RustBelt: Securing the Foundations of the Rust Programming Language

RALF JUNG, MPI-SWS*
JACQUES-HENRI JOURDAN, MPI-SWS*
ROBBERT KREBBERS, Delft University of Technology
DEREK DREYER, MPI-SWS*

Rust is a new systems programming language that promises to overcome the seemingly fundamental tradeoff between high-level safety guarantees and low-level control over resource management. Unfortunately, none of Rust's safety claims have been formally proven, and there is good reason to question whether they actually hold. Specifically, Rust employs a strong, ownership-based type system, but then extends the expressive power of this core type system through libraries that internally use unsafe features. In this paper, we give the first formal (and machine-checked) safety proof for a language representing a realistic subset of Rust. Our proof is extensible in the sense that, for each new Rust library that uses unsafe features, we can say what verification condition it must satisfy in order for it to be deemed a safe extension to the language. We have carried out this verification for some of the most important libraries that are used throughout the Rust ecosystem.

1 INTRODUCTION

Systems programming languages like C and C++ give programmers low-level control over resource management at the expense of safety, whereas most other modern languages give programmers safe, high-level abstractions at the expense of control. It has long been a "holy grail" of programming languages research to overcome this seemingly fundamental tradeoff and design a language that offers programmers both high-level safety *and* low-level control.

Rust [41, 59], developed at Mozilla Research, comes closer to achieving this holy grail than any other industrially supported programming language to date. On the one hand, like C++, Rust supports zero-cost abstractions for many common systems programming idioms and avoids dependence on a garbage collector [52, 62]. On the other hand, like most modern high-level languages, Rust is type-safe and memory-safe. Furthermore, Rust's type system goes beyond that of the vast majority of safe languages in that it statically rules out data races (which are essentially a form of undefined behavior for concurrent programs), as well as common programming pitfalls like iterator invalidation [25]. In other words, compared to mainstream "safe" languages, Rust offers both lower-level control *and* stronger safety guarantees.

At least, that is the hope. Unfortunately, none of Rust's safety claims have been formally proven, and there is good reason to question whether they actually hold. In this paper, we make a major step toward rectifying this situation by giving the first formal (and machine-checked) safety proof for a language representing a realistic subset of Rust. Before explaining our contributions in more detail, and in particular what we mean here by "realistic", let us begin by exploring what makes Rust's type system so unusual, and its safety so challenging to verify.

1.1 Rust's "extensible" approach to safe systems programming

At the heart of the Rust type system is an idea that has emerged in recent years as a unifying concept connecting both academic and mainstream language design: *ownership*. In its simplest form,

* Saarland Informatics Campus.
2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the idea of ownership is that, although multiple aliases to a resource may exist simultaneously, performing certain actions on the resource (such as reading and writing a memory location) should require a "right" or "capability" that is uniquely "owned" by one alias at any point during the execution of the program. Although the right is uniquely owned, it can be "transferred" from one alias to another—*e.g.*, upon calling a function or spawning a thread, or via synchronization mechanisms like locks. In more complex variations, ownership can be *shared* between aliases, but only in a controlled manner (*e.g.*, if shared ownership only permits read access [14]). In this way, ownership allows one to carefully administer the safe usage of potentially aliased resources.

Ownership pervades both academic and mainstream language design for safe(r) systems programming. On the academic side, many proposals have been put forth for using *types* to enforce various ownership disciplines, including "ownership type" systems [16]; region- or typestate-based systems for "safe C" programming in languages like Cyclone [27] and Vault [18]; and substructural type systems like Ynot [44], Alms [61], and Mezzo [9]. Unfortunately, although these languages provide strong safety guarantees, none of them have made it out of academic research into mainstream use. On the mainstream side, "modern C++" (*i.e.*, C++ since the 2011 standard [1]) provides several features—*e.g.*, smart pointers, move semantics, and RAII (Resource Acquisition Is Initialization)—that are essentially mechanisms for controlling ownership. However, while these features encourage safer programming idioms, the type system of C++ is too weak to enforce its ownership disciplines statically, so it is still easy to write programs with unsafe or undefined behavior using these features.

In some sense, the key challenge in developing sound static enforcement of ownership disciplines—and the reason perhaps that academic efforts have not taken off in practice—is that no language can account for the safety of every advanced form of low-level programming that one finds in the wild, because there is no practical way to do so while retaining automatic type checking. As a result, previous designs employ type systems that are either too restrictive (*i.e.*, preventing programmers from writing certain kinds of low-level code they want to write) or too expressive (*i.e.*, encoding such a rich logic in the type structure that programmers must do proofs to appease the type checker).

Rust addresses this challenge by taking a hybrid, *extensible* approach to ownership.

The basic ownership discipline enforced by Rust's type system is a simple one: If ownership of an object (of type `T`) is shared between multiple aliases ("shared references" of type `&T`), then none of them can be used to directly mutate it. This discipline, which is similar in spirit to (if different in detail from) that of several prior academic approaches, is enforceable automatically and eliminates a wide range of common low-level programming errors, such as "use after free", data races, and iterator invalidation. However, it is also too restrictive to account for many low-level data structures and synchronization mechanisms, which fundamentally depend on the ability to mutate aliased state (*e.g.*, to implement mutual exclusion or communication between threads).

Consequently, to overcome this restriction, the implementations of Rust's standard libraries make widespread use of `unsafe` operations, such as "raw pointer" manipulations for which aliasing is not tracked. The developers of these libraries claim that their uses of `unsafe` code have been properly "encapsulated", meaning that if programmers make use of the APIs exported by these libraries but otherwise avoid the use of `unsafe` operations themselves, then their programs should never exhibit any unsafe/undefined behaviors. In effect, these libraries extend the expressive power of Rust's type system by loosening its ownership discipline on aliased mutable state in a modular, controlled fashion: Even though a shared reference of type `&T` may not be used to *directly* mutate the contents of the reference, it may nonetheless be used to *indirectly* mutate them by passing it to one of the observably "safe" (but internally `unsafe`) methods exported by the object's API.

However, there is cause for concern about whether Rust's extensible approach is actually sound. Over the past few years, several soundness bugs have been found in Rust, both in the type system

itself [10, 11, 63] and in libraries that use `unsafe` code [12, 13, 28]. Some of these—such as the Leakpocalypse bug [12]—are quite subtle in that they involve an *interaction* of multiple libraries, each of which is (or seems to be) perfectly safe on its own. To make matters worse, the problem cannot easily be contained by blessing a fixed set of standard libraries as primitive and just verifying the soundness of those; for although it is considered a badge of honor for Rust programmers to avoid the use of `unsafe` code entirely, many nevertheless find it necessary to employ a sprinkling of `unsafe` code in their developments. Of course, it is not unusual for safe languages to provide unsafe escape hatches (*e.g.*, Haskell's `unsafePerformIO`, OCaml's `Obj.magic`) to work around limitations of their type systems. But `unsafe` code plays such a fundamental role in Rust's extensible ownership discipline that it cannot simply be swept aside if one wishes to give a realistic formal account of the language.

The question remains: How can we verify that Rust's extensible approach makes any sense? The standard technology for proving safety properties for high-level programming languages—namely, Wright and Felleisen's method of "progress and preservation" [66]—does not apply to languages in which one can mix safe and unsafe code. (Progress and preservation is a *closed-world* method, which assumes the use of a closed set of typing rules. This assumption is fundamentally violated by Rust's extensible, open-world approach.) So, to account for safe-unsafe interaction, we need a way to specify formally what we are obliged to *prove* if we want to establish that a library employing `unsafe` code constitutes a sound extension of the Rust type system. Luckily, decades of research in semantics and verification have provided us with just the right tools for the job.

1.2 RustBelt: An extensible, semantic approach to proving soundness of Rust

In this paper, we give the first formal (and machine-checked) account of Rust's extensible approach to safe systems programming and how to prove it sound.

For obvious reasons of scale, we do not consider the full Rust language, for which no formal description exists anyway. Instead, after beginning (in §2) with an example-driven tour of the most central and distinctive features of the Rust type system, we proceed (in §3) to describe λ_{Rust}, a core language (of our own design) that formalizes the static and dynamic semantics of these central features. Crucially, λ_{Rust} incorporates Rust's notions of *borrowing*, *lifetimes*, and *lifetime inclusion*—which are fundamental to Rust's ownership discipline—in a manner inspired by Rust's Mid-level Intermediate Representation (MIR). For simplicity, λ_{Rust} omits some orthogonal features of Rust such as traits (which are akin to Haskell type classes); it also avoids the morass of exciting complications concerning relaxed memory, instead adopting a simplified memory model featuring only non-atomic and sequentially consistent atomic operations. Nevertheless, λ_{Rust} is realistic enough that studying it led us to uncover a previously unknown soundness bug in Rust itself [28].

Our core contribution is then to develop an *extensible soundness proof* for λ_{Rust}. The basic idea is to build a *semantic model* of the language—in particular, a *logical relation* [47, 56]. Following recent work on "logical" accounts of logical relations [22, 23, 37, 64], our model interprets λ_{Rust} types as predicates on values in a suitably rich program logic (see §4), and interprets λ_{Rust} typing judgments as logical entailments between these predicates (see §7). This model offers an interpretation of what types *mean* (*i.e.*, what values inhabit them) that is more general than just "what the syntactic typing rules allow"—the model describes when it is *observably* safe to treat a value as having a certain type, even if syntactically that value employs `unsafe` features. With our semantic model—which we call **RustBelt**—in hand, the proof of safety of λ_{Rust} divides into three parts:

(1) Verify that the typing rules of λ_{Rust} are sound when interpreted semantically, *i.e.*, as lemmas establishing that the semantic interpretations of the premises imply the semantic interpretation of the conclusion. This is called *the fundamental theorem of logical relations*.

(2) Verify that, if a closed program is *semantically* well-typed according to the model, its execution will not exhibit any unsafe/undefined behaviors. This is called *adequacy*.
(3) For any library that employs `unsafe` code internally, verify that its implementation satisfies the predicate associated with the semantic interpretation of its interface, thus establishing that the `unsafe` code has indeed been safely "encapsulated" by the library's API. In essence, the semantic interpretation of the interface yields a *library-specific verification condition*.

Together, these ensure that, so long as the only `unsafe` code in a well-typed λ_{Rust} program is confined to libraries that satisfy their verification conditions, the program is safe to execute.

This proof is "extensible" in the sense, that whenever you have a new library that uses `unsafe` code and that you want to verify as being safe to use in Rust programs, RustBelt tells you the verification condition you need to prove about it. Using the Coq proof assistant [17], we have formally proven the fundamental theorem and adequacy once and for all, and we have also proven the verification conditions for (λ_{Rust} ports of) several standard Rust libraries that use `unsafe` code, including `Arc`, `Rc`, `Cell`, `RefCell`, `Mutex`, `RwLock`, `mem::swap`, `thread::spawn`, and `take_mut`.

The essential idea of semantic soundness proofs is hardly new [3, 42], but developing such a proof for a language as subtle and sophisticated as Rust has required us to tackle a variety of technical challenges, more than we can describe in the space of this paper. To focus the presentation, we will therefore not present all these challenges and their solutions in full technical detail (although further details can be found in our technical appendix and Coq development [29]). Rather, we aim to highlight the following key challenges and how we dealt with them.

Challenge #1: Choosing the right logic for modeling Rust. The most fundamental design choice in RustBelt was deciding which logic to use as its target, *i.e.*, for defining semantic interpretations of Rust types. There are several desiderata for such a logic, but the most important is that it should support high-level reasoning about concepts that are central to Rust's type system, such as ownership and borrowing. The logic we chose, **Iris**, is ideally suited to this purpose.

Iris is a language-generic framework for higher-order concurrent separation logic [30–32, 35], which in the past year has been equipped with tactical support for conducting machine-checked proofs of programs in Coq [36] and deployed in several ongoing verification projects [33, 55, 57]. By virtue of being a separation logic [46, 49], Iris comes with built-in support for reasoning modularly about ownership. Moreover, the main selling point of Iris is its support for *deriving* custom program logics for different domains using only a small set of primitive mechanisms (namely, *higher-order ghost state* and *impredicative invariants*). In the case of RustBelt, we used Iris to derive a novel **lifetime logic**, whose primary feature is a notion of *borrow propositions* that mirrors the "borrowing" mechanism for tracking aliasing in Rust. This lifetime logic, which we describe in some detail in §5, has made it possible for us to give fairly direct interpretations of a number of Rust's most semantically complex types, and to verify their soundness at a high level of abstraction.

Challenge #2: Modeling Rust's extensible ownership discipline. As explained above, a distinctive feature of Rust is its extensible ownership discipline: Owning a value of shared reference type `&T` confers different privileges depending on the type `T`. For many simple types, `&T` confers read-only access to the contents of the reference; but for types defined by libraries that use `unsafe` operations, `&T` may in fact confer mutable access to the contents, indirectly via the API of `T`. In Rust lingo, this phenomenon is termed *interior mutability*.

To model interior mutability, RustBelt interprets types `T` in two ways: (1) with an *ownership* predicate that says what it means to own a value of type `T`, and (2) with a *sharing* predicate that says what it means to own a value of type `&T`. Unlike the ownership predicate, the sharing predicate must be a freely duplicable assertion, since Rust allows values of shared reference type to be freely

copied. But otherwise there is a great deal of freedom in how it is defined, thus allowing us to assign very different semantics to &T for different types T. We exploit this freedom in proving semantic soundness of several Rust libraries whose types exhibit interior mutability (see §6).

Challenge #3: Accounting for Rust's "thread-safe" type bounds. Some of Rust's types that exhibit interior mutability use non-atomic rather than atomic memory accesses to improve performance. As a result, however, they are not "thread-safe", meaning that if one could transfer ownership of values of these types between threads, it could cause a data race. Rust handles this potential safety problem by restricting cross-thread ownership transfer to types that satisfy certain *type bounds*: the Send bound classifies types T that are thread-safe, and the Sync bound classifies types T such that &T is thread-safe.

