Aeneas: Rust Verification by Functional Translation

SON HO, Inria, France

JONATHAN PROTZENKO, Microsoft Research, USA

We present Aeneas, a new verification toolchain for Rust programs based on a lightweight functional translation. We leverage 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. 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.

The first contribution of Aeneas is a new approach to borrows and controlled aliasing. We propose a pure, functional semantics for LLBC, a Low-Level Borrow Calculus that captures a large subset of Rust programs. Our semantics is value-based, meaning there is no notion of memory, addresses or pointer arithmetic. Our semantics is also ownership-centric, meaning that we enforce soundness of borrows via a semantic criterion based on *loans* rather than through a syntactic type-based *lifetime* discipline. We claim that our semantics captures the *essence* of the borrow mechanism rather than its current implementation in the Rust compiler.

The second contribution of Aeneas is a translation from LLBC to a pure lambda-calculus. This allows the user to reason about the original Rust program through the theorem prover of their choice, and fulfills our promise of enabling lightweight verification of Rust programs. To deal with the well-known technical difficulty of terminating a borrow, we rely on a novel approach, in which we approximate the borrow graph in the presence of function calls. This in turn allows us to perform the translation using a new technical device called *backward functions*.

We implement our toolchain in a mixture of Rust and OCaml; our chief case study is a low-level, resizing hash table, for which we prove functional correctness, the first such result in Rust. Our evaluation shows significant gains of verification productivity for the programmer. This paper therefore establishes a new point in the design space of Rust verification toolchains, one that aims to verify Rust programs simply, and at scale.

Rust goes to great lengths to enforce static control of aliasing; the proof engineer should not waste any time on memory reasoning when so much already comes "for free"!

CCS Concepts: • **Theory of computation** → **Programming logic**; **Logic and verification**.

Additional Key Words and Phrases: Rust, verification, functional translation

1 INTRODUCTION

In 2006, exasperated by yet another crash of his building's elevator's firmware, and exhausted after walking up 21 flights of stairs, Graydon Hoare set out to design a new programming language [Hoare 2022]. The language, soon to be known as Rust, had two goals. First, to be system-oriented, meaning the programmer would deal with references, pointers, and manually manage memory. Second, to be safe, meaning the compiler's static discipline would rule out memory errors such as use-after-free, or arbitrary memory access. Even though the language evolved a great deal since its inception, these two core premises remain today.

Sixteen years later, Rust enjoys a substantial amount of success, and has ranked as the most loved programming language for six consecutive years on StackOverflow's developer survey [sta 2021]. But as the systems community can attest [Bhargavan et al. 2017; Ferraiuolo et al. 2017; Klein

Authors' addresses: Son Ho, Inria, France, son.ho@inria.fr; Jonathan Protzenko, Microsoft Research, USA, protz@microsoft.com.

et al. 2009; Lorch et al. 2020], memory safety is too weak of a property, no matter how remarkable of an achievement Rust is. Indeed, we oftentimes want to prove deep *correctness properties* of a system. Doing so may involve anything from baseline safety properties, such as the absence of assertion failures or runtime errors, to complex invariants involving concurrent systems.

As a consequence, several verification toolchains have emerged to facilitate proving deep properties about low-level programs. For pragmatic reasons, C is oftentimes a target of choice [Cao et al. 2018; Protzenko et al. 2017]; so is using a custom language, such as Dafny [Hawblitzel et al. 2014]. Alas, whether the tool is based on separation logic [Reynolds 2002] or modifies-clauses [Leino 2010], verification engineers soon find themselves drowning under a sea of memory-related obligations that distract them from the properties of interest. The net result is that verification engineers spend an undue amount of time discharging mundane memory-related proof obligations, because the frontend language does not enforce enough invariants to begin with.

One strategy is to write a checker for an existing language, restricting its usage enough that verification becomes easier. This is the strategy used by e.g., Linear Dafny [Li et al. 2022] or RefinedC [Sammler et al. 2021]. Another strategy is to leverage invariants provided for free by languages with restrictive type systems; that is, to leverage a language like Rust.

Rust is, in effect, trying to reconcile systems programming, and a long tradition of static ownership disciplines [Boyland et al. 2001; Clarke et al. 1998] which traces back to linear types [Wadler 1990], regions [Tofte and Talpin 1997] and the combination thereof [Fluet et al. 2006]. Today, Rust incorporates *ownership* in all aspects of its design; notably, programmers rely on *borrowing* to take references with ownership, and the type system relies on a notion of *lifetime* to enforce the soundness of borrows. This static discipline aims to give the programmer maximum flexibility for common idioms, while still preventing arbitrary aliasing.

The adequacy of Rust as a verification target has not gone un-noticed. There are now several research projects aiming to set up verification frameworks for Rust, using a variety of backends, such as SMT [ver 2022; Matsushita et al. 2020], Viper [Astrauskas et al. 2019; Wolff et al. 2021], or Why [Denis et al. 2021]. With these works, we have gained a deeper understanding of why verifying Rust programs remains difficult, even in the presence of Rust's strong ownership discipline.

First, Rust's type system is not *simply* linear, and features a rich variety of mechanisms, such as: reborrows, two-phase borrows, functions returning borrows, along with many other subtle rules that ensure most common idioms go through the type-checker smoothly. Accounting for all of these is notoriously difficult. Second, the borrow mechanism introduces non-locality, in that ending a borrow requires propagating knowledge backwards, in order to update the previously-borrowed variable with a new value. This central difficulty is handled via a variety of technical devices in other works, such as prophecy variables [Matsushita et al. 2020] or `after` clauses [ver 2022].

In this paper, we propose a new approach to understanding and verifying Rust programs. At the heart of our methodology is a lightweight functional translation of Rust programs. We eschew the complexity of connecting to a separation-logic based backend [Jung et al. 2017], or relying on prophecy variables to produce a logical encoding [Matsushita et al. 2022, 2020]. Instead, we synthesize a pure, functional, executable equivalent of the original Rust program, thus producing a lambda-term that does not rely on memory or special constructs. Our translation handles shared, mutable, two-phase and re-borrows, and thus accounts for a very large fraction of typical Rust programs. We call the conceptual framework, as well as the companion tool, Aeneas.

We wish to emphasize that our functional translation is completely generic. While we demonstrate a possible verification backend by printing our pure programs in F* syntax, many other options are possible. One could easily add additional backends (Coq, Lean, Viper, Why) or devise a contract language for source Rust programs that directly emits SMT proof obligations using our translation. We certainly hope to write some of these in the near future.

To elaborate on the design choices made by Aeneas: we intentionally focus on programs that abide by Rust's static ownership discipline. That is, we do not tackle `unsafe` blocks – we believe such programs are better suited to a sophisticated framework such as RustBelt. We do not tackle interior mutability either, in which the user can use unfettered aliasing, in exchange for a run-time borrow checker. We wish to focus instead on the Rust subset that is functional in essence, meaning we leave treatment of interior mutability up to future work. We believe this places us in a "sweet spot" for verifying Rust programs. The key observation of this work is that for the most part, references and borrows serve the purpose of optimizing either performance (e.g., passing by reference instead of by value), or memory representation (e.g., by controlling aliasing and taking inner pointers within data structures). That is, Rust's references do not serve any semantic purpose; coupled with the fact that the type system is enforcing a linear discipline, such programs are functional in essence, and can be naturally translated to a pure functional equivalent. Wadler observed that a linear type system allows compiling pure programs using imperative updates [Wadler 1990]; we leverage the reverse observation [Charguéraud and Pottier 2008], that is, imperative programs with a strong enough ownership discipline admit a functional equivalent.

To reiterate: the key point of this work is that we give a *functional* semantics, and thus a *pure* translation, to the subset of Rust we consider. Concretely, we define LLBC, the Low-Level Borrow Calculus, to model that subset. Then, we give it an operational semantics that is functional in nature. We do not rely on memory, addresses or pointer arithmetic; rather, we map variables to values, and track aliasing in a very fine-grained manner. We claim that our operational semantics captures the *essence* of borrowing; that is, it does not simply apply the rules dictated by Rust's lifetime discipline. Rather, it establishes what is allowed with regards to ownership in the presence of moves, borrows and copies. As such, our semantics can account not only for the current borrow-checker's behavior, but also for its future evolutions, such as Polonius [The Rust Compiler Team 2021].

Our functional semantics paves the way for our functional translation. We proceed in two steps. First, we tweak our semantics to abstract the aliasing graph in the presence of function calls; to do so, we interpret regions as bags of borrows and loans. Next, we follow the structure of the program in the presence of these region abstractions, and generate a functional translation. To overcome the key difficulty of terminating a borrow, we rely on a technical innovation called *backward functions*, which obviates the need for prophecy variables (as in RustHorn), or `after` clauses (as in Verus).

We have implemented our functional translation approach in a mixture of Rust and OCaml, for a total of 10,000 and 14,000 lines of code, respectively (excluding comments and whitespace). Once the pure translation is synthesized, we emit pure code for the theorem prover of our choice: we currently support F*, and we have a Coq backend in the works.

We evaluate Aeneas on a wide variety of micro-benchmarks for feature-completeness, and verify a resizable hash table as our main case study. We find that the benefits of Aeneas are many. First, with Aeneas, the verification engineer deals with the *intrinsic* difficulty of the proofs, rather than the *incidental* complexity; that is, they can focus on the essence of the proof rather than being mired in the technicalities of memory reasoning. There are no modifies-clause lemmas, and no separation logic framing tactics; these are, by construction, un-necessary. Second, our approach remains *lightweight*: we do not need to design an annotation language for Rust programs, and the properties we prove are not constrained by the expressivity or the usability of said annotation language. Consequently, Aeneas-translated programs can easily integrate within an existing project, and leverage libraries or proof tactics, as opposed to evolving in a closed world whose boundaries are set by a specific annotation language. Third, we are not beholden to one specific verification framework; we envision a world where the verification engineer can simply direct Aeneas towards the proof framework they are the most productive in.

In short, Aeneas reveals the functional essence of Rust programs, and verifies them as such.

We start with an accessible, example-based introduction to Rust and AENEAS (Section 2), then present this paper's contributions:

- an ownership-centric operational semantics for Rust (Section 3),
- a concept of *region abstraction*, which precisely models the interaction of borrows, regions and function calls; region abstractions establish a blueprint for our functional translation, which relies on what we dub *backward functions* (Section 4),
- a complete implementation, based on a Rust plugin and a subsequent OCaml compiler named AENEAS (Section 5); the former extracts all the information we need from the internal Rust AST, while the latter implements all of the steps described in this paper,
- an experimental evaluation that culminates in the verification of a resizable hash table; to the best of our knowledge, the man-hours spent proving that the Rust hash table functionally behaves like a map are extremely modest relative to other similar efforts: AENEAS thus offers substantial gains of productivity (Section 6).

We acknowledge the current limitations of our tool; situate AENEAS relative to other Rust verification efforts, and conclude (Section 7). The implementation of our tools, the verified code and a long version of this paper are available online [Ho and Protzenko 2022a,b,c].

2 AENEAS AND ITS FUNCTIONAL TRANSLATION, BY EXAMPLE

Before jumping into the various facets of our formalism, we keep an eye on the prize, and immediately showcase how AENEAS translates Rust programs to pure equivalents. In this section, and for the remainder of the paper, we use F* syntax for our functional translation; it greatly resembles OCaml and other ML languages, and as such should be familiar to the reader. A brief note about terminology: we adopt the view of Matsakis [2018], and refer to *regions*, emphasizing that a region encompasses a set of borrows and loans at a given program point. The Rust compiler and documentation, however, refer to *lifetimes*, which conveys the idea of a syntactic bracket, and a specific implementation technique to enforce soundness. In this paper, whenever we talk about Rust specifically, we use "lifetime"; whenever we emphasize our semantic view of ownership, we use "region".

Mutable Borrows, Functionally. To warm up, we consider an example that, albeit small, showcases many of Rust's features, including its ownership mechanism. In the Rust program below, `ref_incr` increments a reference, and `test_incr` acts as a representative caller of the function.

```
1  fn ref_incr(x: &mut i32) {
2      *x = *x + 1; }
3
4  fn test_incr() {
5      let mut y = 0i32;
6      ref_incr(&mut y);
7      assert!(y == 1); }
```

The `incr` function operates by reference; that is, it receives the address of a 32-bit signed integer `x`, as indicated by the `&` (reference) type. In addition, `incr` is allowed to modify the contents at address `x`, because the reference is of the `mut` (mutable) kind, which permits memory modification. Finally, the Rust type system enforces that mutable references have a unique owner: the definition of `ref_incr` type-checks, meaning that the function not only *guarantees* it does not duplicate ownership of `x`, but also can *rely* on the fact that no one else owns `x`.

In `test_incr`, we allocate a mutable value (`let mut`) on the stack; upon calling `ref_incr`, we take a mutable reference (`&mut`) to `y`. Statically, `y` becomes unavailable as long as `&mut y` is active. In Rust parlance, `y` is *mutably borrowed* and its ownership has been transfered to the mutable reference. To

type-check the call, the type-checker performs a lifetime analysis: the `ref_incr` function has type (`&'a mut i32) -> ()`, and the `&mut y` borrow has type `&'b mut i32`; both `'a` and `'b` are lifetime variables.

For now, suffices to say that the type-checker ascertains that the lifetime `'b` of the mutable borrow satisfies the lifetime annotation `'a` in the type of the callee, and deems the call valid. Immediately after the call, Rust *terminates* the region `'b`, in effect *relinquishing* ownership of the mutable reference `&'b mut y` so as to make `y` usable again inside `test_incr`. This in turn allows the `assert` to type-check, and thus the whole program. Undoubtedly, this is a very minimalistic program; yet, there are two properties of interest that we may want to establish already. The obvious one: the assertion always succeeds. More subtly, doing so requires us to prove an additional property, namely that the addition at line 2 does not overflow.

The key insight of Aeneas is that even though the program manipulates references and borrows, none of this is informative when it comes to reasoning about the program. More precisely: `x` and `y` are uniquely owned, meaning that there are no stray aliases through which `x` or `y` may be modified; in other words, to understand what happens to `y`, it suffices to track what happens to `&mut y`, and therefore to `x`. Feeding this program to Aeneas generates the following translation, where `i32_add` is an Aeneas primitive that captures the semantics of error-on-overflow in Rust.

```
1  let ref_incr_fwd (x : i32): result i32 =
2    match i32_add x 1 with | Fail -> Fail | Return x0 -> Return x0
3
4  let test_incr : result unit =
5    match ref_incr_fwd 0 with
6    | Fail -> Fail
7    | Return y -> if not (y = 1) then Fail else Return ()
8
9  let _ = assert (test_incr = Return ())
```

This program is semantically equivalent to the original Rust code, but does not rely on the memory: we have leveraged the precise ownership discipline of Rust to generate a functional, pure version of the program. In hindsight, the usage of references in Rust was merely an implementation detail, which is why `ref_incr_fwd` becomes a simple (possibly-overflowing) addition. Should the call to `ref_incr_fwd` (line 5) succeed, its result is bound to `y` (line 7); the `assert` simply becomes a boolean test that may generate a failure in the error monad.

For the purposes of unit-testing, Aeneas inserts an additional assertion for `test_*` functions of type unit → unit: the prover shows instantly that our test always succeeds. (In F*, we execute this assertion directly on the normalizer, without even resorting to SMT; Aeneas produces an executable translation, not a logical encoding.) In the remainder of this section, we use `<--`, F*'s bind operator[1] in the error monad.

Returning a Mutable Borrow, and a Backward Function. Rust programs, however, rarely admit such immediate translations. To see why, consider the following example, where the `choose` function returns a borrow, as indicated by its return type `&'a mut`.

```
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); }
```

---

[1]There are issues in F* related to this notation [fst 2017], which we work around in practice, but ignore here.

The `choose` function is polymorphic over type `T` and lifetime `'a`; the lifetime annotation captures the expectation that both `x` and `y` be in the same region. At call site, `x` and `y` are borrowed (line 6): they become unusable, and give birth to two intermediary values `&mut x` and `&mut y` of type `&'a mut i32`. The value returned by `choose` also lives in region `'a`, i.e., `z` also has type `&'a mut i32`. The usage of `z` (lines 7-8) is valid because the region `'a` still exists; the Rust type-checker infers that region `'a` ought to be terminated after line 8, which ends the borrows and therefore allows the caller to regain full ownership of `x` and `y`, so that the `assert`s at line 9 are well-formed.

At first glance, it appears we can translate `choose` to an obvious conditional. But if we reason about the semantics of `choose` from the caller's perspective, it turns out that the intuitive translation is not sufficient to capture what happens, e.g., to `x` and `y` at lines 9. At call site, `choose` is an opaque, separate function, meaning the caller cannot reason about its precise definition – all that is available is the function type. This type, however, contains precise region information. When performing the function call, the ownership of `x` and `y` is transferred to region `'a` in exchange for `z`; symmetrically, when the lifetime `'a` terminates, `z` is relinquished to region `'a` in exchange for regaining ownership of `x` and `y`. The former operation flows *forward*; the latter flows *backward*. Using a separation-logic oriented analogy: borrows and regions encode a magic wand that is introduced in a function call and eliminated when the corresponding region terminates.

Our point is: both function call and region termination are semantically meaningful. In our earlier example, the `ref_incr` function returned a unit, meaning that AENEAS only emitted a *forward* function (hence the `_fwd` suffix) to translate the function call. With the `choose` example, AENEAS emits both a *forward* and a *backward* function, used for the function call and the end of the `'a` region, respectively.

```
1   let choose_fwd (t : Type) (b : bool) (x : t) (y : t) : result t =
2     if b then Return x else Return y
3
4   let choose_back (t : Type) (b : bool) (x : t) (y : t) (ret : t) : result (t & t) =
5     if b then Return (ret, y) else Return (x, ret)
6
7   let test_choose_fwd : result unit =
8     i <-- choose_fwd i32 true 0 0;
9     z <-- i32_add i 1;
10    massert (z = 1); (* monadic assert *)
11    (x0, y0) <-- choose_back i32 true 0 0 z;
12    massert (x0 = 1);
13    massert (y0 = 0);
14    Return ()
15
16  let _ = assert (test_choose_fwd = Return ())
```

The call to `choose` becomes a call to the forward function `choose_fwd` (line 8); we bind the result of the addition (provided no overflow occurs) to `z` (line 9); then, per the rules of Rust's type-checker, region `'a` terminates which compels us to call the backward function `choose_back`. The intuitive effect of calling `choose_back` is as follows: we relinquish `z`, which was in region `'a`; doing so, we propagate any updates that may have been performed through `z` onto the variables whose ownership was transferred to `'a` in the first place, namely `x` and `y`. This bidirectional approach is akin to lenses [Bohannon et al. 2008], except we propagate the output back to possibly-many inputs; in this case, `z` is a view over either `x` or `y`, and the backward function reflects the update to `z` onto the original variables. Thus, both variables are re-bound (line 11), before chaining the two asserts (lines 12 and 13).

From the caller's perspective, the computational content of `choose` is unknown; but the signature of `choose` reveals the effect it may have onto its inputs `x` and `y`, which in turns allows us to derive

the type of the backward and forward functions from the signature of `choose` itself. The result is a modular, functional translation that does not rely on any sort of cross-function inlining or whole-program analysis. To synthesize `choose_back`, it suffices to invert the direction of assignments; in one case, `z` flows to `x` and `y` remains unchanged; the other case is symmetrical.

