VerusBelt: A Semantic Foundation for Verus's Proof-Oriented Extensions to the Rust Type System

TRAVIS HANCE, MPI-SWS, Germany
LAILA ELBEHEIRY, MPI-SWS, Germany
YUSUKE MATSUSHITA, Kyoto University, Japan
DEREK DREYER, MPI-SWS, Germany

Verus is a verification tool for the Rust programming language that has already been put to use in several significant systems verification efforts. Verus offers a distinctive approach among Rust verification systems in that, in addition to offering fast SMT-based automation, it supports verification of both safe and unsafe Rust code in a single, unified framework. It achieves this by providing users with a range of *proof-oriented types*—types introduced specifically by Verus to aid in verification—on top of which one can then build verified implementations of Rust APIs that would otherwise require the use of unsafe Rust features.

In this paper, we develop VerusBelt, the first semantic soundness proof for a significant subset of Verus. In addition to modeling the full range of Verus's core proof-oriented types—including cells, invariants, resource algebras, and storage protocols—VerusBelt accounts (for the first time) for full-fledged Rust lifetimes, concurrency and thread safety, and mutable borrows. A central challenge involves building a model of shared borrows that is generic enough to support storage protocols—we tackle this by marrying RustBelt's lifetime logic with the Leaf library for temporary resource sharing in Iris. All our proofs are mechanized in Iris/Rocq.

CCS Concepts: • **Theory of computation** → **Type theory**; **Separation logic**.

Additional Key Words and Phrases: Rust, Verus, semantic typing, Leaf, lifetime logic, Iris, Rocq

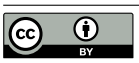
ACM Reference Format:
Travis Hance, Laila Elbeheiry, Yusuke Matsushita, and Derek Dreyer. 2026. VerusBelt: A Semantic Foundation for Verus's Proof-Oriented Extensions to the Rust Type System. *Proc. ACM Program. Lang.* 10, PLDI, Article 247 (June 2026), 25 pages. https://doi.org/10.1145/3808325

1 Introduction

In the past decade, the Rust programming language has garnered a great deal of attention from hackers and researchers alike, due to its powerful support for *safe systems programming*. In particular, like C/C++, Rust supports low-level control over memory management and data layout, but it also prevents leading sources of undefined behavior in C/C++ programs, such as memory safety violations and data races. It does so by employing an ownership-based type system to enforce the principle of *aliasing XOR mutability* (AXM): data can be aliased (*i.e.,* have multiple references to it), or it can be mutable, but it cannot be both at the same time.

Rust's strong type-based guarantees have not only made it a promising tool for building safer systems—they have also made Rust a prime target for *formal verification* of systems code. Specifically, a number of researchers have developed tools that exploit the ownership tracking in Rust types

Authors' Contact Information: Travis Hance, MPI-SWS, Saarland Informatics Campus, Germany, thance@mpi-sws.org; Laila Elbeheiry, MPI-SWS, Saarland Informatics Campus, Germany, lelbehei@mpi-sws.org; Yusuke Matsushita, Kyoto University, Kyoto, Japan, ymat@fos.kuis.kyoto-u.ac.jp; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.



ACM 2475-1421/2026/6-ART247
https://doi.org/10.1145/3808325

in order to simplify or guide the verification of Rust programs [7, 2, 15]. For example, it has been observed that programs that adhere to the AXM discipline can be given "purely functional" specifications in first-order logic rather than requiring the use of a more complex stateful program logic (such as separation logic) that is more difficult to automate [29].

However, there is a fly in the ointment: for implementing certain abstractions (even something as simple as a doubly-linked list), it is necessary to mutate shared state in a way that is in fact safe but that (by definition) violates AXM. In such cases, Rust programmers must make careful use of *unsafe* features of the language, such as raw pointer accesses or type casts, and hope that they are not breaking Rust's language-level safety guarantees when they do so. Unfortunately, these unsafe features also make the problem of verifying Rust programs significantly harder.

Several approaches have been proposed for verifying Rust programs that mix safe and unsafe code. Most of them require APIs implemented using unsafe code to be verified using a different, less automated system than the SMT-based automated system used for safe code [28, 3]. One exception is RefinedRust [10], which supports proofs of safe and unsafe code in a unified Rocq-based system, but it does not offer the same degree of automation as SMT-based verification tools.

In this paper, we focus on **Verus** [22], one of the most prominent and widely deployed systems for verification of Rust code, having been utilized in a number of systems verification efforts [4, 21, 24, 27, 31, 32, 36]. Verus offers a distinctive approach among Rust verification systems in that, in addition to offering fast SMT-based automation, it supports verification of both safe and unsafe Rust code in a single, unified framework. It achieves this combination of features by leveraging the idea of *proof-oriented programming* [9]: rather than taking some pre-existing Rust code that uses unsafe features and trying to verify it *after the fact*, Verus instead provides a set of primitive, *proof-oriented* Rust types with which programmers can employ unsafe features and verify that they are doing so correctly *at the same time*. We call these Verus-specific types "proof-oriented" because they do not exist in standard Rust and are designed specifically to facilitate proofs in Verus.

The proof-oriented types of Verus have already been put to work in a range of nontrivial systems verification projects, including a concurrent memory allocator and node replication algorithm [21], the persistent key-value store CapybaraKV [24], and the VERISMO security module [36]. They have also been used to verify the correctness of many widely used Rust APIs, including `Rc`, `Arc`, `RefCell`, and `RwLock` [22]. However, a key question remains: **Is the Verus type system sound?** The initial work on Verus [22] formalized a small subset of its type system, which modeled some features of Verus but only in a highly idealized way. For example, it only modeled Rust's lifetimes as a two-element lattice, and only formalized a simplified form of the most basic Verus proof-oriented type for single-pointer permissions. In particular, it did not provide any semantic account of the more sophisticated proof-oriented types of Verus that are needed to verify the Rust APIs mentioned above, nor did it model concurrency at all.

Our goal is to fill this gap. Towards that end, we develop **VerusBelt**: a soundness proof for a much more comprehensive subset of Verus. VerusBelt accounts (for the first time) for: full-fledged Rust lifetimes; concurrency and thread safety; mutable borrows; the proof-oriented Verus types `PPtr`, `PCell`, `PointsTo`, `LocalInvariant`, and `AtomicInvariant`, as well as Verus's more complex algebraic mechanisms for ghost state construction (also encoded as proof-oriented types), namely "resource algebras" [20] and "storage protocols" [13].

VerusBelt takes as its starting point the foundational work on RustBelt [18] and RustHornBelt [28]. RustBelt was the first work to offer a soundness proof for a significant subset of the Rust type system, as well as widely-used APIs like `Rc`, `Arc`, `Cell`, and `RwLock`. Specifically, RustBelt defined a somewhat idealized λ-calculus approximation of Rust (called λ_{Rust}) and proved *semantic soundness* of λ_{Rust} [33, 1] by defining a *semantic model* of λ_{Rust} types, interpreting them as predicates in the Iris separation logic [20, 19]. The use of a semantic soundness argument was crucial for verifying

safety of Rust APIs implemented with unsafe features. Building on RustBelt, RustHornBelt went a step further: from safety to functional verification. Specifically, it extended the typing judgment of RustBelt with a functional specification of the behavior of the code being typed. The resulting *type-spec* judgment was used to give a semantic foundation for the Creusot automated verification tool [7], as well as a formal justification (in Iris) for when it is correct to assign a Creusot functional specification to a Rust API built atop unsafe code.

To a first approximation, VerusBelt adopts the approach of RustHornBelt, applying it to develop a semantic foundation for Verus and its portfolio of proof-oriented types.

But the devil is in the details, and there are two primary challenges we encountered in adapting RustHornBelt to Verus, both related to Rust's core mechanism of *borrowing*. In Rust, data of type `T` may be "borrowed" in one of two ways: either through *shared references* (`&T`) or through *mutable references* (`&mut T`). Following the AXM principle, shared references may be freely copied, but (by default) give only read access to the underlying data, whereas mutable references are unique but permit read/write access. Both mechanisms pose challenges for proving the soundness of Verus.

Challenge #1: A uniform approach to modeling shared references. Although shared references by default only allow read access to the underlying data, some Rust APIs (such as `UnsafeCell`, `Cell`, and `RefCell`) allow mutation through a shared reference. One representative example is `Mutex<T>`, which synchronizes access to data of type `T` through a lock: in order to be useful, references to the `Mutex` must be *shared* between threads, but acquiring the lock through such a shared reference will grant the caller a *mutable* reference of the underlying `T` (this is the entire point of the `Mutex`). In Rust lingo, APIs like `Mutex` are thus said to exhibit *interior mutability*.

RustBelt accounts for the safety of interior-mutable APIs by organizing the semantic model of Rust types in an unusual way. In addition to the primary semantic interpretation of a type `T` ($[\![\texttt{T}]\!]$.own)—describing (in separation logic) what it means to "own" a value of type `T`—RustBelt also equips each type with a "shared" interpretation ($[\![\texttt{T}]\!]$.shr). The shared interpretation in effect says what it means to own a shared reference of `T` (hence, $[\![\texttt{\&T}]\!]$.own is defined to be essentially $[\![\texttt{T}]\!]$.shr). This shared interpretation can be flexibly defined in different ways for different types; so although RustBelt provides a default, "read-only" sharing interpretation for most types, it also allows interior-mutable types to define $[\![\texttt{T}]\!]$.shr in such a way that a shared reference can be used to acquire mutable access to the underlying data.

For Verus, however, we need a model of shared references that is less flexible and more uniform than RustBelt's. The reason stems from a fundamental difference in how interior-mutable APIs are implemented in Rust vs. in Verus. In Rust, interior-mutable APIs like `Arc` and `RwLock` are implemented directly using low-level unsafe primitives; correspondingly, RustBelt's proof of soundness of each of these APIs had to be done completely independently—and by defining completely different shared interpretations. In Verus, on the other hand, these APIs are implemented *in safe code* atop a subset of Verus's proof-oriented types. In particular, `Arc` and `RwLock` both leverage an advanced Verus proof-oriented API—*storage protocols* [13]—which provides a highly configurable interface for enforcing safe usage of shared state. Thus, to verify the soundness of Verus, it no longer suffices to give arbitrarily different formal accounts of sharing at different interior-mutable types: we need to develop a unified formal account of the *generic sharing* provided by storage protocols.

We achieve this goal by building on Leaf [13], an Iris library with a generic mechanism for expressing the temporary sharing of resources. Concretely, we show how to merge Leaf's generic account of resource sharing together with RustBelt's "lifetime logic", yielding the *Leaf Lifetime Logic* (LeLiLo). With LeLiLo in hand, we are able to define a simple and uniform model of shared references which encompasses standard Rust types as well as the proof-oriented types of Verus.

Challenge #2: Sound incorporation of mutable references into Verus. At present, Verus has limited support for mutable references, allowing them only in certain contexts. However, it is a longstanding goal of the Verus developers to extend it to support general, Rust-style mutable references. In this paper, we develop a semantic foundation for how Verus can be thus extended.

Our basic idea is to follow RustHornBelt's groundbreaking approach of modeling mutable references using *prophecy variables*. Specifically, RustHornBelt models mutable references as a pair of two values—the *current* value stored in the mutable reference and the *final* value that will be stored in it when the borrow is returned. As the final value is not known ahead of time, it is represented using a prophecy variable.

Unfortunately, we cannot simply adopt RustHornBelt's prophecy-based approach out of the box; doing so naively results in a classic time-travel paradox involving Verus ghost state, in which the future value of a mutable reference can be observed and used to update its current value in an inconsistent manner. In §5, we show how the RustHornBelt model of types can be adjusted so that this paradox can be avoided. The key idea is to impose a distinction between prophecy-dependent and prophecy-independent values in our modeling of Verus types. With this distinction in place, we can place restrictions on certain typing rules so as to limit the flow of prophecy-dependent information, recovering a sound encoding of mutable references in Verus.