We account for these type bounds in RustBelt in a simple and novel way. First, we parameterize both the ownership and sharing predicates in the semantics of types by a *thread identifier*, representing the thread that is claiming ownership. We then define T to be Send if T's ownership predicate does not depend on the thread id parameter (and Sync if T's sharing predicate does not depend on the thread id parameter). Intuitively, this makes sense because, if ownership of a value v of type T is thread-independent, transferring ownership of v between threads is perfectly safe.

All results in this paper have been fully formalized in the Coq proof assistant [29].

2 A TOUR OF RUST

In this section, we give a brief overview of some of the central features of the Rust type system. We do not assume the reader has any prior familiarity with Rust.

2.1 Ownership transfer

Race conditions can only arise from an unrestricted combination of *aliasing* and *mutation* on the same location. In fact, it turns out that ruling out mutation of aliased data also prevents other errors commonplace in low-level pointer-manipulating programs, like use-after-free or double-free. The essential idea of the Rust type system is thus to ensure that aliasing and mutation cannot occur at the same time on any given location, which it achieves by letting *types represent ownership*.

Let us begin with the most basic form of ownership: *exclusive ownership*, in which, at any time, at most one thread is allowed to mutate a given location. Exclusive ownership rules out aliasing entirely, and thus prevents data races. However, just exclusive ownership would not be very expressive, and therefore Rust allows one to *transfer* ownership between threads. To see this principle in practice, consider the following sample program:

```
1  let (snd, rcv) = channel();
2  join(move || {
3         let mut v = Vec::new(); v.push(0); // v: Vec<i32>
4         snd.send(v);
5         // Cannot access v: v.push(1) rejected
6       },
7       move || {
8         let v = rcv.recv().unwrap(); // v: Vec<i32>
9         println!("Received: {:?}", v);
10      });
```

Before we take a detailed look at the way the Rust type system handles ownership here, we briefly discuss syntax: **let** is used to introduce local, stack-allocated variables. These can be made mutable by using **let mut**. The first line uses a pattern to immediately destruct the pair returned

by `channel()` into its components. The vertical bars `||` mark the beginning of an anonymous closure; if the closure would take arguments, they would be declared between the bars.

In this example, one thread sends a shallow copy (*i.e.*, not duplicating data behind pointer indirections) of a vector `v` of type `Vec<i32>` (a resizable heap-allocated array of 32-bit signed integers) over a channel to another thread. In Rust, having a value of some type indicates that we are the *exclusive owner* of the data described by said type, and thus that *nobody else* has any kind of access to this array, *i.e.*, no other part of the program can write to or even read from the array. When ownership is passed to a function (*e.g.*, `send`), the function receives a shallow copy of the data.[1] At the same time, ownership of the data is considered to have *moved*, and thus no longer available in the callee—thus, Rust's variable context is substructural. This is important because the receiver only receives a shallow copy, so if both threads were to use the vector, they could end up racing on the same data.

The function `channel` creates a typed multi-producer single-consumer channel and returns the two endpoints as a pair. The function `join` is essentially parallel composition; it takes two closures and executes them in parallel, returning when both are done.[2] The keyword **`move`** instructs the type checker to move exclusive ownership of the sending end `snd` and receiving end `rcv` of the channel into the first, and, respectively, second closure.

In this example, the first thread creates a new empty `Vec`, `v`, and pushes an element onto it. Next, it sends `v` over the channel. The `send` function takes type `Vec<i32>` as argument, so the Rust type checker considers `v` to be *moved* after the call to `send`. Any further attempts to access `v` would thus result in a compile-time error. The second thread works on the receiving end of the channel. It uses `recv` in order to receive (ownership of) the vector. However, `recv` is a fallible operation, so we call `unwrap` to trigger a *panic* (which aborts execution of the current thread) in case of failure. Finally, we print a debug representation of the vector (as indicated by the format string `"{:?}"`).

One aspect of low-level programming that is distinctively absent in the code above is memory management. Rust does not have garbage collection, so it may seem like our example program leaks memory, but that is not actually the case: Due to ownership tracking, Rust can tell when a variable (say, the vector `v`) goes out of scope without having been moved elsewhere. When that is the case, the compiler automatically inserts calls to a *destructor*, called `drop` in Rust. For example, when the second thread finishes in line 10, `v` is dropped. Similarly, the sending and receiving ends of the channel are dropped at the end of their closures. This way, Rust provides automatic memory management without garbage collection, and with predictable runtime behavior.

2.2 Mutable references

Ownership transfer is a fairly straightforward mechanism for ensuring data-race freedom and related memory safety properties. However, it is also very restrictive. In fact, close inspection shows that even our first sample program does not strictly follow this discipline. Observe that in line 3, we are calling the method `push` on the vector `v`—and we keep using `v` afterwards. Indeed, it would be very inconvenient if pushing onto a vector required explicitly passing ownership to `push` and back. Rust's solution to this issue is *borrowing*, which is the mechanism used to handle reference types. The idea is that `v` is not moved to `push`, but instead borrowed, *i.e.*, passed by reference—granting `push` access to `v` *for the duration of the function call.*

This is expressed in the type of `push`: `fn(&mut Vec<i32>, i32) -> ()`. (Henceforth, we follow the usual Rust style and omit the return type if it is the unit type `()`.) The syntax `v.push(0)`, as used in the example, is just syntactic sugar for `Vec::push(&mut v, 0)`, where `&mut v` creates

[1] Of course, Rust provides a way to do a *deep copy* that actually duplicates the vector, but it will never do this implicitly.
[2] `join` is not in the Rust standard library, but part of Rayon [51], a library for parallel list processing.

a mutable reference to v, which is then passed to push. A mutable reference grants *temporary exclusive access* to the vector, which in the example means that access is restricted to the duration of the call to push. Because the access is temporary, our program can keep using v when push returns. Moreover, the exclusive nature of this access guarantees that no other party will access the vector in any way during the function call, and that push cannot keep copies of the pointer to the vector. Mutable references are always unique pointers.

The type of send, `fn(&mut Sender<Vec<i32>>, Vec<i32>)`, shows another use of mutable references. The first argument is just borrowed, so the caller can use the channel again later. In contrast, the second argument is moved, using ownership transfer as already described above.

2.3 Shared references

Rust's approach to guaranteeing the absence of races and other memory safety is to rule out the combination of aliasing and mutation. So far, we have seen unique ownership (§2.1) and (borrowed) mutable references (§2.2), both of which allow for mutation but prohibit aliasing. In this section we discuss another form of references, namely *shared references*, which form the dual to mutable references: They allow aliasing but prohibit mutation.

Like mutable references, shared references grant *temporary* access to a data structure, and operationally correspond to just pointers. The difference is in the guarantees and permissions provided to the receiver of the reference. While mutable references are exclusive (non-duplicable), shared references can be *duplicated*. In other words, shared references permit aliasing. As a consequence, to ensure data-race freedom and memory safety, shared references are *read-only*; they do not permit mutation.

Practically speaking, shared references behave like unrestricted variables in linear type systems, *i.e.*, just like integers, they can be "copied" (as opposed to just being "moved", which is possible with variables of all types). Rust expresses such properties of types using *bounds*, and the bound that describes unrestricted types is called Copy. Specifically, if a type is Copy, it means that doing a shallow copy (which, remember, is what Rust does to pass arguments) suffices to duplicate elements of the type. Both &T and `i32` are Copy (for any T)—however, `Vec<i32>` is not! The reason for this is that `Vec<i32>` stores data on the heap, and a shallow copy does not duplicate this heap data.

We can see shared references in action in the following example:

```
1 let mut v = Vec::new(); v.push(1);
2 join(|| println!("Thread 1: {:?}", v), || println!("Thread 2: {:?}", v));
3 v.push(2);
```

This program starts by creating and initializing a vector v. It uses a shared reference &v to the vector in two threads, which concurrently print the contents of the vector. This time, the closures are not marked as `move`, which leads to v being captured by-reference, *i.e.*, at type `&Vec<i32>`. As discussed above, this type is Copy, so the type checker accepts using &v in both threads.

The concurrent accesses to v use *non-atomic* reads, which have no synchronization. This is safe because when a function holds a shared reference, it can rely on the data-structure not being mutated—so there cannot be any data races. (Notice that this is a much stronger guarantee than what C provides with `const` pointers: In C, `const` pointers prevent mutation by the current function, however, they do *not* rule out mutation by *other* functions.)

Finally, when join returns, the example program re-gains full access to the vector v and can mutate v again in line 3. This is safe because join will only return when both threads have finished their work, so there cannot be a race between the push and the println. This demonstrates that shared references are powerful enough to *temporarily* share a data structure and permit unrestricted copying of the pointer, but regain exclusive access later.

2.4 Lifetimes

As previously explained, (mutable and shared) references *borrow* ownership and thus grant *temporary* access to a data structure. This immediately raises the question: "How long is *temporary*?" In Rust, this question is answered by equipping every reference with a *lifetime*. The full form of a reference type is actually `&'a mut T` or `&'a T`, where `'a` is the lifetime of the reference. Rust uses a few conventions so that lifetimes can be elided in general, which is why they did not show up in the programs and types we considered so far. However, lifetimes play a crucial role in explaining what happens when the following function is type-checked:

```
1 fn example(v: &/* 'a */mut Vec<i32>) {
2   v.push(21);                    Lifetime 'c
3   { let mut head : &/* 'b */mut i32 = v.index_mut(0);
4     // Cannot access v: v.push(2) rejected
5     *head = 23; }                              Lifetime 'b
6   v.push(42);
7   println!("{:?}", v); // Prints [23, ..., 42]
8 }                                              Lifetime 'a
```

Here we define a function `example` that takes an argument of type `&mut Vec<i32>`. The function uses `index_mut` to obtain a pointer to the first element inside the vector. Writing to `head` in line 5 changes the first element of the vector, as witnessed by the output in line 7. Such pointers directly into a data structure are sometimes called *deep* or *interior* pointers. One has to be careful when using deep pointers because they are a form of aliasing: When `v` is deallocated, `head` becomes a dangling pointer. In fact, depending on the data structure, *any* modification of the data structure could lead to deep pointers being invalidated.[3] This is why the call to `push` in line 4 is rejected.

How does Rust manage to detect this problem and reject line 4 above? To understand this, we have to look at the type of `index_mut`: `for<'b> fn(&'b mut Vec<i32>, usize) -> &'b mut i32`.[4] The `for` is a universal quantifier, making `index_mut` *generic* in the lifetime `'b`. The caller can use any `'b` to instantiate this generic function, limited only by an implicit requirement that `'b` must last at least as long as the call to `index_mut`. Crucially, `'b` is used for *both* the reference passed to the function *and* the reference returned.

In our example, Rust has to infer the lifetime `'b` left implicit when calling `index_mut`. Because the result of `index_mut` is stored in `head`, the type checker infers `'b` to be the scope of `head`, *i.e.*, lines 3-5. As a consequence, based on the type of `index_mut`, the vector must be borrowed *for the same lifetime*. So Rust knows that `v` is mutably borrowed for lines 3-5, which makes the access in line 4 invalid: The lifetime of the reference needed by `push` would overlap with the lifetime of the reference passed to `index_mut`, which violates the rule that mutable references must be unique.

Lifetimes were not visible in the examples discussed so far, but they are always present implicitly. For example, the full type of `push` is given by `for<'c> push(&'c mut Vec<i32>, i32)`. The type checker thus has the freedom to pick *any* lifetime for the reference to the vector, constrained only by the implicit requirement that `'c` has to cover at least the duration of the function call. This is why the vector can be used again immediately after `push` returned.

Notice that, unlike in the previous examples, `v` in this example is just a mutable reference to begin with. Just like `push`, the type of `example` actually involves a generic lifetime `'a`, and `v` has type `&'a mut Vec<i32>`. Despite not being the original owner of `v`, we can still borrow `v` to someone else—a phenomenon dubbed *reborrowing*. All we have to check is that the reborrow ends *before*

[3] One infamous instance of this issue is iterator invalidation, troubling not only low-level languages like C++, but also safe languages like Java.

[4] `usize` is an unsigned integer type of platform-dependent size large enough to cover the address space.

the lifetime of our reference ends. In other words, the lifetime of the reborrow (the `'b` used for `index_mut`, *i.e.*, the scope of `head`) has to be *included* in the lifetime of the reference (`'a`). In this case, we know this to be true by making use of the implicit assumption that `'a` includes this function call, so in particular, it includes `'b`, which is entirely contained within the function call.

2.5 Interior mutability

So far, we have seen how Rust ensures memory safety and data-race freedom by ruling out the combination of aliasing and mutation. However, there are cases where shared mutable state is actually needed to (efficiently) implement an algorithm or a data structure. To support these use-cases, Rust provides some primitives providing shared mutable state. All of these have in common that they permit *mutation* through a *shared reference*—a concept called *interior mutability*.

At this point, you may be wondering—how does this fit together with the story of mutation and aliasing being the root of all memory safety problems? The key point is that these primitives have a carefully controlled API surface. Even though mutation through a shared reference is unsafe *in general*, it can still be safe when appropriate restrictions are enforced by either static or run-time checks. This is where we can see Rust's "extensible" approach to safety in action. Interior mutability is not wired into the type system; instead, the types we are discussing here are implemented in the standard library using `unsafe` code (which we will verify in §6).

2.5.1 Cell. The simplest type with interior mutability is `Cell`. Consider the following example:

```
let c1 : &Cell <i32> = &Cell::new(0);
let c2 : &Cell <i32> = c1;
c1.set(2);
println!("{:?}", c2.get()); // Prints 2
```

The `Cell<i32>` type provides operations for storing and obtaining its content: `set` has type `fn(&Cell<i32>, i32)`, and `get` has type `fn(&Cell<i32>) -> i32`. Both of these only take a shared reference, so they can be called in the presence of arbitrary aliasing. So after we just spent several pages explaining that safety in Rust arises from ruling out aliasing and mutation, now we have `set`, which seems to completely violate this principle. How can this be safe?

The answer to this question has two parts. First of all, `Cell` only allows getting a *copy* of the content via `get`; it is not possible to obtain a pointer into the content. This rules out deep pointers into the `Cell`, making mutation safe. Unsurprisingly, `get` requires the content of the `Cell` to be `Copy`. In particular, `get` cannot be used with cells that contain non-`Copy` types like `Vec<i32>`.