Recursion and Data Structures. It might not be immediately obvious that this translation technique scales up beyond toy examples; to conclude this section, we crank up the complexity and show how Aeneas can handle a wide variety of idioms while still delivering on the original promise of a lightweight functional translation. Our final example is `list_nth`, which allows taking a mutable reference to the *n*-th element of a list, mutating it, and regaining ownership of the list.

```
1   enum List<T> { Cons(T, Box<List<T>>), Nil }
2
3   fn list_nth_mut<'a, T>(l: &'a mut List<T>, i: u32) -> &'a mut T {
4       match l {
5           Nil => { panic!() }
6           Cons(x, tl) => { if i == 0 { x } else { list_nth_mut(tl, i - 1) } } } }
7
8   fn sum(l: & List<i32>) -> i32 {
9       match l {
10          Nil => { return 0; }
11          Cons (x, tl) => { return *x + sum(tl); } } }
12
13  fn test_nth() {
14      let mut l = Cons (1, Box::new(Cons (2, Box::new(Cons (3, Box::new(Nil))))));
15      let x = list_nth_mut(&mut l, 2);
16      *x = *x + 1;
17      assert!(sum(&l) == 7); }
```

This example relies on several new concepts. Parametric data type declarations (line 1) resemble those in any functional programming language such as OCaml or SML. The `Box` type denotes a heap-allocated, uniquely-owned piece of data. Without the `Box` indirection, `List` would describe a type of infinite size and would be rejected. Immutable borrows (line 8) do not sport a `mut` keyword; they do not permit mutation, but the programmer may create infinitely many of them. Only when all shared borrows have been relinquished does full ownership return to the borrowed value. The complete translation is as follows:

```
1   type list_t (t : Type) =
2   | ListCons : t -> list_t t -> list_t t
3   | ListNil : list_t t
4
5   let rec list_nth_mut_fwd (t : Type) (l : list_t t) (i : u32) : result t =
6     begin match l with
7     | ListCons x tl ->
8       begin match i with
9       | 0 -> Return x
10      | _ ->
11        i0 <-- u32_sub i 1;
12        list_nth_mut_fwd t tl i0
13      end
14    | ListNil -> Fail
15    end
16
17  let rec list_nth_mut_back (t : Type) (l : list_t t) (i : u32) (ret : t) : result (list_t t) =
18    begin match l with
19    | ListCons x tl ->
20      begin match i with
```

```
21      | 0 -> Return (ListCons ret tl)
22      | _ ->
23        i0 <-- u32_sub i 1;
24        l0 <-- list_nth_mut_back t tl i0 ret;
25        Return (ListCons x l0)
26      end
27    | ListNil -> Fail
28    end
29
30  let rec sum_fwd (l : list_t i32) : result i32 =
31    begin match l with
32    | ListCons x tl ->
33      i <-- sum_fwd tl;
34      i32_add x i
35      end
36    | ListNil -> Return 0
37    end
38
39  let test_nth_fwd : result unit =
40    let l2 = ListCons (1, ListCons (2, ListConst (3, ListNil))) in
41    i <-- list_nth_mut_fwd i32 l2 2;
42    x <-- i32_add i 1;
43    l2 <-- list_nth_mut_back i32 l2 2 x;
44    i0 <-- sum_fwd l2;
45    massert (not (i0 = 7))
46
47  let _ = assert (test_nth_fwd = Return ())
```

We first focus on the caller's point of view. Continuing with the lens analogy, we focus on (or "get") the *n*-th element of the list via a call to `list_nth_mut_fwd` (line 41); modify the element (line 42); then close (or "put" back) the lens, and propagate the modification back to the list via a call to `list_nth_mut_back` (line 43). The `list_nth_mut_back` function is of particular interest. The function follows the same control-flow as the forward function. However, the key part happens when returning from the `Cons` case: our functional translation performs a *semantic* region analysis, from which it follows that a recursive call to the backward function is needed, along with the construction of a new `Cons` cell.

3 AN OWNERSHIP-CENTRIC SEMANTICS FOR RUST

Before explaining the functional translation above, we must first define our input language and its operational semantics. We now present a series of short Rust snippets, and show in comments how our execution environments model the effect of each statement.

Mutable borrows. After line 1, x points to 0, which we write $x \mapsto 0$. At line 2, `px` *mutably* borrows `x`. As we mentioned earlier, a mutable borrow grants exclusive ownership of a value, and renders the borrowed value unusable for the duration of the borrow. We reflect this fact in our execution environment as follows: `x` is marked as "loaned-out", in a <u>m</u>utable fashion, and `px` is known to be a <u>m</u>utable borrow. Furthermore, ownership of the borrowed value now rests with `px`, so the value within the mutable borrow is 0. Finally, we need to record that `px` is a borrow *of* `x`: we issue a fresh loan identifier ℓ that ties `x` and `px` together. The same operation is repeated at line 3. Value 0 is now held by `ppx`, and `px`, too, becomes "loaned out".

```
1  let mut x = 0;       // x ↦ 0
2  let mut px = &mut x; // x ↦ loan^m ℓ,  px ↦ borrow^m ℓ 0
3  let ppx = &mut px;   // x ↦ loan^m ℓ,  px ↦ loan^m ℓ',    ppx ↦ borrow^m ℓ' (borrow^m ℓ 0)
```

Our environments thus precisely track ownership; doing so, they capture the aliasing graph in an exact fashion. Another point about our style: this representation allows us to adopt a *focused* view of the borrowed value (e.g. 0), solely through its owner (e.g. **ppx**), without worrying about following indirections to other variables. We believe this approach is unique to our semantics; it has, in our experience, greatly simplified our reasoning and in particular the functional translation (Section 4).

We remark that our style departs from Stacked Borrows [Jung et al. 2019], where the modified value remains with x. We also note that our formalism cannot account for unsafe blocks; allowing unfettered aliasing would lead to potential cycles, which we cannot represent. This is an intentional design choice for us: we circumscribe the problem space in order to achieve an intuitive, natural semantics and a lightweight functional translation. Aeneas shines on non-unsafe Rust programs, and can be complemented by more advanced tools such as RustBelt for unsafe parts.

Shared borrows. Shared borrows behave more like traditional pointers. Multiple shared borrows may be created for the same value; each of them grants read-only access to the underlying value. Regaining full ownership of the borrowed value requires terminating all of the borrows. In the example below, the value $(0, 1)$ is borrowed in a shared fashion, at line 2. This time, the value remains with x; but taking an immutable reference to x still requires book-keeping. We issue a new loan ℓ, and record that **px1** is now a shared borrow associated to loan ℓ; to understand which value **px1** points to, we simply look up in the environment who is the owner of ℓ, and read the associated value. Repeated shared borrows are permitted: at line 3, we issue a new loan ℓ' which augments the loan-set of x. At line 4, we anticipate on our internal syntax, where moves and copies are explicit, and copy the first component of x. Values that are loaned immutably, like x, can still be read; in the resulting environment, y points to a copy of the first component, and bears no relationship whatsoever to x. Finally, at line 5, we *reborrow* (through **px1**) the first component of the pair only. First, to dereference **px1**, we perform a lookup and find that x owns ℓ. Then, we perform book-keeping and update the value loaned by x, so as to reflect that its first component has been loaned out.

```
1  let x = (0, 1);     // x ↦ (0, 1)
2  let px1 = &x;       // x ↦ loanˢ {ℓ} (0, 1),          px1 ↦ borrowˢ ℓ
3  let px2 = &x;       // x ↦ loanˢ {ℓ ∪ ℓ'} (0, 1),    px1 ↦ borrowˢ ℓ,   px2 ↦ borrowˢ ℓ'
4  let y = copy x.0;   // x ↦ loanˢ {ℓ ∪ ℓ'} (0, 1),    px1 ↦ borrowˢ ℓ,   px2 ↦ borrowˢ ℓ',   y ↦ 0
5  let z = &(*px1.0);  // x ↦ loanˢ {ℓ ∪ ℓ'} ((loanˢ {ℓ''} 0), 1), px1 ↦ ..., px2 ↦ ..., y ↦ 0, z ↦ borrowˢ ℓ''
```

In our presentation, shared borrows behave like pointers, and every one of them is statically accounted for via the set of loans attached to the borrowed value. The reader might find this design choice surprising: indeed, in Rust, shared borrows behave like immutable values and we ought to be able to treat them as such. Recall, however, that one of our key design goals is to give a *semantic* explanation of borrows; as such, our precise tracking of shared borrows allows us to know precisely when all aliases have been relinquished, and full ownership of the borrowed value has been regained. This allows us to justify why, in the example below, the update to x is sound; without our exact alias tracking, we would have to trust the borrow checker, something we explicitly do not want to do.

```
1  let mut x = 0;  // x ↦ 0
2  let px1 = &x;   // x ↦ loanˢ {ℓ} 0,        px1 ↦ borrowˢ ℓ
3  let px2 = &x;   // x ↦ loanˢ {ℓ ∪ ℓ'} 0,  px1 ↦ borrowˢ ℓ,   px2 ↦ borrowˢ ℓ'
4                  // x ↦ 0,                  px1 ↦ ⊥,   px2 ↦ ⊥
5  x = 1;          // x ↦ 1
```

Finally, we reiterate our remark that our formalism allows keeping track of the aliasing graph in a precise fashion; the discipline of Rust bans cycles, meaning that the aliasing graph is always a tree.

This style of representation resembles Mezzo [Balabonski et al. 2016], where loan identifiers are akin to singleton types, and entries in the environment are akin to permissions.

Rewriting an Old Value, a.k.a. Reborrowing. We now consider a particularly twisted example accepted by the Rust compiler. While the complexity seems at first gratuitous, it turns out that the pattern of borrowing a dereference (i.e., `&mut (*px)`) is particularly common in Rust. The reason is subtle: in the post-desugaring MIR internal Rust representation, moves and copies are explicit, meaning function calls of the form `f(move px)` abound. Such function calls *consume* their argument, and render the user-declared reference `px` unusable past the function call. To offer a better user experience, Rust automatically "reborrows" the contents pointed to by `px`, and rewrites the call into `f(move (&mut (*px)))` at desugaring-time. Thus, only the intermediary value is "lost" to the function call; relying on its lifetime analysis, the Rust compiler concludes that the user-declared reference `px` remains valid past the function call, hence making the programmer's life easier.

Another common pattern is to directly mutate a borrow, i.e. assign a fresh borrow into a variable `x` of type `&mut t` that was *itself* declared as `let mut`. Capturing the semantics of such an update must be done with great care, in order to preserve precise aliasing information.

We propose an example that combines both patterns; the fact that we make `px` reborrow itself is what makes the example "twisted". Rust accepts this program; we now explain with our semantics *why* it is sound. In the example below, after line 2, the environment offers no surprises. Justifying the write at line 3 requires care. We borrow $*px$, which modifies px to point to $\mathrm{borrow}^m \ \ell' \ 0$, and returns $\mathrm{loan}^m \ \ell'$; the value about to be overwritten is stored in a fresh variable px_{old}, and $\mathrm{loan}^m \ \ell'$ gets written to px.

```
1   let mut x = 0;        // x ↦ 0
2   let mut px = &mut x;  // x ↦ loanᵐ ℓ,   px ↦ borrowᵐ ℓ 0
3   px = &mut (*px);      // x ↦ loanᵐ ℓ,   px_old ↦ borrowᵐ ℓ (loanᵐ ℓ'),   px ↦ borrowᵐ ℓ' 0
4                         // x ↦ loanᵐ ℓ,   px_old ↦ borrowᵐ ℓ 0,            px ↦ ⊥
5   assert!(x==0);        // x ↦ 0,         px_old ↦ ⊥,                       px ↦ ⊥
```

Saving the old value is crucial for line 4. For the assertion, we need to regain full ownership of x. To do so, we first terminate ℓ'. This *reorganizes* the environment, with two consequences. First, px becomes unusable, which we write $px \mapsto \bot$. Second, px_{old}, which we had judiciously kept in the environment, becomes $\mathrm{borrow}^m \ \ell \ 0$. We reorganize the environment again, to terminate ℓ; the effect is similar, and results in $x \mapsto 0$, i.e. full ownership of x. This example illustrates a key characteristic of our approach, which is that we reorganize borrows in a lazy fashion, and don't terminate a borrow until we need to get the borrowed value back.

An Illegal Borrow. We offer an final example which leverages our reorganization rules; further-more, the example illustrates how our semantics reaches the same conclusion as `rustc`, though by different means, on a borrowing error. At line 2, a borrow is introduced, which results in a fresh loan ℓ. To make progress, we terminate borrow ℓ at line 3. Line 4 then type-checks, with a fresh borrow ℓ'. Then, we error out at line 5: px_1 has been terminated, and we cannot dereference $\bot$.

The Rust compiler proceeds differently, and implements an analysis which requires computing borrow constraints for an entire function body. The compiler notices that lifetime ℓ must go on until line 6 (because of the assert), which prevents a new borrow from being issued at line 4. Rust thus ascribes the error to an earlier location than we do, that is, line 4. We remark that the Rust behavior is semantically equivalent to ours; however, our lazy approach which terminates borrows only as needed has the advantage that evaluation can proceed in a purely forward fashion, without requiring a non-local analysis. This on-demand approach is similar to Stacked Borrows.

```
1   let mut x = 0;     // x ↦ 0
2   let px1 = &mut x;  // x ↦ loanᵐ ℓ,   px₁ ↦ borrowᵐ ℓ 0
```

```
3                        // x ↦ 0,   px₁ ↦ ⊥
4    let px2 = &mut x; // aeneas: x ↦ loanᵐ ℓ′,   px₁ ↦ ⊥,   px₂ ↦ borrowᵐ ℓ′ 0
5                        // rustc: error: cannot borrow `px1` as mutable more than once at a time
6    assert!(*px1 == 0); // aeneas: error, attempt to deference unusable variable px1
```

3.1 The Low-Level Borrow Calculus

We now formally introduce and define our semantics of Rust programs. We start with the Low-Level Borrow Calculus ("LLBC", Figure 1), our source language. LLBC is in large part inspired by MIR, Rust's post-desugaring internal representation, notably: all local variables $\vec{x}_{\text{local}}$ are bound at the beginning of the function declaration; returning a value from a function amounts to writing into the special variable x_{ret}, followed by return; all subexpressions have been named so as to fit within MIR's statement/r-value/operand categories; and all variables within expressions have been desugared to either a move, a copy or a borrow. However, and unlike MIR, LLBC retains some high-level constructs: control-flow remains structured (LLBC statements are thus the fusion of MIR's statements and terminators); and case analysis on data types is exposed via a limited form of (complete) pattern matching, as opposed to a low-level integer switch on the tag. We remark that data types may match on a *path* only; this merely imposes that the scrutinee be let-bound before examining it, something that MIR does internally. Finally, we use pure expressions to allocate data types, rather than the progressive (mutable) initialization pattern used by MIR; and we see structures as data types equipped with a single constructor, for conciseness. Staying close to MIR is a design choice in line with other Rust-related works [Jung et al. 2019]; it allows for fewer, simpler rules, and a more precise description of what happens from the point of view of ownership.

At the heart of LLBC is a notion of *place*, i.e. the combination of a base variable (e.g. x) and a series of field offsets and indirections known as a *path* (e.g. *_.f). A place is akin to the notion of "lvalue" in, e.g., C. Assigning or returning from a function can only be done into a *place*. The grammar of rvalues and operands (Rust's limited form of expression) is very explicit, in that every use of a variable is performed through a copy, a move, or a borrow of a place.

3.2 A Structured Memory Model

Rust marries high-level concepts, such as ownership, a strong notion of value, and data types, with low-level concepts such as moves and copies, paths through a base address, and modifications at depth throughout the store. We propose a semantics that operates exclusively in terms of values (that is, no memory addresses), yet still permits fine-grained memory mutations as allowed by Rust.

Figure 2 presents our environments, which we write Ω, and our values, which we write v. We do not distinguish between store and environment, and use the two terms interchangeably. The store maps variable names x to values v: we have no notion of arbitrary memory addresses, or pointer arithmetic. Our values v are carefully crafted to model the semantics of borrows and ownership tracking in Rust; several of them already appeared in our earlier examples.

The combination of places, environments and values allows us to define reads and writes already. Reads and writes are defined in terms of our *structured* memory model: we do not have any notion of memory address, but *do* have a notion of path combined with a base "address" (variable) x, that is, a place; this permits reads and writes, at depth, through references. We present selected rules for reading (R-*) and writing (W-*) in Figure 3.

For *reading*, we write $\Omega(p) \Rightarrow v$, meaning reading from Ω at place p produces v. Doing so requires looking up the "base pointer" (variable) x found in p (READ), then deferring to an auxiliary judgment of the form $\Omega \vdash P(v_x) \Rightarrow v$, meaning following path P into v_x produces value v. We can follow path P as long as the value v_x is of the right shape (R-MUT-BORROW, R-FIELD). Reading from a mutable borrow requires no additional operation, since the mutable borrow uniquely owns the

τ	::=	type
	bool | uint32 | int32 | . . .	base types
	$\&^{\rho}\mathsf{mut}\ \tau$	mutable borrow
	$\&^{\rho}\ \tau$	immutable (shared) borrow
	$T\ \vec{\tau}$	type application
	$\alpha, \beta, \ldots$	type variables
	$(\vec{\tau})$	tuple $(\mathrm{len}(\vec{\tau}) > 1)$ or unit $(\mathrm{len}(\vec{\tau}) = 0)$
T	::=	type constructorapplication
	t	user-defined data type
	Box	boxed type
s	::=	statement
	$\emptyset$	empty statement (nil)
	$s; s$	sequence (cons)
	$p := rv$	assignment
	$p := f\ \vec{op}$	function call
	if op then s else s	conditional
	match p with $\overrightarrow{C \to s}$	data typecase analysis
	return	function exit
	panic	unrecoverable error
	. . .	loops, etc.
rv	::=	assignable "r" values
	op	operand
	$\&\mathsf{mut}\ p$	mutable borrow
	$\&\ p$	immutable (shared) borrow
	$!op \mid op + op \mid op - op \mid \ldots$	operators
op	::=	operand
	move p	ownership transfer
	copy p	scalar copy
	true | false | $n_{\mathsf{i}32}$ | $n_{\mathsf{u}32}$ | . . .	constants
	$C[\vec{f} = \vec{op}]$	data type constructor
	$(\vec{op})$	tuple $(\mathrm{len}(\vec{op}) > 1)$or unit $(\mathrm{len}(\vec{op}) = 0)$
x		variable
p	::= $P[x]$	place
P	::=	path
	$[.]$	base case
	$*^{s}P$	deref shared borrow
	$*^{m}P$	deref mutable borrow
	$*^{b}P$	deref box
	$P.f$	field selection
	$P.n$	field selection (tuple)
D	::=	top-level declaration
	fn $f\ \langle\vec{\rho}\rangle\ (\vec{x}_{\mathrm{arg}} : \vec{\tau})\ (\vec{x}_{\mathrm{local}} : \vec{\tau})\ (x_{\mathrm{ret}} : \tau) = s$	function declaration
	type $t\ \vec{\alpha} = C[\vec{f} : \vec{\tau}] \mid \ldots$	data type declaration