Contributions. The main contributions of **VerusBelt** are as follows:

- A comprehensive semantic soundness proof for a subset of Verus (as approximated in the idealized λ_{Rust}), focused primarily on Verus's proof-oriented types (`PPtr`, `PCell`, `PointsTo`, `LocalInvariant`, `AtomicInvariant`, resource algebras, and storage protocols) together with a standard suite of Rust primitive types (§3–§4).
- The Leaf Lifetime Logic (LeLiLo): a marriage of two Iris libraries (Leaf [13] and the "lifetime logic" from RustBelt [18]), allowing resources to be shared *generically* and *temporally* (§3.3).
- A uniform model of shared references, needed for modeling Verus's highly generic APIs like storage protocols and enabled by LeLiLo's $\&^{\alpha}P$ primitive (§3.3).
- The Points-To Cell Logic: An extension of points-to resources to support chains of abstract cells $\ell \vdash [\vec{\delta}] \mapsto v$, enabling compositional reasoning about interior mutability (§3.4).
- A semantic foundation for extending Verus with full-blown mutable references by adapting RustHornBelt's prophecy-based model of mutable references [28] and stratifying it to avoid time-travel paradoxes (§5).
- A mechanization of our proofs in the Rocq proof assistant (see our artifact).

Non-goals. We do not attempt to verify the actual implementation of Verus itself; rather, our efforts are focused on providing a semantic foundation for Verus's proof-oriented types. Furthermore, we do not attempt to verify common Rust libraries (like `Vec` or `HashMap`), which are also currently axiomatized by Verus. See §6 for more on the gap between VerusBelt and Verus.

We begin with a brief overview of Verus and its proof-oriented types (§2), and conclude the paper with a summary of our results and limitations (§6) and a comparison with related work (§7).

2 Verus Overview

Verus allows Rust code to be extended with proofs and specifications via a syntactic extension to Rust. For example, we can add a specification to a function via a `requires` clause (precondition) and an `ensures` clause (postcondition):

```
1 fn example_add(a: u64, b: u64) -> (return_value: u64)
2     requires a <= 1000, b <= 900, ensures return_value <= 1900,
3 { return a + b; }
```

(The named return value is also part of the syntactic extension.) Verus checks the proof obligations associated with the program; in this case, it would check the `ensures` clause holds at the end of the function, conditional on the `requires` clause holding at the beginning. The expressions in these clauses are written in the *Verus specification language*. This language deliberately resembles Rust, and it is checked partly via Rust's own type checker, but it is a pure, functional language. Being non-stateful, it is not subject to Rust's ownership or lifetime checking.

Verus generally uses the Z3 theorem prover [5] to dispatch proof obligations, although a Verus user can opt-in to specialized solvers for particular circumstances, such as bit-vector solving or integer ring theories. All verification is modular at the function level; Verus treats every function call as opaque and only reasons about it via its declared `requires` and `ensures`.

To generate efficient verification conditions, Verus leans heavily on the AXM property of Rust's type system. For example, Verus can easily confirm the following assertion:

```
fn example_references(x: &u64, y: &mut u64) {
    let a = *x;
    *y = 20;                // Type system ensures y distinct from x
    some_other_function();  // Type system ensures this cannot modify x
    let b = *x;
    assert(a == b);
}
```

The `assert` statement, like `requires` and `ensures`, is written in the Verus specification language. Verus proves this one easily, knowing that `x`, being a shared reference, cannot possibly be mutated, no matter what `some_other_function` is. This reasoning depends crucially on the AXM principle.

Ghost permissions. Unfortunately, AXM is not the end of the story. In more advanced Rust programming patterns, we often need to violate AXM; that is, we need to support mutation of aliased state. In Rust, there are two main *unsafe features* we can use to do this:

(1) A **raw pointer** of type `*mut T` is a pointer to a shared object in memory. The pointer itself does not own the `T` that it points to, and can thus be freely copied, but it can also be used to (unsafely!) mutate the `T`. Raw pointers are useful (among other things) for implementing "pointer-based data structures with internal sharing" [35] like doubly-linked lists, in which any given node may be mutable but still have multiple nodes pointing to it.
(2) A **cell** of type `UnsafeCell<T>` is essentially a wrapper around `T` allowing it to be (unsafely!) mutated through a shared reference of type `&UnsafeCell<T>`. Unlike a pointer, a cell of type `UnsafeCell<T>` actually *owns* its contents `T`; thus, moving/destructing it also moves/destructs its contents. `UnsafeCell<T>` is the core unsafe building block from which a number of safe interior-mutable types (like `Mutex`, `RwLock`, `Cell`, and `RefCell`) are built.

Verus aims to support programming with both these mechanisms, albeit in a way that can be verified to be *safe*. It achieves this via two non-standard *proof-oriented* types:

(1) `PPtr<T>`, which has the same layout as `*mut T`,[1] (specifically, a pointer to a malloc-backed heap allocation).
(2) `PCell<T>`, which has the same layout as `UnsafeCell<T>`.

[1]The original `PPtr` described by Lattuada et al. [22] operates on discrete heap allocations of statically-determined size. The latest version of Verus has a more elaborate pointer library with a more fungible treatment of memory, supporting pointer arithmetic and pointer-to-int-to-pointer roundtrip casts. In VerusBelt, we support a pointer type with fixed-sized allocations similar to the original Verus `PPtr`, with no int-to-pointer casting. A fuller treatment of pointers here would be best served with a rich treatment of Rust's dynamic semantics concerning pointers (*e.g.*, using Stacked Borrows [17] or Tree Borrows [34]). Such a treatment is beyond the scope of this paper.

```rust
fn ptr_example() {
 // ptr: PPtr<u64>, points_to: PointsTo<u64>
 let (ptr, Tracked(points_to)) =
    PPtr::new(5); // alloc function

 assert(points_to.addr() == ptr.addr());
 assert(points_to.value() == 5);

 let ptr1 = ptr;
 let ptr2 = ptr;

 // *ptr1 = 7;
 ptr1.write(Tracked(&mut points_to), 7);
 assert(points_to.value() == 7);

 // *ptr2 = 9;
 ptr2.write(Tracked(&mut points_to), 9);
 assert(points_to.value() == 9);

 let ptr3 = ptr;


 // let x = *ptr3;
 let x: &u64 = ptr3.borrow(Tracked(&points_to));
 assert(*x == 9);
}
```

```rust
fn pcell_example() {
 // pcell: PCell<u64>, points_to: PointsTo<u64>
 let (pcell, Tracked(points_to)) = PCell::new(5);


 assert(points_to.id() == pcell.id());
 assert(points_to.value() == 5);

 let ref1 = &pcell;
 let ref2 = &pcell;

 // *ref1 = 7;
 ref1.write(Tracked(&mut points_to), 7);
 assert(points_to.value() == 7);

 // *ref2 = 9;
 ref2.write(Tracked(&mut points_to), 9);
 assert(points_to.value() == 9);

 let pcell_b = pcell;
 let ref3 = &pcell_b;

 // let x = *ref3;
 let x: &u64 = ref3.borrow(Tracked(&points_to));
 assert(*x == 9);
}
```

Fig. 1. **Verus examples with `PPtr`, `PCell`, and `PointsTo` primitives.** *The* `Tracked` *keyword indicates a variable use or variable binding of a purely ghost type (here of type* `PointsTo`*). We use* `assert` *statements to illustrate what facts (written in the Verus spec langauge) Verus can deduce at each point. Notice the payload behind the pointer or in the cell is reasoned via the* `PointsTo`*. When performing a write, the* `PointsTo` *is correspondingly mutated; where performing a read, Verus determines the return value (*x*) as a function of the* `PointsTo` *being read (*`points_to.value()`*).*

Perhaps surprisingly, and despite the fact that pointers and cells serve different purposes, Verus reasons about `PPtr<T>` and `PCell<T>` almost identically. This can be seen at a glance in Figure 1. To elaborate on this point, we will cover the `PPtr` snippet first, and then compare.

The `PPtr` *primitive.* The `PPtr` example shows a clear violation of AXM: we have multiple aliases (`ptr1`, `ptr2`) but also mutation. To deal with the AXM violation, Verus requires all accesses to be mediated by a *ghost object* (with *ghost type* `PointsTo`), which relates the pointer `ptr` (in specs, `points_to.addr()`) to the value located at that memory address (in specs, `points_to.value()`).

By "ghost object," we mean that an object is erased from the code before compilation, and which thus is only used for type-checking (including ownership and lifetime checking) and verification. Most proof-oriented types we consider in this paper are ghost types. The only exceptions are `PPtr` and `PCell`, which interact closely with ghost types but which are physical types themselves. Verus also refers to ghost objects as "tracked ghost objects," to emphasize that they are subject to ownership and lifetime tracking.[2] We use purple for ghost types like `PointsTo` and green for runtime types likes `PPtr`.

Here, the `PointsTo` ghost object plays two essential roles. First, it lets Verus reason about the value stored at that memory address. This is usually needed to prove rich correctness properties on `PPtr`-based code. For example, Lattuada et al. [22] show how to use `PPtr` to verify a doubly-linked list, which requires nontrivial invariants over a collection of `PointsTo` objects. Second, the use of the ghost object ensures safe access to the memory. For this to work, Verus requires that the `addr()` of the `PointsTo` match the pointer being accessed. In the snippet, the accesses are permitted because Verus automatically reasons `ptr1` and `ptr2` are equal to `ptr`. In more complex examples, this may require nontrivial proof. Verus also requires the `PointsTo` have the appropriate mutability

[2]The first Verus paper [22] referred to these as "proof mode" objects.

mode: `&mut PointsTo` for writing, and `&PointsTo` for reading. This system rules out memory safety violations like use-after-free as well as data races.

Observe that the existence of ghost state allows us to "shift our perspective" in a useful way. The `ptr_example` function would ordinarily be considered AXM-violating (mutating the same data through two aliased pointers), but since in Verus we view the data through the ghost object, and the ghost object is held mutably, the AXM principle is restored in this shifted perspective. This is (intuitively) why Verus can apply its usual reasoning principles even when `PPtr` is involved.

The `PCell` primitive. So, what about `PCell`? As illustrated in Figure 1, Verus treats a `PCell` nearly identically to `PPtr`, using a similar `PointsTo` type. (When necessary, we use `PPtr::PointsTo` and `PCell::PointsTo` to disambiguate the two similarly named `PointsTo` types.) There are still a few key differences to point out. First of all, the actual data layout is different. The `u64` payload is stored in the `PCell`, so it never leaves the stack. Verus doesn't care about the data layout, though; it cares about the specifications, and in specifications, a `PCell` is treated similarly to a pointer.

There *is* still an important difference in the specifications, however. Observe that Verus cannot use the physical memory address as the "key" to relate a `PCell` to its `PCell::PointsTo`. This is because the cell might *move* (as it does on Line 20), but this should still work as expected, with the same `PointsTo` object for the same `PCell` at the new address. Therefore, Verus instead uses an abstract *cell ID* (`pcell.id()`) for this purpose. The cell ID is a strictly logical notion, used in specifications but not physically represented in executable code.

Obtaining ownership via shared references. Now, when verifying a pointer- or cell-based data structure, the name of the game is obtaining access to the `PointsTo` ghost objects at the appropriate times and with the appropriate mutability level. This is often nontrivial—if it were trivial, the user could probably have just used a `Box` or other common safe type in the first place.

Towards this end, Verus provides a variety of mechanisms for manipulating ghost state. One such mechanism is the proof-oriented type called `LocalInvariant<T>`, a kind of ghost "container" that holds a ghost `T` satisfying a user-declared *invariant predicate*. The user can "open" this container even when it is behind a shared reference (`&LocalInvariant<T>`) to obtain ownership of the `T`.

Take, for example, `&(PCell<u64>, LocalInvariant<PointsTo<u64>>)`. With this type, a user could open the invariant, obtain full ownership of the `PointsTo`, write to the `PCell`, and then close the invariant. To satisfy Verus's checks, they would use the invariant predicate to constrain the cell ID of the `PointsTo` to agree with the `PCell`. In this way, they can mutate memory despite starting with only a shared reference (`&`) type—a useful pattern for implementing many `Cell`-like types.

As with the `ptr_example` above, this may seem at first glance to be another clear violation of AXM, but thanks to Verus's shift of perspective, it isn't. The essential point is that Verus cannot "see" inside the container when it is closed, so there is no violation of AXM from Verus's perspective—no issue with the fact that the contents may change even when we are holding onto the container via a shared reference. Rather, we can only learn the specific `T` value when we *open* the invariant, and at that point, we only know that it satisfies the invariant predicate.