However, there is still a potential source of problems, which arises from Rust's support for multithreading. In particular, the following program must *not* be accepted:

```
let c = &Cell::new(0);
join(|| c.set(1), || println!("{:?}", c.get()));
```

The threads perform conflicting unsynchronized accesses to `c`, *i.e.*, this program has a data race.

To rule out programs like the one above, Rust has a notion of types being "sendable to another thread". Such types satisfy the `Send` bound. The type of `join` demands that the environment captured by the closure satisfies `Send`. For example, `Vec<i32>` is `Send` because when the vector is moved to another thread, the previous owner is no longer allowed to access the vector—so it is fine for the new owner, in a different thread, to perform any operation whatsoever on the vector.

In the case above, the closure captures a shared reference to `c` of type `&Cell<i32>`. To check whether shared references are `Send`, there is another bound called `Sync`, with the property that type `&T` is `Send` if and only if `T` is `Sync`. Intuitively, a type is `Sync` if it is safe to have shared references to the same instance of the type in different threads. In other words, all the operations available on

&T have to be thread-safe. For example, Vec<i32> is Sync because shared references only permit reading the vector, and it is fine if multiple threads do that at the same time. However, Cell<i32> is *not* Sync because set is not thread-safe. As a consequence, &Cell<i32> is not Send, which leads to the program above being rejected.

2.5.2 Mutex. The Cell type is a great example of interior mutability and a zero-cost abstraction as it comes with no overhead: get and set compile to plain unsynchronized accesses, so the compiled program is just as efficient as a C program using shared mutable state. However, as we have seen, Cell pays for this advantage by not being thread-safe. The Rust standard library also provides primitives for thread-safe shared mutable state, one being Mutex, which implements mutual exclusion (via a standard lock) for protecting access to one shared memory location. Consider the following example:

```
let mutex = Mutex::new(Vec::new());
join( || { let mut guard = mutex.lock().unwrap();
           guard.deref_mut().push(0) },
      || { let mut guard = mutex.lock().unwrap();
           println!("{:?}", guard.deref_mut()) } );
```

This program starts by creating a mutex of type Mutex<Vec<i32>> initialized with an empty vector. The mutex is then shared between two threads (implicitly relying on Mutex<Vec<i32>> being Sync). The first thread acquires the lock, and pushes an element to the vector. The second thread acquires the lock just to print the contents of the vector.

The guard variables are of type MutexGuard<'a, Vec<i32>> where 'a is the lifetime of the shared mutex reference passed to lock (this ensures that the mutex itself will stay around for at least as long as the guard). Mutex guards serve two purposes. Most importantly, if a thread owns a guard, that means it holds the lock. To this end, guards provide a method deref_mut which turns a mutable reference of MutexGuard into a mutable reference of Vec<i32> *with the same lifetime*. Very much unlike Cell, however, the Mutex type permits obtaining deep pointers into the data guarded by the lock. In fact, the compiler will insert calls to deref_mut automatically where appropriate, making MutexGuard<'a, Vec<i32>> behave essentially like &'a mut Vec<i32>.

Moreover, the guards are set up to release the lock when their destructors are called, which will happen automatically when the guards go out of scope. This is safe because, just like with index_mut (§2.4), the compiler ensures that deep pointers obtained through deref_mut have all expired by the time the guard is dropped.

3 LANGUAGE AND TYPE SYSTEM

In this section, we introduce λ_{Rust}: our formal version of Rust. The Rust surface language comes with significant syntactic sugar (some of which we have already seen). To simplify the formalization, λ_{Rust} features only a small set of primitive constructs, and requires the advanced constructs of Rust's surface language to be desugared into primitive constructs. Indeed, something very similar happens in the compiler itself, where surface Rust is lowered into the *Mid-level Intermediate Representation (MIR)* [39]. λ_{Rust} is much closer to MIR than to surface Rust.

Before we present the syntax (§3.1), operational semantics (§3.2) and type system (§3.3) of λ_{Rust}, we highlight some of its key features:

- Programs are represented in continuation-passing style. This choice enables us to represent complex control-flow constructs, like labeled breaks and early returns, as present in the Rust surface language. Furthermore, following the correspondence of CPS and control-flow graphs [5], this makes λ_{Rust} easier to relate to MIR.

- The individual instructions of our language perform a single operation. By keeping the individual instructions simple and avoiding large composed expressions, it becomes possible to describe the type system in a concise way.
- The memory model of λ_{Rust} supports pointer arithmetic and ensures that programs with data races or illegal memory accesses can reach a stuck state in the operational semantics. In particular, programs that cannot get stuck in any execution—a guarantee established by the adequacy theorem of our type system (Theorem 7.2)—are data-race free.

3.1 The syntax

The syntax of λ_{Rust} is as follows:

$$
\begin{aligned}
\mathit{Path} \ni p &::= x \mid p.n \\
\mathit{Val} \ni v &::= \mathbf{false} \mid \mathbf{true} \mid z \mid \ell \mid \mathbf{funrec}\, f(\overline{x})\, \mathbf{ret}\, k := F \\
\mathit{Instr} \ni I &::= v \mid p \mid p_1 + p_2 \mid p_1 - p_2 \mid p_1 \le p_2 \mid p_1 = p_2 \mid \mathbf{new}(n) \mid \mathbf{delete}(n, p) \\
&\quad \mid {}^{*}p \mid p_1 := p_2 \mid p_1 :=_n {}^{*}p_2 \mid p \overset{\text{inj } i}{:=} () \mid p_1 \overset{\text{inj } i}{:=} p_2 \mid p_1 \overset{\text{inj } i}{:=}_n {}^{*}p_2 \mid \ldots \\
\mathit{FuncBody} \ni F &::= \mathbf{let}\, x = I\, \mathbf{in}\, F \mid \mathbf{letcont}\, k(\overline{x}) := F_1\, \mathbf{in}\, F_2 \mid \mathbf{newlft}; F \mid \mathbf{endlft}; F \\
&\quad \mid \mathbf{if}\, p\, \mathbf{then}\, F_1\, \mathbf{else}\, F_2 \mid \mathbf{case}\, {}^{*}p\, \mathbf{of}\, \overline{F} \mid \mathbf{jump}\, k(\overline{x}) \mid \mathbf{call}\, f(\overline{p})\, \mathbf{ret}\, k
\end{aligned}
$$

We let path offsets n and integer literals z range over the integers, and sum indices i range over the natural numbers. The language has two kinds of variables: program variables, which are written as x or f, and continuation variables, which are written as k.

We distinguish four classes of expressions: *function bodies* F consist of *instructions* I that operate on *paths* p and *values* v. Only the most basic values can be written as literals: the Booleans **false** and **true**, integers z, locations ℓ (see §3.2 for further details), and functions $\mathbf{funrec}\, f(\overline{x})\, \mathbf{ret}\, k := F$. There are no literals for products or sums, as these only exist in memory, represented by sequences of values and tagged unions, respectively. Paths are used to express the values that instructions operate on. The common case is to directly refer to a local variable x. Beyond this, paths can refer to parts of a compound data structure laid out in memory: Offsets $p.n$ perform pointer arithmetic, incrementing the pointer expressed by p by n memory cells.

Function bodies mostly serve to chain instructions together and manage control flow, which is handled through continuations. Continuations are declared using $\mathbf{letcont}\, k(\overline{x}) := F_1\, \mathbf{in}\, F_2$, and called using $\mathbf{jump}\, k(\overline{x})$. The parameters $\overline{x}$ are instantiated when calling the continuation. We allow continuations to be recursive, so as to model looping constructs, like **while** and **for**.

The "ghost instructions" **newlft** and **endlft** start and end lifetimes. These instructions have interesting typing rules, but do not do anything operationally.

Functions can be declared using $\mathbf{funrec}\, f(\overline{x})\, \mathbf{ret}\, k := F$, where f is a binder for the recursive call, $\overline{x}$ is a list of binders for the arguments, and k is a binder for the return continuation. The return continuation takes one argument for the return value. Functions can be called using $\mathbf{call}\, f(\overline{p})\, \mathbf{ret}\, k$, where k is the continuation that should be called when the function returns.

Local variables of λ_{Rust}—as represented by **let** bindings—are pure and subject to β-reduction. This is different from local variables in Rust (and MIR), which are mutable and addressable. Hence, to correctly model Rust's local variables, we allocate them on the heap. It is worth noting that similar to prior work on low-level languages [34, 38], we do not make a distinction between the stack and the heap, as these are semantically the same. In practice, this looks as follows:

```
fn option_as_mut
    (o: &mut Option<i32>) ->
    Option<&mut i32> {
  match *o {
    None => None,
    Some(ref mut t) => Some(t)
  }
}
```

```
funrec option_as_mut(o) ret ret :=
  let r = new(2) in
  letcont k() := delete(1, o); jump ret(r) in
  let o' = *o in case *o' of
  – r :=^{inj 0} (); jump k()
  – r :=^{inj 1} o'.1; jump k()
```

We see that the function argument `o` is a pointer, which is dereferenced when used and deallocated before the function returns. Similarly, a pointer `r` is allocated for the return value.

The λ_{Rust} language has instructions for the usual arithmetic operations, memory allocation, and deallocation, as well as loading from memory ($^*p$) and storing a value into memory ($p_1 := p_2$). The `memcpy`-like instruction $p_1 :=_n {}^*p_2$ copies the contents of n memory locations from p_2 to p_1. All of these accesses are *non-atomic*, *i.e.*, they are not thread-safe. We will come back to this point in §3.2.

The previous example demonstrates how sums are represented by tagged unions in memory. Values of the `Option<i32>` type are represented by a sequence of two base values: an integer value that represents the tag (0 for `None` and 1 for `Some`) and, if the tag is 1, a value of type `i32` for the argument `t` of `Some(t)`. The instructions $p_1 :=^{\text{inj}\,i} p_2$ and $p_1 :=_n {}^{\text{inj}\,i} {}^*p_2$ can be used to assign to a pointer p_1 of sum type, setting both the tag i and the value associated with this variant of the union, while $p_1 :=^{\text{inj}\,i} ()$ is used for variants that have no data associated with them (like `None`).

There are more instructions available in the underlying core language, *e.g.*, instructions to spawn threads or perform atomic accesses, including CAS (compare-and-swap). However, the type system does not provide any typing rules for these instructions, so they can only be used by `unsafe` code.

3.2 The operational semantics

The operational semantics of λ_{Rust} is given by translation into a core language. The core language is a lambda calculus equipped with primitive values, pointer arithmetic, and concurrency. We define the semantics this way for three reasons. First of all, we can model some of the λ_{Rust} constructs (*e.g.*, $p_1 :=_n {}^*p_2$) as sequences of simpler instructions in the core language. Secondly, we can reduce both continuations and functions to plain lambda terms. Finally, the core language supports a substitution-based semantics, which makes reasoning more convenient, whereas the CPS grammar given above is not actually closed under substitution. The details of the core language are fully spelled out in our technical appendix [29].

The memory model is inspired by CompCert [38] in order to properly support pointer arithmetic. On top of this, we want the memory model to detect and rule out data races. Following C++11 [1], we provide both *non-atomic* memory accesses, on which races are considered undefined behavior, and *atomic* accesses, which may be racy; however, for simplicity, we only provide sequentially consistent (SC) atomic operations, avoiding consideration of C++11's relaxed atomics in this paper. We consider a program to have a data race if there are ever two concurrent accesses to the same location, at least one of which is a write, and at least one of which is non-atomic. To detect such data races, every location is equipped with some additional state (resembling a reader-writer lock), which is checked dynamically to see if a particular memory access is permitted. We have shown in Coq that if a program has a data race, then it has an execution where these checks fail. As a consequence, if we prove that a program cannot get stuck (which implies that the checks always succeed, in all executions), then the program is data-race free.

3.3 The type system

The type system of λ_{Rust} is shown in Figure 1 and its types and contexts are as follows:

$$\begin{array}{lll} \mathit{Lft} \ni \kappa ::= \alpha \mid \mathbf{static} & \mathbf{E} ::= \emptyset \mid \mathbf{E}, \kappa \sqsubseteq_{\mathrm{e}} \kappa' & \mathbf{T} ::= \emptyset \mid \mathbf{T}, p \lhd \tau \mid \mathbf{T}, p \lhd^{\dagger\kappa} \tau \\ \mathit{Mod} \ni \mu ::= \mathbf{mut} \mid \mathbf{shr} & \mathbf{L} ::= \emptyset \mid \mathbf{L}, \kappa \sqsubseteq_{\mathrm{l}} \overline{\kappa} & \mathbf{K} ::= \emptyset \mid \mathbf{K}, k \lhd \mathbf{cont}(\mathbf{L}; \overline{x}.\, \mathbf{T}) \end{array}$$

$$\mathit{Type} \ni \tau ::= T \mid \mathbf{bool} \mid \mathbf{int} \mid \mathbf{own}_n\, \tau \mid \&^{\kappa}_{\mu}\, \tau \mid \lightning_n \mid \Pi\overline{\tau} \mid \Sigma\overline{\tau} \mid \forall\overline{\alpha}.\, \mathbf{fn}(\text{ϝ} : \mathbf{E}; \overline{\tau}) \to \tau \mid \mu\, T.\, \tau$$

There are two kinds of pointer types: *owned pointers* $\mathbf{own}_n\, \tau$ and *(borrowed) references* $\&^{\kappa}_{\mu}\, \tau$.

Owned pointers are used to represent Rust's local, stack-allocated variables. As already mentioned, we model the stack using heap allocations. Effectively, $\mathbf{own}_n\, \tau$ is the type of a stack slot containing a variable of type τ. The subscript n is used to track the size of the allocation, which is relevant for deallocation. For further details, see our technical appendix [29].

