drop elaboration closely follows the description in this text. Unfortunately, its formal correctness proof relies on a large number of unproved lemmas. The `no-tuple-support-complete-proof` directory, on the other hand, stores a simpler version of the elaboration which only supports scalar values and no tuples and which therefore does not use many of the concepts we have introduced here. However, this variant is accompanied by a full correctness proof.