A number of subtle checks are needed to ensure this whole system is sound. We will discuss these more in §4, along with `LocalInvariant`'s cousin, `AtomicInvariant`, which is used for implementing concurrent data structures.

Switching between mutable and read-only access. It is also common to have types that obey AXM—with resources switching between unique mutable and shared read-only states—but only for highly nontrivial reasons. This is most obvious in the reader-writer lock type (`RwLock`), though it also holds for types like `RefCell`, `Arc`, and `Rc`.

To be specific, let us look at the (slightly simplified) type signatures in Rust's `RwLock` interface:

```
impl<V> RwLock<V> {
    fn new(v: V) -> RwLock<V>;

    fn read(&self) -> ReadGuard<V>;
    fn borrow<'a>(guard: &'a ReadGuard<V>) -> &'a V;
    fn release_read(&self, guard: ReadGuard<V>);

    fn write(&self) -> WriteGuard<V>;
    fn borrow_mut<'a>(guard: &'a mut WriteGuard<V>) -> &'a mut V;
    fn release_write(&self, guard: WriteGuard<V>);
}
```

When the reader requests a read-lock (`read`) they obtain a special `ReadGuard` object which can be used to obtain a shared reference to the protected `V` data (`borrow`). The write-lock works similarly, but with a mutable reference.

What stops the user from unsafely using the references after the locks have been released? Essential to this question is the use of Rust's *lifetime variables*, such as the `'a` in `borrow`. This lifetime variable is used to bound the lifetime of the output reference in terms of the input reference; this ensures that the borrowed data cannot be accessed after the `ReadGuard` is destroyed.

To implement and verify such a library seems highly nontrivial, since its correctness apparently lies on reasoning about all the ways a well-typed client might make use of these `ReadGuard` and `WriteGuard` objects. *Intuitively*, the `RwLock` interface is sound because the lock library obeys the AXM principles (*i.e.,* it ensures that read locks and write locks are mutually exclusive), but the reason it obeys AXM principles relies on the lock being implemented correctly (*i.e.,* through some atomic synchronization algorithm). However, the latter is not handled by typical use of Rust's type system. Thus, we need a formal reasoning principle that takes us from this intuition of the lock being correct because it obeys AXM to the complex type signatures above.

In Verus, this formal reasoning principle is called a *storage protocol* (SP), a general but complex algebraic structure based on the structure of the same name from the Leaf logic [13] (also the same logic we will be using to define VerusBelt).

In short, the way an SP works is this: the user defines a protocol and proves that it obeys AXM principles; this allows Verus to safely embed the protocol into Rust's type system via additional ghost state. A central element to this is the ability to define special ghost "reader objects" equipped with functions for accessing shared references. For example, to verify a `RwLock`, the SP might construct a reader with this signature:

```
fn borrow_ghost<'a, T>(resource: &'a Reader<T>) -> &'a PointsTo<T>
    requires ...  ensures ...
```

Such a function with this lifetime bound is called a *guard function*. By combining guard functions with other Verus building blocks, we can verify functions like `ReadGuard`'s `borrow`. For example:

```
struct ReadGuard<'a, V> { resource: Tracked<Reader<V>>, pcell: &'a PCell<V> }
fn borrow<'a>(guard: &'a ReadGuard<'a, V>) -> &'a V
    requires ... ensures ...
{
    guard.pcell.borrow(Tracked(borrow_ghost(&guard.resource)))
}                          // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ &PointsTo<V>
```

This function (like all functions in Verus) is then proved correct by a combination of theorem-proving, and (crucially here) Rust's lifetime checking. In this way, the storage protocol gives us the `(resource: &'a Reader<T>) -> &'a PointsTo<T>` as an elementary building block with which to build higher-level functions.

Lattuada et al. [21] showed this method to be effective in larger Verus developments as well, allowing the developer to isolate AXM-style reasoning and thereby leverage shared ownership in their proofs. In one instance, following the earlier IronSync [14], they use an SP in their "Node Replication" system to manage a message buffer, where each entry is written by a single writer and then read by multiple readers. In another instance, they use SPs to manage the shared, inter-thread components of a concurrent memory allocator.

3 Foundations for Verus

What makes a suitable formalism for verifying the soundness of Verus's proof-oriented types? We posit that a suitable starting point lies in the *type-spec* system of RustHornBelt [28]. We will start with an overview of how RustHornBelt's specification system works, and then discuss what it means to adapt its methodology to Verus.

RustHornBelt and type-spec judgments. RustHornBelt considers the semantics of programs written in a Rust-inspired λ-calculus called λ_{Rust}, originally defined by RustBelt [18]. λ_{Rust} has concurrent, thread-interleaving semantics, uses continuation-passing style for control flow, and has heap semantics that support sequentially-consistent atomic accesses and non-atomic accesses.

Now, in RustHornBelt, all λ_{Rust} code (from individual instructions to entire functions) is typed together with its *specification*. For example, a function taking a single argument might be typed as:

$$f : \mathit{fn}(\mathsf{T}) \to \mathsf{U} \rightsquigarrow \Phi$$

Here, Φ is a *specification* in the form of a *predicate transformer*:

$$\Phi : (\lfloor \mathsf{U} \rfloor \to \mathit{Prop}) \to (\lfloor \mathsf{T} \rfloor \to \mathit{Prop})$$

Here, $\lfloor \mathsf{T} \rfloor$ and $\lfloor \mathsf{U} \rfloor$ are the *interpretation sorts*, used for reasoning about types T and U in specifications.

We interpret Φ roughly as follows. For a *postcondition predicate* $\phi : \lfloor \mathsf{U} \rfloor \to \mathit{Prop}$, giving a condition that we want to hold on the return value $u : \mathsf{U}$, we get a *precondition predicate* $\Phi(\phi) : \lfloor \mathsf{T} \rfloor \to \mathit{Prop}$ that must hold on the input value to ensure safe execution of `f` and have ϕ hold at the end.

In most cases, it is straightforward to encode a Verus-style specification into a RustHorn type-spec. This usually takes a form like $\Phi = \lambda\phi.\ \lambda \mathit{args}.\ \mathit{requires} \wedge (\mathit{ensures} \Rightarrow \phi(\mathit{output}))$. For example, take `PCell::borrow` (as used in Figure 1), specialized to `u64`. This function is a Verus primitive, so we can't prove it sound in Verus itself; thus, we are interested in proving it sound in VerusBelt.

```
1 fn borrow(pcell: &PCell<u64>, points_to: &PCell::PointsTo<u64>) -> (res: &u64)
2     requires pcell.id() == points_to.id()
3     ensures *res == points_to.value()
```

In specifications, a `u64` is represented by an integer ($\mathbb{Z}$), a `PCell` is represented by its *CellId*, and a `PCell::PointsTo<u64>` links a *CellId* to the integer stored in the cell. Thus:

$$\lfloor \texttt{u64} \rfloor \triangleq \mathbb{Z} \qquad \lfloor \texttt{PCell<u64>} \rfloor \triangleq \mathit{CellId} \qquad \lfloor \texttt{PCell::PointsTo<u64>} \rfloor \triangleq \mathit{CellId} \times \mathbb{Z}$$

Now, `borrow` takes references for all its arguments. In RustHornBelt, $\lfloor \texttt{\&U} \rfloor = \lfloor \mathsf{U} \rfloor$ for all U; *i.e.,* it considers the actual reference address irrelevant. VerusBelt actually differs here, but this is not important yet, so we stick with $\lfloor \texttt{\&U} \rfloor = \lfloor \mathsf{U} \rfloor$ for our first example. Thus we type this function:

$$\texttt{borrow} : \mathit{fn}(\texttt{\&PCell<u64>}, \texttt{\&PCell::PointsTo<u64>}) \to \texttt{\&u64} \rightsquigarrow \Phi$$
$$\text{where } \Phi : (\mathbb{Z} \to \mathit{Prop}) \to (\mathit{CellId} \to (\mathit{CellId} \times \mathbb{Z}) \to \mathit{Prop})$$
$$\Phi \triangleq \lambda\phi.\ \lambda\gamma, (\gamma', z).\ \gamma = \gamma' \wedge \phi(z)$$

Here, the cell IDs are given by γ and γ', so the precondition is represented by $\gamma = \gamma'$.

Now, the reader might wonder if there is a problem with applying this system towards *unsafe* functions. `PCell::borrow`, for example, is not a safe function, at least not in the usual Rust sense. Rust takes "safe" to mean *unconditionally* safe, *i.e.,* safe for any valid input from a well-typed client program. `PCell::borrow`, however, is only safe when the precondition is satisfied, not for any random `PCell` and `PCell::PointsTo`. Thus, it must be checked by Verus.

The type-spec system works the same way. RustHornBelt's "type-spec soundness theorem" only says that `f` is safe to execute if it can be typed $\texttt{f} : \mathit{fn}() \to () \rightsquigarrow (\lambda\phi.\ \phi)$. This does not mean functions with nontrivial Φ specifications are useless. Such functions can still be used as part of a larger program in order to construct functions that *are* unconditionally safe to call. Using such functions requires one to check their preconditions at their invocations—just as Verus does.

Our objective. With the type-spec framework in hand, a specific objective becomes clear for verifying Verus's type system. Namely, we want to define interpretation sorts $\lfloor\cdot\rfloor$ for Verus's proof-oriented types, translate the specs of major API functions into type-specs, and prove them sound together with the standard Rust typing rules and a type-spec soundness theorem. To state this theorem, we also need to define the (dynamic) semantics of Verus's proof-oriented operations.

In most cases, translating Verus specifications to type-specs is straightforward (and we will talk about the exceptional cases later). Verus uses fairly standard weakest-preconditions, so for any given function, we can (manually) compute the weakest precondition from its Verus signature to determine the predicate transformer Φ; however, we do not attempt to establish a formal correspondence. Furthermore, the semantics of most Verus operations are straightforward to define, as they are already expressible in the "core language" of λ_{Rust}. VerusBelt's language, λ_{Verus}, is thus nearly identical to λ_{Rust}, with one technical addition relating to `AtomicInvariant` (§4.1). The main challenge, then, lies in proving the soundness of the type-specs. So, how *do* we go about proving this? In the next section, we take a look at the VerusBelt proof framework.

3.1 VerusBelt Basics: Semantic Models for Rust Types

VerusBelt begins with the notion of a *semantic model* of a type `T`. A semantic model effectively answers the question, "What does it mean for a value of type `T` to be valid?" or more specifically, "What does it mean for a value of type `T`, *represented by a spec-value* $t : \lfloor\texttt{T}\rfloor$, to be valid?" In VerusBelt, this question is answered for a type `T` through two functions, the "phys" and "own" functions:

$$[\![\texttt{T}]\!].\text{phys} : \lfloor\texttt{T}\rfloor \to \mathit{list}\ \mathit{FancyValue}$$
$$[\![\texttt{T}]\!].\text{own} : \lfloor\texttt{T}\rfloor \times \mathit{ThreadId} \to \mathit{iProp}$$

The first of these, the *physical representation*, $[\![\texttt{T}]\!].\text{phys}$, returns a list of "fancy values." We will discuss what makes these "fancy" later, but for now, the reader can think of this as a vector $\vec{v}$ of word-values that make up the in-memory representation of the type. (This treatment of the physical value $\vec{v}$ is slightly different than in RustHornBelt, which defines valid $\vec{v}$ by a relation.)

The second (and usually more interesting) is the *ownership predicate* $[\![\texttt{T}]\!].\text{own}$, given as a proposition in the *Iris program logic* [20, 19], in which we carry out most of our proofs. The ownership predicate represents the *resources* (*e.g.,* memory) whose ownership is captured in the type.

Let us show the models of a few common types to get the concept. One of the simplest types in Rust/Verus is the humble `bool`:

$$\lfloor\texttt{bool}\rfloor \triangleq \mathbb{B}$$
$$[\![\texttt{bool}]\!].\text{phys}(b) \triangleq [\textbf{if}\ b\ \textbf{then}\ 1\ \textbf{else}\ 0]$$
$$[\![\texttt{bool}]\!].\text{own}(b, \theta) \triangleq \text{True}$$

(Here, $\mathbb{B}$ is the discrete set of boolean values, {true, false}, while True denotes truth in the Iris separation logic.) Thus, this essentially says we represent true by the physical value 1, and false by 0. Since this is a primitive type with no ownership, the ⟦bool⟧.own proposition is trivially true.