References $\&^{\kappa}_{\mu}\, \tau$ are qualified by a *modifier* μ, which is either **mut** (for mutable references, which are unique) or **shr** (for shared references), and a *lifetime* κ. The **static** lifetime lasts for the execution of the entire program (corresponding to `'static` in Rust, which plays the same role), whereas lifetime variables α can be allocated and deallocated dynamically using the **`newlft`** and **`endlft`** ghost instructions. References $\&^{\kappa}_{\mu}\, \tau$ are borrowed for lifetime κ and, as such, can only be used as long as the lifetime κ is alive, *i.e.*, still ongoing.

The type $\lightning_n$ describes arbitrary sequences of n base values. This type represents uninitialized memory. For example, when allocating an owned pointer (rule S-NEW), its type is $\mathbf{own}_n \lightning_n$. Uninitialized memory also comes up when data has been moved away, as witnessed by TREAD-OWN-MOVE. Owned pointers permit strong updates, which means their type τ can change when the memory gets (re-)initialized. Note that this is sound because ownership of owned pointers is unique.

The types $\Pi\overline{\tau}$ and $\Sigma\overline{\tau}$ represent n-ary products and sums, respectively. In particular, this gives rise to a unit type (the empty product $\Pi[]$) and the empty type (the empty sum $\Sigma[]$). We use $\tau_1 \times \tau_2$ and $\tau_1 + \tau_2$ as notation for binary products ($\Pi[\tau_1, \tau_2]$) and sums ($\Sigma[\tau_1, \tau_2]$), respectively.

Function types $\forall\overline{\alpha}.\, \mathbf{fn}(\text{ϝ} : \mathbf{E}; \overline{\tau}) \to \tau$ can be polymorphic over lifetimes $\overline{\alpha}$. The external lifetime context $\mathbf{E}$ can be used to demand that one lifetime parameter be included in another one. The lifetime ϝ here is a binder than can be used in $\mathbf{E}$ to refer to the lifetime of this function. For example, $\forall\alpha.\, \mathbf{fn}(\text{ϝ} : \text{ϝ} \sqsubseteq \alpha; \&^{\alpha}_{\mathbf{mut}}\, \mathbf{int}) \to \Pi[]$ is the type of a function that takes a mutable reference to an integer with any lifetime that covers this function call (matching the implicit assumption Rust makes), and returns a unit, *i.e.*, nothing. Note that, to allow passing and returning objects of arbitrary size, both the parameters and the return value are transmitted via owned pointers; this calling convention is universally applied and hence does not show up in the function type.

Finally, λ_{Rust} supports *recursive types* $\mu\, T.\, \tau$, with the restriction (enforced by the well-formedness judgment shown in the appendix [29]) that the recursive occurrence T is below a pointer type.

To keep the type system of λ_{Rust} focused on our core objective (modeling borrowing and lifetimes), there is no support for type-polymorphic functions. Instead, we handle polymorphism on the meta-level: In our shallow embedding of the type system in Coq, we can quantify any definition and theorem over arbitrary semantic types (§4). We exploit this flexibility when verifying the safety of Rust libraries that use `unsafe` features (§6). These libraries are typically polymorphic, and by keeping the verification similarly polymorphic, we can prove that functions and libraries are safe to use at any instantiation of their type parameters.

Judgments. The typing judgments for function bodies F and instructions I have the shape $\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F$ and $\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{T}_1 \vdash I \dashv x.\, \mathbf{T}_2$.

The *variable context* Γ is the only binding context. It introduces all variables that are free in the judgment and keeps track of whether they are program variables ($x : \mathbf{val}$), lifetime variables

Rules for subtyping and type coercions:

T-BOR-LFT
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \&_\mu^{\kappa'} \tau \Rrightarrow \&_\mu^{\kappa} \tau}$$

C-SUBTYPE
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau \Rrightarrow \tau'}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash p \lhd \tau \overset{\text{ctx}}{\Rrightarrow} p \lhd \tau'}$$

C-COPY
$$\frac{\tau \text{ copy}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash p \lhd \tau \overset{\text{ctx}}{\Rrightarrow} p \lhd \tau, p \lhd \tau}$$

C-SPLIT-BOR
$$\Gamma \mid \mathbf{E}; \mathbf{L} \vdash p \lhd \&_\mu^\kappa (\tau_1 \times \tau_2) \overset{\text{ctx}}{\Rrightarrow} p.0 \lhd \&_\mu^\kappa \tau_1,\ p.\text{size}(\tau_1) \lhd \&_\mu^\kappa \tau_2$$

C-SHARE
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \text{ alive}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash p \lhd \&_{\mathbf{mut}}^\kappa \tau \overset{\text{ctx}}{\Rrightarrow} p \lhd \&_{\mathbf{shr}}^\kappa \tau}$$

C-BORROW
$$\Gamma \mid \mathbf{E}; \mathbf{L} \vdash p \lhd \mathbf{own}_n \tau \overset{\text{ctx}}{\Rrightarrow} p \lhd \&_{\mathbf{mut}}^\kappa \tau, p \lhd^{\dagger\kappa} \mathbf{own}_n \tau$$

C-REBORROW
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa' \sqsubseteq \kappa}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash p \lhd \&_{\mathbf{mut}}^\kappa \tau \overset{\text{ctx}}{\Rrightarrow} p \lhd \&_{\mathbf{mut}}^{\kappa'} \tau, p \lhd^{\dagger\kappa'} \&_{\mathbf{mut}}^\kappa \tau}$$

Rules for type reading and writing:

TREAD-OWN-COPY
$$\frac{\tau \text{ copy}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{own}_n \tau \circ\!\!-^\tau \mathbf{own}_n \tau}$$

TREAD-OWN-MOVE
$$\frac{n = \text{size}(\tau)}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{own}_m \tau \circ\!\!-^\tau \mathbf{own}_n \not\downarrow n}$$

TREAD-BOR
$$\frac{\tau \text{ copy} \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \text{ alive}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \&_\mu^\kappa \tau \circ\!\!-^\tau \&_\mu^\kappa \tau}$$

TWRITE-OWN
$$\frac{\text{size}(\tau) = \text{size}(\tau')}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{own}_n \tau' \multimap^\tau \mathbf{own}_n \tau}$$

TWRITE-BOR
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \text{ alive}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \&_{\mathbf{mut}}^\kappa \tau \multimap^\tau \&_{\mathbf{mut}}^\kappa \tau}$$

Rules for typing of instructions:

S-NUM
$$\Gamma \mid \mathbf{E}; \mathbf{L} \mid \emptyset \vdash z \dashv x.\, x \lhd \mathbf{int}$$

S-NAT-LEQ
$$\Gamma \mid \mathbf{E}; \mathbf{L} \mid p_1 \lhd \mathbf{int}, p_2 \lhd \mathbf{int} \vdash p_1 \le p_2 \dashv x.\, x \lhd \mathbf{bool}$$

S-NEW
$$\Gamma \mid \mathbf{E}; \mathbf{L} \mid \emptyset \vdash \mathtt{new}(n) \dashv x.\, x \lhd \mathbf{own}_n \not\downarrow n$$

S-DELETE
$$\frac{n = \text{size}(\tau)}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid p \lhd \mathbf{own}_n \tau \vdash \mathtt{delete}(n, p) \dashv \emptyset}$$

S-DEREF
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \circ\!\!-^\tau \tau_1' \qquad \text{size}(\tau) = 1}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid p \lhd \tau_1 \vdash {}^*p \dashv x.\, p \lhd \tau_1', x \lhd \tau}$$

S-ASSGN
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^\tau \tau_1'}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid p_1 \lhd \tau_1, p_2 \lhd \tau \vdash p_1 := p_2 \dashv p_1 \lhd \tau_1'}$$

Rules for typing of function bodies:

F-CONSEQUENCE
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T} \overset{\text{ctx}}{\Rrightarrow} \mathbf{T}' \qquad \mathbf{K}' \subseteq \mathbf{K} \qquad \Gamma \mid \mathbf{E}; \mathbf{L}' \mid \mathbf{K}'; \mathbf{T}' \vdash F}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F}$$

F-LET
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{T}_1 \vdash I \dashv x.\, \mathbf{T}_2 \qquad \Gamma, x : \mathbf{val} \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}_2, \mathbf{T} \vdash F}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}_1, \mathbf{T} \vdash \mathtt{let}\, x = I\, \mathtt{in}\, F}$$

F-LETCONT
$$\frac{\Gamma, k, \overline{x} : \mathbf{val} \mid \mathbf{E}; \mathbf{L}_1 \mid \mathbf{K}, k \lhd \mathbf{cont}(\mathbf{L}_1; \overline{x}.\, \mathbf{T}'); \mathbf{T}' \vdash F_1 \qquad \Gamma, k : \mathbf{val} \mid \mathbf{E}; \mathbf{L}_2 \mid \mathbf{K}, k \lhd \mathbf{cont}(\mathbf{L}_1; \overline{x}.\, \mathbf{T}'); \mathbf{T} \vdash F_2}{\Gamma \mid \mathbf{E}; \mathbf{L}_2 \mid \mathbf{K}; \mathbf{T} \vdash \mathtt{letcont}\, k(\overline{x}) := F_1\, \mathtt{in}\, F_2}$$

F-JUMP
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T} \overset{\text{ctx}}{\Rrightarrow} \mathbf{T}'[\overline{y}/\overline{x}]}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid k \lhd \mathbf{cont}(\mathbf{L}; \overline{x}.\, \mathbf{T}'); \mathbf{T} \vdash \mathtt{jump}\, k(\overline{y})}$$

F-NEWLFT
$$\frac{\Gamma, \alpha : \mathbf{lft} \mid \mathbf{E}; \mathbf{L}, \alpha \sqsubseteq_1 \overline{\kappa} \mid \mathbf{K}; \mathbf{T} \vdash F}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash \mathtt{newlft}; F}$$

F-ENDLFT
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}' \vdash F \qquad \mathbf{T} \Rrightarrow^{\dagger\kappa} \mathbf{T}'}{\Gamma \mid \mathbf{E}; \mathbf{L}, \kappa \sqsubseteq_1 \overline{\kappa} \mid \mathbf{K}; \mathbf{T} \vdash \mathtt{endlft}; F}$$

F-CALL
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T} \overset{\text{ctx}}{\Rrightarrow} \overline{p} \lhd \overline{\mathbf{own}}\, \overline{\tau}, \mathbf{T}' \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \overline{\kappa} \text{ alive} \qquad \Gamma, \varphi : \mathbf{lft} \mid \mathbf{E}, \varphi \sqsubseteq_e \overline{\kappa}; \mathbf{L} \vdash \mathbf{E}'}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid k \lhd \mathbf{cont}(\mathbf{L}; y.\, y \lhd \mathbf{own}\, \tau, \mathbf{T}'); \mathbf{T}, f \lhd \mathbf{fn}(\varphi : \mathbf{E}'; \overline{\tau}) \to \tau \vdash \mathtt{call}\, f(\overline{p})\, \mathtt{ret}\, k}$$

Fig. 1. A selection of the typing rules of λ_{Rust}.

($\alpha : \mathbf{lft}$), or type variables ($T : \mathbf{type}$). All the remaining contexts state facts and assert ownership related to variables introduced here, but they do not introduce additional binders.

As we have already seen with function types, the external lifetime context $\mathbf{E}$ contains assumptions about lifetimes including each other. The local lifetime context $\mathbf{L}$, on the other hand, contains lifetime declarations $\kappa \sqsubseteq_{\mathbf{l}} \overline{\kappa}$. These declarations state that the lifetime κ was created using **`newlft`** by the current function (see F-NEWLFT), and is furthermore a sub-lifetime of all the lifetimes in the list $\overline{\kappa}$. The lifetimes appearing on the left-hand side in the local lifetime context are lifetimes *within* the current function. In particular, such lifetimes can be ended with **`endlft`** (see F-ENDLFT).

The *typing context* $\mathbf{T}$ is in charge of describing ownership: It contains type assignments $p \lhd \tau$ and *lifetime-blocked* type assignments $p \lhd^{\dagger\kappa} \tau$. Blocked type assignments can only be used again when the lifetime κ has ended, as expressed by the unblocking judgment $\mathbf{T} \Rightarrow^{\dagger\kappa} \mathbf{T}'$. It is important to stress that typing contexts are substructural: Type assignments describe ownership, and thus they can only be duplicated if the type satisfies τ copy (C-COPY), corresponding to Rust's `Copy`.

The judgment for function bodies does not include a "return" type because function bodies are in CPS and therefore do not return. The return type of a function type ends up being the argument type of its return continuation in $\mathbf{K}$. Instructions, however, *do* return and produce a value, so their typing judgment features two typing contexts: The typing context $\mathbf{T}_1$ is consumed before execution, and the typing context $\mathbf{T}_2$ is produced after execution.

Finally, the continuation context $\mathbf{K}$ tracks continuations $k \lhd \mathbf{cont}(\mathbf{L}; \overline{x}.\, \mathbf{T})$, where $\mathbf{L}$ is the local lifetime context and $\mathbf{T}$ the typing context that is required to call the continuation.

Subtyping is described by $\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \Rightarrow \tau_2$. The main forms of subtyping supported in Rust are lifetime inclusion (T-BOR-LFT) and (un)folding recursive types. Apart from that, there are the usual structural rules witnessing covariance and contravariance of type constructors. On the type context level, $\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T}_1 \overset{\text{ctx}}{\Rightarrow} \mathbf{T}_2$ lifts subtyping (C-SUBTYPE) while also adding a few coercions that can only be applied at the top-level type. Most notably, a mutable reference can be coerced to a shared reference (C-SHARE), an owned pointer can be borrowed (C-BORROW) to create a mutable reference, and a mutable reference can be reborrowed (C-REBORROW).

The judgments $\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa' \sqsubseteq \kappa$ and $\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa$ alive are used to prove that a lifetime is included in another, and that a lifetime is alive. External lifetime context satisfaction $\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{E}'$ is used on function calls to check the assumptions made by the callee. The judgments $\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\tau} \tau_2$ and $\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \circ\!\!-\!\!\!\multimap^{\tau} \tau_2$ describe the type-level effect of writing to and reading from memory.