Fig. 1. The Low-Level Borrow Calculus: Syntax

$$
\begin{array}{lll}
v & ::= & & \text{value} \\
& & \text{true} \mid \text{false} \mid n_{\text{i32}} \mid n_{\text{u32}} \mid \dots & \text{booleans and integers} \\
& & \text{borrow}^m \, \ell \, v & \text{mutable borrow} \\
& & \text{borrow}^s \, \ell & \text{shared borrow} \\
& & \text{borrow}^r \, \ell & \text{reserved borrow} \\
& & \text{loan}^m \, \ell & \text{mutable loan} \\
& & \text{loan}^s \, \{\vec{\ell}\} \, v & \text{shared loan} \\
& & \bot & \text{inaccessible value} \\
& & C \, [\overrightarrow{f = \vec{v}}] & \text{data type constructor} \\
& & (\vec{v}) & \text{tuple constructor } (\text{len}(\vec{v}) > 1) \text{ or unit } (\text{len}(\vec{v}) = 0) \\
r & ::= & & \text{result} \\
& & \text{panic} & \text{unrecoverable error state} \\
& & \text{return} \, v & \text{successful, possibly-early exit} \\
& & \dots & \text{loop states, etc.} \\
\ell & & & \text{loan identifier} \\
\Omega & ::= & & \text{evaluation context} \\
& & \emptyset & \text{empty} \\
& & x \mapsto v, \Omega & \text{new mapping}
\end{array}
$$

Fig. 2. The Low-Level Borrow Calculus: Reduction Environments, Values

READ
$$\frac{p = P[x] \qquad x \mapsto v_x \in \Omega \qquad \Omega \vdash P(v_x) \Rightarrow v}{\Omega(p) \Rightarrow v}$$

R-BOX
$$\frac{\Omega \vdash P(v_P) \Rightarrow v}{\Omega \vdash (*^b P)(\text{Box} \, v_P) \Rightarrow v}$$

R-MUT-BORROW
$$\frac{\Omega \vdash P(v_P) \Rightarrow v}{\Omega \vdash (*^m P)(\text{borrow}^m \, \ell \, v_P) \Rightarrow v}$$

R-SHARED-BORROW
$$\frac{\text{loan}^s \, \{\ell \cup \_\} \, v_P \in \Omega \qquad \Omega \vdash P(v_P) \Rightarrow v}{\Omega \vdash (*^s P)(\text{borrow}^s \, \ell) \Rightarrow v}$$

R-FIELD
$$\frac{\Omega \vdash P(v_P) \Rightarrow v}{\Omega \vdash (P.f)(C \, [\overrightarrow{f = v_P}]) \Rightarrow v}$$

R-BASE
$$\frac{}{\Omega \vdash [.](v) \Rightarrow v}$$

R-SHARED-LOAN
$$\frac{P \neq [.] \qquad \Omega \vdash P(v_P) \Rightarrow v}{\Omega \vdash P(\text{loan}^s \, \{\_\} \, v_P) \Rightarrow v}$$

WRITE
$$\frac{\begin{array}{cc} x \mapsto v_x \in \Omega & p = P[x] \\ \Omega \vdash P(v_x) \leftarrow v \Rightarrow v'_x & \Omega' = \Omega[x \mapsto v'_x] \end{array}}{\Omega(p) \leftarrow v \Rightarrow \Omega'}$$

W-BOX
$$\frac{\Omega \vdash P(v_P) \leftarrow v \Rightarrow v'_P}{\Omega \vdash (*^b P)(\text{Box} \, v_P) \leftarrow v \Rightarrow \text{Box} \, v'_P}$$

W-MUT-BORROW
$$\frac{\Omega \vdash P(v_P) \leftarrow v \Rightarrow v'_P}{\Omega \vdash (*^m P)(\text{borrow}^m \, \ell \, v_P) \leftarrow v \Rightarrow \text{borrow}^m \, \ell \, v'_P}$$

W-BASE
$$\frac{}{\Omega \vdash [.](v_P) \leftarrow v \Rightarrow v}$$

W-FIELD
$$\frac{f = f_i \qquad \Omega \vdash P(v_{P,i}) \leftarrow v \Rightarrow v'_{P,i} \qquad v'_{P,j} = v_{P,j} \text{ for } j \neq i}{\Omega \vdash (P.f)(C \, [\overrightarrow{f = v_P}]) \leftarrow v \Rightarrow C \, [\overrightarrow{f = v'_P}]}$$

Fig. 3. Reading From and Writing To Our Structured Memory Model

value it points to (R-Mut-Borrow). Conversely, reading from a shared borrow requires looking up the owner of the loan to find the value being pointed to (R-Shared-Borrow).

Rule R-Base is our base case: if we have reached the end of the path P, we simply return the value found there. One subtlety occurs in the case of shared *loans*. Rule R-Shared-Loan permits reading from a value that is currently immutably borrowed, which is allowed in Rust. However, we only do so if necessary; that is, if we must follow further indirections in P (i.e., $P \neq [.]$). If there are no further indirections (i.e., $P = [.]$), R-Base kicks in and returns a value of the form loan^s. This is intentional, and will prove useful for E-Shared-Or-Reserved-Borrow, as we shall see shortly.

For *writing*, we write $\Omega(p) \leftarrow v \Rightarrow \Omega'$, meaning assigning value v into Ω at place p produces an updated environment Ω' (Write). As before, we follow the structure of v_x (Write), and defer to an auxiliary judgment of the form $\Omega \vdash P(v_x) \leftarrow v \Rightarrow v'_x$, which from v_x computes an updated value v'_x where only the subexpression selected by P is updated with v. We update x's entry in the environment to map to the new value v'_x, denoted $\Omega[x \mapsto v'_x]$ (Write). As before, the shape of P determines which rule applies: we may only write through a mutable borrow (W-Mut-Borrow) or a box (W-Box). We eventually apply W-Base. We elide the remaining rules (tuples, fields, etc.).

3.3 Semantics of Ownership and Borrows

At the heart of our operational semantics is our treatment of borrows, which captures ownership transfer. We now introduce the core of our operational rules in Figure 4, and describe in detail the essential operations: borrows, moves, copies, and assignments. Our rules start with E-, for evaluation rules. Both the rules and our earlier syntax manipulate a third flavor of borrows, which we dub "reserved" borrows, denoted as borrow^r. A technical device, they account for the two-phase borrows introduced by the Rust compiler in the process of desugaring to MIR. We explain those later, in Section 3.4; they can be safely ignored for now.

A few preliminary remarks about notation: we rely on the auxiliary notion of a *ghost update*, denoted $\Omega[p \mapsto v]$ – this extends our earlier notation of $\Omega[x \mapsto v]$ – for updating an entry in the environment. In contrast to run-time writes, previously introduced as $\Omega(p) \leftarrow v \Rightarrow \Omega'$ (Write), ghost updates do not have any effect at run-time (Figure 8). Instead, they allow us to perform the necessary book-keeping to statically keep track of ownership and aliases. The definition of ghost updates is almost identical to run-time writes (Write), the only difference being that ghost updates can follow a shared borrow to perform an administrative update underneath the shared loan. As we will see shortly, this is leveraged by e.g. E-Shared-Or-Reserved-Borrow, to track borrowing underneath a shared borrow. We write $\Omega \vdash op \rightsquigarrow v \dashv \Omega'$ to indicate that in environment Ω, operand op reduces to v and produces updated environment Ω'. The $\rightsquigarrow$ judgment is overloaded for other syntactic categories. Finally, we write e.g. loan $\notin v$, with no arguments, to indicate that no kind of loan should appear in v; we have similar syntactic conventions for other restrictions.

For mutable borrows (E-Mut-Borrow), we disallow: borrowing already-borrowed values (no loan); borrowing moved, uninitialized values (no $\bot$) or reserved borrows; and borrowing through a shared borrow (no $*^s$ in p). This latter requirement refers to the place p, not the value v found at p: a value reachable via a shared borrow is, inevitably, shared, and therefore cannot be uniquely owned by means of a mutable borrow. If these premises are satisfied, we perform a ghost update of the environment, and mark p as loaned with identifier ℓ. The borrow evaluates to borrow$^m \ell v$, which embodies unique ownership of value v thanks to the exclusive loan ℓ.

For immutable borrows (E-Shared-Or-Reserved-Borrow), we disallow moved or uninitialized values (no $\bot$) and reserved borrows, but rule out mutable loans only: it is always legal in Rust to create another shared borrow from a value that has already been shared. The borrow evaluates to borrow$^s \ell$, a borrow without value ownership. We need to record the fact that a fresh loan has been handed out; we perform a ghost update on the environment, to either augment the loan-set of the

E-Mut-Borrow
$$\frac{\Omega(p) \Rightarrow v \qquad \{\bot, \mathsf{loan}, \mathsf{borrow}^r\} \notin v}{*^s \notin p \qquad \ell \text{ fresh} \qquad \Omega(p) \leftarrow \mathsf{loan}^m \ell \Rightarrow \Omega'}{\Omega \vdash \&\mathsf{mut}\, p \rightsquigarrow \mathsf{borrow}^m \ell\, v \dashv \Omega'}$$

E-Shared-Or-Reserved-Borrow
$$\Omega(p) \Rightarrow v \qquad \{\bot, \mathsf{loan}^m, \mathsf{borrow}^r\} \notin v$$
$$\ell \text{ fresh} \qquad \Omega' = \Omega[p \mapsto v']$$
$$v' = \begin{cases} \mathsf{loan}^s\, \{\vec{\ell} \cup \ell\}\, v'' & \text{if } v = \mathsf{loan}^s\, \{\vec{\ell}\}\, v'' \\ \mathsf{loan}^s\, \{\ell\}\, v & \text{otherwise} \end{cases}$$
$$\frac{}{\Omega \vdash \&\, p \rightsquigarrow \mathsf{borrow}^{r,s} \ell \dashv \Omega'}$$

E-Move
$$\Omega(p) \Rightarrow v \qquad \{\bot, \mathsf{loan}, \mathsf{borrow}^r\} \notin v$$
$$\frac{\{*^m, *^s\} \notin p \qquad \Omega(p) \leftarrow \bot \Rightarrow \Omega'}{\Omega \vdash \mathsf{move}\, p \rightsquigarrow v \dashv \Omega'}$$

E-Copy
$$\Omega(p) \Rightarrow v \qquad \{\bot, \mathsf{loan}^m, \mathsf{borrow}^{m,r}\} \notin v$$
$$\frac{\Omega \vdash \mathsf{copy}\, v \Rightarrow v' \dashv \Omega'}{\Omega \vdash \mathsf{copy}\, p \rightsquigarrow v' \dashv \Omega'}$$

E-Constructor
$$\frac{\Omega_i \vdash op_i \rightsquigarrow v_i \dashv \Omega_{i+1}}{\Omega_0 \vdash C[\overrightarrow{f = op}] \rightsquigarrow C[\overrightarrow{f = v}] \dashv \Omega_n}$$

E-Return
$$\Omega_i \vdash x_{\mathsf{local},i} := \bot \rightsquigarrow () \dashv \Omega_{i+1}$$
$$\frac{\Omega_n(x_{\mathsf{ret}}) \Rightarrow v \qquad \{\bot, \mathsf{loan}, \mathsf{borrow}^r\} \notin v}{\Omega_0 \vdash \mathsf{return} \rightsquigarrow \mathsf{return}\, v \dashv \Omega_n}$$

E-IfThenElse-T
$$\Omega \vdash op \rightsquigarrow \mathsf{true} \dashv \Omega'$$
$$\frac{\Omega' \vdash s_1 \rightsquigarrow r \dashv \Omega''}{\Omega \vdash \mathsf{if}\, op \,\mathsf{then}\, s_1 \,\mathsf{else}\, s_2 \rightsquigarrow r \dashv \Omega''}$$

E-Assign
$$\Omega \vdash rv \rightsquigarrow v \dashv \Omega' \qquad \Omega'(p) \Rightarrow v_p$$
$$v_p \text{ has no outer loans} \qquad x_{\mathsf{old}} \text{ fresh}$$
$$\frac{\Omega'(p) \leftarrow v \Rightarrow \Omega'' \qquad \Omega''' = \Omega''[x_{\mathsf{old}} \mapsto v_p]}{\Omega \vdash p := rv \rightsquigarrow () \dashv \Omega'''}$$

E-Free
$$\frac{\Omega(p) \Rightarrow \mathsf{Box}\, v \qquad \Omega \vdash p := \bot \rightsquigarrow () \dashv \Omega'}{\Omega \vdash \mathsf{free}\, p \rightsquigarrow () \dashv \Omega'}$$

E-Match
$$\frac{\Omega \vdash p \overset{s}{\Rightarrow} C[\overrightarrow{f = v}] \qquad \Omega \vdash s \rightsquigarrow r \dashv \Omega'}{\Omega \vdash \mathsf{match}\, p \,\mathsf{with}\, \ldots \mid C \rightarrow s \mid \ldots \rightsquigarrow r \dashv \Omega'}$$

Fig. 4. Selected Reduction Rules for LLBC. We omit: tuples (similar to constructor), sequences (trivial). We also omit the handling of results – these prevent further execution and simply get carried through. Boxes behave like regular ADT constructors, except for the **free** Rust function, which receives primitive treatment, above.

R-Not-Shared
$$\frac{\Omega(p) \Rightarrow v \qquad v \neq \mathsf{loan}^s\, \{\vec{l}\}\, v'}{\Omega(p) \overset{s}{\Rightarrow} v}$$

R-Shared
$$\frac{\Omega(p) \Rightarrow \mathsf{loan}^s\, \{\vec{l}\}\, v}{\Omega(p) \overset{s}{\Rightarrow} v}$$

Fig. 5. Auxiliary Judgment: Reading a Possibly Immutably-Shared Value. Rust allows matching on a value for which there are oustanding *shared* borrows; the auxiliary $\overset{s}{\Rightarrow}$ read allows reading underneath a loan^s, if applicable.

borrowed value with ℓ, or to introduce a new loan at p to account for the fact that the value at that place is now immutably borrowed. The r, s in the conclusion indicates that the rule may produce *either* a reserved or a shared borrow; again, it is safe to ignore the "reserved" variant for now.

For moves (E-Move), we disallow moving: $\bot$, already-borrowed values (no loans), or reserved borrows. We also forbid moving *through* a dereference. The former prevents invalidating stray borrows; simply said, if a value has unterminated borrows, we cannot obtain full ownership of it in

C-Shared-Borrow
$$\frac{\ell' \text{ fresh} \qquad \text{loan}^s \{\ell \cup \vec{\ell}\}\, v \in \Omega}{\Omega' = \left[\text{loan}^s \{\ell \cup \ell' \cup \vec{\ell}\}\, v / \text{loan}^s \{\ell \cup \vec{\ell}\}\, v\right] \Omega}{\Omega \vdash \text{copy borrow}^s \ell \Rightarrow \text{borrow}^s \ell' \dashv \Omega'}$$

C-Shared-Loan
$$\frac{\Omega \vdash \text{copy } v \Rightarrow v' \dashv \Omega'}{\Omega \vdash \text{copy loan}^s \{\vec{\ell}\}\, v \Rightarrow v' \dashv \Omega'}$$

C-Scalar
$$\frac{v = \text{true or false or } n_{i32} \text{ or } n_{u32} \text{ or } \dots}{\Omega \vdash \text{copy } v \Rightarrow v \dashv \Omega}$$

C-None
$$\frac{}{\Omega \vdash \text{copy None} \Rightarrow \text{None} \dashv \Omega}$$

C-Some
$$\frac{\Omega \vdash \text{copy } v \Rightarrow v' \dashv \Omega'}{\Omega \vdash \text{copy Some } v \Rightarrow \text{Some } v' \dashv \Omega'}$$

C-Tuple
$$\frac{\Omega_i \vdash \text{copy } v_i \Rightarrow v'_i \dashv \Omega_{i+1}}{\Omega_0 \vdash \text{copy } (\vec{v}) \Rightarrow (\vec{v}') \dashv \Omega_n}$$

Fig. 6. Auxiliary Judgment: Copying. We mimic MIR and see the copy of options and tuples as primitive operations. The judgment is undefined for any other construct as Rust's MIR only permits copying primitive data.

A-Shorthand
$$\frac{\ominus v_p}{v_p \text{ has no outer loans}}$$

A-Scalar
$$\frac{v = \text{true or false or } n_{i32} \text{ or } n_{u32} \text{ or } \dots}{\ominus v}$$

A-Tuple
$$\frac{\ominus v_i}{\ominus (\vec{v})}$$

A-Constructor
$$\frac{\ominus v_i}{\ominus C[\vec{f} = \vec{v}]}$$

A-Borrow-M
$$\frac{}{\ominus \text{borrow}^m \ell\, v}$$

A-Borrow-R-S
$$\frac{}{\ominus \text{borrow}^{r,s} \ell}$$

A-Bot
$$\frac{}{\ominus \perp}$$

Fig. 7. Auxiliary Judgment: Absence of Outer Loans. We use shorthand notation $\ominus$ for this figure. Enforcing this criterion ensures that, at assignment-time, the memory we are about to write does not contained loaned-out data, as this would be unsound. This judgement is defined by omission, and is never valid for values of the form loan _. Such values, however, may appear underneath borrows, as the A-Borrow-* rules enforce no preconditions.

Write-G
$$\frac{p = P[x] \qquad x \mapsto v_x \in \Omega \\ \Omega \vdash p(v_x) \leftarrow v \xRightarrow{g} v'_x \dashv \Omega' \qquad \Omega'' = \Omega'[x \mapsto v'_x]}{\Omega[p \mapsto v] = \Omega''}$$

W-G-Shared-Borrow
$$\frac{\text{loan}^s \{\ell \cup \_\}\, v_p \in \Omega \qquad \Omega \vdash p(v_p) \leftarrow v \xRightarrow{g} v'_p \dashv \Omega' \\ \Omega'' = \left[\text{loan}^s \{\ell \cup \_\}\, v'_p / \text{loan}^s \{\ell \cup \_\}\, v_p\right] \Omega}{\Omega \vdash (*^s p)(\text{borrow}^s \ell) \leftarrow v \xRightarrow{g} \text{borrow}^s \ell \dashv \Omega''}$$

Fig. 8. Auxiliary Judgment: Ghost Write. This judgment inherits all of the rules of the form W-*.

order to perform the move. The latter replicates Rust's constraint that no moves are allowed under a borrow.

For copies (E-Copy), we disallow copying mutable or reserved borrows, or mutably-loaned values; we rely on an auxiliary judgment of the form $\Omega \vdash \text{copy } v \Rightarrow v' \dashv \Omega'$, meaning creating a copy of v in Ω produces v' and returns a fresh environment Ω'. This judgment behaves like the earlier Read, except for shared borrows. When copying a shared borrow, we automatically allocate a new loan (C-Shared-Borrow) and augment the loan-set via a replacement (we use the substitution notation) – that is, we automatically perform a shared reborrow. When copying a shared loan (C-Shared-Loan),

we simply copy the actual value without performing any shared-loan tracking; the ownership information that regards the old value is irrelevant for the newly-copied value.

For matches (E-Match), we peek at the enum tag via $\overset{s}{\Rightarrow}$; actual transfer of ownership with moves and copies takes place in the suitable branch while executing s. We note that in general, our Read judgment may return values of the form loan^s: this is useful e.g., to enforce that a value is not loaned out, as in the premise of E-Move. For matches, however, we merely need to read the enum tag; for this, we automatically dereference shared loans via $\overset{s}{\Rightarrow}$.

All of our rules are in an explicit style: we prefer to add premises, rather than rely on an implicit invariant by omission. For instance, we add many premises to E-Copy rather than rely on the fact that the copy auxiliary judgment has no rules for mutable borrows, reserved mutable borrows and mutable loans. This also guides our lazy implementation of reorganizations.

We are now ready to define the semantics of assignments (E-Assign). We reduce rvs first, and remark that to obtain v, the various rules for the rv syntactic category must succeed. For instance, if the right-hand side is a move , then E-Move enforces all of its preconditions. This means E-Assign operates with ownership of v, which maps to our intuition for assignments in the presence of ownership and, naturally, also corresponds to the Rust semantics. What we do enforce, however, is that the value v_p found at place p should not have any (outer) loans (Figure 7). Overwriting a value that is currently loaned-out would violate safety; we need to rule this out. More precisely: loans may only appear behind pointer indirections; the value itself that is being overwritten may not contain any loan. The assignment rule ends by writing v at the new place (using Write). The rule for function call is identical, except it deals with binding the arguments, locals and return variable.