What about product types? We can define the semantic model for a pair `(S, T)`:

$$\begin{aligned}
\lfloor \texttt{(S, T)} \rfloor &\triangleq \lfloor \texttt{S} \rfloor \times \lfloor \texttt{T} \rfloor \\
[\![\texttt{(S, T)}]\!].\mathrm{phys}((s,t)) &\triangleq [\![\texttt{S}]\!].\mathrm{phys}(s) +\!\!+ [\![\texttt{T}]\!].\mathrm{phys}(t) \\
[\![\texttt{(S, T)}]\!].\mathrm{own}((s,t),\theta) &\triangleq [\![\texttt{S}]\!].\mathrm{own}(s,\theta) * [\![\texttt{T}]\!].\mathrm{own}(t,\theta)
\end{aligned}$$

Thus, a product type is represented in-memory by the two constituent types laid out consecutively.

Using this method, the semantic proof of any particular judgment always reduces to some proposition in the Iris logic. For example, the type-spec judgment $f : \mathit{fn}(\texttt{T}) \to \texttt{S} \rightsquigarrow \Phi$ reduces to proving a Hoare triple like

$$\begin{aligned}
&\forall t, \theta, \phi.\ \{[\![\texttt{T}]\!].\mathrm{own}(t,\theta) * p \mapsto [\![\texttt{T}]\!].\mathrm{phys}(t) * \Phi(\phi)(t)\}\ q := f(p) \\
&\quad \{\exists s.\ [\![\texttt{S}]\!].\mathrm{own}(s,\theta) * q \mapsto [\![\texttt{S}]\!].\mathrm{phys}(s) * \phi(s)\}
\end{aligned}$$

modulo details regarding lifetime contexts and other contextual boilerplate. For "ghost" operations, *i.e.,* those that are operationally no-ops, this essentially amounts to proving that we can *update* the ghost resources of T to the ghost resources of S, as denoted by the Iris "update" operator:

$$\forall t, \theta.\ [\![\texttt{T}]\!].\mathrm{own}(t,\theta) \Rrightarrow \exists s.\ [\![\texttt{S}]\!].\mathrm{own}(s,\theta)$$

With this in mind, let us now take a look at a type with a more interesting ownership predicate: Box<T>. A Box is represented by a single pointer, ℓ, as well as ownership of T at the address ℓ. Thus, the Box owns resources for the memory at address ℓ, together with all the resources of T:[3]

$$\begin{aligned}
\lfloor \texttt{Box<T>} \rfloor &\triangleq \mathit{Loc} \times \lfloor \texttt{T} \rfloor \\
[\![\texttt{Box<T>}]\!].\mathrm{phys}((\ell,x)) &\triangleq [\ell] \\
[\![\texttt{Box<T>}]\!].\mathrm{own}((\ell,x),\theta) &\triangleq \ell \mapsto [\![\texttt{T}]\!].\mathrm{phys}(x) * \triangleright [\![\texttt{T}]\!].\mathrm{own}(x,\theta)
\end{aligned}$$

With these resources we can read and modify the in-memory T.

3.2 Semantic Model for PPtr

Now we are ready to look at our first proof-oriented types: PPtr and PPtr::PointsTo. It turns out these have a close relationship to Box. Recall that a PPtr is just an address, while PointsTo is a ghost object that gives us ownership at the address. In other words: a PPtr is just like a Box without any of the owned resources; a PointsTo is just like a Box but without a physical representation.

$$\begin{aligned}
\lfloor \texttt{PPtr} \rfloor &\triangleq \mathit{Loc} & \lfloor \texttt{PointsTo<T>} \rfloor &\triangleq \mathit{Loc} \times \lfloor \texttt{T} \rfloor \\
[\![\texttt{PPtr}]\!].\mathrm{phys}(\ell) &\triangleq [\ell] & [\![\texttt{PointsTo<T>}]\!].\mathrm{phys}((\ell,x)) &\triangleq [] \\
[\![\texttt{PPtr}]\!].\mathrm{own}(\ell,\theta) &\triangleq \mathrm{True} & [\![\texttt{PointsTo<T>}]\!].\mathrm{own}((\ell,x),\theta) &\triangleq \ell \mapsto [\![\texttt{T}]\!].\mathrm{phys}(x) * \triangleright [\![\texttt{T}]\!].\mathrm{own}(x,\theta)
\end{aligned}$$

We observe that a Box is really just a PPtr plus a PointsTo. A Box can always be destructed into a pair `(PPtr, PointsTo)`; we can always go the other way as long as the locations match. This immediately gives us a way to read and write through a PPtr and PointsTo.

[3]We elide details related to deallocation of memory. The (▷) symbol is the *later modality*, needed for technical reasons, though from this point on we will mostly elide these as well.

3.3 Borrowed Propositions and Lifetimes

Our next—and arguably most important—topic will be the semantic model of *shared references*, but it is useful first to review how RustBelt and related works model references. RustBelt, by leveraging Iris's capacity for higher-order ghost state, constructs a derived logic called *the lifetime logic*, which contains various *borrow propositions* subject to an algebra of "lifetimes."

In the lifetime logic, a borrow proposition $\&^\alpha P$, for an arbitrary Iris proposition P, is itself an Iris proposition that represents the ability to access P as long as the lifetime α is active. This access may be limited or restricted in some way, depending on the specific variety of borrow.

To allow reasoning about whether a lifetime α is alive, the lifetime logic provides two key propositions: the *lifetime token* $[\alpha]$, a resource that indicates the lifetime α is still active, and the *dead token* $[\dagger\alpha]$, which indicates that α has expired. A key property of a borrow is that we can *regain* ownership of a proposition after the borrow expires. This is represented in the law for instantiating a borrow, which (depending on the borrow kind) typically looks something like this:

$$\triangleright P \Rrightarrow \&^\alpha P * ([\dagger\alpha] \mathrel{-\!\!*\!\!\!\Rrightarrow} \triangleright P)$$

The $\Rrightarrow$ symbol denotes an *update*, *i.e.,* that we can exchange the left hand side for the right hand side, while $\mathrel{-\!\!*\!\!\!\Rrightarrow}$ denotes an *affine update*, *i.e.,* an update which can only be performed once, analogously to the wand operator ($-\!\!*$). Thus, this borrowing law "splits" the proposition in two: one which is usable while the lifetime is active, and one which is usable after it has expired. This is useful because we can regain ownership of P even after "losing track of" ownership of $\&^\alpha P$.

So what kinds of borrows are there, and what can you do with them? RustBelt originally defined a variety of different borrow kinds, notably, *full borrows* ($\&^\alpha_{\text{full}} P$) used to model mutable references, and a handful of others for modeling shared references: *fractured borrows*, *atomic borrows*, and *non-atomic borrows*, which are individually selected on a per-type basis. Unfortunately, as noted in the introduction, this *ad hoc* system makes it difficult to implement models for Verus's most general sharing mechanism: its storage protocols. This brings us to one of our key technical advances: we consolidate all sharing into a single *shared borrow proposition*, $\&^\alpha_{\text{shr}} P$, constructed using the *Leaf logic* [13] for temporarily shared resources. This $\&^\alpha_{\text{shr}} P$ proposition will then form the basis of our semantic model for shared borrows in VerusBelt.

Figure 2 shows the Leaf Lifetime Logic (LeLiLo) that we use in VerusBelt, which marries aspects of Leaf and the RustBelt lifetime logic. The basics of LeLiLo are *mostly* true to the presentation of RustBelt's lifetime logic [18]. We have the ability to allocate a new lifetime token $[\alpha]$ and the ability to "expire" it, replacing it with $[\dagger\alpha]$. The lifetimes form an algebra under "lifetime intersection" ($\sqcap$). We also have a *lifetime inclusion* proposition, $\alpha \sqsubseteq \beta$, read "β outlives α" to express constraints on the ordering of lifetime expiration.

The first majorly novel aspect is the existence of the *shared borrow* proposition $\&^\alpha_{\text{shr}} P$. This proposition forms the basis for our model of shared references in VerusBelt:[4]

$$\begin{aligned}
\lfloor \&\alpha\ \mathsf{T} \rfloor &\triangleq \mathit{Loc} \times \lfloor \mathsf{T} \rfloor \\
[\![\&\alpha\ \mathsf{T}]\!].\mathrm{phys}((\ell, t)) &\triangleq [\ell] \\
[\![\&\alpha\ \mathsf{T}]\!].\mathrm{own}((\ell, t), \theta) &\triangleq \&^\alpha_{\text{shr}} \left(\ell \mapsto [\![\mathsf{T}]\!].\mathrm{phys}(t)\right) * \&^\alpha_{\text{shr}} \left([\![\mathsf{T}]\!].\mathrm{own}(t, \theta)\right) * \cdots
\end{aligned}$$

This notably looks a lot like the semantic model of `Box<T>`, except each of our resources is now behind a $\&^\alpha_{\text{shr}}$. But what exactly can we *do* with a $\&^\alpha_{\text{shr}} P$ proposition, and how do we use it to prove the variety of properties on the shared reference type `&T`?

[4] For technical reasons, we need one additional term in the own predicate, not shown, corresponding to the "persistent" component of $[\![\mathsf{T}]\!].\mathrm{own}(t, \theta)$. Furthermore, we continue to elide details relating to the later modality ($\triangleright$).

The Leaf Lifetime Logic

Propositions: $[\alpha]$ **Persistent Propositions:** $[\dagger\alpha]$ $\alpha \sqsubseteq \beta$ $\&^{\alpha}_{\mathsf{shr}} P$

(where α, β : *Lifetime*, P : *iProp*)

LeLiLo-Begin
$\mathsf{True} \Rrightarrow\!\!\!\!\ast\; \exists \alpha.\ [\alpha] * ([\alpha] \mathrel{-\!\!\!\ast\!\!\!\Rrightarrow} [\dagger\alpha])$

LeLiLo-Not-Own-End
$[\alpha] * [\dagger\alpha] \vdash \mathsf{False}$

LeLiLo-Tok-Inter
$[\alpha \sqcap \beta] \dashv\vdash [\alpha] \wedge [\beta]$

LeLiLo-End-Inter
$[\dagger(\alpha \sqcap \beta)] \dashv\vdash [\dagger\alpha] \vee [\dagger\beta]$

LeLiLo-Incl-Dead
$([\alpha] \sqsubseteq [\beta]) * [\dagger\beta] \Rrightarrow [\dagger\alpha]$

LeLiLo-Incl-Guard
$\alpha \sqsubseteq \beta \dashv\vdash [\alpha] \twoheadrightarrow [\beta]$

LeLiLo-Shr-Borrow
$\triangleright P \Rrightarrow\!\!\!\!\ast\; (\&^{\alpha}_{\mathsf{shr}} P) * ([\dagger\alpha] \Rrightarrow\!\!\!\!\ast\; \triangleright P)$

LeLiLo-Shr-Guard
$\&^{\alpha}_{\mathsf{shr}} P \dashv\vdash [\alpha] \twoheadrightarrow P$

LeLiLo-Shr-Pers
$\mathsf{Persistent}(\&^{\alpha}_{\mathsf{shr}} P)$

Fig. 2. **The Leaf Lifetime Logic (LeLiLo).** *Selected laws from our Leaf Lifetime Logic, including base lifetime laws and shared borrowing laws.*

Leaf Guarding Logic

Persistent Propositions: $P \twoheadrightarrow Q$ (where P, Q : *iProp*)

Guard-Refl
$P \twoheadrightarrow P$

Guard-Trans
$(P \twoheadrightarrow Q) * (Q \twoheadrightarrow R) \vdash (P \twoheadrightarrow R)$

Guard-Sep
$(P * Q) \twoheadrightarrow P$

Heap-Read-Guarded
$(G \twoheadrightarrow \ell \mapsto v) \vdash \{G\}\ x := {}^{*}\ell\ \{(x = v) * G\}$

Fig. 3. **The Leaf Guarding Logic.** *Selected laws from Leaf base logic and an example Leaf-based program logic. We elide details related to invariant masks. For a more complete picture, including SP laws, see Leaf [13].*

At first, it may look like there is not much to $\&^{\alpha}_{\mathsf{shr}} P$. We have LeLiLo-Shr-Borrow, the borrowing rule like the one we described above used to initiate a new borrow. Secondly, we have the somewhat cryptic LeLiLo-Shr-Guard, that $\&^{\alpha}_{\mathsf{shr}} P$ is equivalent to $[\alpha] \twoheadrightarrow P$. In our model, the flexibility of shared borrows comes entirely from this operator.