4 RUSTBELT: A SEMANTIC MODEL OF λ_{Rust} TYPES IN IRIS

Our proof of soundness of λ_{Rust} proceeds by defining a *logical relation*, which interprets the types and typing judgments of λ_{Rust} as logical predicates in an appropriate semantic domain. We focus here on the interpretation of types, leaving the interpretation of typing judgments and the statements of our main results to §7. First, in §4.1, we give a simplified version of the semantic domain of types. In §4.2, we give the semantic interpretation of some representative λ_{Rust} types. Finally, in §4.3, we focus on the interpretation of shared reference types. It will turn out that we have to generalize our semantic domain of types to account for them.

4.1 A simplified semantic domain of types

The semantic domain of types answers the question "What *is* a type?". Usually, the answer is that *a type denotes a set of values*—or, equivalently, a predicate over values. Fundamentally, this is also the case for λ_{Rust}, but the details get somewhat more complicated. First of all, our model of the type system of λ_{Rust} expresses types not as predicates in "plain mathematics" (*e.g.*, the usual higher-order logic), but as predicates in *Iris*. As discussed in the introduction, Iris is a higher-order separation logic designed to prove correctness of complex concurrent programs. Using Iris to express types

has the advantage that concepts like ownership are already built into the underlying framework, so the model itself does not have to take care of them.

Rather than try to explain all the features of Iris here, we will introduce them *en passant*, as needed. However, one that is worth mentioning up front is the ability to define predicates by *guarded recursion*. This means that a predicate can refer to itself recursively, but only below a $\triangleright$ ("later") modality [8] or some other appropriate "guard". The use of a guard ensures that the circular definition can be solved—regardless of whether the recursive reference occurs positively, negatively, or both—using the technique of "step-indexing" [7]. For this reason, $\triangleright$ appears in various places in our model; the placement of these $\triangleright$'s is important for soundness, but is not otherwise relevant to our high-level exposition, so we will mostly ignore it in the rest of the paper.

Our interpretation of types associates to every type τ an Iris predicate $[\![\tau]\!].\mathrm{own} \in \mathit{TId} \times \mathit{list}(\mathit{Val}) \to \mathit{iProp}$. This predicate takes two parameters and returns an Iris proposition (of type *iProp*). The second parameter is the *list* of values we are considering. It turns out that types in Rust do not just cover a single value: In general, data is laid out in memory and spans multiple locations. However, we have to impose some restrictions on the lists of values accepted by a type: we require that every type has a fixed *size* $[\![\tau]\!].\mathrm{size}$. This size is used to compute the layout of compound data structures, *e.g.*, for product types.[5] We require that a type only accepts lists whose length matches the size:

$$[\![\tau]\!].\mathrm{own}(t, \overline{v}) \Rightarrow |\overline{v}| = [\![\tau]\!].\mathrm{size} \qquad (\textsc{ty-size})$$

Furthermore, for `Copy` types we require that $[\![\tau]\!].\mathrm{own}(t, \overline{v})$ be *persistent*. In Iris, a proposition is considered persistent if it does not describe ownership of any exclusive right or resource, and can therefore be freely copied and shared among several parties.

The first parameter of the predicate (of type *TId*) permits types to moreover depend on the *thread identifier* of the thread that claims ownership. This is used for types like `&Cell` that cannot be sent to another thread. In other words, *ownership is (in general) thread-relative*. As we explained in §1.2, this provides a very natural way of modeling `Send`: Semantically speaking, a type τ is `Send` if $[\![\tau]\!].\mathrm{own}$ does *not* depend on the thread id. We will see more details about this in §6.1, when we give the interpretation of `Cell`, a type that cannot be shared across threads.

4.2 Interpreting types

Now that we have a semantic domain of types, we can define their semantic interpretation as a function from syntactic types τ into the semantic domain. In this paper, we focus on the most representative types. The full interpretation can be found in the technical appendix [29].

Booleans. To get started, let us consider a very simple type: **bool**. It should not come as a surprise that $[\![\mathbf{bool}]\!].\mathrm{size} := 1$. The semantic predicate of a Boolean is defined as follows:

$$[\![\mathbf{bool}]\!].\mathrm{own}(t, \overline{v}) := \overline{v} = [\mathtt{true}] \lor \overline{v} = [\mathtt{false}]$$

In other words, a Boolean can only be a singleton list (which is already expressed by its size), and that list has to contain either `true` or `false`.

Unsurprisingly, the semantic interpretation of integers is similar and equally straightforward.

Products. Given two types τ_1 and τ_2, we define the semantics of their binary product $\tau_1 \times \tau_2$ as that of the two types laid out one after the other in memory. This definition can be iterated to yield the interpretation of n-ary products.

For the size, we have $[\![\tau_1 \times \tau_2]\!].\mathrm{size} := [\![\tau_1]\!].\mathrm{size} + [\![\tau_2]\!].\mathrm{size}$. The semantic predicate associated with $\tau_1 \times \tau_2$ uses *separating conjunction* ($P * Q$), the defining feature of separation logic, to join the

[5] Full Rust supports "unsized types" under some restrictions, but these are not covered by our model.

semantic predicates of both types. The separating conjunction ensures that they describe ownership of disjoint pieces of memory. (Here, $+\!\!+$ is list concatenation.)

$$[\![\tau_1 \times \tau_2]\!].\mathrm{own}(t, \overline{v}) := \exists \overline{v}_1, \overline{v}_2.\, \overline{v} = \overline{v}_1 +\!\!+ \overline{v}_2 * [\![\tau_1]\!].\mathrm{own}(t, \overline{v}_1) * [\![\tau_2]\!].\mathrm{own}(t, \overline{v}_2)$$

Owned pointers. In order to give a semantic interpretation to the type $\mathbf{own}_n\, \tau$ of owned pointers, we use the standard *points-to* proposition of separation logic, $\ell \mapsto \overline{v}$. It states that, starting at location ℓ, the memory contains the values $\overline{v}$, and asserts *ownership* of this memory region. With this ingredient, the interpretation is given by $[\![\mathbf{own}_n\, \tau]\!].\mathrm{size} = 1$ and the following semantic predicate:

$$[\![\mathbf{own}_n\, \tau]\!].\mathrm{own}(t, \overline{v}) := \exists \ell.\, \overline{v} = [\ell] * \exists \overline{w}.\, \ell \mapsto \overline{w} * \triangleright\, [\![\tau]\!].\mathrm{own}(t, \overline{w}) * \triangleright\, \mathrm{DeallocSize}(\ell, n, [\![\tau]\!].\mathrm{size})$$

Rust supports recursive types whenever the recursive occurrence is below a pointer indirection. To properly model this using Iris' guarded recursive definitions, we have to make sure that all uses of τ are guarded—in this case, by adding a $\triangleright$.

The proposition $\mathrm{DeallocSize}(\ell, n, [\![\tau]\!].\mathrm{size})$ in the semantic predicate above manages the right to deallocate the location ℓ. These details can be found spelled-out in our technical appendix [29].

Mutable references. Mutable references, like owned pointers, are unique pointers to something of type τ. The key difference is that mutable references are *borrowed*, not owned, and hence they come with a lifetime indicating when they expire. In standard separation logic, an assertion always represents ownership of some part of the heap, for an unlimited duration (or until the owner actively decides to give it to another party). Instead, a mutable reference in Rust represents ownership *for a limited period of time*. When this lifetime of the reference is over, a mutable reference becomes useless, because the original owner gets back the full ownership.

To handle this new notion of "ownership with an expiry date", we developed a custom logic for reasoning about lifetimes and borrowing. It is called the *lifetime logic*. This logic is embedded and proven correct in Iris, and we describe it in §5. Most importantly, for an Iris assertion P and a lifetime κ, the lifetime logic defines an assertion $\&^{\kappa}_{\mathbf{full}}\, P$, called a *full borrow*, representing ownership of P for the duration of lifetime κ. Using full borrows, the interpretation of the type of mutable references is as follows:

$$[\![\&^{\kappa}_{\mathbf{mut}}\, \tau]\!].\mathrm{size} := 1 \qquad [\![\&^{\kappa}_{\mathbf{mut}}\, \tau]\!].\mathrm{own}(t, \overline{v}) := \exists \ell.\, \overline{v} = [\ell] * \&^{[\![\kappa]\!]}_{\mathbf{full}}\big(\exists \overline{w}, \ell \mapsto \overline{w} * [\![\tau]\!].\, \mathrm{own}(t, \overline{w})\big)$$

This is very similar to the interpretation of $\mathbf{own}_n\, \tau$, except that the assertion describing ownership of the contents of the reference $(\exists \overline{w}, \ell \mapsto \overline{w} * [\![\tau]\!].\, \mathrm{own}(t, \overline{w}))$ is wrapped in a full borrow (at lifetime κ) instead of being owned directly. Finally, it turns out that $\&^{[\![\kappa]\!]}_{\mathbf{full}}\, P$ already functions as a guard of P, so there is no need for us to add any extra later modality $\triangleright$.

4.3 Interpreting shared references

The interpretation of shared references $\&^{\kappa}_{\mathbf{shr}}\, \tau$ requires more work than the types we considered so far. Usually, we would proceed as we did above: Define $[\![\&^{\kappa}_{\mathbf{shr}}\, \tau]\!].\mathrm{own}$ based on $[\![\tau]\!].\mathrm{own}$ such that all the typing rules for $\&^{\kappa}_{\mathbf{shr}}\, \tau$ work out. Most of the time, this does not leave much room for choice; the primitive operations available for the type almost define it uniquely. This is decidedly not the case for shared references, for it turns out that, in Rust, there are hardly any primitive operations on `&T`. The only properties that hold for `&T` in general is that it can be created from a `&mut T`, it is `Copy`, it has size 1, and its values have to be memory locations. However, as we have seen in §2, types like `Cell` or `Mutex` *do* provide some very interesting operations on shared references, *e.g.*, providing indirect mutable access through a shared reference.

To account for this freedom, we permit every type to pick its own *sharing predicate*. We then use the sharing predicate of τ to define $[\![\&^{\kappa}_{\mathbf{shr}}\, \tau]\!].\mathrm{own}$. This permits, for every type, a different set of operations on its shared references. For example, the sharing predicate for basic types like **bool**

allows read-only access, while the sharing predicate for `Mutex<T>` allows read and write accesses to the underlying object of type `T` once the lock has been acquired.

More formally, we extend the semantic domain of types and associate to each of them another predicate $[\![\tau]\!].\mathrm{shr} \in \mathit{Lft} \times \mathit{TId} \times \mathit{Loc} \to \mathit{iProp}$, and use it directly to model shared references:

$$[\![\&^{\kappa}_{\mathbf{shr}}\,\tau]\!].\mathrm{size} := 1 \qquad [\![\&^{\kappa}_{\mathbf{shr}}\,\tau]\!].\mathrm{own}(t, \overline{v}) :=\exists \ell.\, \overline{v} = [\ell] * [\![\tau]\!].\mathrm{shr}([\![\kappa]\!], t, \ell)$$

The $[\![\tau]\!].\mathrm{shr}$ predicate takes three parameters: the lifetime κ of the shared reference, the thread identifier t, and the location ℓ constituting the shared reference itself. Just like `Send` expresses that $[\![\tau]\!].\mathrm{own}$ does not actually depend on the thread identifier (see §4.1), we define `Sync` to mean that $[\![\tau]\!].\mathrm{shr}$ does not depend on the thread identifier. To support the aforementioned primitive operations on `&T`, the sharing predicate has to satisfy the following properties:

$$\mathrm{persistent}([\![\tau]\!].\mathrm{shr}(\kappa, t, \ell)) \tag{TY-SHR-PERSIST}$$

$$\&^{\kappa}_{\mathbf{full}}\big(\exists \overline{w}.\, \ell \mapsto \overline{w} * [\![\tau]\!].\mathrm{own}(t, \overline{w})\big) * [\kappa]_q \Rrightarrow [\![\tau]\!].\mathrm{shr}(\kappa, t, \ell) * [\kappa]_q \tag{TY-SHARE}$$

$$\kappa' \sqsubseteq \kappa \land [\![\tau]\!].\mathrm{shr}(\kappa, t, \ell) \Rightarrow [\![\tau]\!].\mathrm{shr}(\kappa', t, \ell) \tag{TY-SHR-MONO}$$

First, TY-SHR-PERSIST requires that $[\![\tau]\!].\mathrm{shr}$ be persistent, which implies that $[\![\&^{\kappa}_{\mathbf{shr}}\,\tau]\!].\mathrm{own}(t, \overline{v})$ is persistent. This corresponds to the fact that, in Rust, shared references are always `Copy`.

Second, TY-SHARE asserts that shared references can be created from mutable references: This is the main ingredient for proving the rule C-SHARE of the type system. Looking at this rule more closely, its first premise is a full borrow of an owned pointer to τ. This is exactly $[\![\&^{\kappa}_{\mathbf{mut}}\,\tau]\!].\mathrm{own}(t, [\ell])$. Its second premise is a *lifetime token* $[\kappa]_q$, which, as we will explain in §5, witnesses that the lifetime is alive and permits accessing borrows. Given these premises, TY-SHARE states that we can perform an *update*, denoted by the Iris connective $\Rrightarrow$.[6] This update will safely transform the resources described by the premises into those described by the conclusion, namely τ's sharing predicate along with the same lifetime token that was passed in.

Third, TY-SHR-MONO requires that $[\![\tau]\!].\mathrm{shr}$ be monotone with respect to the lifetime parameter. This is important for proving the subtyping rule T-BOR-LFT.

The addition of the sharing predicate completes our description of the semantic domain of types: Each type τ is interpreted by a tuple $[\![\tau]\!] = (\mathrm{size}, \mathrm{own}, \mathrm{shr})$ of a natural number and two Iris predicates that satisfy TY-SIZE, TY-SHR-PERSIST, TY-SHARE and TY-SHR-MONO. Let us now go back to the types we already considered above and define their sharing predicates.

Sharing predicate for products. The sharing predicate for products is simply the separating conjunction of the sharing predicates of the two components:

$$[\![\tau_1 \times \tau_2]\!].\mathrm{shr}(\kappa, t, \ell) := [\![\tau_1]\!].\mathrm{shr}(\kappa, t, \ell) * [\![\tau_2]\!].\mathrm{shr}(\kappa, t, \ell + [\![\tau_1]\!].\mathrm{size})$$

The location used for the second component is shifted by $[\![\tau_1]\!].\mathrm{size}$, reflecting the memory layout.