One key point of E-Assign is that we retain the old value in the environment Ω'', under a fresh name x_{old} not accessible to the user-written program. We call this retained value x a *ghost value*, because its only purpose is to avoid discarding useful ownership knowledge; operationally, this is memory that can be actually reclaimed since it isn't reachable anymore. Our third example at the beginning of the section leverages this fact.

3.4 Reorganizing Environments and Terminating Borrows

We now present the final conceptual portion of our operational semantics: reorganizing the environment, which we used in our earlier examples to terminate borrows. We present rules in a declarative style, to highlight the *semantics* of Rust as opposed to the *implementation* of borrow-checking. A consequence of our declarative approach is that we do not need to follow Rust's behavior to the letter; rather, we reorganize borrows in a lazy fashion, and don't terminate a borrow unless we need to get the borrowed value back. Concretely, our rules allow reorganization before and after every statement (we have elided this from Figure 4 for clarity). This has two concrete consequences. First, we ignore the drop nodes from MIR – indeed, they do not appear in Figure 1. Second, we claim that this captures a general semantics of borrows; we substantiate that claim by showing, in Section 6, how our semantics can validate a Rust program checked with Polonius, an ongoing rewrite of the borrow checker to allow for a larger class of Rust programs to be accepted.

We define reorganizing via a set of rewriting rules that operate on the environment Ω (Figure 9). Since these rules are syntactic in nature, we rely on value contexts $V[v]$, rather than our earlier semantic notions of reads, writes and ghost updates. We omit administrative rules for re-ordering environments at will. Our judgments are of the form $\Omega \hookrightarrow \Omega'$, meaning Ω may be reorganized into Ω'. We indulge in some syntax overload; whenever used on the left-hand side of $\hookrightarrow$, we understand $\Omega[x \mapsto v]$ to pattern-match on Ω to select a mapping. This considerably simplifies notation.

Our rules either render a value unusable ($\bot$), or strengthen it (borrow$_m$, in the case of reserved borrows). For these reasons, we demand unique ownership of the value in V via Not-Borrowed;

Not-Borrowed

$$\frac{\nexists V', V''. \; V[\cdot] = V'[\mathsf{borrow}^m \; \_ \; (V''[\cdot])]}{\nexists V', V''. \; V[\cdot] = V'[\mathsf{loan}^s \; \{\_\} \; (V''[\cdot])]}$$
$$\mathsf{not\_borrowed\_value} \; V$$

End-Shared-Or-Reserved-1

$$\frac{\mathsf{not\_borrowed\_value} \; V}{\Omega[x_1 \mapsto V[\mathsf{borrow}^{r,s} \ell], x_2 \mapsto V'[\mathsf{loan}^s \; \{\ell\} \; v]] \hookrightarrow}$$
$$\Omega[x_1 \mapsto V[\bot], \qquad x_2 \mapsto V'[v]]$$

Not-Shared

$$\frac{\nexists V', V''. \; V[\cdot] = V'[\mathsf{loan}^s \; \{\_\} \; (V''[\cdot])]}{\mathsf{not\_shared\_value} \; V}$$

End-Shared-Or-Reserved-2

$$\frac{\mathsf{not\_borrowed\_value} \; V \qquad \ell \notin \vec{\ell}}{\Omega[x_1 \mapsto V[\mathsf{borrow}^{r,s} \ell], x_2 \mapsto V'[\mathsf{loan}^s \; \{\ell \cup \vec{\ell}\} \; v]] \hookrightarrow}$$
$$\Omega[x_1 \mapsto V[\bot], \qquad x_2 \mapsto V'[\mathsf{loan}^s \; \{\vec{\ell}\} \; v]]$$

End-Mut

$$\frac{\{\mathsf{loan}, \mathsf{borrow}^r\} \notin v \qquad \mathsf{not\_borrowed\_value} \; V}{\Omega[x_1 \mapsto V[\mathsf{borrow}^m \; \ell \; v], x_2 \mapsto V'[\mathsf{loan}^m \; \ell]] \hookrightarrow}$$
$$\Omega[x_1 \mapsto V[\bot], \qquad x_2 \mapsto V'[v]]$$

Activate-Reserved

$$\frac{\{\mathsf{loan}, \mathsf{borrow}^r\} \notin v \qquad \mathsf{not\_shared\_value} \; V'}{\Omega[x_1 \mapsto V[\mathsf{borrow}^r \; \ell], \quad x_2 \mapsto V'[\mathsf{loan}^s \; \{\ell\} \; v]] \hookrightarrow}$$
$$\Omega[x_1 \mapsto V[\mathsf{borrow}^m \; \ell \; v], x_2 \mapsto V'[\mathsf{loan}^m \; \ell]]$$

Fig. 9. Reorganizing Environments

that is, we can only terminate borrows for values that are not themselves borrowed. Doing so, we precisely capture the constraints of Rust with regards to reborrows. We now review the rules.

When ending a shared borrow, we render the borrow unusable henceforth, and replace it with $\bot$. Then, two situations arise. If this is the last borrow (End-Shared-Or-Reserved-1), i.e., if the loan-set is the singleton set $\{\ell\}$, we replace the shared loan with the previously-shared value. If there are more borrows out there (End-Shared-Or-Reserved-2), we simply decrease the loan-set of the shared loan to reflect that the borrow has been ended.

When ending a mutable borrow (End-Mut), we enforce that we own the value we are about to return (i.e. not loaned). The borrow then becomes unusable, and the borrowed value is returned to its rightful owner.

This high-level approach to the Rust semantics allows us to very naturally account for an oft-used Rust feature, namely two-phase borrows, which are introduced in many places when desugaring to MIR. We account for those through what we call *reserved* borrows. Reserved borrows are created just like shared borrows (E-Shared-Or-Reserved-Borrow). However, reserved borrows cannot be copied, dereferenced, or written into. Therefore, the only way to use a reserved borrow is to strengthen it into a mutable borrow, which is legal, as long as all other (shared or reserved) borrows have ended (Activate-Reserved). Reserved borrows enable a variety of very common idioms [The Rust Compiler Team 2022] without resorting to more advanced desugarings.

These rules are declarative and non-ordered; in practice, our tool performs a syntax-directed reorganization guided by the various preconditions on our rules. For instance, whenever loan $\notin v$ appears as a premise, we perform a traversal of v to end whichever loans we encounter. This is another reason why we prefer the "explicit" style of rules (i.e. with copious premises): they clearly state expectations, and thus allow for a straightforward implementation.

4 SYMBOLIC ABSTRACTIONS AND FUNCTIONAL TRANSLATION

Our semantics allows us to keep track of borrows and ownership in an exact fashion. We now ask: if we adopt a modular approach and treat function calls as opaque, how much can we leverage borrows and regions to still enable precise tracking of ownership and aliasing? We answer that question with a region-centric shape analysis that abstracts away the effect of a function call on the ownership graph, via a notion of *region abstraction*. We dub the result our "symbolic semantics"; it

is, obviously, less precise than our earlier concrete semantics; yet, it contains enough ownership and aliasing information that we can generate a functional translation from it. Our symbolic semantics very much resembles the concrete semantics; this time, however, we turn out attention to the region information provided by function signatures to abstract away subsets of the ownership graph.

From this section onwards, we introduce a few additional restrictions on the subset of Rust we can handle. Our concrete semantics supports loops, but our symbolic semantics does not; we *do*, however, support recursive functions. We disallow nested borrows in function *signatures*, but users can still manipulate arbitrarily nested borrows within function bodies. We also disallow instantiating a polymorphic function with a type argument that contains a borrow. Finally, we do not allow type declarations that contain borrows. We believe most of these issues can be addressed with suitable amounts of engineering; we discuss these limitations in detail in Section 7.

4.1 Symbolic Semantics by Example

Symbolic Values; Matches. A first concept we need to add to our toolkit is that of a symbolic variable; that is, a variable whose type is known, but not its value: we write $(\sigma : \tau)$. We now illustrate how symbolic variables behave, notably in the presence of `match`es, which refine our static knowledge about a symbolic variable. From here on, we make many constructions explicit so as to study a valid LLBC program; importantly, moves are now materialized.

```
1   fn f(mut o: Option<i32>) {   // o ↦ (σ : Option i32)
2     let po = &mut o;           // o ↦ loanᵐ ℓ;   po ↦ borrowᵐ ℓ (σ : Option i32)
3
4     match *po {
5       None => {                // o ↦ loanᵐ ℓ;   po ↦ borrowᵐ ℓ (σ : None)
6         panic!() }
7
8       Some => {                // o ↦ loanᵐ ℓ;   po ↦ borrowᵐ ℓ (Some (σ′ : i32))
9         let r =
10          &mut (*po).Some.0;   // o ↦ loanᵐ ℓ;   po ↦ borrowᵐ ℓ (Some loanᵐ ℓ′);   o_ref ↦ borrowᵐ ℓ′ (σ′ : i32)
11         *r = 1; }};}          // o ↦ loanᵐ ℓ;   po ↦ borrowᵐ ℓ (Some loanᵐ ℓ′);   o_ref ↦ borrowᵐ ℓ′ 1
```

In the example above, σ stands in for the function parameter whose concrete value is unknown at run-time; σ behaves like any other value from our previous examples, and can be borrowed mutably (line 2).

A key requirement for the soundness of our semantics is to forbid changing the enum variant of `o`, while its value or one of its fields is borrowed: this disallows leftover borrows pointing to data of the wrong (previous) type. We enforce this soundness criterion as follows. Assume, for the sake of example, that the user at line 3 decides to mutate `o`, e.g. by doing `o = None`. Our semantics for assignments looks up the symbolic value for the left-hand side of the assignment (i.e., `o`), and demands that the symbolic value have no oustanding "outer" loans (we formally define this criterion in Figure 7). In order to satisfy this criterion, we must terminate the borrow `po` in order to obtain $o \mapsto \sigma; po \mapsto \perp$, which then prevents any further use of `po` – we have successfully prevented a type-incorrect usage.

At line 4, we perform a case analysis; at this stage, all we know is that the scrutinee `*po` evaluates to symbolic variable σ, of the correct type Option i32. In order to check the branches, we treat each one of them individually, in each case refining σ with a more precise value according to the constructor of the branch. Simply said, in the `None` case, we replace every occurrence of σ with None, and in the `Some` case, we replace every occurrence of σ with Some $(\sigma′ : i32)$, where $\sigma′$ is a fresh symbolic variable.

More interesting pointer manipulations follow in the `Some` branch. We borrow the value within the option via `r`, using a projector syntax inspired by MIR's internal representation of projectors.

This borrowing incurs no loss in precision in our alias tracking: because we refined σ earlier (in effect, performing a strong-update of σ), we know that *both* o and po are unusable as long as r lives. More specifically, and in the vein of our remark above: should the user, for the sake of example, decide to mutate via po, e.g. to change the enum variant by doing *po = None at line 10, our symbolic semantics would give up ownership of r, in order to regain $po \mapsto \text{borrow}^m \ell$ (Some 1), which by virtue of containing no "outer" loans would make the update valid (see 3.3, discussion of E-Assign).

Function Calls: Single region case. We now switch from the callee to the caller's perspective, and turn our attention to function calls. We introduce a new concept of region abstraction to our borrow graph. An abstraction owns borrows and loans, but does so abstractly; that is, we have no aliasing information about values in an abstraction. Region abstractions allow us to retain ownership and aliasing information in the presence of function calls; they are introduced when a call takes place, upon which they assume ownership of the call's arguments; they are terminated whenever the caller relinquishes ownership of the return value, upon which ownership flows back to the original arguments.

Before modifying the semantics from Section 3, we illustrate region abstractions with an example. We revisit our earlier test_choose function (Section 2).

```
1   let mut x = 0;    let mut y = 0;
2   let px = &mut x; let py = &mut y; // x ↦ loan^m ℓ_x, y ↦ loan^m ℓ_y, px ↦ borrow^m ℓ_x 0, py ↦ borrow^m ℓ_y 0
3   let pz = choose(true, move px, move py);
4   // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,   pz ↦ borrow^m ℓ_r (σ : uint32),
5   // A(ρ) {  borrow^m ℓ_x 0,   borrow^m ℓ_y 0,   loan^m ℓ_r  }
6   *pz = *pz + 1;
7   // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,   pz ↦ borrow^m ℓ_r (σ' : uint32),        step 0
8   // A(ρ) {  borrow^m ℓ_x 0,   borrow^m ℓ_y 0,   loan^m ℓ_r  }
9   // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,   pz ↦ ⊥,                                 step 1
10  // A(ρ) {  borrow^m ℓ_x 0,   borrow^m ℓ_y 0,   σ'  }
11  // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,   pz ↦ ⊥,                                 step 2
12  // px' ↦ borrow^m ℓ_x σ_x,   py' ↦ borrow^m ℓ_y σ_y
13  // x ↦ σ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,   pz ↦ ⊥,   px' ↦ ⊥,   py' ↦ borrow^m ℓ_y σ_y   step 3
14  assert!(x == 1);
```

Up to line 2, the usual set of rules apply and yield an environment that is consistent with Section 3. Our abstract rules come in at line 3, where we are faced with a function call. We now need to abstract the call, that is, precisely capture how the function call affects the borrow graph, without looking at the definition of the function itself. To do so, we have only one piece of information at our disposal: the type of choose, namely $(\text{bool}, \&^\rho \text{mut uint32}, \&^\rho \text{mut uint32}) \rightarrow \&^\rho \text{mut uint32}$.

The type of choose conveys two key pieces of information: first, it *consumes* two mutable borrows in order to *produce* a fresh (abstract) return value; second, the borrows and the return value belong to the same *region* ρ. We proceed as follows. We allocate a fresh region abstraction $A(\rho)$, which owns the consumed arguments pertaining to region ρ; in our case, $\text{borrow}^m \ell_x 0$ and $\text{borrow}^m \ell_y 0$. (In the case of multiple regions per function type, we need to project the ownership of the arguments along their respective regions; we handle this case formally in § 4.2.) We know that the return value pz has type $\&^\rho \text{mut uint32}$; furthermore, the region in the type tells us that the owner of this abstract value is the abstraction $A(\rho)$. We perform a *symbolic expansion* (also detailed in the next section) to give pz the shape $\text{borrow}^m \ell_r (\sigma : \text{uint32})$, pointing into a $\text{loan}^m \ell_r$ for the return value owned by $A(\rho)$. We use σ to denote a "symbolic value"; such values are not statically known, and receive a special treatment during the translation. We obtain the environment at lines 4-5, where the ownership of both px and py has been transferred to the region abstraction; and where pz has full ownership of a value loaned from the region abstraction. Intuitively, a region abstraction is a bag containing borrows (what has been consumed) and loans (what has been produced).

At line 6, the mutation type-checks, and does not affect the abstract environment: the symbolic value σ borrowed through pz is simply replaced by a fresh symbolic value σ' stemming from the addition. At that stage, we cannot read from x since it is mutably loaned; we therefore need to reorganize the environment to make the assertion succeed. Since we do not have any precise knowledge about the aliasing relationship between x, y and pz, we cannot return ownership to x directly; we must return ownership *en masse* by terminating region $A(\rho)$. We do so by terminating the borrow for pz, which returns the abstract value σ' to $A(\rho)$ (step 1, lines 9-10). Now that $A(\rho)$ has no outstanding loans left, we can terminate $A(\rho)$ itself. This reintroduces in the environment borrows l_x and l_y with fresh values, and replaces the borrowed values they held (0 in both cases) with fresh symbolic values to account for potential modifications (lines 11-12). These borrows are ghost values, i.e. not directly accessible to the user; they once again ensure we do not lose ownership information. A final reorganization of the environment terminates ℓ_x, and makes x usable again (line 13).

Multiple region case. We now study a call to the swap function, which permutes the two components of a single tuple, located in two different regions.

$$\text{swap} : (z : (\&^\alpha \text{mut uint32}, \&^\beta \text{mut uint32})) \rightarrow (\&^\beta \text{mut uint32}, \&^\alpha \text{mut uint32})$$

We examine a call `let r = swap (move z)` in the following environment:

$$x \mapsto \text{loan}^m \ell_x, \quad y \mapsto \text{loan}^m \ell_y, \quad z \mapsto (\text{borrow}^m \ell_x \; 0, \text{borrow}^m \ell_y \; 0)$$

This time, the presence of two regions forces us to be more precise. We introduce a new notion of projector, which comes in three flavors. We use an *input borrow projector* to dispatch each component of the argument to its respective region abstraction. We use a *loan projector* to dispatch each component of the returned value to its respective region abstraction. And we use an *output borrow projector* to determine the shape of the return value based on its type information. (We write these proj_{in}, proj_{l} and proj_{out}, respectively.) Doing so, we rely on *expansion rules* to destruct and name the components of various tuples as needed. Thus, the environment after the function call is:

$$x \mapsto \text{loan}^m \ell_x, \quad y \mapsto \text{loan}^m \ell_y, \quad z \mapsto \bot, \quad r \mapsto \text{proj}_{\text{out}} (\sigma : (\&^\beta \text{mut uint32}, \&^\alpha \text{mut uint32}))$$
$$A(\alpha)\{$$
$$\quad \text{proj}_{\text{in}} ((\text{borrow}^m \ell_x \; 0, \text{borrow}^m \ell_y \; 0) : (\&^\alpha \text{mut uint32}, \&^\beta \text{mut uint32}))$$
$$\quad \text{proj}_{\text{l}} (\sigma : (\&^\beta \text{mut uint32}, \&^\alpha \text{mut uint32}))$$
$$\}$$
$$A(\beta)\{$$
$$\quad \text{proj}_{\text{in}} ((\text{borrow}^m \ell_x \; 0, \text{borrow}^m \ell_y \; 0) : (\&^\alpha \text{mut uint32}, \&^\beta \text{mut uint32}))$$
$$\quad \text{proj}_{\text{l}} (\sigma : (\&^\beta \text{mut uint32}, \&^\alpha \text{mut uint32}))$$
$$\}$$

In the resulting environment, z has been consumed. The return value r needs to be decomposed using an output borrow projector, according to its type. And both region abstractions have the same content, which need to be projected along their respective regions α and β.

A first, new reorganization rule (Figure 11) allows us to refine the symbolic value σ to be a tuple (σ_l, σ_r). Next, the two input projectors reduce based on the type: the left component in $A(\alpha)$ remains, as it belongs to the enclosing region α, and the right component reduces to _, an ignored value that does not belong to α. (The case of β is symmetrical.) The loan projector behaves similarly and retains only the components pertaining to the enclosing region; finally, the output borrow projector generates a pair of borrows pointing to the corresponding abstract values. The resulting

environment is therefore as follows.

$$x \mapsto \text{loan}^m \ell_x, \quad y \mapsto \text{loan}^m \ell_y, \quad z \mapsto \bot, \quad r \mapsto (\text{borrow}^m \ell_l \ \sigma_l, \ \text{borrow}^m \ell_r \ \sigma_r),$$

$$A(\alpha)\{ \quad (\text{borrow}^m \ell_x \ 0, \_) \quad (\_, \text{loan}^m \ell_l) \quad \},$$

$$A(\beta)\{ \quad (\_, \text{borrow}^m \ell_y \ 0) \quad (\text{loan}^m \ell_r, \_) \quad \}$$

Discussion of examples. We see our region abstractions as a form of magic wands; a function call consumes part of the memory, and returns a magic wand (the region abstraction) along with its argument (the returned value). Regaining ownership of the consumed memory requires applying the magic wand to its argument, hence surrendering access to the returned value.