The $\twoheadrightarrow$ symbol, pronounced *guards*, is the primary operator of the Leaf logic. At a very intuitive level, $A \twoheadrightarrow B$ roughly means that "we can treat the proposition A as a shared, read-only version of B." The interesting properties of shared borrows thus are all derived from properties of the $\twoheadrightarrow$ operator (Figure 3). Let us work through a few examples now.

Example: Reading from memory. From a $\&^{\alpha}_{\mathsf{shr}}(\ell \mapsto v)$ we would like to be able to read from ℓ. This will let us prove that we can copy data from a `&T` (when `T` is `Copy`). We expect this to be possible using the intuition of "$\&^{\alpha}_{\mathsf{shr}}$ gives us read-only access," and indeed, this is the case. Leaf has a general read rule for the $\twoheadrightarrow$ operator:

$$(G \twoheadrightarrow (\ell \mapsto v)) \vdash \{G\}\ x := {}^{*}\ell\ \{(x = v) * G\}$$

As a special case, this shows how shared borrows can be used to read from memory:

$$\&^{\alpha}_{\mathsf{shr}}(\ell \mapsto v) \vdash \{[\alpha]\}\ x := {}^{*}\ell\ \{(x = v) * [\alpha]\}$$

That is: with $\&^{\alpha}_{\mathsf{shr}}(\ell \mapsto v)$, we can read from location ℓ so long as the lifetime α is active.

Example: Splitting borrows. From `&(T, U)`, one should be able to obtain `&T` and `&U`. This follows from *borrow splitting*, *i.e.*, $\&^{\alpha}_{\mathsf{shr}}(P * Q) \Rrightarrow \&^{\alpha}_{\mathsf{shr}} P * \&^{\alpha}_{\mathsf{shr}} Q$. This in turn follows from transitive chaining (Guard-Trans): We chain $[\alpha] \twoheadrightarrow (P * Q)$ and $(P * Q) \twoheadrightarrow P$ (Guard-Sep) to get $\&^{\alpha}_{\mathsf{shr}} P$. Likewise, $\&^{\alpha}_{\mathsf{shr}} Q$ follows similarly. (Since $\&^{\alpha}_{\mathsf{shr}}$ is persistent, we easily get both.)

Example: Lifetime shortening. The Rust type system features a lifetime *inclusion* relationship. If α is shorter than β, then `&`β `T` is a subtype of `&`α `T`. At the Iris level, we express this as $\alpha \sqsubseteq \beta * \&^{\beta}_{\mathsf{shr}} P \Rrightarrow \&^{\alpha}_{\mathsf{shr}} P$. Again, this follows from transitivity, here using the fact that we also describe $\sqsubseteq$ in terms of the guards operator: $\alpha \sqsubseteq \beta$ is just $[\alpha] \twoheadrightarrow [\beta]$ (LeLiLo-Incl-Guard).

Example: Storage protocols. Recall (§2) that Verus's storage protocol laws allow users to prove signatures with complex lifetime bounds, e.g., `(resource: &α Reader<T>) -> &α PointsTo<T>`. This reduces to $\&^{\alpha}_{\mathsf{shr}} P \Rrightarrow \&^{\alpha}_{\mathsf{shr}} Q$ for some ownership predicates P and Q. Again, we can apply Leaf transitivity; in this case it suffices to show that $P \twoheadrightarrow Q$. This guard relation can then be derived from Leaf's flexible SP mechanism (on which Verus's SP mechanism is based). For more details on the SP algebraic structure, we refer to Leaf [13].

3.4 Semantic Model for `PCell`

As explained in §2, Verus treats `PCell` similarly to `PPtr`, and indeed VerusBelt does as well. In both cases, the "physical representation" of the type is identical to its mathematical representation—a physical memory location ℓ in the case of `PPtr`, and a logical cell ID γ in the case of `PCell`. Moreover, in both cases the ownership interpretation of the type is trivial because all ownership of the underlying content is mediated by the corresponding ghost `PointsTo` type.

However, the devil is in the details: there is something very unusual about modeling the "physical representation" of a `PCell` using a logical cell ID γ, namely that γ is not a physical value! Both ⟦`Box<PCell<T>>`⟧.own and ⟦`&PCell<T>`⟧.own involve an $\ell \mapsto$ ⟦`PCell<T>`⟧.phys(γ) proposition, but if ⟦`PCell<T>`⟧.phys(γ) is just γ, then we have an assertion of the form $\ell \mapsto \gamma$ here. What does this even mean? To answer this question, we develop a novel extension of the points-to logic, which as we will see, allows us to say not only "ℓ points to a value v," but also "ℓ points to a *cell*." We introduce the Points-To Cell Logic first, then return afterwards to the semantic model of a `PCell`.

The Points-To Cell Logic. Roughly speaking, the role of the Points-To Cell Logic is to let us "split" a points-to proposition $\ell \mapsto v$ into two pieces via a *cell* (γ): $\ell \mapsto \gamma$ and $\gamma \mapsto v$, whose ownership may then be manipulated separately.

Formally speaking, in our new points-to logic, the $\mapsto$ operator takes on its left-hand side a *fancy location*, which may either be a real *location* or a *cell ID*, and it takes on the right-hand side a *fancy value*, which may either be a real *value* or a *cell ID*. A letter may wear a hat to indicate fanciness. Thus, ℓ is always a concrete location; $\hat{\ell}$ is either a concrete location or a cell; v is always a concrete value; and $\hat{v}$ is either a concrete value or a cell. We use γ to denote a cell ID. The general form of the points-to, then, is $\hat{\ell} \mapsto \hat{v}$, which more specifically may be either $\ell \mapsto v$ or $\ell \mapsto \gamma$ or $\gamma \mapsto v$ or $\gamma_1 \mapsto \gamma_2$. (Actually, the fully general form allows ℓ to point to a block of n fancy values on the r.h.s. of the $\mapsto$, but to simplify the presentation we focus here on the case where there is just one.)

The possibility of a cell pointing to a cell ($\gamma_1 \mapsto \gamma_2$) comes up when dealing with nested cells, *e.g.*, `PCell<PCell<T>>`. (This is uncommon, but it may come up when nesting cell-based abstractions.) For a sequence $\vec{\gamma} = \gamma_1, \ldots, \gamma_k$, we write $\hat{\ell} \vdash\!\!\lfloor \vec{\gamma} \rfloor\!\!\rightarrow \hat{v}$ to denote:

$$(\hat{\ell} \mapsto \gamma_1) * (\gamma_1 \mapsto \gamma_2) * \cdots * (\gamma_{k-1} \mapsto \gamma_k) * (\gamma_k \mapsto \hat{v})$$

The laws for the Cell Logic are given in Figure 4. It must be kept in mind that a "cell," here, is a strictly logical notion, not an element of the program state. For example, Cell-Fresh, which instantiates a new cell, is purely an update on ghost state, not tied to any particular program step.

Let us next observe how the usual Hoare laws for manipulating the heap are impacted by the presence of cells. We already saw that, with Leaf, we are able to extend the heap-read rule to allow reading when the points-to is behind a guard. With cells in play, we can see something similar *partially* happen for the heap-write rule. In particular, to write to a location ℓ with a chain of cells $\ell \mapsto \gamma_1 \mapsto \cdots \mapsto \gamma_k \mapsto v$, we only need ownership over the *last* link in this chain (Heap-Write2). For the remaining links, read-only access suffices.

By combining Heap-Write2 with the lifetime logic, we can quickly derive:

$$\vdash \left\{[\alpha] * \&^{\alpha}_{\mathsf{shr}}(\ell \vdash\!\!\lfloor \vec{\delta} \rfloor\!\!\rightarrow \gamma) * \gamma \mapsto v\right\} {}^{*}\ell := v' \left\{[\alpha] * \&^{\alpha}_{\mathsf{shr}}(\ell \vdash\!\!\lfloor \vec{\delta} \rfloor\!\!\rightarrow \gamma) * \gamma \mapsto v'\right\}$$

Points-To Cell Logic
Propositions: $\hat{\ell} \vdash\!\lfloor \vec{\delta} \rfloor\!\mapsto \hat{v}$

(where $\ell : \mathit{Loc}$, $v : \mathit{Value}$, $\hat{\ell} : \mathit{Loc} + \mathit{CellId}$, $\hat{v} : \mathit{Value} + \mathit{CellId}$, $\gamma : \mathit{CellId}$, $\vec{\delta} : \mathit{list}\ \mathit{CellId}$)

Cell-Sep
$\left(\hat{\ell} \vdash\!\lfloor \vec{\delta}_1 \rfloor\!\mapsto \gamma\right) * \left(\gamma \vdash\!\lfloor \vec{\delta}_2 \rfloor\!\mapsto \hat{v}\right) \dashv\vdash \left(\hat{\ell} \vdash\!\lfloor \vec{\delta}_1 \gamma \vec{\delta}_2 \rfloor\!\mapsto \hat{v}\right)$

Cell-And
$\left(\hat{\ell} \vdash\!\lfloor \vec{\delta}_1 \rfloor\!\mapsto \gamma\right) \wedge \left(\gamma \vdash\!\lfloor \vec{\delta}_2 \rfloor\!\mapsto \hat{v}\right) \dashv\vdash \left(\hat{\ell} \vdash\!\lfloor \vec{\delta}_1 \gamma \vec{\delta}_2 \rfloor\!\mapsto \hat{v}\right)$

Cell-Fresh
$\hat{\ell} \mapsto \hat{v} \Rrightarrow \exists \gamma.\ \hat{\ell} \vdash\!\lfloor \gamma \rfloor\!\mapsto \hat{v}$

Heap-Read
$(G \twoheadrightarrow (\ell \vdash\!\lfloor \vec{\delta} \rfloor\!\mapsto v)) \vdash \{G\}\ x := {}^*\ell\ \{(x = v) * G\}$

Heap-Write1
$\{\ell \mapsto v\}\ {}^*\ell := v'\ \{\ell \mapsto v'\}$

Heap-Write2
$(G \twoheadrightarrow (\ell \vdash\!\lfloor \vec{\delta} \rfloor\!\mapsto \gamma)) \vdash \{G * (\gamma \mapsto v)\}\ {}^*\ell := v'\ \{G * (\gamma \mapsto v')\}$

Heap-Move-Cell
$\{\ell \mapsto \gamma * \ell' \mapsto v'\}\ {}^*\ell' := {}^*\ell\ \{\exists v.\ \ell \mapsto v * \ell' \mapsto \gamma\}$

Fig. 4. **Points-To Cell Logic.** *Selected laws for the Points-To Cell Logic.*

From this, we can already start to see how the cell logic enables us to write to a cell when it is behind a shared reference.

Finally, Heap-Move-Cell says that we can *move* a cell from one location ℓ to another ℓ' by copying from ℓ to ℓ'. Notably, this rule does not even require ownership of the second half ($\gamma \mapsto v$); this helps us show that we can move a `PCell` without ownership of its `PCell::PointsTo`.

The model of `PCell` *and Cell* `PCell::PointsTo`. Now, with the Points-To Cell logic in hand, we can present the model for `PCell` (to simplify the presentation, we assume here `T` has size 1):

$$\lfloor \texttt{PCell<T>} \rfloor \triangleq \mathit{CellId}$$
$$[\![\texttt{PCell<T>}]\!].\mathrm{phys}(\gamma) \triangleq [\gamma]$$
$$[\![\texttt{PCell<T>}]\!].\mathrm{own}(\gamma, \theta) \triangleq \mathsf{True}$$

Here, we leverage the fact that the output of $[\![\cdot]\!]$.phys is a list of *fancy values*, and may thus contain cell IDs. Thus, the $\ell \mapsto \gamma$ token will appear in the ownership predicate of the pointer that *points to* the `PCell`, *e.g.,* $[\![\texttt{Box<PCell<T>>}]\!]$.own will contain $\ell \mapsto \gamma$, while $[\![\texttt{\&}\alpha\ \texttt{PCell<T>}]\!]$.own will contain $\&^{\alpha}_{\mathsf{shr}}(\ell \mapsto \gamma)$.