Sharing predicate for simple types. It turns out that there is a common pattern for defining the sharing predicates of many basic types: Indeed, when no interior mutability is at play, a shared reference provides read-only access. This sharing predicate can be used for any `Copy` type τ of size 1. In this case, $[\![\tau]\!].\mathrm{own}(t, \overline{v})$ can be written in the following form:

$$[\![\tau]\!].\mathrm{own}(t, \overline{v}) = \exists v.\, \overline{v} = [v] * \Phi_\tau(t, v)$$

where Φ_τ is a persistent predicate. This is the case, for example, for **bool**, **int**, function types, and shared references $\&^{\kappa}_{\mathbf{shr}}\,\tau$ themselves. For these types, we use the following sharing predicate:

$$[\![\tau]\!].\mathrm{shr}(\kappa, t, \ell) := \exists v.\ \&^{\kappa}_{\mathbf{frac}}(\lambda q.\, \ell \overset{q}{\mapsto} v) * \triangleright\, \Phi_\tau(t, v)$$

[6] The connective $P \Rrightarrow Q$ is in fact a shorthand for $P \mathrel{-\!\!*} \Rightarrow Q$ in Iris [31].

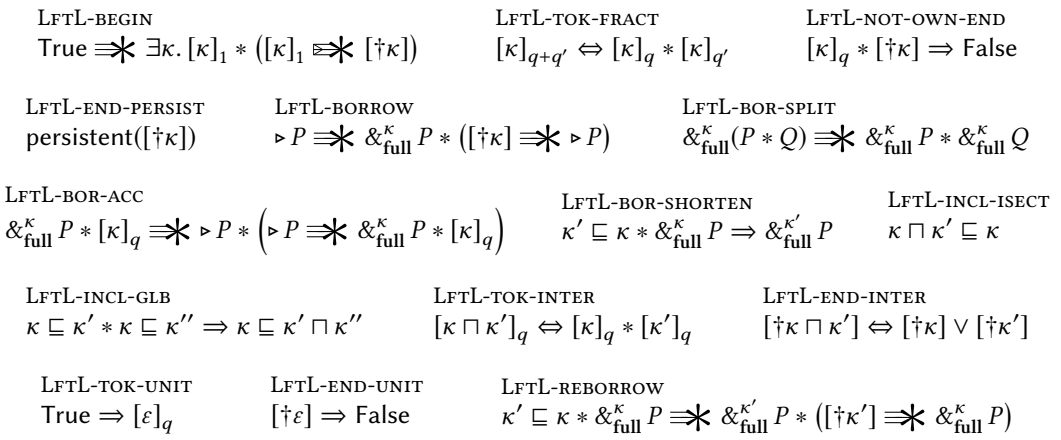

Fig. 2. Selected rules of the lifetime logic.

This definition says that there exists a fixed value v (the current value the reference points to) such that Φ_τ holds under the later modality $\triangleright$ (recall that shared references are pointers, and hence occurrences of τ need to be guarded to enable construction of recursive types), and that we have a *fractured borrow* $\&^\kappa_{\text{frac}}(\lambda q.\, \ell \overset{q}{\mapsto} v)$ of the ownership of the memory cell.

Fractured borrows are another notion provided by the lifetime logic: Similarly to full borrows, they represent temporary ownership of some resource, limited by a given lifetime. The difference is that they are persistent, but only grant *some fraction* of the content. Fortunately, that is all that is needed in order to support a read of the shared reference.

5 LIFETIME LOGIC

In §4, we gave a semantic model for λ_{Rust} types, but we left some important notions undefined. In particular, we used the notion of a *full borrow* $\&^\kappa_{\text{full}} P$ in the interpretation of mutable references to reflect that this kind of ownership is temporary and will "expire" when lifetime κ ends; we mentioned *lifetime tokens* $[\kappa]_q$ as a resource used to witness that a lifetime is ongoing; and we employed *fractured borrows* $\&^\kappa_{\text{frac}} \Phi$ in the sharing predicate of simple types.

In this section, we describe the *lifetime logic*, a library we have developed in Iris to support these notions. In the paper, we focus on discussing the proof rules provided by the library and show how the lifetime logic can be used to model temporary and potentially shared ownership of Iris resources. More details can be found in our technical appendix and in our Coq development [29].

We start by presenting the two core notions of *lifetimes* and *full borrows* in §5.1. We then continue in §5.2, explaining how lifetimes can be *compared* and *intersected*. Finally, in §5.3, we present *fractured borrows*, which we have already seen as being useful for defining sharing predicates.

5.1 Full borrows and lifetime tokens

Figure 2 shows the main rules of the lifetime logic. We explain them by referring to the following Rust example, similar to the one in §2.4:

```
1 let mut v = Vec::new(); v.push(0);
2 { let mut head = v.index_mut(0); *head = 23; }
3 println!("{:?}", v);
```

Recall the type of `index_mut`: `for<'a> fn(&'a mut Vec<i32>, usize) -> &'a mut i32`. To call this function, we need a borrow at some lifetime κ (which we will use to instantiate `'a`). To get started, we need to create this lifetime. This is the role of LftL-begin: it lets us perform an Iris update to create a fresh lifetime κ and gives us the full *lifetime token* $[\kappa]_1$ witnessing that this lifetime is ongoing. (This token can then be split into fractional lifetime tokens $[\kappa]_q$—see below.) It also provides the update $[\kappa]_1 \Rrightarrow\!\!\!\!\!\!\Rightarrow [\dagger\kappa]$: we will use this update later to end κ by exchanging the full lifetime token $[\kappa]_1$ for a *dead token*, written $[\dagger\kappa]$, indicating that κ has ended.[7]

Once the lifetime has been created, we can borrow the vector `v` at the lifetime κ in order to pass a borrowed reference to `index_mut`. This is allowed by LftL-borrow, really the core rule of the lifetime logic. This rule splits ownership of a resource P (in our example, the vector `v`) into the separating conjunction of a *full borrow* $\&^\kappa_{\text{full}} P$ and an *inheritance* $[\dagger\kappa] \Rrightarrow\!\!\!\!\!\!\Rightarrow \triangleright P$. The borrow grants access to P *during* the lifetime κ, while the inheritance allows us to retrieve ownership of P *after* κ has ended. In other words, LftL-borrow *splits ownership in time*. The separating conjunction indicates that the two operands are "disjoint", which means we can safely transfer ownership of the borrow to `index_mut` and keep ownership of the inheritance for ourselves to use later. Except here, this is not disjointness in space (*e.g.*, in the memory), since both the borrow and the inheritance grant access to the same shared resource. Rather, it is disjointness *in time*: The lifetime κ is *either* ongoing *or* ended, so the borrow and the inheritance are never useful at the same time.

We do not give the actual implementation of `index_mut` in this paper. However, here is what `index_mut` does with respect to ownership. First, the ownership of the memory used by the vector ("inside" the full borrow) is split into two parts: (1) The ownership of the accessed vector position, and (2) the ownership of the rest of the vector. Then, the rule LftL-bor-split is used to split the full borrow into two full borrows dedicated to each of these parts. The full borrow of part (1) is returned to the caller; this matches the return type of `index_mut`. On the other hand, the full borrow of part (2) is dropped.[8] This means that the ownership of the rest of the vector is effectively lost until the lifetime ends, at which point it can be recovered using the inheritance.

The next step of our program is the write to `*head` on line 2. Recall that the type of `head` is `&mut i32`, which represents ownership of a full borrow of a single memory location. In order to perform this write, we need to *access* this full borrow and get the resource it contains (in particular, the maps-to predicate $\ell \mapsto \overline{v}$). This is what LftL-bor-acc does: If we give it a full borrow $\&^\kappa_{\text{full}} P$ and a lifetime token $[\kappa]_q$, witnessing that κ is alive, then we get the resource P. Moreover, we also get the update $\triangleright P \Rrightarrow\!\!\!\!\!\!\Rightarrow \&^\kappa_{\text{full}} P * [\kappa]_q$: This can later be used when we are done with P, in order to reconstitute the full borrow and get back the lifetime token. In our proofs, we will always be forced to give back all the lifetime tokens that we obtained; this makes sure that we properly close all borrows again. This can be seen, for example, in Ty-share: A token is provided as a premise to this update, but the same token must also be returned again in the conclusion.

Finally, at the end of line 2 of our example, `head` goes out of scope and it is time to end κ. To this end, we apply the update $[\kappa]_1 \Rrightarrow\!\!\!\!\!\!\Rightarrow [\dagger\kappa]$ that we obtained when κ was created. In doing so, we have to give up the lifetime token $[\kappa]_1$ (ensuring that all borrows are closed again), but we get back the dead token $[\dagger\kappa]$, which can be used to prove that κ has indeed ended. Now that κ has ended, we can use our inheritance $[\dagger\kappa] \Rrightarrow\!\!\!\!\!\!\Rightarrow \triangleright P$ to get back the ownership of `v` before printing it. Note that the dead token $[\dagger\kappa]$ is persistent (LftL-end-persist), so it can be used multiple times—this is

[7] Note that the ending update uses an "update that takes a step" $\Rrightarrow\!\!\!\!\!\!\Rightarrow$ rather than a normal update $\Rrightarrow$. This connective, which is defined in the appendix [29] and is required for technical reasons related to step-indexing, restricts the update to only be used in conjunction with reasoning about a physical step of computation.

[8] Iris is an *affine* logic, in which it is possible to give up ownership of resources at any time, *i.e.*, Iris has the law $P * Q \vdash P$.

important since there may be many borrows (and thus many inheritances we wish to use) at the same lifetime. Each inheritance, however, may only be used once.

One important feature of the lifetime logic that this example does not demonstrate is the parameter q, a fraction. Lifetime tokens can always be split into smaller parts, in a reversible fashion (LftL-tok-fract). This is needed when we want to access several full borrows with the same lifetime at the same time, or to witness that a lifetime is ongoing in several threads simultaneously. Moreover, unsurprisingly, a lifetime cannot be both dead and alive at the same time (LftL-not-own-end).

5.2 Lifetime inclusion

In §2 and §3, we have seen that Rust relates lifetimes by *lifetime inclusion*. This is used for subtyping (T-bor-lft) and reborrowing (C-reborrow).

What does it mean for a lifetime κ to be "included" in another κ'? The key property of lifetime inclusion is that when the shorter κ is still alive, then so is the longer κ'. From the perspective of lifetime tokens, this means that, given a token for κ, we should be able to obtain a token for κ'. Conversely, given a dead token for κ', we should be able to obtain a dead token for κ, as well. This is reflected in the definition of lifetime inclusion:

$$\kappa \sqsubseteq \kappa' \quad := \quad \Box\left(\left(\forall q.\, [\kappa]_q \Rrightarrow \exists q'.\, [\kappa']_{q'} * \left([\kappa']_{q'} \Rrightarrow [\kappa]_q\right)\right) \quad * \quad \left([\dagger\kappa'] \Rrightarrow [\dagger\kappa]\right)\right)$$

The first part says that we can trade a fraction of the token of κ for a potentially different fraction of the token of κ'. It also provides a way to revert this trading to recover the original token of κ, so that no token is permanently lost. The second part of this definition is the analogue for dead tokens. Note that since dead tokens are persistent, it is not necessary to provide a way to recover the dead token that is passed in. The entire definition is wrapped in Iris's always modality $\Box$ to make lifetime inclusion a persistent assertion that can be reused as often as needed.

It is easy to show that lifetime inclusion is a preorder. Inclusion can be used to *shorten* a full borrow (LftL-bor-shorten): If a full borrow is valid for a long lifetime, then it should also be valid for the shorter one. This rule justifies subtyping based on lifetimes in λ_{Rust}.

An even stronger use of lifetime inclusion is *reborrowing*, expressed by LftL-reborrow. This rule is used to prove the reborrowing rule in the type system, C-reborrow. Unlike shortening, reborrowing provides an inheritance to regain the initial full borrow after the shorter lifetime has ended. This may sound intuitively plausible, but turns out to be extremely subtle. In fact, most of the complexity in the model of the lifetime logic arises from reborrowing.

5.2.1 Lifetime intersection. Beyond having a preorder, it turns out that lifetimes also have a greatest lower bound: Given two lifetimes κ and κ', their *intersection* $\kappa \sqcap \kappa'$ is the lifetime that ends whenever either of the operands ends.

Lifetime intersection is particularly useful to create a fresh lifetime that is a sublifetime of some existing κ. We invoke the rule LftL-begin to create an auxiliary lifetime α_0, and then we use the intersection $\alpha := \alpha_0 \sqcap \kappa$ as our new lifetime. It follows that $\alpha \sqsubseteq \kappa$. In the type system, we use this in the proof of F-newlft to create a new lifetime α that is shorter than all the lifetimes in $\overline{\kappa}$.

Intersection of lifetimes interacts well with lifetime tokens: A token of the intersection is composed of tokens of both operands, at the same fraction (LftL-tok-inter). In other words, in order to prove that an intersection is alive, we have to prove that both operands are alive. Similarly, in order to prove that an intersection has ended, it suffices to prove that either operand has ended (LftL-end-inter). These laws let us do the token trading required by lifetime inclusion, showing that intersection indeed is the greatest lower bound for $\sqsubseteq$ (LftL-incl-isect, LftL-incl-glb).

Furthermore, intersection has a unit ε, the never-ending lifetime. It can never be ended (LftL-end-unit) and we can freely get tokens for it (LftL-tok-unit). We use ε to model the **static** lifetime.

LftL-bor-fracture
$$\&^{\kappa}_{\mathbf{full}}\, \Phi(1) \Rrightarrow \&^{\kappa}_{\mathbf{frac}}\, \Phi$$

LftL-fract-acc
$$\frac{\forall q_1, q_2.\ \Phi(q_1 + q_2) \Leftrightarrow \Phi(q_1) * \Phi(q_2)}{\&^{\kappa}_{\mathbf{frac}}\, \Phi * [\kappa]_q \Rrightarrow \exists q'.\ \triangleright \Phi(q') * \left(\triangleright \Phi(q') \Rrightarrow [\kappa]_q\right)}$$

Fig. 3. Selected rules for fractured borrows.