Naturally, region abstractions set the stage for our functional translation: introducing an abstraction translates to a call to a *forward function*, while terminating an abstraction translates to a call to a *backward function*. But in order to get to the functional translation, we must first define region abstractions more formally.

4.2 From Concrete to Symbolic Semantics

We define our symbolic semantics as an extension of our earlier formalism, along with a new rule for function calls. First, we extend the value category to account for symbolic values, denoted σ (Figure 10), as well as our three kinds of projectors.

Following the intuition from our second example, above, we recall that the chief goal of input and loan projectors is to distribute ("project"), at function call-time, each component of a value to their corresponding abstraction region, while output borrow projectors allow destructuring the caller's view of the return value according to its type and region. Projectors enjoy some duality: if we switch to the point of view of the callee, output and loan projectors capture the fact that the effective arguments are "on loan" from the callee's context.

We keep the syntactic overhead to a minimum. Our rules (and our implementation) enforce strong invariants: for instance, input borrow and loan projectors may only appear within region abstractions; and only a very restricted form of values may appear under projectors. But to keep notation lightweight, we refrain from adding extra syntactic categories. In the same spirit, we introduce some syntactic sugar. Whenever type annotations are not needed, we skip τ in $(\sigma : \tau)$. Conversely, whenever we need to access the type of a value (to make a region apparent), we allow $(v : \tau)$ as a convenient way to bind the type. Finally, we start leveraging region annotations in borrow types, and introduce a new form of path $A(\rho) \mapsto$ to select an element from an abstraction.

The new rewriting rules capture the behavior demonstrated with earlier examples. A symbolic value may be decomposed structurally (Decompose-Tuple); we use a substitution notation to indicate that there may be several occurrences of σ, and all of them must be substituted at the same time. (Our second example showcased this situation.) We tack the value σ being destructed and the pattern used for that purpose (σ_l, σ_r) onto the arrow, so that they can be conveniently recalled to generate a suitable let-binding in the functional translation. Symmetrically, any kind of projector descends along the structure of the symbolic value (Proj-Tuple).

Input borrow projectors, once confronted with a borrow value, either discard it (as in either Proj-I-Mut-No-Match or Proj-I-Shared-No-Match) or retain it (as in either Proj-I-Mut-Match or Proj-I-Shared-Match). As we forbid nested borrows for now, projections stop upon the first borrow they encounter; if the projector retains the borrow, we keep the whole borrowed value.

Loan and output borrow projectors reduce in lockstep: if the environment contains an output borrow projector and a region abstraction contains a corresponding loan projector over the same symbolic value σ, then we may turn them into a borrow and a loan, respectively (Proj-Unfold-Mut-Match, Proj-Unfold-Shared-Match). To give back ownership of a return value

$$v \quad ::=$$

	$\dots$	(as before)
	$(\sigma : \tau)$	symbolic value
	$\text{proj}_{\text{in}}\, v$	input borrow projector
	$\text{proj}_{\text{l}}\, v$	loan projector
	$\text{proj}_{\text{out}}\, v$	output borrow projector

$$\Omega \quad ::=$$

	$\dots$	(as before)
	$A(\rho)\{\vec{v}\}, \Omega$	new region abstraction for region ρ

$$\rho \quad ::= \qquad\qquad\qquad \text{region identifier}$$

Fig. 10. Abstract Semantics: Environments, Values

DECOMPOSE-TUPLE
$$\frac{\sigma_l, \sigma_r \text{ fresh}}{\Omega \overset{\sigma}{\underset{(\sigma_l,\sigma_r)}{\hookrightarrow}} \left[((\sigma_l, \sigma_r) : (\tau_1, \tau_2)) \big/ (\sigma : (\tau_1, \tau_2)) \right] \Omega}$$

PROJ-TUPLE
$$\frac{}{\begin{array}{l} \Omega[A(\rho) \mapsto \text{proj}_{\text{in,l,out}}\, (\sigma_l, \sigma_r)] \hookrightarrow \\ \Omega[A(\rho) \mapsto (\text{proj}_{\text{in,l,out}}\, \sigma_l, \text{proj}_{\text{in,l,out}}\, \sigma_r)] \end{array}}$$

PROJ-I-MUT-MATCH
$$\frac{}{\begin{array}{l} \Omega[A(\rho) \mapsto \text{proj}_{\text{in}}\, (\text{borrow}^m\, \ell\, \sigma : \&^\rho \text{mut}\, \tau)] \hookrightarrow \\ \Omega[A(\rho) \mapsto \text{borrow}^m\, \ell\, (\sigma : \tau)] \end{array}}$$

PROJ-I-MUT-NO-MATCH
$$\frac{}{\begin{array}{l} \Omega[A(\rho) \mapsto \text{proj}_{\text{in}}\, (\text{borrow}^m\, \_ : \&^\mu \text{mut}\, \tau)] \hookrightarrow \\ \Omega[A(\rho) \mapsto \_] \end{array}}$$

PROJ-I-SHARED-MATCH
$$\frac{}{\begin{array}{l} \Omega[A(\rho) \mapsto \text{proj}_{\text{in}}\, (\text{borrow}^s\, \ell : \&^\rho\, \tau)] \hookrightarrow \\ \Omega[A(\rho) \mapsto \text{borrow}^s\, \ell] \end{array}}$$

PROJ-I-SHARED-NO-MATCH
$$\frac{}{\begin{array}{l} \Omega[A(\rho) \mapsto \text{proj}_{\text{in}}\, (\text{borrow}^s\, \ell : \&^\mu\, \tau)] \hookrightarrow \\ \Omega[A(\rho) \mapsto \_] \end{array}}$$

PROJ-I-SYMB
$$\frac{\& \notin \tau}{\Omega[A(\rho) \mapsto \text{proj}_{\text{in}}\, (\sigma : \tau)] \hookrightarrow \Omega[A(\rho) \mapsto \sigma]}$$

PROJ-UNFOLD-MUT-MATCH
$$\frac{\sigma', \ell \text{ fresh}}{\begin{array}{l} \Omega[p \mapsto \text{proj}_{\text{out}}\, (\sigma : \&^\rho \text{mut}\, \tau), A(\rho) \mapsto \text{proj}_{\text{l}}\, \sigma] \hookrightarrow \\ \Omega[p \mapsto \text{borrow}^m\, \ell\, \sigma', \qquad A(\rho) \mapsto \text{loan}^m\, \ell] \end{array}}$$

PROJ-UNFOLD-SHARED-MATCH
$$\frac{\sigma', \ell \text{ fresh}}{\begin{array}{l} \Omega[p \mapsto \text{proj}_{\text{out}}\, (\sigma : \&^\rho\, \tau), A(\rho) \mapsto \text{proj}_{\text{l}}\, \sigma] \hookrightarrow \\ \Omega[p \mapsto \text{borrow}^s\, \ell, \qquad A(\rho) \mapsto \text{loan}^s\, \{\ell\}\, \sigma'] \end{array}}$$

PROJ-L-NO-MATCH
$$\frac{\& \notin \tau}{\begin{array}{l} \Omega[A(\rho) \mapsto \text{proj}_{\text{l}}\, (\sigma : \tau)] \hookrightarrow \\ \Omega[A(\rho) \mapsto \_] \end{array}}$$

END-ABSTRACTION
$$\frac{\begin{array}{c} \text{borrows}^m(A(\rho)) = \overrightarrow{\text{borrow}^m\, \ell\, \_} \\ \text{borrows}^s(A(\rho)) = \overrightarrow{\text{borrow}^s\, \ell'} \\ \text{loan}, \text{proj}_{\text{l}} \notin A(\rho).v_r \qquad \vec{\sigma}' \text{ fresh} \qquad \vec{x}_g, \vec{y}_g \text{ fresh} \end{array}}{A(\rho), \Omega \hookrightarrow \Omega, \overrightarrow{x_g \mapsto \text{borrow}^m\, \ell\, \sigma'}, \overrightarrow{y_g \mapsto \text{borrow}^s\, \ell'}}$$

END-ABSTRACT-MUT
$$\frac{\text{not\_borrowed\_value}\, V}{\begin{array}{l} \Omega[x \mapsto V[\text{borrow}^m\, \ell\, v], A(\rho) \mapsto \text{loan}^m\, \ell] \hookrightarrow \\ \Omega[x \mapsto V[\bot], \qquad A(\rho) \mapsto v] \end{array}}$$

Fig. 11. Reorganizing Environments with Abstract Values and Projectors

to the abstraction, End-Abstract-Mut folds a mutable borrow back into an abstraction. Ending a region abstraction itself is done via End-Abstraction; we use the borrow notation to collect all the borrows, at once from all the values in the abstraction. We require two things. First, that the abstraction has no outstanding loans, i.e. the contents of the abstraction are either symbolic values or borrows. Second, that the abstraction contains no loan projectors. This latter precondition avoids dangling output borrow projectors once the abstraction has disappeared. Terminating the region abstraction returns ownership of the borrows, with fresh symbolic values, to fresh (ghost) variables.

4.3 From Symbolic Semantics to Functional Code

At last, we explain how Aeneas, using the symbolic semantics, generates a pure translation of the original LLBC program. We make several hypotheses at this stage; beyond the restrictions we already mentioned, we now assume every disjunction in the control-flow is in terminal position. This is a strong restriction, which in practice requires duplicating the continuation of conditionals and matches. This is also a well-understood problem, known as "computing a join" in abstract interpretation for shape analysis domains [Rival 2011], or the "merge problem" in Mezzo [Protzenko 2014]. Coupled with the fact that problem space is highly constrained by Rust's lifetime discipline, we are confident that this can be addressed systematically and predictably.

4.3.1 Translation Example: Forward Function. As a starting example, we consider the forward translation of the `call_choose` function below, now presented in LLBC syntax with explicit writes and moves, along with a fully-explicit return variable `x_ret`.

```
1  fn call_choose(mut p : (u32, u32)) -> (x_ret: u32) {
2    let px = &mut p.0;
3    let py = &mut p.1;
4    let pz = choose(true, move px, move py);
5    *pz = *pz + 1;
6    x_ret = move p.0;
7    return;
8  }
```

The translation of the forward function is carried out by performing a symbolic execution on `call_choose`, and synthesizing an AST in parallel to reflect the effect of applying the symbolic execution rules.

A key insight about our synthesis rules is that they are exclusively concerned with *symbolic values*; the actual variables from the source program ($x \mapsto \ldots$) are mere book-keeping devices and have no relevance to the translated program. Symbolic values σ, however, cannot be determined statically; they thus compute at run-time, and as such are let-bound in the target program.

We start the translation by initializing an environment, where p maps to a symbolic value σ_0. In parallel, we synthesize the function as an AST with a hole to be progressively filled as we make progress throughout the translation. We present both environment and translation side-by-side to reflect the fact that they both make progress in parallel. We point, whenever possible, to the specific rules that apply; these pointers can be skipped upon a first reading, but might prove useful later, once the reader has encountered the formal definition of the translation.

$p \mapsto (\sigma_0 : (u32, u32))$

```
1  let call_choose_fwd (s0 : (u32 & u32)) : u32 =
2    [.]
```

At line 2, accessing field 0 requires us to expand σ_0; we first do so, which leads to the introduction of a let-binding in the translation (T-Destruct, Decompose-Tuple).

$p \mapsto (\sigma_1 : u32, \sigma_2 : u32)$

```
1  let call_choose_fwd (s0 : (u32 & u32)) : u32 =
2    let (s1, s2) = s0 in
3    [.]
```

The expansion above then allows us to evaluate the two mutable borrows on lines 2-3 via E-Mut-Borrow. It is important to notice that borrows and assignments just lead to bookkeeping in the environment: the synthesized translation is left unchanged. We get the following:

$p \mapsto (\mathrm{loan}^m\,\ell_1, \mathrm{loan}^m\,\ell_2)$
$px \mapsto \mathrm{borrow}^m\,\ell_1\,(\sigma_1 : \mathrm{u32})$
$py \mapsto \mathrm{borrow}^m\,\ell_2\,(\sigma_2 : \mathrm{u32})$

```
1  let call_choose_fwd (s0 : (u32 & u32)) : u32 =
2    let (s1, s2) = s0 in
3    [.]
```

We then reach the function call at line 4. To account for the call, we do several things (T-Call-Forward). As before (Section 4.1), we introduce a region abstraction to account for 'a; transfer ownership of the effective arguments to the abstraction; then introduce a fresh symbolic value $\sigma_3 : \&^\alpha \mathrm{mut}\,\mathrm{u32}$ to account for the returned value stored in pz.

Specifically, we transfer ownership of the effective arguments of the call to region α, using three input projections to retain only the borrows that pertain to α. The first argument gets projected as $\mathrm{proj}_{\mathrm{in}}\,\alpha$ bool, which reduces to _ (irrelevant value). The next two arguments, each of type $\&^\alpha \mathrm{mut}\,\mathrm{u32}$, reduce according to Proj-I-Mut-Match.

To handle the return value, we introduce a loan projector within the abstraction, and an output borrow projector in the caller's context for pz; both refer to the same symbolic value σ_3, meaning that they will both be refined simultaneously via the Proj-Unfold- rules.

In parallel, we introduce a call to choose_fwd in the synthesized translation. As previously mentioned, the borrow types are translated to the identity; the input arguments of choose_fwd are thus simply s1 and s2. Our translation is monadic, meaning we obtain:

$p \mapsto (\mathrm{loan}^m\,\ell_1, \mathrm{loan}^m\,\ell_2)$
$px \mapsto \bot$
$py \mapsto \bot$
$pz \mapsto \mathrm{proj}_{\mathrm{out}}\,(\sigma_3 : \&^\alpha \mathrm{mut}\,\mathrm{u32})$
$A(\alpha)\{$

$\quad \overset{\rightharpoonup}{}$
$\quad \mathrm{borrow}^m\,\ell_1\,(\sigma_1 : \mathrm{u32}),$
$\quad \mathrm{borrow}^m\,\ell_2\,(\sigma_2 : \mathrm{u32}),$
$\quad \mathrm{proj}_{\mathrm{l}}\,(\sigma_3 : \&^\alpha \mathrm{mut}\,\mathrm{u32}),$
$\}$

```
1  let call_choose_fwd (s0 : (u32 & u32)) : u32 =
2    let (s1, s2) = s0 in
3    s3 <-- choose_fwd true s1 s2;
4    [.]
```

Upon evaluating line 5, we expand $\sigma_3 : \&^\alpha \mathrm{mut}\,\mathrm{u32}$ to $\mathrm{borrow}^m\,\ell_3\,(\sigma_4 : \mathrm{u32})$, following its type (Proj-Unfold-Mut-Match). Once again, as the borrow types are translated to the identity, the expansion of σ_3 simply introduces a variable reassignment (Pure-Mut-Borrow). This gives us:

$p \mapsto (\mathrm{loan}^m\,\ell_1, \mathrm{loan}^m\,\ell_2)$
$px \mapsto \bot$
$py \mapsto \bot$
$pz \mapsto \mathrm{borrow}^m\,\ell_3\,(\sigma_4 : \mathrm{u32})$
$A(\alpha)\{$

$\quad \overset{\rightharpoonup}{}$
$\quad \mathrm{borrow}^m\,\ell_1\,(\sigma_1 : \mathrm{u32}),$
$\quad \mathrm{borrow}^m\,\ell_2\,(\sigma_2 : \mathrm{u32}),$
$\quad \mathrm{loan}^m\,\ell_3,$
$\}$

```
1  let call_choose_fwd (s0 : (u32 & u32)) : u32 =
2    let (s1, s2) = s0 in
3    s3 <-- choose_fwd true s1 s2;
4    let s4 = s3 in
5    [.]
```

Now that σ_3 has been decomposed, we can symbolically execute the increment, which merely introduces a call to u32_add and generates a fresh variable σ_5 which stands for the unknown result of the addition:

$p \mapsto (\text{loan}^m \ell_1, \text{loan}^m \ell_2)$
$px \mapsto \bot$
$py \mapsto \bot$
$pz \mapsto \text{borrow}^m \ell_3 \ (\sigma_5 : \text{u32})$
$A(\alpha)\{$

    $\overrightarrow{\text{borrow}^m \ell_1 \ (\sigma_1 : \text{u32}),}$
    $\text{borrow}^m \ell_2 \ (\sigma_2 : \text{u32}),$
    $\text{loan}^m \ell_3,$

$\}$

```
1  let call_choose_fwd (s0 : (u32 & u32)) : u32 =
2    let (s1, s2) = s0 in
3    s3 <-- choose_fwd true s1 s2;
4    let s4 = s3 in
5    s5 <-- u32_add s4 1;
6    [.]
```

Finally, the move at line 6 requires retrieving the ownership of `p.0`. Doing so requires ending the region abstraction $A(\alpha)$ (introduced by the call to **choose**), which in turns requires ending the loan inside the abstraction (End-Abstract-Mut). Followingly, we first end ℓ_3 leading to the environment below, where we notice σ_5 is the value given back *to* **choose**. Ending a loan (or a borrow, depending of the point of view) leaves the synthesized code unchanged.

$p \mapsto (\text{loan}^m \ell_1, \text{loan}^m \ell_2)$
$px \mapsto \bot$
$py \mapsto \bot$
$pz \mapsto \bot$
$A(\alpha)\{$

    $\overrightarrow{\text{borrow}^m \ell_1 \ (\sigma_1 : \text{u32}),}$
    $\text{borrow}^m \ell_2 \ (\sigma_2 : \text{u32}),$
    $(\sigma_5 : \text{u32})$

$\}$

```
1  let call_choose_fwd (s0 : (u32 & u32)) : u32 =
2    let (s1, s2) = s0 in
3    s3 <-- choose_fwd true s1 s2;
4    let s4 = s3 in
5    s5 <-- u32_add s4 1;
6    [.]
```

Then, we actually end the region abstraction $A(\alpha)$ by moving back the borrows ℓ_1 and ℓ_2 in the environment, with fresh symbolic values σ_6 and σ_7 (End-Abstraction). Those are the values given back *by* **choose**. We mentioned earlier that ending a region abstraction translates as a call to a backward function; we thus synthesize a call to **choose_back** using T-Call-Backward. The **choose_back** function receives both the original arguments given to the call to **choose_fwd** (that is, **true**, **s1** and **s2**, which replicate the control-flow of the forward function) and the value returned *to* the region abstraction (that is, **s5**). The **choose_back** function produces a pair of values, namely those given back *by* the region abstraction (that is, **s6** and **s7**).