The "other half," $\gamma \mapsto v$, will of course be owned by the `PCell::PointsTo`:

$$\lfloor \texttt{PCell::PointsTo<T>} \rfloor \triangleq \mathit{CellId} \times \lfloor \mathsf{T} \rfloor$$
$$[\![\texttt{PCell::PointsTo<T>}]\!].\mathrm{phys}((\gamma, x_0)) \triangleq []$$
$$[\![\texttt{PCell::PointsTo<T>}]\!].\mathrm{own}((\gamma, x_0), \theta) \triangleq \gamma \mapsto [\![\mathsf{T}]\!].\mathrm{phys}(x_0) * [\![\mathsf{T}]\!].\mathrm{own}(x_0, \theta)$$

One immediately observes a parallel to the `PPtr`/`PPtr::PointsTo` models defined above; the key is to allow the cell IDs $\vec{\gamma}$ to play the role of a pointer.

The high-level point is this: when we have a *shared reference* `&PCell<T>` and full ownership of a `PCell::PointsTo<T>`, we have, through, the ownership predicates:

$$\left(\&^{\alpha}_{\mathsf{shr}}(\ell \mapsto \gamma)\right) * \gamma \mapsto v$$

which (despite the partial sharedness) is enough to write to the cell. But when we have `&PCell<T>` and `&PCell::PointsTo<T>`, we only have:

$$\left(\&^{\alpha}_{\mathsf{shr}}(\ell \mapsto \gamma)\right) * \left(\&^{\alpha}_{\mathsf{shr}}(\gamma \mapsto v)\right)$$

which is only enough to *read* from the cell. In other words, a shared `&PCell` is always sufficient; it is the sharedness of the `PointsTo` that matters.

Verifying cell borrow. To elaborate on this, we will step through the VerusBelt verification of the `PCell::borrow` function in detail. Before we can proceed, though, we need to make a slight correction to our prior definition of shared borrows. The previous definition was built on the presumption that any data can be accessed via $\ell \mapsto v$, but now we know that we may need $\ell \vdash[\vec{\delta}]\mapsto v$ for some chain of cells $\vec{\delta}$. Our new definition accounts for this:[5]

$$\lfloor \&\alpha\ \mathsf{T} \rfloor \triangleq \mathit{Loc} \times \mathit{list}\ \mathit{CellId} \times \lfloor \mathsf{T} \rfloor$$

$$[\![\&\alpha\ \mathsf{T}]\!].\mathrm{phys}((\ell, \vec{\delta}, t)) \triangleq [\ell]$$

$$[\![\&\alpha\ \mathsf{T}]\!].\mathrm{own}((\ell, \vec{\delta}, t), \theta) \triangleq \&^{\alpha}_{\mathsf{shr}} \left(\ell \vdash[\vec{\delta}]\mapsto [\![\mathsf{T}]\!].\mathrm{phys}(t)\right) * \&^{\alpha}_{\mathsf{shr}} \left([\![\mathsf{T}]\!].\mathrm{own}(t, \theta)\right)$$

Likewise, let's flesh out our `PCell::borrow` type-spec from the beginning of §3 to account for the full definition of $\lfloor \&\alpha\ \mathsf{T} \rfloor$.

$$\texttt{borrow} : \mathit{fn}(\texttt{\&PCell<T>}, \texttt{\&PointsTo<T>}) \longrightarrow \texttt{\&T} \rightsquigarrow$$
$$\lambda\phi.\ \lambda(\ell, \vec{\delta}, \gamma), (\_, \_, (\gamma', t)).\ \gamma = \gamma' \wedge \phi(\ell, \vec{\delta}\gamma, t)$$

For a reference to ghost state, the location is insignificant, so we blank it out.

Now, let us proceed with the proof. The primary formal obligation is to prove in Iris that:

$$[\![\&\alpha\ \texttt{PCell<T>}]\!].\mathrm{own}((\ell, \vec{\delta}, \gamma), \theta) * [\![\&\alpha\ \texttt{PCell::PointsTo<T>}]\!].\mathrm{own}((\_, \_, (\gamma, t)), \theta)$$
$$\Rrightarrow [\![\&\alpha\ \mathsf{T}]\!].\mathrm{own}((\ell, \vec{\delta}\gamma, t), \theta)$$

Now, when we expand these definitions, we find that we need to show:

$$\left(\&^{\alpha}_{\mathsf{shr}}(\ell \vdash[\vec{\delta}]\mapsto \gamma) * \&^{\alpha}_{\mathsf{shr}}\mathsf{True}\right) * \left(\&^{\alpha}_{\mathsf{shr}}(\_ \vdash[\_]\mapsto [\,]) * \&^{\alpha}_{\mathsf{shr}}(\gamma \mapsto [\![\mathsf{T}]\!].\mathrm{phys}(t) * [\![\mathsf{T}]\!].\mathrm{own}(t, \theta))\right)$$
$$\Rrightarrow \&^{\alpha}_{\mathsf{shr}}(\ell \vdash[\vec{\delta}\gamma]\mapsto [\![\mathsf{T}]\!].\mathrm{phys}(t)) * \&^{\alpha}_{\mathsf{shr}}[\![\mathsf{T}]\!].\mathrm{own}(t, \theta)$$

This reduces to showing that we can combine the shared borrows for both "halves" of the points-to:

$$\&^{\alpha}_{\mathsf{shr}}(\ell \vdash[\vec{\delta}]\mapsto \gamma) * \&^{\alpha}_{\mathsf{shr}}(\gamma \mapsto [\![\mathsf{T}]\!].\mathrm{phys}(t)) \Rrightarrow \&^{\alpha}_{\mathsf{shr}}(\ell \vdash[\vec{\delta}\gamma]\mapsto [\![\mathsf{T}]\!].\mathrm{phys}(t))$$

How do we actually combine two shared references? In general, shared references do not admit a rule like $\&^{\alpha}_{\mathsf{shr}}P * \&^{\alpha}_{\mathsf{shr}}Q \Rrightarrow \&^{\alpha}_{\mathsf{shr}}(P * Q)$. Since $\&^{\alpha}_{\mathsf{shr}}P$ is persistent, and therefore $\&^{\alpha}_{\mathsf{shr}}P \vdash \&^{\alpha}_{\mathsf{shr}}P * \&^{\alpha}_{\mathsf{shr}}P$, such a rule would let us derive $\&^{\alpha}_{\mathsf{shr}}P \Rrightarrow \&^{\alpha}_{\mathsf{shr}}(P * P)$, which is clearly nonsense.

Leaf's solution is to instead join state behind $\twoheadrightarrow$ operators using the ordinary (*i.e.,* nonseparating) conjunction ($\wedge$). This allows us to show (subject to some technical conditions on P and Q) that $\&^{\alpha}_{\mathsf{shr}}P * \&^{\alpha}_{\mathsf{shr}}Q \Rrightarrow \&^{\alpha}_{\mathsf{shr}}(P \wedge Q)$. In particular, we can apply Cell-And, which gives us that $(\ell \vdash[\vec{\delta}]\mapsto \gamma) \wedge (\gamma \mapsto [\![\mathsf{T}]\!].\mathrm{phys}(t)) \dashv\vdash (\ell \vdash[\vec{\delta}\gamma]\mapsto [\![\mathsf{T}]\!].\mathrm{phys}(t))$, completing the proof.

4 Soundness of Invariant Types

In this section, we discuss the soundness of Verus's `LocalInvariant` and `AtomicInvariant` types. It turns out that for these types, most of the challenges lie in specifying their desired behavior correctly. The semantic models (which we will not go into) are, if not trivial, at least less interesting compared to those of other types.

Recall that a `LocalInvariant<T>` is like a container that we can open to obtain full ownership of `T`, even when the container is behind a shared reference. The difficulty is that Verus must make sure the user does not open the same container twice at the same time. This is nontrivial since we can have multiple aliases to the same container. Thus, to prevent double-access, Verus maintains a

[5]The reader might observe that we seem to be "leaking" internal model details into the interpretation sort, $\lfloor \&\alpha\ \mathsf{T} \rfloor$. Indeed, this is so; however, it is technically convenient, and it is ultimately not a big deal because it only leads to strictly more precise specifications.

(ghost) global variable called the *invariant mask* which tracks the names of all invariants that are presently open.

We need an equivalent concept in the VerusBelt type-spec system. Therefore, we add the mask—a set of names—as an additional parameter in the VerusBelt predicate transformer:

$$\Phi : (\lfloor \mathsf{U} \rfloor \to \mathcal{P}(\mathit{Name}) \to \mathit{Prop}) \to (\lfloor \mathsf{T} \rfloor \to \mathcal{P}(\mathit{Name}) \to \mathit{Prop})$$

This now allows us to write specifications on operations that interact with the mask.

The second challenge is that there are multiple ways to destroy a container and reclaim the contents `T` for another purpose. The most obvious way this can happen is with the primitive destructor function (`LocalInvariant::into_inner`). A far more subtle way this can happen is when ownership of a `LocalInvariant` is passed between threads, which implicitly moves it from one thread's mask system to another's.

We need to rule out the possibility of such actions occurring while the invariant is open, but it is difficult to use the mask system (at least for the second issue). Instead, Verus prevents these bad cases through *lifetimes*: whenever we open a $\&\alpha$ `LocalInvariant`, Verus makes sure the lifetime α outlives the duration for which the invariant is open. However, *this* check turns out to be difficult to get right; in fact, we discovered over the course of VerusBelt that Verus got this check wrong for several years. The problem is that Verus originally attempted to ensure soundness by applying additional lifetime constraints at invariant-close time. The issue is that, in the event of nonterminating code, the invariant might *never be closed*, in which case such constraints would have no effect. In VerusBelt, we show how to specify these constraints properly by adding an *invariant lifetime context* to the program typing judgments, with which we can describe the complete set of constraints that must be enforced.

4.1 Handling Atomics

`LocalInvariant` is not thread-safe, but it has a thread-safe cousin, `AtomicInvariant`, which can be shared across multiple threads at the same time (*i.e.,* it implements `Sync`), which can be used for verifying concurrent, lock-free data structures built on atomic operations (*e.g.,* compare-and-swap). In order to be thread-safe, `AtomicInvariant` comes with an additional restriction: when an `AtomicInvariant` is opened, the user can perform at most one atomic operation before they have to close it. However, this region can still include an arbitrary amount of ghost code.

To prove the safety of `AtomicInvariant`, we need to treat this entire region as if it were genuinely atomic. The problem is that ghost operations are still technically "steps" in the operational semantics, even if they happen to be no-ops. Thus, they can be interleaved with other threads. To resolve this, we employ a 'trick' in the operational semantics of our model language, λ_{Verus}. Specifically, we add a global "atomic lock," which must be taken in order to open any `AtomicInvariant`. Of course, no such lock actually exists in the Rust/Verus runtime. Therefore, we need an additional argument: namely, that if any atomic region contains only a single non-ghost atomic operation, then execution without the global lock is equivalent to execution with the global lock. However, we leave this argument as an informal one, as λ_{Verus} does not have any formal syntactic distinction between ghost and non-ghost code.

Finally, we say a bit about the actual concurrency semantics of λ_{Verus}. Following λ_{Rust}, λ_{Verus} supports atomics with "sequentially consistent" ordering (`SeqCst`), together with non-atomic ordering, in which any data race is considered undefined behavior. To handle non-atomic memory in our proofs, it is necessary to prove the heap-based Hoare triples (*e.g.,* Heap-Read and others in Figure 4) sound even for non-atomic reads and writes. This is somewhat complex as it involves the intersection of Leaf, the cell logic, and the non-atomic semantics. Once these rules are proven, however, the details of the memory model are abstracted away for the bulk of the typing proofs.