5.3 Fractured borrows

Full borrows and lifetimes are powerful tools for modeling temporary ownership in Iris. However, they cannot be used as is for modeling Rust's shared references. In §4.3, we used the notion of *fractured borrows* as our key notion for defining the default read-only sharing predicate. Figure 3 gives the main reasoning rules for fractured borrows.

To make it possible to use them as a sharing predicate, fractured borrows are persistent and, just like full borrows (LftL-bor-shorten), they can be shortened. Because they are persistent, fractured borrows can potentially be accessed simultaneously by several parties. As such, they cannot provide access to the full underlying resource. Instead, LftL-fract-acc provides access only to *some fraction* of the borrowed content.

To express this, fractured borrows work on a *predicate* Φ over fractions that has to be compatible with addition: $\Phi(q_1 + q_2) \iff \Phi(q_1) * \Phi(q_2)$. When using LftL-fract-acc to access the content of the fractured borrow, we get $\Phi(q)$ for some unknown fraction q. This works because no matter how many threads access the same fractured borrow at the same time, it is always possible to give out *some* tiny fraction of Φ and keep some remainder available for the next thread. Similarly to full borrows, LftL-fract-acc requires a lifetime token for witnessing that the lifetime is alive, and gives back the lifetime token only when the resource is returned.

Fractured borrows can be created from a full borrow of $\Phi(1)$ using the LftL-bor-fracture rule.

6 MODELING TYPES WITH INTERIOR MUTABILITY

As we have discussed in §2.5, the standard library of Rust provides types with *interior mutability*. These types, written in Rust using `unsafe` features, can nonetheless be used safely because the interface they provide to client code encapsulates these unsafeties behind well-typed abstractions. We have proven the safety of several such libraries, namely: `Cell`, `RefCell`, `Mutex`, `RwLock`, `Rc`, and `Arc`.[9] To fulfill this goal, we had to first pick semantic interpretations for the abstract types exported by these libraries (*e.g.*, `Cell<T>`). We then proved that each publicly exported function from these libraries satisfies the semantic interpretation of its type.

Usually, when modeling types with interior mutability, the most difficult definition is that of the sharing predicate $[\![\tau]\!]$.shr. Indeed, these types use a sharing predicate which is different from the default, read-only one that we described in §4.3. The sharing predicates vary greatly depending on which operations are allowed. Most of them use a new variant of borrow propositions, called *persistent borrows*, which we present in this section. As it turns out, all the variants of borrow propositions (including full and fractured borrows) are encodable in terms of a single internal mechanism, called *indexed borrows*, but the explanation of this encoding would take us too far afield (details are explained in the technical appendix [29]). We focus our explanations on two representative forms of interior mutability that we have already presented in §2: `Cell` and `Mutex`.

[9]Note that some simplifications of our setup make the proof of some of these libraries simpler. More precisely, we are not handling unwinding after panics, and all atomic memory operations are sequentially consistent, while Rust's standard library uses weaker atomic accesses.

LftL-bor-na
$\&^{\kappa}_{\text{full}} P \Rrightarrow \&^{\kappa/t}_{\text{na}} P$

LftL-na-acc
$\&^{\kappa/t}_{\text{na}} P * [\kappa]_q * [\text{Na} : t] \Rrightarrow \triangleright P * \left(\triangleright P \Rrightarrow [\kappa]_q * [\text{Na} : t]\right)$

Fig. 4. Selected rules for non-atomic persistent borrows.

6.1 Cell

In §2.5.1, we have seen that `Cell<T>` stores values of type `T` and provides two functions: `get` and `set`, which can be used for reading from and writing to the cell. It turns out that ownership and size of **cell**(τ), the equivalent of `Cell<T>` in λ_{Rust}, are the same as τ. In fact, Rust's standard library provides two functions for converting between `T` and `Cell<T>`, `Cell::new` and `Cell::into_inner`, both of which are effectively the identity function.

The sharing predicate is where things get interesting. Remember that `get` and `set` can be called *even if you only have a shared reference* to a `Cell<T>`. This means that `Cell<i32>` must use a very different sharing predicate than `i32`, which just provides read-only access. In contrast, to verify `set`, we need temporary full access for the duration of the function call. However, it is also important that all shared references to a `Cell` are confined to a single thread, since the `get` and `set` operations are not thread-safe. Recall that Rust enforces this by declaring that `Cell` is not `Sync`, which is equivalent to saying that `&Cell` is not `Send`, so that shared references to it cannot be sent to another thread—they must stay in the thread they have initially been created in.

In order to encode this idea, we use *non-atomic persistent borrows*, another kind of borrow derived from the lifetime logic. Some of their rules are presented in Figure 4. Like fractured borrows, non-atomic persistent borrows are persistent, can be created from full borrows, and support shortening. However, the rule to access the borrows is different: LftL-na-acc gives *full* access to the borrowed content, so it is important that concurrent threads not be allowed to access the same borrow simultaneously. Toward this end, the borrows depend on a thread identifier t. Accessing them requires a *non-atomic token* $[\text{Na} : t]$ bound to that thread identifier. This token is created at the birth of the thread, and threaded through all of its control flow. That is, every function receives it and has to return it. The token is required to open a borrow, and not returned until the borrow is closed, making it impossible to open it twice at the same time.

We can now use non-atomic persistent borrows to give the sharing predicate of **cell**(τ):

$$[\![\mathbf{cell}(\tau)]\!].\text{shr}(\kappa, t, \ell) := \&^{\kappa/t}_{\text{na}} \left(\exists \overline{v}.\, \ell \mapsto \overline{v} * [\![\tau]\!].\text{own}(t, \overline{v})\right)$$

Note, in particular, that our model of `Cell<T>` reflects the fact that it is never `Sync`, since its sharing predicate depends on the thread identifier t. However, if $[\![\tau]\!].\text{own}(t, \overline{v})$ does not depend on t, then neither does $[\![\mathbf{cell}(\tau)]\!].\text{own}(t, \overline{v})$—just like in Rust, `Cell<T>` is `Send` if and only if `T` is.

6.2 Mutex

`Mutex` is the other example of interior mutability that we presented in §2.5. `Mutex<T>` uses a lock to safely grant multiple threads read and write access to a shared object of type `T`.

We start by giving its size and ownership predicate:

$$[\![\mathbf{mutex}(\tau)]\!].\text{size} := 1 + [\![\tau]\!].\text{size} \qquad [\![\mathbf{mutex}(\tau)]\!].\text{own}(t, \overline{v}) := [\![\mathbf{bool} \times \tau]\!].\text{own}(t, \overline{v})$$

That is, when it is not shared, **mutex**(τ) is exactly the same as a pair of a **bool** (representing the status of the lock[10]), and of an object of type τ (the content).

[10]The actual implementation in Rust uses the locking primitives of the operating system. We use our own spinlock-based implementation to model that.

The sharing predicate is more complex: It cannot use fractured borrows, because we cannot afford getting only a fraction of ownership, and it cannot use non-atomic persistent borrows, because mutexes are thread-safe. Instead, it uses yet another kind of borrow, *atomic persistent borrows*, whose rules can be found in the appendix [29]. Again, they are distinguished from the other borrows in the rule granting access to the borrowed content. Here, the mechanism used to prevent two threads accessing the same borrow at the same time is *atomicity*: The proof rules enforce that an atomic persistent borrow cannot be opened for longer than a single, atomic instruction. Thus, during the execution of any given instruction, only one thread can be accessing the borrow. Returning to `Mutex`'s sharing predicate, the content of its borrow will only get accessed when changing the status of the lock, and doing so will require atomic memory accesses. Of course, this corresponds to the fact that, in our spinlock implementation, we are only using atomic sequentially consistent instructions to read or write the status flag. Using non-atomic accesses would lead to data races.

Using atomic persistent borrows, we can give the sharing predicate for mutexes:

$$\begin{aligned}&[\![\mathbf{mutex}(\tau)]\!].\mathrm{shr}(\kappa, t, \ell) := \exists \kappa'.\, \kappa \sqsubseteq \kappa' * \\ &\qquad \&^{\kappa}_{\mathrm{at}}\left(\ell \mapsto \mathbf{true} \quad \vee \quad \ell \mapsto \mathbf{false} * \&^{\kappa'}_{\mathrm{full}}\left(\exists \overline{v}.\, (\ell+1) \mapsto \overline{v} * [\![\tau]\!].\mathrm{own}(t, \overline{v})\right)\right)\end{aligned}$$

This is quite a mouthful: First, we use an existential quantification at the beginning to close the predicate under shorter lifetimes, satisfying TY-SHR-MONO. We use an atomic persistent borrow to share ownership of the status flag at location ℓ. This defines an invariant that is maintained *until* κ ends. The invariant can be in one of two states: In the first state, the flag is **`true`**, in which case the lock is locked and no other resource is stored in the borrow. Ownership of the content is currently held by whichever thread acquired the lock. In the second state, the flag is **`false`**. This means the lock is unlocked, and the borrow also stores the ownership of the content at type τ at location $\ell + 1$. When acquiring or releasing the lock, we can atomically open the persistent borrow and change the branch of the disjunction, thus acquiring or releasing ownership of the content.

Curiously, ownership of the content is wrapped in a full borrow. One might expect instead that it should be directly contained in the outer persistent borrow. In this case, acquiring the lock would result in acquiring full (unborrowed) ownership of the content of the mutex. That, however, does not work: Imagine κ' ends while the lock is held. (That is possible, for example, if the `MutexGuard` is leaked and hence its destructor never gets called.) In this case, ownership of the content would never be returned to the borrow. However, when κ' ends, the `Mutex` is again fully owned by someone, which means *they* expect to be the exclusive owner of the content! This is why the full borrow is necessary: When taking the lock, one gets the inner resource only under a borrow at lifetime κ', guaranteeing that ownership is returned when κ' ends.

To conclude, observe that if the inner type does not depend on the thread identifier t (which corresponds to saying that it is `Send`), then neither $[\![\mathbf{mutex}(\tau)]\!].\mathrm{own}$ nor $[\![\mathbf{mutex}(\tau)]\!].\mathrm{shr}$ do, so that $\mathbf{mutex}(\tau)$ is both `Send` and `Sync`. This exactly corresponds to Rust's behavior.

7 PROOF OF SOUNDNESS

Having defined the semantics of types in §4, we can finish up our formal development by defining semantic interpretations of the judgments presented in §3.3. We focus on the two most important ones: typing of instructions and typing of function bodies. Their interpretations use Hoare triples:

$$\begin{aligned}&\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{T}_1 \models I \dashv x.\, \mathbf{T}_2 := \\ &\qquad \forall \gamma, t.\, \{[\![\mathbf{E}]\!]_\gamma * [\![\mathbf{L}]\!]_\gamma * [\mathrm{Na} : t] * [\![\mathbf{T}_1]\!]_\gamma(t)\}\ I\ \{v.\, [\![\mathbf{L}]\!]_\gamma * [\mathrm{Na} : t] * [\![\mathbf{T}_2]\!]_{\gamma[x \leftarrow v]}(t)\} \\ &\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \models F := \forall \gamma, t.\, \{[\![\mathbf{E}]\!]_\gamma * [\![\mathbf{L}]\!]_\gamma * [\mathrm{Na} : t] * [\![\mathbf{T}]\!]_\gamma(t) * [\![\mathbf{K}]\!]_\gamma(t)\}\ F\ \{\mathrm{True}\}\end{aligned}$$

In the preconditions, we can find the interpretations of the various contexts, together with the thread-local token [Na : t] used to access the non-atomic persistent borrows of thread t. The instruction judgment nicely demonstrates how this token is threaded through, alongside the local lifetime context $[\![\mathbf{L}]\!]_\gamma$ which contains all the lifetime tokens. The function judgment, on the other hand, has a trivial post-condition—remember that functions do not return, they call a continuation. The Hoare triple for that continuation is provided by $[\![\mathbf{K}]\!]_\gamma(t)$, and it will require [Na : t] as well as $[\![\mathbf{L}]\!]_\gamma$ in its precondition, which is how the tokens travel between functions. The interpretation $[\![\mathbf{E}]\!]_\gamma$ just uses the lifetime inclusion from the lifetime logic; this makes it persistent, so it does not have to be threaded through. Finally, $[\![\mathbf{T}]\!]_\gamma(t)$ uses the semantic interpretation of types as defined in the previous sections, tying them to the current thread t.

With this defined, we can state two core theorems showing the soundness of our type system. The first one shows a deep connection between the semantic judgments and their syntactic counterparts.

Theorem 7.1 (Fundamental theorem of logical relations). *For any inference rule of the type system, when we replace all $\vdash$ by $\models$, the resulting Iris theorem holds.*

One important corollary of the fundamental theorem is that if a judgment can be derived syntactically, then it also holds semantically. However, Theorem 7.1 is much stronger than this, because we can use it to glue together safe and unsafe code. Given a program that is syntactically well-typed except for certain components that are only semantically (but not syntactically) well-typed, the fundamental theorem tells us that the entire program is semantically well-typed.

The second theorem is an adequacy theorem, relating the logical relation to program behavior:

Theorem 7.2 (Adequacy). *Let f be a λ_{Rust} function such that $\emptyset \mid \emptyset; \emptyset \mid \emptyset \models f \mathrel{=\!|} x.\, x \lhd \mathbf{fn}() \to \Pi[]$ holds. Then when we execute f with the default continuation (which is just a no-op), no execution ends in a stuck state.*

In particular, the adequacy theorem guarantees that a semantically well-typed program is memory and thread safe: It will never perform any invalid memory access and will not have data races.