$p \mapsto (\text{loan}^m \ell_1, \text{loan}^m \ell_2)$
$px \mapsto \bot$
$py \mapsto \bot$
$pz \mapsto \bot$
$px_g \mapsto \text{borrow}^m \ell_1 \ (\sigma_6 : \text{u32})$
$py_g \mapsto \text{borrow}^m \ell_2 \ (\sigma_7 : \text{u32})$

```
1  let call_choose_fwd (s0 : (u32 & u32)) : u32 =
2    let (s1, s2) = s0 in
3    s3 <-- choose_fwd true s1 s2;
4    let s4 = s3 in
5    s5 <-- u32_add s4 1;
6    (s6, s7) <-- choose_back true s1 s2 s5;
7    [.]
```

We can finally end borrow ℓ_1 and evaluate the **return**, which ends the translation (T-Return-Forward). As we save meta-information about the assignments to generate suitable names for the variables, and also inline unnecessary let-bindings, Aeneas actually generates the following function:

```
1  let call_choose_fwd (p : (u32 & u32)) : u32 =
2    let (px, py) = p in
3    pz <-- choose_fwd true px py;
4    pz0 <-- u32_add pz 1;
5    (px0, _) <-- choose_back true px py pz0;
6    Return px0
```

4.3.2 Translation Example: Backward Function `choose_back`. We now proceed with the synthesis of the *backward* translation of `choose` from section 2. We recall its definition here for the sake of clarity.

```
1  fn choose<'a, T>(b : bool, x : &'a mut T, y : &'a mut T) -> &'a mut T {
2    if b {
3      return x;
4    }
5    else {
6      return y;
7    }
8  }
```

Unlike `call_choose`, the `choose` function takes borrows as input parameters. We thus need to track their provenance, from the point of view of the callee. As a consequence, we initialize the environment by introducing an abstraction containing loans so as to model the values owned by the caller and *loaned* to the function for as long as α lives (T-FUN-BACKWARD). This is the dual of the caller's point of view: the abstraction contains loan projectors (that stand in for the effective arguments), and the context contains output borrow projectors (that stand in for the formal parameters).

The synthesized function itself receives the same input parameters as the forward translation of `choose` (that is to say `b`, `x` and `y`), together with an additional parameter (`ret`) for the value which will be given back to the backward function upon ending α.

$$A_{\text{input}}(\alpha)\{ \quad \_, \quad \text{loan}^m \ell_x, \quad \text{loan}^m \ell_y, \quad \}$$
$$b \mapsto \sigma_b$$
$$x \mapsto \text{borrow}^m \ell_x \ (\sigma_x : t)$$
$$y \mapsto \text{borrow}^m \ell_y \ (\sigma_y : t)$$

```
1  let choose_back
2    (t : Type) (b : bool) (x : t) (y : t)
3    (ret : t) : result (t & t) =
4    [.]
```

We then evaluate the `if` at line 2 (T-IFTHENELSE). This requires branching over symbolic value σ_b. We duplicate the environment and substitute σ_b with true for the first branch, and false for the second branch. This substitution is a generic refinement which comes in handy for data types, where it introduces binders for constructor arguments. Of course, this introduces a branching in the synthesized translation. Below, we show the environment for the first branch of the `if`:

$$A_{\text{input}}(\alpha)\{ \quad \_, \quad \text{loan}^m \ell_x, \quad \text{loan}^m \ell_y, \quad \}$$
$$b \mapsto \text{true}$$
$$x \mapsto \text{borrow}^m \ell_x \ (\sigma_x : t)$$
$$y \mapsto \text{borrow}^m \ell_y \ (\sigma_y : t)$$

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

We proceed with the evaluation of the first branch. Upon reaching line 3, we return `x`, that is to say the value $\text{borrow}^m \ell_x \ (\sigma_x : t)$. This is the crucial part of the translation. We first need to model the fact that after the return, the caller is free to use the mutable borrow ℓ_x to perform in-place modifications until they end region α. Then, upon ending α, we need to propagate the properly updated values that were loaned to `choose` at call site back to their original owners. We proceed in two steps (T-RETURN-BACKWARD).

We first replace σ_x with σ_{ret}, a symbolic value reserved for this purpose and corresponding to the `ret` parameter in the translation. We thus model the transfer of ownership from the caller back to its callee, which happens when the caller ends region α. At this moment, the callee observes that σ_x has been updated to σ_{ret}. This gives us the environment:

$$A_{\text{input}}(\alpha)\{ \quad \_, \quad \text{loan}^m \ell_x, \quad \text{loan}^m \ell_y, \quad \}$$
$$b \mapsto \text{true}$$
$$x \mapsto \text{borrow}^m \ell_x \ (\sigma_{\text{ret}} : t)$$
$$y \mapsto \text{borrow}^m \ell_y \ (\sigma_y : t)$$

For the second step, we end the loans in the input abstraction A_{input} to determine which values are given back to the caller (in our case, the values borrowed by x and y), in place of the values loaned to choose (and whose provenance is captured by loan^m ℓ_x and loan^m ℓ_y). Ending those loans (END-ABSTRACT-MUT) potentially leads to ending region abstractions (END-ABSTRACTION), introducing calls to backward functions (this happens for list_nth_mut). We get the following environment:

$$A_{\text{input}}(\alpha)\{ \quad \_, \quad \sigma_{\text{ret}}, \quad \sigma_y, \quad \}$$
$$b \mapsto \text{true}$$
$$x \mapsto \bot$$
$$y \mapsto \bot$$

From this environment we finally deduce the first branch of the translation:

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

We omit the translation of the second branch, which is similar.

4.3.3 Synthesis Rules. The rules are in Figure 13, where T- stands for translation; they describe a process in which we traverse the source program in a forward fashion, simultaneously updating our symbolic environment and generating pure λ-terms. Our final judgment operates over LLBC statements s and takes two forms. For statements that are in terminal position, we write $M, \Omega \vdash s \updownarrow e$, meaning statement s compiles down to pure expression e in environment Ω and translation meta-data M. But for statements that are not in terminal position (i.e., on the left-hand side of a semicolon), we are faced with the usual mismatch between statement-based languages and let/expression-based languages. We solve the issue by allowing expressions to contain a hole, to receive a continuation – we write $E[\cdot]$. Our judgement for non-terminal statements is thus of the form $M, \Omega \vdash s \updownarrow E[\cdot] \dashv M', \Omega'$ – we note that this form produces an updated environment M', Ω' to allow chaining with subsequent statements. In practice, our implementation uses continuation-passing style to keep things readable, as opposed to an AST definition with holes for our target language. In both cases, we let $\updownarrow$ be either $\downarrow$, for translating a forward function, or $\uparrow^\rho$, for translating the backward function associated to region ρ. We spare the reader the grammar of expressions e, which is a standard lambda-calculus; suffices to say that we use $\leftarrow$ to denote the monadic bind operator, as in Section 2; monadic returns appear as ret. The synthesis of variables is captured by the rules PURE-*; as we alluded to earlier, our translation never encounters, nor produces, a source variable x; rather, they structurally visit a symbolic value and map source *symbolic* variables to variables in the target λ-calculus (PURE-SYMB). Naturally, in practice, we use heuristics to pick sensible names for the symbolic variables, thus guaranteeing that the output of our translation is readable. Conversion of types from source to target is almost the identity, except for Box τ, $\&^\rho$mut τ and $\&^\tau$ which become τ, consistently with PURE-BOX, PURE-MUT-BORROW and PURE-SHARED-BORROW.

Another insight about our rules: in order to make progress, we may synthesize fresh bindings at any time via a reorganization. For instance, if a symbolic value with a tuple type is refined into the tuple of its components (DECOMPOSE-TUPLE), we need to mirror this fact in the generated program (T-DESTRUCT). We use a degenerate judgment of the form $M, \Omega \vdash \emptyset \updownarrow \ldots$, which appears in T-DESTRUCT, T-REORG-ANYTIME and T-CALL-BACKWARD.

The rest of the rules leverage our earlier concepts of region abstractions, projections, and symbolic environments to precisely capture the relationship between a function body and its parameters (callee), or a function application and its arguments (caller). The rules for synthesis generally apply both for generating forward and backward functions, with the exception of T-RETURN and T-FUN.

We adopt the perspective of the caller and begin with function calls. In T-CALL-FORWARD, we follow the procedure outlined in our earlier examples. First, we allocate a fresh σ_r to stand in for

the return value of the call; we have one abstraction region per region in the function type. Each abstraction region gains ownership of the relevant part of the function arguments ($\text{proj}_{\text{in}}\ \vec{v}$), and loans out whichever part of the symbolic return value originates from that region ($\text{proj}_{|}\ \sigma_r$). These abstractions augment the symbolic environment, along with an output projector for the return value σ_r. We synthesize a monadic bind which introduces σ_r in the generated program, and record in the meta-data M that this call happened with effective arguments $\vec{v}$. Rule T-Call-Backward is not syntax-directed and may happen at any time; in practice, we apply it lazily. We wish to terminate region ρ, associated to an earlier function call found in M. We require a successful application of End-Abstraction, so as to terminate abstraction region $A(\rho)$. This returns ownership to us (the caller) of various borrows of symbolic values $\vec{\sigma}'$. We thus need to synthesize a call to the backward function for region ρ of f, in order to compute in the translated code what is the value of $\vec{\sigma}'$. The backward function receives the *original* effective arguments to f (found in M), and the symbolic values that stands for the terminated projector loans from $A(\rho)$.

We now switch to the perspective of the *callee* and study function definitions. We once again rely on region abstractions to explain the ownership relationship between arguments and function body. In T-Fun-Forward, we synthesize f_{fwd}. In our initial environment, we have one abstraction per region in the type of f; each abstraction region owns the parts of each argument that are along region ρ_i; we execute the body in an environment where the formal arguments $\vec{x}_{\text{arg}}$ are each wrapped in an output projector. The backward functions are synthesized in a similar way, except with extra arguments.

Matches are delicate, and come in two flavors. We remark that our matches are made up of non-nested, constructor patterns; by the time we examine Rust's internal MIR language, nested patterns have already been desugared.

If our symbolic environment has enough static knowledge to determine which particular branch of the data type we are in (T-Match-Concrete), we do not bother with generating a trivial match and simply generate code for the corresponding branch. If the scrutinee is not known statically, then it is a symbolic variable (T-Match-Symbolic). For each branch i, we effectively perform a strong update of σ, replacing it with a constructor value whose fields are themselves fresh symbolic variables (the $\vec{\sigma}_i$). The same $\vec{\sigma}_i$ appear in the translated code (our target lambda calculus *can* bind arguments to constructors), thus preserving proper lexical binding in the generated code. We remark that refining into a constructor value is important for soundness – lacking this precise substructural tracking, we would lose precision in our borrow checking and would allow type-incorrect programs.

We finally explain the rules for returning. In T-Return-Forward, the value found in the special return variable dictates what we return, so long as we have ownership of this value. In T-Return-Backward, our goal is now to map sub-parts of v_{ret} back to their original locations, using our region abstraction analysis. We "symbolize" v_{ret} using an auxiliary function (Figure 12) ; intuitively, sym replaces the sub-parts of v_{ret} (what we symbolically know about the callee's return value x_{ret}, so far) with matching sub-parts from σ_{ret} (the caller-updated return value, bound in T-Fun-Backward, provided at call site in T-Call-Backward). For instance:

$$\text{sym}(\alpha, (\sigma_l, \sigma_r), (\text{borrow}^m\ \ell\ v : \&^\alpha \text{mut}\ \tau, \text{borrow}^m\ \ell'\ v' : \&^\beta \text{mut}\ \tau')) = (\text{borrow}^m\ \ell\ \sigma_l, \bot)$$

We then allow reorganizing the environment, so as to to allow destructuring σ_{ret} suitably; this may also trigger calls to T-Call-Backward, as is the case for `list_nth_back`. If the region abstraction $A(\rho)$ is fully closed, the values it now holds determine the tuple we pass to the monadic return.

Those are arguably the most important rules: we illustrate them with `choose` (Section 2). In `choose_back`, σ_{ret} corresponds to the `ret` argument. In the `true` branch, v_{ret} is $\text{borrow}^m\ \ell_x\ \sigma_x$. Calling sym produces $\text{borrow}^m\ \ell_x\ \sigma_{\text{ret}}$. We end ℓ_x, and thus propagate σ_{ret} back to the loan for ℓ_x. Besides,

$$\begin{array}{lcl}
\mathsf{sym}(\alpha, (\vec{\sigma}), (\vec{v})) & = & \overrightarrow{(\mathsf{sym}(\alpha, \sigma, v))} \\
\mathsf{sym}(\alpha, \sigma, \mathsf{borrow}^m\, \ell\, v : \&^\alpha \mathsf{mut}\, \tau) & = & \mathsf{borrow}^m\, \ell\, \sigma \\
\mathsf{sym}(\alpha, \_, \mathsf{borrow}^m\, \ell\, v : \&^\beta \mathsf{mut}\, \tau) & = & \bot \qquad\qquad\qquad \alpha \neq \beta
\end{array}$$

Fig. 12. The sym function, used for the generation of backward functions. Specifically, $\mathsf{sym}(\alpha, \sigma, v)$ models the caller invoking the backward function for α with the value originally returned by the forward function to said caller. We abstract away the concrete view of the return value (v) into a symbolic view (modeled by σ) – essentially saying that the caller may have arbitrarily mutated the return value while it owned it.

ending ℓ_y gives back the unchanged σ_y and thus closes the region abstraction $A(\alpha)$, which now contains $\{\sigma_{\mathrm{ret}}, \sigma_y\}$; we return the symbolic values in the exact same order, that is, (ret, y).

5   IMPLEMENTING AENEAS

Our implementation is written in a mixture of Rust and OCaml. A first tool, dubbed CHARON, performs the translation from Rust's MIR internal representation to LLBC. Concretely, CHARON is a Rust compiler plugin that performs a large amount of mundane, tedious tasks, such as: computing a dependency graph, reordering definitions, grouping mutually recursive definitions together, reconstructing data type creation, and generally getting rid of the idioms that are definitely too low-level for LLBC (Section 3.1). Once this is done, CHARON dumps a JSON file to disk containing the LLBC AST. We plan to switch to a more efficient binary format in the future. CHARON totals 9.5 kLoC (lines of code, excluding whitespace and comments). CHARON lives as a separate project because we believe it has an existence of its own outside of AENEAS; we could easily see other projects re-using a lot of the engineering work we performed in order to share the implementation burden. We hope to present CHARON to several other tool authors in the near future.

AENEAS picks up the CHARON-generated AST, and implements the transformations described in Section 3 and Section 4. Practically speaking, we have a single interpreter that runs in two modes, either concrete or symbolic. The former produces a final value, if running a closed term; the latter produces a translated program. We currently extract to F*, and have a Coq backend in the works, along with plans for an HOL4 and a Lean backend. Effectively, our symbolic interpreter acts as a borrow checker for Rust programs using our semantic notion of borrows; we plan to investigate whether we can isolate this checker to validate, e.g., bare C programs that would fit within our admissible subset. We have written AENEAS in OCaml, a language much better suited to the manipulation of ASTs than Rust. The implementation of AENEAS totals 13.5kLoC. Of those, 6kLoC are for the interpreters, and 4kLoC for the translation and extraction. The rest contains library functions.

The implementation is, naturally, trusted. However, we have taken extraordinary care to ensure that it is trustworthy. Notably, after every application of one of the rules, we verify a large amount of invariants, such as: the environment is well-typed; borrows are consistent; shared values don't contain a mutable loan, etc. In practice, those invariants are extremely tight, and have led to the great level of detail that our rules exhibit. Should one turn off those invariant checks, the whole CHARON + AENEAS invocation becomes almost instantaneous, as opposed to a few seconds per file when constantly checking invariants. In addition to those 13.5kLoC, we have 6kLoC of comments; our implementation is truly written with great care. Curious readers can find both CHARON and AENEAS in the supplementary material.

Pure-Mut-Borrow
$$\frac{\Omega \vdash v \updownarrow e}{\Omega \vdash \mathsf{borrow}^m\, \ell\, v \updownarrow e}$$

Pure-Const
$$\frac{}{\Omega \vdash n_{\mathsf{i32}} \updownarrow n_{\mathsf{i32}}}$$

Pure-Tuple
$$\frac{\Omega \vdash \vec{v} \updownarrow \vec{e}}{\Omega \vdash (\vec{v}) \updownarrow (\vec{e})}$$

Pure-Symb
$$\frac{}{\Omega \vdash \sigma \updownarrow \sigma}$$

Pure-Box
$$\frac{\Omega \vdash v \updownarrow e}{\Omega \vdash \mathsf{Box}\, v \updownarrow e}$$

Pure-Shared-Borrow
$$\frac{\mathsf{loan}^s\,\{\vec{\ell}\}\, v \in \Omega \qquad \ell \in \vec{\ell} \qquad \Omega \vdash v \updownarrow e}{\Omega \vdash \mathsf{borrow}^s\, \ell \updownarrow e}$$

T-Destruct
$$\frac{\vec{\sigma}'\ \mathsf{fresh} \qquad M, \Omega \xrightarrow[(\vec{\sigma}')]{\sigma} M', \Omega'}{M, \Omega \vdash \emptyset \updownarrow \mathsf{let}\,(\vec{\sigma}') = \sigma\ \mathsf{in}\ [\cdot] \dashv M', \Omega''}$$

T-Return-Forward
$$\frac{\Omega_i \vdash x_{\mathsf{local},i} := \bot \rightsquigarrow () \dashv \Omega_{i+1} \qquad \Omega_n(x_{\mathsf{ret}}) \Rightarrow v \qquad \Omega_n \vdash v \downarrow e \qquad \{\mathsf{loan}, \bot, \mathsf{borrow}^r\} \notin v}{M, \Omega_0 \vdash \mathsf{return} \downarrow e}$$

T-IfThenElse
$$\frac{M, \Omega \vdash op \updownarrow \dashv M', \Omega' \qquad \Omega' \xrightarrow[\mathsf{true}]{\sigma} \Omega_1 \qquad \Omega' \xrightarrow[\mathsf{false}]{\sigma} \Omega_2 \qquad M', \Omega_1 \vdash s_1 \updownarrow e_1 \qquad M', \Omega_2 \vdash s_2 \updownarrow e_2}{M, \Omega \vdash \mathsf{if}\ op\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \updownarrow \mathsf{if}\ \sigma\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2}$$

T-Seq
$$\frac{M, \Omega \vdash s_1 \updownarrow E[\cdot] \dashv M', \Omega' \qquad M', \Omega' \vdash s_2 \updownarrow e \dashv M'', \Omega''}{M, \Omega \vdash s_1; s_2 \updownarrow E[e] \dashv M'', \Omega''}$$

T-Call-Forward
$$\frac{A(\rho_i) = \left\{\overrightarrow{\mathsf{proj}_{\mathsf{in}}\, v}, \mathsf{proj}_{\mathsf{l}}\, \sigma_{\mathsf{ret}}\right\} \quad (\sigma : \tau) \in \vec{v}\ \mathsf{implies}\ \&\notin \tau \quad \vec{\rho}\ \mathsf{fresh} \qquad \sigma_{\mathsf{ret}}\ \mathsf{fresh} \qquad \Omega \vdash v_i \updownarrow e_i \qquad \Omega, \overrightarrow{A(\rho)} \vdash p := \mathsf{proj}_{\mathsf{out}}\, \sigma_{\mathsf{ret}} \rightsquigarrow () \dashv \Omega'}{M, \Omega \vdash p := f\langle\vec{\rho}\rangle\, \vec{v} \updownarrow \sigma_{\mathsf{ret}} \leftarrow f_{\mathsf{fwd}}\, \vec{e};\ [\cdot] \dashv (f\langle\vec{\rho}\rangle\, \vec{e}), M; \Omega'}$$

T-Reorg-Anytime
$$\frac{M, \Omega \vdash \emptyset \updownarrow E[\cdot] \dashv M', \Omega' \qquad M', \Omega' \vdash s \updownarrow E'[\cdot] \dashv M'', \Omega''}{M, \Omega \vdash s \updownarrow E[E'[\cdot]] \dashv M'', \Omega''}$$

T-Call-Backward
$$\frac{\Omega = A(\rho), \Omega' \qquad M = f\langle\ldots, \rho, \ldots\rangle\, \vec{v}, M' \qquad \Omega \vdash A(\rho).v_{\mathsf{ret}} \updownarrow e_{\mathsf{ret}} \qquad A(\rho), \Omega' \hookrightarrow \Omega'', x_g \mapsto \mathsf{borrow}^m\, \ell\, \sigma', \overrightarrow{y_g \mapsto \ldots}\ \mathsf{via}\ \textsc{End-Abstraction}}{M, \Omega \vdash \emptyset \updownarrow (\vec{\sigma}') \leftarrow f_{\mathsf{back}(\rho)}\, \vec{e}\, e_{\mathsf{ret}};\ [\cdot] \dashv M', \Omega''}$$