5 Mutable References

Verus, at present, has only limited support for mutable references, allowing them only in specific contexts. In particular, Verus can handle mutable references only as *arguments to functions*; this case is "easy" because when a function takes ownership of `&mut T` only as input, it is similar to taking ownership of `T` as input and returning ownership of `T` in the output, which makes the translation straightforward. However, in many other situations, it is difficult to formulate the "return ownership"; as a result, Verus does not allow (i) functions that return mutable references, (ii) putting mutable references in containers, (iii) instantiating generic parameters with mutable references, or (iv) pattern matching on mutable references. Even with these restrictions, a developer can get pretty far, but it often involves clunky workarounds. For example, to mutate the element of a container, the user has to move the element out of the container, mutate it, and then move it back.

It is thus unsurprising that there are frequent requests to extend Verus with improved support for mutable references, but it is highly nontrivial to devise a suitable encoding. The difficulty is that the final value of a mutable reference must be "transmitted" back to the location it was originally borrowed from, but these might be spatially separated at the time the borrow expires.

An elegant solution was proposed in the RustHorn encoding [29], which closes the gap of all four limitations mentioned above, and which was proved sound by RustHornBelt [28]. Their solution posits that the best time to relate a mutable reference to its borrowed-from location is at the one point where they are together: at *borrow time*. Of course, the final value of the mutable reference is not known at this point, and therefore, to reason about this value at borrow time, we need to use a *prophecy variable*. Though the word "prophecy" sounds magical, it really just means we are introducing an unknown variable which will be constrained at a later program step.

Specifically, in the RustHorn method, we have $\lfloor \texttt{\&mut T} \rfloor = \lfloor \texttt{T} \rfloor \times \lfloor \texttt{T} \rfloor$, where the first $\lfloor \texttt{T} \rfloor$ gives us the current value, and the second $\lfloor \texttt{T} \rfloor$ gives us the prophesied value of the borrow upon its expiration. Here, we represent these by `current` and `future`, respectively. This method gives us a clean way to write specifications on functions that manipulate mutable references; for example:

```
fn get_first_element_mut<'a, T, U>(pair: &'a mut (T, U)) -> (fst: &'a mut T)
  ensures current(fst) == current(pair).0, current(pair).1 == future(pair).1,
    // The eventual value of `fst` will be the eventual value of `pair.0`, even
    // though the specific value is not known until some time after the call ends.
    future(fst) == future(pair).0,
{ &mut pair.0 }
```

The use of the RustHorn encoding together with automated theorem proving is not novel; this has been done by Creusot [7]. What has *not* been shown is that this encoding is sound together with Verus's spectrum of proof-oriented types, and in particular, the *interactions* between extended mutable references and Verus proof-oriented types have not been investigated.

For example, the introduction of extended mutable references could give Verus a more convenient way to manipulate the contents of a cell. Specifically, we propose the following new primitive operation for Verus, a mutable counterpart for `borrow` (§3), which gives the caller a mutable reference to the interior of a `PCell`. This is not possible in current Verus because it involves returning a mutable reference (limitation (i)).

```
fn borrow_mut(pcell: &PCell<T>, points_to: &mut PCell::PointsTo<T>) -> (r: &mut T)
    requires pcell.cell_id() == points_to.cell_id(),
    ensures current(r) == current(points_to).value(),
            future(r) == future(points_to).value(),
            future(points_to).cell_id() == current(points_to).cell_id(),
```

To ensure such interactions are sound, and to support the inclusion of extended mutable reference support in Verus, we aim to incorporate the RustHorn prophecy encoding into VerusBelt, and in particular, we aim to show the soundness of laws such as our proposed `borrow_mut`. Fortunately, RustHornBelt gives us a starting point for incorporating prophecies into the semantic models of Rust types; however, there are additional challenges that arise from the interaction between prophecies and Verus's proof-oriented types. We discuss these challenges first.

5.1 A Paradox with Prophecies

The example here illustrates a paradox that can occur if RustHornBelt-style prophecies are naively integrated with Verus-style ghost state. The problem is present in many proof-oriented types, *e.g.,* `LocalInvariant`, `AtomicInvariant`, and storage protocol types; for the section, we will focus on the simplest type that can illustrate the problem: `Ghost<T>`. `Ghost<T>` is a ghost type which is specified the same as `T` (*i.e.,* $\lfloor$`Ghost<T>`$\rfloor = \lfloor$`T`$\rfloor$), but which owns no resources (*i.e.,* not even those resources normally associated with the type `T`). This proof-oriented type has the unusual feature that it takes a *specification expression* as input, allowing it to carry out ghost computations over the interpretations ($\lfloor\cdot\rfloor$) of available variables. The problem is that (naively) we could use `Ghost` to perform arbitrary computation over a prophesied value. For example:

```
1 fn prophecy_paradox() {
2   let mut var = Ghost(0);
3   let mut_ref = &mut var;
4   let x = Ghost(*future(mut_ref) + 1);
5   *mut_ref = x;
6   assert(var == var + 1);
7 }
```

This is essentially a "time travel paradox," wherein a prophecy variable is resolved to a value that depends on itself. Specifically, `future(mut_ref)` is a prophecy variable ρ; we set x to $\rho + 1$ then resolve ρ to x. Thus, $\rho = \rho + 1$, a contradiction. By contrast, RustHornBelt's specifications do not permit this kind of arbitrary computation over prophecies; rather, RustHornBelt only manipulates prophecies through specific, carefully-crafted typing rules that avoid bad cycles.

Ultimately, we need to enforce that the input to `Ghost(...)` cannot depend on any prophesied values. To this end, we can add an explicit notion of a "prophetic" function to the Verus specification language—*e.g.,* `future` would be prophetic, and `current` would be a non-prophetic (normal) spec function. Then we can add a requirement that the input to `Ghost` must always be nonprophetic.

To prove that `Ghost`, restricted in this way, is sound, we need to reason about "non-prophetic" values in the type-spec system. One approach is to use functions that exclude prophetic components (*e.g.,* project only to the first element of the $\lfloor$`&mut T`$\rfloor = \lfloor$`T`$\rfloor \times \lfloor$`T`$\rfloor$ pair), but these functions are inherently lossy, and having to reason about them would greatly complicate generic code that uses `Ghost<U>` (since `U` could be instantiated with `&mut T`).

Instead, we make a fundamental change to the type-spec system to make nonprophetic values easier to describe. In VerusBelt, *every* interpretation sort $\lfloor\cdot\rfloor$ is considered nonprophetic, but crucially, still has enough information to recover the prophecies in the appropriate context. Specifically, we now define the interpretation of mutable references as $\lfloor$`&mut T`$\rfloor = \lfloor$`T`$\rfloor \times \mathit{ProphVar}$, where $\mathit{ProphVar}$ is the sort of prophecy variable "identifiers." These identifiers alone do not give us access to prophesied values. We can only access the future value if we have access to the "lookup table" mapping variables to their prophesied values. We thus make our second and final addition to the predicate transformer Φ, by adding this "lookup table" ($\mathit{ProphAsn}$) as a parameter:

$$\Phi : (\lfloor \mathtt{U} \rfloor \to \mathcal{P}(\mathit{Name}) \to \mathit{ProphAsn} \to \mathit{Prop}) \to (\lfloor \mathtt{T} \rfloor \to \mathcal{P}(\mathit{Name}) \to \mathit{ProphAsn} \to \mathit{Prop})$$

Is anything even different about this system, if we still have access to the prophecies? Yes! Now that prophetic information is split into a separate argument, we can formulate the `Ghost` constructor so that its input is a deterministic function of all the arguments *other* than *ProphAsn*. Such a function is nonprophetic *by construction*, and this formulation is easily proved sound.

5.2 Semantic Model for Mutable Borrows

Now that we have refined what exactly we intend to prove, we are ready to discuss the semantic model of mutable references. There are several ingredients that we need. First, we need the logic of *full borrows* ($\&^{\alpha}_{\text{full}} P$), an essential component of the lifetime logic developed for RustBelt [18], which formed the basis of its model for mutable references. We also need to adapt it to incorporate the Leaf Lifetime Logic. Second, we need the system of *parametric prophecies* developed by RustHornBelt [28]. (Fortunately, the developments in the previous section do not profoundly impact the fundamentals of the prophecy models.) However, either of these topics in detail would take us too far afield, so for these, we will refer to their respective papers.

Finally, we need to carefully incorporate the Points-To Cell Logic to support the desired interactions between mutable references and cells. To do this, the most important lesson to keep in mind is the lesson from Heap-Write2—that to read from ℓ, we only need to mutate the last "link" of a cell chain, i.e., it suffices to have read only access to $\ell \vdash\!\!\lfloor_{\gamma_1,\ldots,\gamma_{k-1}}\!\!\rfloor\!\!\mapsto \gamma_k$, and mutable access to $\gamma_k \mapsto v$. To define the model of mutable borrows, we put the $\gamma_k \mapsto v$ component in a full borrow, and the $\ell \vdash\!\!\lfloor_{\gamma_1,\ldots,\gamma_{k-1}}\!\!\rfloor\!\!\mapsto \gamma_k$ inside a shared borrow. By structuring the model this way, we are able to prove the soundness of our proposed `borrow_mut` function.

6 Results and Limitations

In VerusBelt, we prove type-specs for all the major primitive operations for Verus's APIs `PCell`, `PPtr`, `PointsTo`, `LocalInvariant`, and `AtomicInvariant`, as well as resource algebras and storage protocols. For `PCell` (and `PCell::PointsTo`), this means operations to create a cell, destroy a cell, borrow immutably from a cell, and borrow mutably from a cell. For `PPtr` (and `PPtr::PointsTo`), the four analogous functions: allocate, free, borrow immutably, and borrow mutably. For the invariant types: to create, destroy, and access. For RAs and SPs, see our accompanying artifact for details.

We proved these together with standard Rust types: boxes, product types, sum types, shared borrows, mutable borrows (using a RustHorn-style prophecy encoding), and most of their standard typing rules. We also support a function to spawn threads and fixpoint theorems for instantiating recursive types. We proved most relevant trait bounds relating to marker traits `Copy`, `Send`, and `Sync`, both for standard and for Verus-specific types.

Finally, we prove the **type-spec soundness theorem**, again much like RustHornBelt's but modified slightly for the additional parameters ($m : \mathcal{P}(\mathit{Name})$ and $\pi : \mathit{ProphAsn}$) used in VerusBelt. The theorem describes the behavior of a function `f` typed with the "trivial precondition."

Theorem 6.1. *Consider a function* `f`*, taking no arguments and returning unit, typed as:*

$$\texttt{f} : \mathit{fn}() \to () \rightsquigarrow \Phi_{\mathit{triv}} \qquad \text{where} \qquad \Phi_{\mathit{triv}} \triangleq \lambda\phi, m, \pi.\ \phi(())(m)(\pi)$$

$$(\Phi_{\mathit{triv}} : (\lfloor () \rfloor \to \mathcal{P}(\mathit{Name}) \to \mathit{ProphAsn} \to \mathit{Prop}) \to (\mathcal{P}(\mathit{Name}) \to \mathit{ProphAsn} \to \mathit{Prop}))$$

Then, `f` *executing under VerusBelt semantics will not exhibit undefined behavior.*

The type-spec theorem enables a two-phase approach to verifying a piece of code. In the first phase, *type-checking*, we end up with some judgment like $\texttt{f} : \mathit{fn}() \to () \rightsquigarrow \Phi$ for some Φ that "accumulates" all the predicate transformers of individual typing rules. We then enter the second, *spec-checking*, phase, in which we need to prove that $\forall \phi, m, \pi.\ \Phi_{\mathit{triv}}(\phi)(m)(\pi) \Rightarrow \Phi(\phi)(m)(\pi)$. This will in turn guarantee (by weakening of the type-spec judgment) that $\texttt{f} : \mathit{fn}() \to () \rightsquigarrow \Phi_{\text{triv}}$,

thus enabling us to apply the type-spec theorem. While in VerusBelt everything is done in Rocq, this two-phase approach nevertheless resembles what Verus actually does: to check a function, it typechecks it (using Rust), translates the function into a verification condition using a weakest precondition transformer, and then checks the condition with an automated solver, usually Z3 [5].