Put together, these theorems establish that, *if the only code in a λ_{Rust} program that is not syntactically well-typed appears in semantically well-typed libraries, then the program is safe to execute.*

8 RELATED WORK

Substructural type systems for state, and their soundness proofs. Over the past decades, numerous languages and type systems have been developed that use linear types [65], ownership [16], and/or regions [24] to guarantee safety of heap-manipulating programs. These include Cyclone [27], Vault [18], and Alms [61]. Much of this work has influenced the design of Rust, but a detailed discussion of that influence is beyond the scope of this paper. The key point for our purposes is that these systems are *closed-world*, meaning that they are defined by a fixed set of rules and are proven sound using *syntactic* techniques [66]. As explained in §1.1, Rust's extensible type system fundamentally does not fit into this paradigm. In a related but very different line of work, systems like Ynot [44], FCSL [43], and F* [54] integrate variants of separation logic into dependent type theory. These systems are aimed at full functional verification of low-level imperative code and thus require a significant amount of manual proof and/or type annotations compared to Rust.

Mezzo [9] can be placed somewhere between these two approaches. It comes with a substructural type system whose expressivity parallels that of a separation logic. Its soundness proof is modular in the sense that the authors start by verifying a core type system, and then add various extensions. This relies on an abstract notion of resources called *monotonic separation algebras*. Nevertheless, Mezzo's handling of types remains entirely syntactic (*e.g.*, based on the grammar of types); there is no semantic account for types that would permit "adding" new types without revisiting the proofs.

Cogent [4, 45] is a purely functional, linearly typed language designed to implement file systems and verify their functional correctness. Its linear type system permits efficient compilation to machine code using in-place updates, while the purely functional semantics enables equational reasoning. Its design is such that missing functionality can be implemented in C functions (much like unsafe code in Rust), which are given types to enforce correct usage in the Cogent program. These C functions are then manually verified to implement an equational specification and to follow the guarantees of the type system. However, the language and the type system are much simpler than Rust's (*e.g.*, there is no support for recursion, iteration, borrowing, or mutable state).

Formal results for Rust. Patina [48] is a formalization of the Rust type system, with accompanying partial proofs of progress and preservation. Being syntactic, these proofs do not scale to account for `unsafe` code. To keep our formalization feasible, we did not reuse the syntax and type system of Patina, but rather designed λ_{Rust} from scratch in a way that better fits the Iris setup.

CRUST [60] is a bounded model checker designed to verify the safety of Rust libraries implemented using unsafe code. It checks that all clients calling up to n library methods do not trigger memory safety faults. This provides an easy-to-use, automated way of checking unsafe code, before attempting a full formal proof. Their approach has successfully re-discovered some soundness bugs that had already been fixed in Rust's standard library. However, by only considering one library at a time, it cannot find bugs like [12] that arise from the interaction of multiple libraries.

Concurrent separation logics. RustBelt builds on the Iris framework [31], which in turn incorporates several great advances made in the past decade in the area of concurrent separation logics [6, 19–21, 43, 46, 53]. In particular, RustBelt depends crucially on Iris's support for: (1) custom notions of logical resource (*i.e.*, "fictional separation" [26]), which we use to model novel abstract predicates like the various forms of borrow propositions; (2) impredicative invariants [53], which we use to model higher-order state; and (3) support for tactical proofs in Coq [36], without which a verification of the scale and complexity of RustBelt would not be possible.

One recent innovation in separation logics is *temporary read-only permissions* [15]. The authors introduce a duplicable "read-only" modality with rules that resemble ours for shared references at "simple" types like `i32`. However, since shared references permit interior mutability, the read-only permission is not suited to directly modeling shared references. Nevertheless, it would be interesting to explore whether this approach can facilitate the tracking of lifetime tokens, just like read-only permissions eliminate the bookkeeping involved in fractional permissions. One challenge here is that λ_{Rust} supports non-lexical lifetimes [40], whereas read-only permissions are strictly lexical.

9 CONCLUSION

We have described λ_{Rust}, a formal version of the Rust type system that we used to study Rust's ownership discipline in the presence of `unsafe` code. We have shown that various important Rust libraries with unsafe implementations, many of them involving *interior mutability*, are safely encapsulated by their type. We had to make some concessions in our modeling: We do not model (1) more relaxed forms of atomic accesses, which Rust uses for efficiency in libraries like `Arc`; (2) Rust's trait objects (comparable to interfaces in Java), which can pose safety issues due to their interactions with lifetimes; or (3) stack unwinding when a panic occurs, which causes issues similar to exception safety in C++ [2]. We proved safety of the destructors of the verified libraries, but do not handle automatic destruction, which has already caused problems [11] for which the Rust community still does not have a modular solution [58]. The remaining omissions are mostly unrelated to ownership, like traits (Rust's take on type classes) and type-polymorphic functions.

Despite these limitations, we believe we have captured the essence of Rust's ownership discipline. The framework provided by the lifetime logic proved flexible enough to handle functions that are correct for subtle reasons, like `Ref::map` and `RefMut::map`, part of `RefCell`, which had to have

their signature changed from the initial design to ensure soundness [50]. In fact, our verification work resulted in uncovering and fixing a bug in Rust's standard library [28], demonstrating that our model of Rust is realistic enough to be useful. Furthermore, our type system already handles features that are still being sketched for Rust itself, like non-lexical lifetimes [40], and we are in active discussion with the Rust community on these topics.

In ongoing and future work, we plan to fill some of the gaps mentioned above and to bring λ_{Rust} closer to MIR, the most important intermediate language in the Rust compiler. Concretely, we would like to make the fact that all local variables are heap-allocated more implicit, and to extend paths to include dereferencing a pointer. That should permit us to reduce the number of primitive instructions, making each of them correspond to exactly one construct in MIR.

ACKNOWLEDGMENTS

We wish to thank the Rust community in general, and Aaron Turon and Niko Matsakis in particular, for their feedback and countless helpful discussions.

This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289).

REFERENCES

[1] ISO Working Group 21. 2011. Programming languages – C++. (2011).
[2] David Abrahams. 1998. Exception-safety in generic components. In *International Seminar on Generic Programming (LNCS)*. https://doi.org/10.1007/3-540-39953-4_6
[3] Amal Jamil Ahmed. 2004. *Semantics of types for mutable state.* Ph.D. Dissertation. Princeton University.
[4] 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. In *ASPLOS*. ACM. https://doi.org/10.1145/2872362.2872404
[5] Andrew W. Appel. 2007. *Compiling with Continuations.* Cambridge University Press.
[6] Andrew W. Appel (Ed.). 2014. *Program Logics for Certified Compilers.* Cambridge University Press.
[7] Andrew W. Appel and David McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS* 5 (2001). https://doi.org/10.1145/504709.504712
[8] Andrew W. Appel, Paul-André Melliès, Christopher Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *POPL*. https://doi.org/10.1145/1190216.1190235
[9] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2016. The design and formalization of Mezzo, a permission-based programming language. *TOPLAS* 38, 4 (2016). https://doi.org/10.1145/2837022
[10] Ariel Ben-Yehuda. 2015. Can mutate in match-arm using a closure. (2015). Rust issue #27282. https://github.com/rust-lang/rust/issues/27282.
[11] Ariel Ben-Yehuda. 2015. dropck can be bypassed via a trait object method. (2015). Rust issue #26656. https://github.com/rust-lang/rust/issues/26656.
[12] Ariel Ben-Yehuda. 2015. std::thread::JoinGuard (and scoped) are unsound because of reference cycles. (2015). Rust issue #24292. https://github.com/rust-lang/rust/issues/24292.
[13] Christophe Biocca. 2017. std::vec::IntoIter::as_mut_slice borrows &self, returns &mut of contents. (2017). Rust issue #39465. https://github.com/rust-lang/rust/issues/39465.
[14] John Boyland. 2003. Checking interference with fractional permissions. In *SAS (LNCS)*. https://doi.org/10.1007/3-540-44898-5_4
[15] Arthur Charguéraud and François Pottier. 2017. Temporary read-only permissions for separation logic. In *ESOP (LNCS)*. https://doi.org/10.1007/978-3-662-54434-1_10
[16] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership types for flexible alias protection. In *OOPSLA*. https://doi.org/10.1145/286936.286947
[17] The Coq team. 2017. The Coq proof assistant. (2017). https://coq.inria.fr/.
[18] Robert DeLine and Manuel Fähndrich. 2001. Enforcing high-level protocols in low-level software. In *PLDI*. ACM. https://doi.org/10.1145/381694.378811

[19] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: Compositional reasoning for concurrent programs. In *POPL*. ACM. https://doi.org/10.1145/2429069.2429104

[20] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent abstract predicates. In *ECOOP (LNCS)*. https://doi.org/10.1007/978-3-642-14107-2_24

[21] Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. 2009. Deny-guarantee reasoning. In *ESOP (LNCS)*. https://doi.org/10.1007/978-3-642-00590-9_26

[22] Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical step-indexed logical relations. *LMCS* 7, 2:16 (2011). https://doi.org/10.2168/LMCS-7(2:16)2011

[23] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. 2010. A relational modal logic for higher-order stateful ADTs. In *POPL*. ACM. https://doi.org/10.1145/1706299.1706323

[24] Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. 2006. Linear regions are all you need. In *ESOP (LNCS)*. https://doi.org/10.1007/11693024_2

[25] Douglas Gregor and Sibylle Schupp. 2003. Making the usage of STL safe. In *IFIP TC2 / WG2.1 Working Conference on Generic Programming*. https://doi.org/10.1007/978-0-387-35672-3_7

[26] Jonas Braband Jensen and Lars Birkedal. 2012. Fictional separation logic. In *ESOP (LNCS)*. https://doi.org/10.1007/978-3-642-28869-2_19

[27] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*.

[28] Ralf Jung. 2017. MutexGuard<Cell<i32>> must not be Sync. (2017). Rust issue #41622. https://github.com/rust-lang/rust/issues/41622.

[29] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language – Technical appendix and Coq development. (2017). http://www.mpi-sws.org/~dreyer/papers/rustbelt/appendix.zip.

[30] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. ACM. https://doi.org/10.1145/2951913.2951943

[31] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2017. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. (2017). Submitted for publication.

[32] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*. https://doi.org/10.1145/2676726.2676980

[33] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In *ECOOP*.

[34] Robbert Krebbers. 2015. *The C standard formalized in Coq*. Ph.D. Dissertation. Radboud University.

[35] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The essence of higher-order concurrent separation logic. In *ESOP (LNCS)*. https://doi.org/10.1007/978-3-662-54434-1_26

[36] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. ACM. https://doi.org/10.1145/3009837.3009855

[37] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A relational model of types-and-effects in higher-order concurrent separation logic. In *POPL*. ACM. https://doi.org/10.1145/3009837.3009877

[38] Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert memory model, version 2*. Technical Report RR-7987. Inria.

[39] Nicholas D. Matsakis. 2016. Introducing MIR. (2016). https://blog.rust-lang.org/2016/04/19/MIR.html.

[40] Nicholas D. Matsakis. 2016. Non-lexical lifetimes: Introduction. (2016). http://smallcultfollowing.com/babysteps/blog/2016/04/27/non-lexical-lifetimes-introduction/.

[41] Nicholas D. Matsakis and Felix S. Klock II. 2014. The Rust language. In *SIGAda Ada Letters*, Vol. 34. ACM. https://doi.org/10.1145/2663171.2663188

[42] Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978). https://doi.org/10.1016/0022-0000(78)90014-4

[43] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating state transition systems for fine-grained concurrent resources. In *ESOP (LNCS)*. https://doi.org/10.1007/978-3-642-54833-8_16

[44] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: Dependent types for imperative programs. In *ICFP*. ACM. https://doi.org/10.1145/1411204.1411237

[45] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. 2016. Refinement through restraint: Bringing down the cost of verification. In *ICFP*. ACM. https://doi.org/10.1145/2951913.2951940

[46] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theoretical computer science* 375, 1-3 (2007). https://doi.org/10.1016/j.tcs.2006.12.035

[47] Gordon Plotkin. 1973. Lambda-definability and logical relations. (1973). Unpublished manuscript.
[48] Eric Reed. 2015. *Patina: A formalization of the Rust programming language.* Master's thesis. University of Washington.
[49] John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *LICS*. IEEE. https://doi.org/10.1109/LICS.2002.1029817
[50] Simon Sapin. 2015. Add std::cell::Ref::map and friends. (2015). Rust PR #25747. https://github.com/rust-lang/rust/pull/25747#issuecomment-105175235.
[51] Josh Stone and Nicholas D. Matsakis. 2017. The Rayon library. (2017). https://crates.io/crates/rayon.
[52] Bjarne Stroustrup. 2012. Foundations of C++. In *ESOP*. https://doi.org/10.1007/978-3-642-28869-2_1
[53] Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *ESOP (LNCS)*. https://doi.org/10.1007/978-3-642-54833-8_9
[54] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent types and multi-monadic effects in F*. In *POPL*. ACM. https://doi.org/10.1145/2837614.2837655
[55] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. Conditionally accepted to OOPSLA.
[56] W. W. Tait. 1967. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic* 32, 2 (1967). https://doi.org/10.2307/2271658
[57] Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A higher-order logic for concurrent termination-preserving refinement. In *ESOP (LNCS)*. https://doi.org/10.1007/978-3-662-54434-1_10
[58] The Rust team. 2016. Drop Check. In *The Rustonomicon*. https://doc.rust-lang.org/nomicon/dropck.html.
[59] The Rust team. 2017. The Rust programming language. (2017). http://rust-lang.org/.
[60] John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. CRUST: A bounded verifier for Rust. In *ASE*. IEEE. https://doi.org/10.1109/ASE.2015.77
[61] Jesse A. Tov and Riccardo Pucella. 2011. Practical affine types. In *POPL*. ACM. https://doi.org/10.1145/1926385.1926436
[62] Aaron Turon. 2015. Abstraction without overhead: Traits in Rust. (2015). https://blog.rust-lang.org/2015/05/11/traits.html.
[63] Aaron Turon. 2015. Implied bounds on nested references + variance = soundness hole. (2015). Rust issue #25860. https://github.com/rust-lang/rust/issues/25860.
[64] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*. ACM. https://doi.org/10.1145/2500365.2500600
[65] Philip Wadler. 1990. Linear types can change the world! In *Programming Concepts and Methods*.
[66] Andrew K Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and computation* 115, 1 (1994). https://doi.org/10.1006/inco.1994.1093