T-Return-Backward
$$\frac{\begin{array}{c}\Omega_i \vdash x_{\mathsf{local},i} := \bot \rightsquigarrow () \dashv \Omega_{i+1} \\ \Omega_n(x_{\mathsf{ret}}) \Rightarrow v_{\mathsf{ret}} \qquad \{\mathsf{loan}, \bot, \mathsf{borrow}^r\} \notin v_{\mathsf{ret}} \\ \Omega' = \Omega_n[x_{\mathsf{ret}} \mapsto \mathsf{sym}(\rho, \sigma_{\mathsf{ret}}, v_{\mathsf{ret}})] \\ \Omega' \vdash \emptyset \uparrow^\rho E[\cdot] \dashv \Omega'' \qquad \Omega''[A(\rho)] = \{\vec{v}\} \\ \{\mathsf{loan}, \mathsf{proj}\} \notin \vec{v} \qquad \Omega \vdash \vec{v} \uparrow \vec{e}\end{array}}{\Omega_0 \vdash \mathsf{return} \uparrow^\rho E[\mathsf{ret}\ (\vec{e})]}$$

T-Fun-Backward
$$\frac{A(\rho_i) = \left\{\overrightarrow{\mathsf{proj}_{\mathsf{l}}\,(\sigma : \tau)}\right\} \qquad \vec{\sigma}\ \mathsf{fresh} \qquad \emptyset, \Omega \vdash s \uparrow^\rho e \qquad \Omega = \overrightarrow{A(\rho)}, \overrightarrow{x \mapsto \mathsf{proj}_{\mathsf{out}}\, \sigma}, \vec{x}_{\mathsf{local}} \mapsto \bot, x_{\mathsf{ret}} \mapsto \bot}{\mathsf{fn}\ f\ \langle\vec{\rho}\rangle\ (\vec{x} : \vec{\tau})\ (\vec{x}_{\mathsf{local}} : \vec{\tau}_{\mathsf{local}})\ (x_{\mathsf{ret}} : \tau_{\mathsf{ret}}) = s \uparrow^\rho \quad \mathsf{def}\ f_{\mathsf{back}(\rho)} = \lambda(\vec{\sigma} : \vec{\tau})(\sigma_{\mathsf{ret}} : \mathsf{proj}_\rho\, \tau_{\mathsf{ret}}).e}$$

T-Fun-Forward
$$\frac{A(\rho_i) = \left\{\overrightarrow{\mathsf{proj}_{\mathsf{l}}\,(\sigma : \tau)}\right\} \qquad \vec{\sigma}\ \mathsf{fresh} \qquad \emptyset, \Omega \vdash s \downarrow e \qquad \Omega = \overrightarrow{A(\rho)}, \overrightarrow{x \mapsto \mathsf{proj}_{\mathsf{out}}\, \sigma}, \vec{x}_{\mathsf{local}} \mapsto \bot, x_{\mathsf{ret}} \mapsto \bot}{\mathsf{fn}\ f\ \langle\vec{\rho}\rangle\ (\vec{x} : \vec{\tau})\ (\vec{x}_{\mathsf{local}} : \vec{\tau}_{\mathsf{local}})\ (x_{\mathsf{ret}} : \tau_{\mathsf{ret}}) = s \downarrow \quad \mathsf{def}\ f_{\mathsf{fwd}} = \lambda(\vec{\sigma} : \vec{\tau}).e}$$

T-Match-Concrete
$$\frac{\Omega(p) \xRightarrow{s} C[\overrightarrow{f = v}] \qquad M, \Omega \vdash s \updownarrow e}{M, \Omega \vdash \mathsf{match}\, p\ \mathsf{with}\ \ldots \mid C \to s \mid \ldots \updownarrow e}$$

T-Match-Symbolic
$$\frac{\Omega(p) \xRightarrow{s} (\sigma : t\ \vec{\tau}) \qquad \mathsf{type}\ t\ \vec{\alpha} = \overrightarrow{C[\overrightarrow{f : \vec{\tau}_f}]} \qquad \vec{\sigma}_i\ \mathsf{fresh} \qquad M, \left[\overrightarrow{(C_i[\overrightarrow{f_i : \vec{\sigma}_i}] : t\ \vec{\tau})}\middle/ (\sigma : t\ \vec{\tau})\right]\Omega \vdash s_i \updownarrow e_i}{M, \Omega \vdash \mathsf{match}\, p\ \mathsf{with}\ \overrightarrow{C \to s} \updownarrow \mathsf{match}\ \sigma\ \mathsf{with}\ \overrightarrow{C\ \vec{\sigma} \to e}}$$

Fig. 13. Functional Translation via our Symbolic Semantics

Project	General borrows	Return borrows	Loops	Closures, traits	Termination	I/O	Borrow check.	Extrinsic	Executable
Aeneas	✓	✓	-	-	✓	✓	✓	✓	✓
Electrolysis [Ullrich 2016]	-	-	✓	✓	✓	-	✓	✓	✓
Creusot [Denis et al. 2021]	✓	✓	✓	✓	-	-	-	-	-
Prusti [Wolff et al. 2021]	✓	✓	✓	✓	-	-	-	-	-

Fig. 14. **Comparison of Verification Frameworks Targeting Safe Rust**

General borrows: arbitrarily nested borrows and reborrows are allowed in function bodies; Electrolysis supports a very restricted subset of such operations. Return borrows: a function can return borrows; Electrolysis supports a very restricted subset of such functions. Loop: the framework has support for loops. Closures, traits: the framework has support for function pointers, closures and traits. Termination: the framework allows to reason about termination. I/O: we can reason about the external world like I/O. Borrow check.: the framework doesn't trust nor need the Rust borrow checker; Prusti extracts the lifetime information computed by the Rust borrow checker, then checks that those lifetime are correct. Extrinsic: the framework allows extrinsic proofs rather than intrinsic proofs; as stated, we believe such proofs are more amenable to modular reasoning and thus verification at scale. Executable: the framework generates an executable translation of the Rust program.

Supported Aeneas Features. Aeneas follows the design philosophy that no annotations should be added to the Rust code; therefore, we have devised a few concrete mechanisms to make integration of Aeneas-generated code easier.

Whenever translating a recursive function, Aeneas emits a `decreases` annotation in the generated F* code. The annotation refers to a yet-to-be-defined lemma that proves semantic termination. This allows for a natural style, wherein the user invokes Aeneas to generate e.g. `Foo.fst`; the file references `Foo.Lemmas.fst`, which is then filled out and maintained by the user. For Coq, we intend to rely on a fuel parameter controlled by the user using a similar fashion.

We also provide support for interacting with the external world. Users of Aeneas can choose to mark some modules in a given crate as "opaque"; the resulting functions appear as an interface file only, meaning that they are *assumed*. The user can then provide a hand-maintained library of lemmas that characterize the behavior of these external functions. We handle external dependencies in a similar fashion, generating an interface which contains exactly those external functions and types which are called from within the crate. This relieves the tool authors from having to maintain wrappers for the entire Rust standard library.

Finally, we provide support for enhancing our working monad with an external world. That is, rather than working in the error monad, we generate code for a combined world and error monad. Combined with the opaque feature, this allows the user to modularly state their assumptions about the world, and gives us in practice a lightweight monadic effect system.

6 CASE STUDIES

Hash Table. To assess the efficiency of Aeneas as a verification platform, we study a resizing hash table equipped with `insert`, `get` (immutable lookup), `get_mut` (mutable lookup) and `remove`. Each bucket is a linked list; `insert` replaces the existing binding, if any; resizing is automatic once a certain threshold is reached. Our table is polymorphic in the type of the stored elements, but right now, keys have a fixed type `usize` (the proofs do not rely on that). We plan to make the implementation generic over the key type once we support traits.

The functional property we prove is that the hash table functionally behaves like a map. In this regards, the interesting function is `insert`. This function delegates insertion to `insert_no_resize`, then checks if the max load has been reached, in which case it calls `try_resize`. We first prove that `insert_no_resize` behaves as expected, that is to say: the binding for the inserted key maps to the

new value, and all the other bindings are preserved. We then prove that `try_resize` is functionally the identity. Interestingly, a function like `try_resize` leverages many low-level features of Rust: we have to be wary of arithmetic overflows when computing the new size, we mutably borrow the slots vectors for in-place updates, and move elements so as not to perform reallocations.

```
1   fn try_resize<'a>(&'a mut self) {
2       // Check that we can resize (prevent overflows)
3       let capacity = self.slots.len();
4       let n = usize::MAX / 2;
5       if capacity <= n / self.max_load_factor.0 {
6           // Create a new table with a bigger capacity
7           let mut ntable = HashTable::new_with_capacity(
8               capacity * 2,
9               self.max_load_factor.0,
10              self.max_load_factor.1,
11          );
12          // Move the elements to the new table
13          HashTable::move_elements(&mut ntable, &mut self.slots, 0);
14          // Replace the current table with the new table
15          self.slots = ntable.slots;
16          self.max_load = ntable.max_load;
17      }
18  }
```

This proof is built on top of two invariants. First, in every slot, all the keys are pairwise disjoint. This allows us to prove that `insert_no_resize` is correct. Second, all the keys in a slot have the proper hash. Combined with the first invariant, it gives us that all the keys in the table are pairwise disjoint, which we need to prove that `try_resize` is correct. In combination with those invariants, stated in `hash_map_t_inv`, we alternate between two high-level views of the hash table. The first view is a list of lists, which corresponds to the low-level structure of the hash table in rust. The second, which we use in the proof of `try_resize`, is a flattened associative list.

Due to the low-level nature of Rust, precise arithmetic preconditions pop up in the specification of the hash table. More specifically, inserting in the hash table doesn't fail, if and only if the internal counter which tracks the number of entries doesn't overflow.

```
1   val hash_map_insert_fwd_lem (#t : Type) (self : hash_map_t t) (key : usize) (value : t) :
2     Lemma (requires (hash_map_t_inv self)) (ensures (
3       match hash_map_insert_fwd t self key value with
4       | Fail -> (* We fail only if: *)
5         None? (find_s self key) /\ (* the key is not already in the map *)
6         size_s self = usize_max (* and we can't increment `num_entries` *)
7       | Return hm' -> (* In case of success: *)
8         hash_map_t_inv hm' /\ (* The invariant is preserved *)
9         find_s hm' key == Some value /\ (* [key] maps to [value] *)
10        (forall k'. k' <> key ==> find_s hm' k' == find_s self k') /\ (* Other bindings unchanged *)
11        (match find_s self key with (* The size is incremented, iff we inserted a new key *)
12         | None -> size_s hm' = size_s self + 1
13         | Some _ -> size_s hm' = size_s self)))
```

By virtue of working with a (translated) pure program, we were able to focus on the *functional* behavior of the hash table and the important proof obligations, such as the absence of arithmetic overflows, rather than memory reasoning. Furthermore, the functional translation allowed us to prove the specifications in an *extrinsic* style, where we establish lemmas about the behavior of existing functions, rather than in an *intrinsic* style, where we sprinkle the Rust code with assertions, calls to auxiliary lemmas, and possibly materialize extra variables to aid reasoning. We find that the extrinsic style not only yields a modular proof, but also allows the control-flow of the proof to

refine the control-flow of the code. For instance, `insert_no_resize` exhibits two logically different behaviors (key is present or not), even though the code does not branch. Rather than materialize two run-time code paths with different assertions, we perform the inversion in the proof only.

The proofs took a total of 4 person-days, for an implementation of 201 LoC without blanks and comments. We were hindered by some design choices of F*, which generates proof obligations for the Z3 SMT solver; this mode of operation is well-suited to intrinsic reasoning, but there is right now no way for the user to have an interactive proof context like Coq. We are eager to investigate the usability of Aeneas when coupled with backends that rely on true interaction with tactics like Coq, and appear better suited for this kind of extrinsic, functional proofs.

To the best of our knowledge, our implementation is the first verified hash table in Rust. To obtain points of comparison, we therefore asked other non-Rust verification experts or tool authors to estimate the effort to verify a hash table using their respective frameworks. VST's hash table exercise [Andrew W Appel 2021] can be reasonably completed by students in three days; this is, however, a non-resizing hash table. A hash table verified using CFML [Pottier 2017] required a week of work to establish functional correctness; the original code, however, is in OCaml, which is a higher-level language than C. Based on these comparisons, we conclude that Aeneas is very competitive with other, more established verification frameworks.

A final point of comparison is with other data structures in Low*, the subset of F* that compiles to C. A recent paper [Ho et al. 2021] verifies an imperative map using an associative linked list. The authors reveal that this required several weeks of full-time work, so we can confidently claim that we vastly improve the state of the art for verifying low-level programs in F*.

Non-Lexical Lifetimes (Polonius). We recently implemented a Bϵ-tree [Bender et al. 2015], and ran it through Aeneas; our verification efforts are ongoing. In the process of doing so, we bumped into a limitation of the current Rust borrow checking algorithm; interestingly, this limitation does not appear with Aeneas, owing to our semantic checking of borrows.

The `get_suffix_at_x` function, below, looks for an element in a list, and returns a mutable borrow to the suffix of this list starting at this element, for in-place modifications in a C-style fashion. The current Rust borrow-checker, based on lexical lifetimes, is too coarse to notice that the `Cons` branch is valid. More specifically, it considers that the reborrows performed through `hd` and `tl` should last until the end of the lexical scope, that is, until the end of the `Cons` branch. We inserted the error message printed by Rust in the comments.

Instead, one may notice that is it possible to end those borrows earlier, after evaluating the conditional, in order to retrieve full ownership of the value borrowed by `ls` in the first branch of the `if`, and make the example type-check. The ongoing replacement of the borrow-checker in Rust, named Polonius, implements a more refined lifetime analysis, and accepts this program. However, the status of Polonius is unclear.

More interestingly, our semantic approach of borrows makes this program type-check without issues; since our discipline is not lexical, but based on symbolic execution and a semantic approach to loans, we accept the example without troubles, and are resilient to further syntactic tweaks of the program.

```
1   fn get_suffix_at_x<'a>(ls: &'a mut List<u32>, x: u32) -> &'a mut List<u32> {
2       match ls {
3           Nil => { ls }
4           Cons(hd, tl) => { // first mutable borrow occurs here
5               if *hd == x { ls // second mutable borrow occurs here
6               } else { get_suffix_at_x(tl, x) } } } }
```

I/O and External Dependencies. Real-world applications rely on external libraries and often need to interact with the external world through I/O or sockets. We elegantly model interaction with the outside environment using opaque modules, and a state type that combines memory, IO and the outside world. In other words, Aeneas allows reasoning about such applications by lifting the generated code into a combined-state + error monad, and relying on module signatures to model the interaction with external functions.

When we designate a module as opaque, Aeneas treats all the definitions coming from this module and reachable from the root module as opaque, and generates an interface file accordingly, that is, with declarations (`val`s), but no definitions (no `let`s). The user is then free to provide models for those declarations, or simply state assumptions by means of assumed lemmas. This feature illustrates why a modular, type-directed translation like Aeneas' is important: the user doesn't need to reveal any information about the function's definition (or model its behavior using a specification language); rather, the user can work post-translation in the comfort of their favorite theorem prover. Moreover, while it is possible to add annotations to function signatures in a local crate, this possibility falls short when it comes to dealing with external dependencies, over which the user has no control! Handling this oftentimes requires tediously wrapping such dependencies in properly annotated modules. In contrast, this work, and Charon and Aeneas, simply treat the external dependencies as opaque by looking up the types and functions that are needed in the (non-opaque modules of the) local crate, and generating corresponding declarations.

Finally, using a state-error monad allows us to introduce stateful reasoning when this is really needed, for instance when the code uses I/O functions. The `state` type, which models the external world, is also an opaque type for which the user is free to provide a model or write assumptions, in a fashion similar to opaque modules. In practice, this gives us a lightweight effect system.

Let us illustrate those possibilities with the following example. We set out to serialize our earlier hash table to the disk. To account for this, we author `serialize` and `deserialize` functions in a separate opaque module outside of the scope of verification. We mark the module as opaque, meaning Aeneas generates the following declarations. First, `insert_on_disk`, below, simply loads the map from the disk, inserts a new entry, and stores the updated table back on disk.

```
1  fn insert_on_disk(key: Key, value: u64) {
2      let mut hm = deserialize();
3      hm.insert(key, value);
4      serialize(hm); }
```

Aeneas generates the following declarations to model the disk state and the serialization and deserialization functions.

```
1  type state : Type
2  val deserialize_fwd : state -> result (state & hash_map_t u64)
3  val serialize_fwd : hash_map_t u64 -> state -> result (state & unit)
```

Those definitions generate the following translation of `insert_on_disk`:

```
1  let insert_on_disk_fwd (key : usize) (value : u64) : state -> result (state & unit) =
2    hm <-- hashmap_utils_deserialize_fwd;
3    hm <-- hash_map_insert_fwd_back u64 hm key value;
4    _ <-- hashmap_utils_serialize_fwd hm;
5    return ()
```

In order to reason about the function above, we define the (assumed) lemmas below, which capture our assumption that once we store a table on the disk, all subsequent loads return the same table, provided they succeed.

```
1  val state_v : state -> hash_map_t u64
2
3  val serialize_lem (hm : hash_map_t u64) (st : state) : Lemma (
```

```
4     match serialize_fwd hm st with | Fail -> True | Return (st', ()) -> state_v st' == hm)
5
6   val deserialize_lem (t : Type) (st : state) : Lemma (
7     match deserialize_fwd st with
8     | Fail -> True | Return (st', hm) -> hm == state_v st /\ state_v st' == state_v st)
```

Relying on those lemmas, we can prove properties like the one below:

```
1   val insert_on_disk_fwd (key : usize) (value : u64) (st : state) : Lemma (
2     match hash_map_main_insert_on_disk_fwd key value st with
3     | Fail -> True
4     | Return (st', ()) ->
5       let hm = state_v st' in
6       match hash_map_insert_fwd_back u64 hm key value with
7       | Fail -> False
8       | Return hm' -> hm' == state_v st')
```

Precise Reborrows (Discussion). Due to its precise management of borrows, we explained above that Aeneas can handle programs which are supported by Polonius, and not by the current implementation of the borrow checker. Because of the way we handle reborrows, there are actually cases of programs deemed invalid even by Polonius, but supported by Aeneas.

For instance, in the example below, we first create a shared borrow that we store in pp at line 2, then a reborrow of a subvalue borrowed by pp that we store in px at line 5. Upon evaluating the assignment at line 9, px simply maps to a shared borrow of the first component of p (environment at lines 6-8). Importantly, even though px reborrows part of pp, there are no links between px and pp: our semantics does not track the hierarchy between borrows and their subsequent reborrows. In other words, line 5 is equivalent to `let px = &p.0`, where we borrow directly from p without resorting to pp. This implies that, upon ending the borrow ℓ_p stored in pp at line 9, we do not need to end ℓ_x stored in px, which in turn allows us to legally evaluate the assertion at line 13.

```
1   let mut p = (0, 1);
2   let pp = &p;
3   // p  ↦ loanˢ {ℓₚ} (0, 1)
4   // pp ↦ borrowˢ ℓₚ
5   let px = &(*pp.0);
6   // p  ↦ loanˢ {ℓₚ} (loanˢ {ℓₓ} 0, 1)
7   // pp ↦ borrowˢ ℓₚ
8   // px ↦ borrowˢ ℓₓ
9   p.1 = 2;
10  // p  ↦ (loanˢ {ℓₓ} 0, 2)
11  // pp ↦ ⊥
12  // px ↦ borrowˢ ℓₓ
13  assert!(*px == 1);
```

When we attempt to borrow check this program (with Polonius or the current implementation of the borrow checker), the Rust borrow checker considers that px reborrows pp, and thus needs to end before pp ends. It consequently fails with the following error message:

```
cannot assign to p.1 because it is borrowed
   |
 2 |      let pp = &p;
   |               -- borrow of p.1 occurs here
 5 |      let px = &(pp.0);
 9 |      p.1 = 2;
   |      ^^^^^^^ assignment to borrowed p.1 occurs here
13 |      assert!(*px == 1);
   |               --- borrow later used here
```

The code snippet below illustrates a similar example with mutable borrows. We create a borrow $px1$ of (the value of) x at line 2, then reborrow this value through $px2$ at line 3. At line 8, we then update $px1$ to borrow y. At this point, $px2$ still borrows x. The important point to notice is that upon performing this update, we remember the old value of $px1$ in the ghost variable $px1_{old}$ to not lose information about the borrow graph (environment at lines 9-13). Similarly to the previous example with shared borrows, the resulting environment doesn't track the fact that $px2$ was created by reborrowing the value initially borrowed by $px1$: there are no links between those two variables. Consequently, upon ending borrow ℓ_y (stored in $px1$) to access y at line 16, we don't need to end ℓ_2 (stored in $px2$). This in return allows us to legally derefence $px2$ at line 22.

```
1   let mut x = 0;
2   let mut px1 = &mut x;
3   let px2 = &mut (*px1); // Reborrow: px2 now borrows (the value of) x
4   // x     ↦ loanᵐ ℓ₁
5   // px1 ↦ borrowᵐ ℓ₁ (loanᵐ ℓ₂)
6   // px2 ↦ borrowᵐ ℓ₂ 0
7   let mut y = 1;
8   px1 = &mut y; // Update px1 to borrow y instead of x
9   // x       ↦ loanᵐ ℓ₁
10  // px1_old ↦ borrowᵐ ℓ₁ (loanᵐ ℓ₂)
11  // px2     ↦ borrowᵐ ℓ₂ 0
12  // y       ↦ loanᵐ ℓ_y
13  // px1     ↦ borrowᵐ ℓ_y 1
14  assert!(*px1 == 1);
15  assert!(*px2 == 0);
16  assert!(y == 1); // End the borrow of y through px1 (shouldn't impact px2!)
17  // x       ↦ loanᵐ ℓ₁
18  // px1_old ↦ borrowᵐ ℓ₁ (loanᵐ ℓ₂)
19  // px2     ↦ borrowᵐ ℓ₂ 0
20  // y       ↦ 1
21  // px1     ↦ ⊥
22  assert!(*px2 == 0); // Considered invalid by rustc, but accepted by Aeneas
```

When attempting to borrow check this code snippet, we get the following error:

```
cannot use y because it was mutably borrowed
   |
8  |       px1 = &mut y; // Update px1 to borrow y instead of x
   |             ------ borrow of y occurs here
...
16 |       assert!(y == 1); // End the borrow of y through px1 (shouldn't impact px2!)
   |               ^ use of borrowed y
22 |       assert!(*px2 == 0); // Considered invalid by rustc, but accepted by Aeneas
   |               ---- borrow later used here
```

The two examples above exemplify cases where the Rust borrow checker deems a program as invalid, while Aeneas accepts it. We do not claim that this is a strong limitation of the Rust borrow checker: these use cases seem quite anecdotal and are probably useless in practice. However, we believe the ability of Aeneas to precisely capture the behavior of such use cases supports our claim that our semantics really captures the essence of the borrow mechanism.

7 FUTURE WORK

We have admitted many of Aeneas' current limitations through this paper. We plan to address loops and disjunctions in the control-flow as soon as possible, using the techniques we referenced earlier. Doing so should bring us closer to feature-parity with Creusot and Prusti. Next are traits,

our Coq backend, and large-scale use-cases. We believe all of the above to be engineering tasks; the semantic insights are in this paper.

Our formalization provides a precise semantics of ownership in Rust; as we alluded to earlier, we can explain not only extensions of the Rust borrow-checker (Polonius), but trickier programs that Polonius cannot yet account for. Our next unit of work is a proof of soundness of our semantics and translation, possibly against Stacked Borrows and/or RustBelt. Doing so would establish Aeneas as an alternate borrow checker for Rust, possibly informing future evolutions.

We now review in detail the remaining restrictions about the subset of Rust we can handle. We believe function pointers and closures fit naturally within our approach. Our reasoning is entirely type-based, so a function can be fully characterized by a forward function and a list of backward functions, one for each region variable (lifetime). From there, adding support for traits should mostly be a matter of engineering.

Our current restriction about nested borrows in function types is overly conservative, and aims to simplify and unambiguously rule out the truly difficult case, which is that of nested mutable borrows living in different regions, e.g., `&'a mut &'b mut T`. Manipulating such functions is made difficult by operations like what we dub *borrow overwrites*. Borrow overwrites happen when we update an *inner* borrow so as to borrow a different value, as exemplified by line 10 in the example below. Such operations are hard to account for in our abstract, symbolic semantics.

We extended our rules to handle such cases, but testing those in our implementation revealed they were slightly imprecise. More specifically and as a teaser, Aeneas was able to perform a symbolic execution on the example below *without* line 12. However, adding the chained function call would eventually lead to a state where the interpreter is unable to legally apply any rule because the borrow graph is too imprecise, making it get stuck and fail. Further investigation seemed to indicate that our region abstractions are too monolithic and need to be split into *subregion* abstractions. We will investigate this solution in future work.

```
1   fn f<'a, 'b>(ppx : &'a mut &'b mut u32) : &'a mut &'b mut u32;
2
3   let mut x = 0;
4   let mut px = &mut x;
5   let ppx = &mut px;
6   let mppx = &mut (*ppx);
7   let ppy = f(move mppx); // First function call
8   **ppy = 1;
9   let mut y = 2;
10  *ppy = &mut y; // Borrow overwrite - hard to track because ppy was returned by a function call
11  let mppy = &mut (*ppy);
12  let ppz = f(&move mppy); // Chained function call
```

For the time being, having support for nested mutable borrows doesn't seem to be extremely relevant in practice, with regards to the expressivity of the subset we support. It often happens that Rust programmers manipulate functions which receive as input a mutable borrow of a structure which contains itself mutable borrows. However, such functions generally do not perform borrow overwrites, in which case it is possible to split such nested borrows into non-nested borrows, i.e., split the input parameter into several input parameters. In other words, the semantic expressivity provided by nested mutable borrows doesn't seem to be exploited much by Rust developers. We plan to investigate if there are actual use cases which rely on this feature.

Our lack of support for borrows within ADTs mostly comes from engineering concerns. When an ADT contains borrows, ending those borrows requires to give back *exactly* those values which were borrowed. This sometimes requires introducing some "backward" type definitions, which only contain such fields. We forbid the instantiation of type parameters with types containing borrows

for the same reasons, and to control the introduction of nested borrows in a conservative manner. Moreoever, instantiating a type parameter with a type containing mutable borrows transforms any function into a function which requires the generation of backward translations. This can be addressed by introducing a "generic" backward function for every type parameter. In short, this would require generating higher-order backward functions. Again, doing so will require a fair amount of engineering time.

A final unit of future work is to strengthen the Charon tool; many authors of Rust verification tools seem to re-implement comparable compiler plugins. Sharing engineering efforts can only benefit the wider community.

8 RELATED WORK; CONCLUSION

Electrolysis [Ullrich 2016] most resembles Aeneas. The tool translates Rust programs to a pure lambda-calculus, then targets the Lean [Moura et al. 2015] proof assistant. It relies on lenses to model mutable borrows, and as such comes with severe restrictions (Figure 14); for instance, functions may only return borrows to their first argument. Electrolysis does not come with a formal model, and thus does not make a case for semantic correctness. As such, it resembles a very pragmatic "transpiler" rather than a compiler; for instance, traits map to type classes, because they, at a high-level, work in a similar fashion.

RustBelt [Jung et al. 2017] targets a different problem than Aeneas: proving the soundness of Rust's type system, and proving the correctness of unsafe Rust programs using the Iris framework [Jung et al. 2018]. RustBelt is an impressive framework and allows composing safe code with unsafe code using a notion of semantic typing. We see RustBelt as the exact complement of Aeneas: RustBelt allows proving fiendishly difficult, small pieces of unsafe Rust code, while Aeneas allows reasoning about large amounts of safe Rust, without resorting to a full-fledged framework like Iris.

RustHorn [Matsushita et al. 2020] uses a device called *prophecy variables* to generate a pure, non-executable logical encoding of Rust programs. The original paper contains a non-mechanized proof that their logical encoding is sound with respects to a memory-based semantics of the original Rust program. RustHornBelt [Matsushita et al. 2022] uses the RustBelt framework to mechanically prove the soundness of the RustHorn-style logical encoding. We plan to investigate using this style of proof to mechanically establish the soundness of our ownership-centric semantics.

Creusot [Denis et al. 2021] is a tool that builds on RustHornBelt to generate proof obligations that can then be discharged to SMT; the authors have a proof of soundness formalized in Coq for a simplified model of MIR. Their design chooses *automated*, *intrinsic* proofs: they introduce an annotation language for specifications, wrap a large part of the standard library in it, then rely on requires/ensures clauses and annotations to perform the proofs. This style emphasizes a logical encoding as opposed to an executable specification, and a closed-world approach where the verified code cannot naturally be integrated as part of, e.g., a large Coq development. The advantage of this style is that they can easily require annotations for, e.g., loop invariants.

Prusti's frontend [Astrauskas et al. 2019] is very similar to Creusot's. The tool uses Rust's type system to guide the application of rules in Viper [Juhasz et al. 2014], which means they rely on the Rust borrow checker for lifetime inference but do not need to trust its results. Doing so, they automate the application of memory reasoning rules and thus avoid general-purpose memory proof search. Creusot, however, by virtue of its dedicated encoding that directly leverages lifetime information, appears to offer better verification performance than the Prusti frontend for the general-purpose Viper tool [Denis et al. 2021, §5.3].

Our presentation of a low-level language equipped with a system of permissions, followed by an embedding into a theorem prover, is reminiscent of RefinedC [Sammler et al. 2021]. RefinedC relies on magic wands to make up for the lack of borrows; wands, by virtue of being very general, require

the use of heuristics. RefinedC, however, focuses on the subset of C code that obeys its permission discipline; and it relies on a memory model in Coq rather than a functional translation. RefinedC is foundational, and requires user annotations in an intrinsic style, while Aeneas generates a trusted, pure translation for extrinsic proofs. Crucially, both RefinedC and Aeneas rely on the fact that they never need to backtrack after applying rules. For RefinedC, this is made possible by restricting the user annotations to a carefully crafted, yet extensible, fragment of separation logic, along with alias types in the style of Mezzo's permissions [Protzenko 2014]. For Aeneas, we never backtrack because we leverage the borrow mechanism to lazily terminate borrows.

Aeneas is inspired by the Mezzo language [Pottier and Protzenko 2013]. In Mezzo, the type-checker is flow-sensitive and operates by maintaining a permission environment for each program point, possibly losing information for function calls and disjunctions in the control-flow. This design is similar to our symbolic semantics. This also further illuminates our earlier claim, which is that our symbolic interpreter acts as a borrow-checker, and further encourages us to investigate a standalone Rust borrow-checker based on the Aeneas formalism.

Heapster [He et al. 2021], just like Aeneas, attempts to extract pure code from a low-level program. In the case of Heapster, the input is LLVM internal code, and the output is logical (non-executable) specifications. The user must guide extraction by adding type annotations and loop invariants to their programs. A key difference lies in the way they handle pointers to elements inside recursive data-structures, a typical use case for borrows in Rust, by requiring the user to define what they dub *reachability permissions*. Those permissions act as predicates describing the link between a pointer and the value it points to inside a structure, and allow them to reason about the effect of destructive updates. Reasoning-wise, reachability permissions fulfill a role similar to our backward functions. With Aeneas, however, we leverage Rust's borrow discipline to automate the generation of those functions.

Cogent [Amani et al. 2016; O'Connor et al. 2021] is a domain-specific language equipped with a linear type system. The Cogent compiler produces: C code; a high-level Isabelle/HOL specification; and a proof of refinement from the former to the latter. By virtue of producing an Isabelle/HOL specification, Cogent seamlessly composes with existing developments in that language, and can thus be integrated into a larger project, something Aeneas also enables. However, unlike Aeneas, the Cogent compiler does not need to be trusted since it produces a proof of translation correctness for each compilation run. We also remark that the linear type system of Cogent is significantly less expressive than Rust's; notably, Cogent does not seem to allow an equivalent of mutable borrows.

Stacked Borrows [Jung et al. 2019] give a semantics to the notion of borrows in Rust, but sets out to achieve different goals than Aeneas: namely, to provide a set of rules that Rust developers can follow and validate their code against when writing unsafe code. The work comes with an extensive evaluation, which establishes both that the tool can detect incorrect uses (bugs were found), and that it can prove that some optimizations written using unsafe code are correct. This work adopts a very low-level view of memory, and it is unclear whether it can be used productively at the scale that we envision for Aeneas. The value of the work, however, lies in its precise, memory-based semantics of borrows; we are evaluating the feasibility of proving our semantics against it.

Figure 14 compares some of the tools above to Aeneas. If anything, the table reveals that each tool adopts a unique stance on what kind of programs they aim to verify, and with what kind of toolchain. For Aeneas, the stance is as follows: we target safe Rust programs, and we believe in extrinsic reasoning. Doing so, we hit what we believe is a "sweet spot", where the functional encoding is lightweight and accessible, and where proof engineers can be most productive.

ACKNOWLEDGMENTS

REFERENCES

2017. Not possible to bind to a pattern. https://github.com/FStarLang/FStar/issues/1288.

2021. StackOverflow Developer Survey. https://insights.stackoverflow.com/survey/2021.

2022. Verus, an Experimental Verification Framework for Rust-like code. https://github.com/secure-foundations/verus/blob/004eadd8c31b60c886f7a8c8b568806e46a49a78/source/docs/design/returning-mutable-references.md.

Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. *SIGARCH Comput. Archit. News* 44, 2 (mar 2016), 175–188. https://doi.org/10.1145/2980024.2872404

Qinxiang Cao Andrew W Appel, Lennart Beringer. 2021. Verifiable C. https://softwarefoundations.cis.upenn.edu/vc-current/index.html.

V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification, In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). *Proc. ACM Program. Lang.* 3, OOPSLA, 147:1–147:30. https://doi.org/10.1145/3360573

Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2016. The design and formalization of Mezzo, a permission-based programming language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 38, 4 (2016), 1–94.

Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. 2015. An Introduction to Bϵ-trees and Write-Optimization. *login Usenix Mag.* 40 (2015).

Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, et al. 2017. Everest: Towards a verified, drop-in replacement of HTTPS. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Aaron Bohannon, J Nathan Foster, Benjamin C Pierce, Alexandre Pilkiewicz, and Alan Schmitt. 2008. Boomerang: resourceful lenses for string data. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 407–419.

John Boyland, James Noble, and William Retert. 2001. Capabilities for sharing. In *European Conference on Object-Oriented Programming*. Springer, 2–27.

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W Appel. 2018. VST-Floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning* 61, 1 (2018), 367–422.

Arthur Charguéraud and François Pottier. 2008. Functional translation of a calculus of capabilities. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. 213–224.

David G Clarke, John M Potter, and James Noble. 1998. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 48–64.

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2021. *The Creusot Environment for the Deductive Verification of Rust Programs*. Research Report RR-9448. Inria Saclay - Île de France. https://hal.inria.fr/hal-03526634

Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 287–305.

Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear regions are all you need. In *European Symposium on Programming*. Springer, 7–21.

Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps:{End-to-End} Security via Automated {Full-System} Verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 165–181.

Paul He, Eddy Westbrook, Brent Carmer, Chris Phifer, Valentin Robert, Karl Smeltzer, Andrei Ştefănescu, Aaron Tomb, Adam Wick, Matthew Yacavone, and Steve Zdancewic. 2021. A Type System for Extracting Functional Specifications from Memory-Safe Imperative Programs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 135 (oct 2021), 29 pages. https://doi.org/10.1145/3485512

Son Ho and Jonathan Protzenko. 2022a. Aeneas: A Verification Toolchain for Rust Programs. https://github.com/sonmarcho/aeneas.

Son Ho and Jonathan Protzenko. 2022b. *Aeneas: Rust Verification by Functional Translation.* https://doi.org/10.5281/zenodo.6672939

Son Ho and Jonathan Protzenko. 2022c. Aeneas: Rust Verification by Functional Translation (Long Version). https://doi.org/10.48550/ARXIV.2206.07185

Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. 2021. Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations. (2021).

Graydon Hoare. 2022. https://twitter.com/graydon_pub/status/1492792051657629698.

Uri Juhasz, Ioannis T Kassios, Peter Müller, Milos Novacek, Malte Schwerhoff, and Alexander J Summers. 2014. *Viper: A verification infrastructure for permission-based reasoning.* Technical Report. ETH Zurich.

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked borrows: an aliasing model for Rust. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.

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

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles.* 207–220.

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning.* Springer, 348–370.

Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2022. Linear Types for Large-Scale Systems Verification. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).*

Jacob R Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R Wilcox, and Xueyuan Zhao. 2020. Armada: low-effort verification of high-performance concurrent programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 197–210.

Niko Matsakis. 2018. Regions are Sets of Loans. http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/.

Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHorn-Belt: A semantic foundation for functional verification of Rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2020. RustHorn: CHC-Based Verification for Rust Programs.. In *ESOP.* 484–514.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction.* Springer, 378–388.

LIAM O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. Cogent: uniqueness types and certifying compilation. *Journal of Functional Programming* 31 (2021).

François Pottier. 2017. Verifying a hash table and its iterators in higher-order separation logic. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs.* 3–16.

François Pottier and Jonathan Protzenko. 2013. Programming with permissions in Mezzo. *ACM SIGPLAN Notices* 48, 9 (2013), 173–184.

Jonathan Protzenko. 2014. *Mezzo: a typed language for safe effectful concurrent programs.* Ph. D. Dissertation. Université Paris Diderot-Paris 7.

Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, et al. 2017. Verified low-level programming embedded in F. *Proc. ACM program. lang.* 1, ICFP (2017), 17–1.

John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science.* IEEE, 55–74.

Xavier Rival. 2011. Abstract Domains for the Static Analysis of Programs Manipulating Complex Data Structures. *Habilitation à diriger des recherches, École Normale Supérieure* (2011).

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 158–174.

The Rust Compiler Team. 2021. The Polonius Book. https://rust-lang.github.io/polonius/.

The Rust Compiler Team. 2022. Guide to rustc development. https://rustc-dev-guide.rust-lang.org/borrow_check/two_phase_borrows.html.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and computation* 132, 2 (1997), 109–176.

Sebastian Ullrich. 2016. Simple verification of rust programs via functional purification. *Master's Thesis, Karlsruher Institut für Technologie (KIT)* (2016).

Philip Wadler. 1990. Linear types can change the world!. In *Programming concepts and methods*, Vol. 3. Citeseer, 5.

Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. 2021. Modular Specification and Verification of Closures in Rust. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 145 (oct 2021), 29 pages. https://doi.org/10.1145/3485522