Differences between VerusBelt and real Verus/Rust. For technical reasons, we were not able to prove a `Send` bound for `&mut T`, outside of static lifetimes. This limitation arises in part from complexities in the semantic definition of `Send` that are necessary to support `LocalInvariant`. There are also some minor differences between the VerusBelt specs and the Verus specs for RAs and storage protocols; the specifics are documented in our artifact. At present, we do not believe either of these represents a genuine error in the Verus specs, but bridging these gaps may require additional adjustments to our models. Furthermore, like RustBelt and RustHornBelt, VerusBelt models a language that is simplified in many ways compared to "real" Rust (*e.g.,* simplified type layout, pointer provenance rules, no "two-phase borrows").

We emphasize again that VerusBelt primarily targets Verus's proof-oriented types and their interactions with Rust's references. Verus also provides axiomatic specifications for many standard library types—*e.g.,* `Vec` and `HashMap`—which are out of scope for VerusBelt. (We argue these would be better suited for verification in Verus itself.) Verus's translation of Rust source to verification conditions is also entirely trusted code, and we do not attempt to formalize the connection between predicate transformers (Φ) and Verus's translation.[6] Finally, Verus relies on a number of automated solvers (such as Z3 or Singular [6]), which likewise are not verified or proof-producing.

On executable and ghost code. One major limitation of VerusBelt is the lack of a proper distinction between "executable code" and "ghost code." Practically speaking, this means that VerusBelt would correspond more closely to a hypothetical version of Verus which treated all ghost types (like `LocalInvariant` and `PointsTo`) as ordinary zero-sized Rust types (ZSTs), compiled in the normal way with `rustc`. In reality, Verus has a fairly elaborate compilation scheme where it *erases* all such ghost code before applying codegen.

This means that potential soundness issues in Verus's erasure scheme are outside the scope of VerusBelt. For example, there could be a bug where Verus erases an infinite loop, changing the semantics of the program. Addressing this properly—*i.e.,* stating and proving a realistic erasure theorem—would require accounting for a number of features and challenges that we currently do not consider in VerusBelt. One such feature is Verus's *decreases-measure*, together with its associated verification conditions, which ensure termination via well-founded recursion. Another such feature is Verus's *invariant-credit* system, which it uses to prevent nontermination in the style of Landin's knot, an issue which becomes relevant in the presence of higher-order ghost functions (also not treated by VerusBelt). We leave these exciting technical challenges to future work.

7 Related Work

The first Verus publication [23] presented a formalism for Verus via an idealized λ-calculus. VerusBelt complements this original formalism in some ways and supersedes it in others. First of all, the original λ-calculus formalizes the *mode* system, *i.e.,* it properly distinguishes executable code from ghost code and proves termination of ghost code, as well as an erasure-validity theorem, which VerusBelt does not do. (However, it does not address all the challenges described in §6.) On the other hand, the original formalism supports only a primitive form of shared borrows, without a

[6] Specifically, VerusBelt specifications for individual operations are based on Verus weakest preconditions, but these translations are entirely manual. Furthermore, Verus and VerusBelt differ in the handling of control flow. Verus operates on loops and conditionals, derived directly from the Rust source, whereas RustBelt (and hence VerusBelt) uses a continuation passing style that more closely reflects the arbitrary control flow graphs of Rust's internal representation, MIR.

rich lifetime system, and without mutable borrows, concurrency, interior mutability, or storage protocols. It also does not handle *unsafe* functions that (like most features in this paper) need checked preconditions to be considered safe. In contrast, VerusBelt accounts for all these features.

There is at this point a wide range of systems for formal verification of Rust code [18, 2, 28, 7, 10, 25, 8, 3]. Some of these systems, such as Prusti [2] and Flux [25], have specifically targeted safe Rust code. In contrast, VerusBelt builds on a line of work aimed at foundationally verifying Rust programs with *unsafe* code, beginning with RustBelt [18] and RustHornBelt [28].

The original goal of RustBelt was to prove soundness of a core subset of Rust, as well as a variety of essential Rust APIs (mostly interior-mutable) that were implemented using unsafe features. RustHornBelt extended RustBelt to serve as a foundation for *functional verification*. Specifically, it was developed to prove *correctness* of Rust APIs implemented using unsafe code, so that such APIs could be safely linked with safe Rust code that was automatically verified with Creusot [7].

The verifications of safety and/or correctness of internally-unsafe Rust APIs in RustBelt and RustHornBelt were done manually in Iris/Rocq, but subsequent work has explored how to automate them. RefinedRust [10] is a framework for semi-automated verification of Rust programs with unsafe code; it employs tactic-based automation following prior work on RefinedC [30], but also uses a model of mutable references based on the prophecy encoding of RustHornBelt. It is fully "foundational" (mechanized completely in Rocq), but with less powerful automation than other SMT-based systems. Foroushaani and Jacobs [8] extend the VeriFast tool [16] to support proofs of semantic safety for a variety of Rust APIs, effectively automating the RustBelt proofs using modular symbolic execution (but with the larger trusted computing base of VeriFast compared to Rocq). Analogously, Gillian-Rust [3] shows how the verification conditions of RustHornBelt can be automated with symbolic execution to support a "hybrid" approach to Rust verification (semi-automated proofs of unsafe APIs in Gillian-Rust, automated proofs in Creusot). Finally, in very recent work [11], Creusot itself has been extended with Verus-style ghost types to support a similar style of verification to Verus (though not including storage protocols); we believe VerusBelt could also potentially serve as a semantic foundation for this extension of Creusot.

Though Verus builds heavily on RustHornBelt, our goals differ in a subtle but important way from RustHornBelt (and from other hybrid approaches). Whereas RustHornBelt would propose using separation logic (specifically, Iris) to verify an unsafe library like `Cell` or `Arc`, VerusBelt uses Iris only to verify the proof-oriented primitive types of Verus, and expects developers to verify these libraries by (safely!) building on top of these proof-oriented types in Rust/Verus. This approach enables a greater deal of automation by leveraging Rust's type system together with Verus's SMT-based theorem proving, as quantified by Lattuada et al. [21]. On the other hand, Verus *does* require users to write "nonstandard" Rust code using Verus's bespoke, proof-oriented types, while most other tools target "standard" unsafe Rust code.

GhostCell [35] is a Rust API that was proposed for supporting safer programming of pointer-based data structures with internal sharing. Much like Verus's proof-oriented `PCell` type, `GhostCell` "separates permissions from data." Unlike `PCell`, however, it ties cells to their permission tokens using *branded types*, which are encoded by exploiting higher-rank polymorphism over Rust lifetimes. This means it does not need an additional "verification layer," but the system is less flexible in general, as it is not intended for general functional verification. The soundness proof for GhostCell is done via an extension to RustBelt, based on the standard RustBelt lifetime logic.

Though not a Rust verification tool *per se*, IronSync [14] is a concurrency verification framework for Linear Dafny [26] with some ghost primitives resembling Verus's proof-oriented types. It was the first framework to contain a mechanism recognizable as a storage protocol (there called a "Safety-Deposit State Machine"). It leveraged function signatures conceptually similar to Rust's `(&'a T) -> &'a U`, though they were still not quite as flexible as Rust's shared references.

Acknowledgments

We would like to thank Ralf Jung and Bryan Parno for their valuable feedback. We also thank the anonymous reviewers for their valuable feedback. This research was supported in part by the Hakubi Project at Kyoto University and JSPS KAKENHI Grant Number JP24KJ0133 for the third author.

Data Availability Statement

Our documented Rocq artifact is available online [12].

References

[1] Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic foundations for typed assembly languages. *TOPLAS* 32, 3 (2010), 7:1–7:67. doi:10.1145/1709093.1709094

[2] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 147:1–147:30. doi:10.1145/3360573

[3] Sacha-Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. 2025. A Hybrid Approach to Semi-automated Rust Verification. *Proc. ACM Program. Lang.* 9, PLDI, Article 186 (June 2025), 23 pages. doi:10.1145/3729289

[4] Yi Cai, Pratap Singh, Zhengyao Lin, Jay Bosamiya, Joshua Gancher, Milijana Surbatovich, and Bryan Parno. 2025. Vest: verified, secure, high-performance parsing and serialization for rust. In *Proceedings of the 34th USENIX Conference on Security Symposium* (Seattle, WA, USA) *(SEC '25)*. USENIX Association, USA, Article 355, 19 pages.

[5] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (LNCS, Vol. 4963)*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[6] Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann. 2022. Singular 4-3-0 — A computer algebra system for polynomial computations. http://www.singular.uni-kl.de.

[7] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verication of Rust Programs. In *Proceedings of ICFEM 2022 - International Conference on Formal Engineering Methods (Lecture Notes in Computer Science)*. Springer Verlag, Madrid, Spain. doi:10.1007/978-3-031-17244-1_6

[8] Nima Rahimi Foroushaani and Bart Jacobs. 2022. Modular Formal Verification of Rust Programs with Unsafe Blocks. arXiv:2212.12976 [cs.LO] https://arxiv.org/abs/2212.12976

[9] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. 2021. Steel: proof-oriented programming in a dependently typed concurrent separation logic. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. doi:10.1145/3473590

[10] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proc. ACM Program. Lang.* 8, PLDI, Article 192 (jun 2024), 25 pages. doi:10.1145/3656422

[11] Arnaud Golfouse, Armaël Guéneau, and Jacques-Henri Jourdan. 2026. Using Ghost Ownership to Verify Union-Find and Persistent Arrays in Rust. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Rennes, France) *(CPP '26)*. Association for Computing Machinery, New York, NY, USA, 383–397. doi:10.1145/3779031.3779086

[12] Travis Hance, Laila Elbeheiry, Yusuke Matsushita, and Derek Dreyer. 2026. *VerusBelt: A Semantic Foundation for Verus's Proof-Oriented Extensions to the Rust Type System (Artifact)*. doi:10.5281/zenodo.19613067

[13] Travis Hance, Jon Howell, Oded Padon, and Bryan Parno. 2023. Leaf: Modularity for Temporary Sharing in Separation Logic. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 223 (2023), 28 pages. doi:10.1145/3622798

[14] Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. 2023. Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 911–929. https://www.usenix.org/conference/osdi23/presentation/hance

[15] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust Verification by Functional Translation. *Proc. ACM Program. Lang.* 6, ICFP (2022), 711–741. doi:10.1145/3547647

[16] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 41–55. doi:10.1007/978-3-642-20398-5_4

[17] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (dec 2019), 32 pages. doi:10.1145/3371109

[18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. doi:10.1145/3158154

[19] Ralf Jung, R. Krebbers, Jacques-Henri Jourdan, A. Bizjak, L. Birkedal, and Derek Dreyer. 2018. Iris From The Ground Up: A Modular Foundation For Higher-Order Concurrent Separation Logic. *Journal of Functional Programming* 28 (2018).

[20] 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 *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. doi:10.1145/2676726.2676980

[21] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 438–454. doi:10.1145/3694715.3695952

[22] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs Using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 85 (2023), 30 pages. doi:10.1145/3586037

[23] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types (extended version). (2023). doi:10.48550/ARXIV.2303.05491

[24] Hayley LeBlanc, Jacob R. Lorch, Chris Hawblitzel, Cheng Huang, Yiheng Tao, Nickolai Zeldovich, and Vijay Chidambaram. 2025. PoWER never corrupts: tool-agnostic verification of crash consistency and corruption detection. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation* (Boston, MA, USA) *(OSDI '25)*. USENIX Association, USA, Article 46, 19 pages.

[25] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. *Proc. ACM Program. Lang.* 7, PLDI, Article 169 (June 2023), 25 pages. doi:10.1145/3591283

[26] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2022. Linear types for large-scale systems verification. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 69 (April 2022), 28 pages. doi:10.1145/3527313

[27] Zhengyao Lin, Michael McLoughlin, Pratap Singh, Rory Brennan-Jones, Paul Hitchcox, Joshua Gancher, and Bryan Parno. 2025. Towards practical, end-to-end formally verified X.509 certificate validators with verdict. In *Proceedings of the 34th USENIX Conference on Security Symposium* (Seattle, WA, USA) *(SEC '25)*. USENIX Association, USA, Article 259, 17 pages.

[28] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: A Semantic Foundation for Functional Verification of Rust Programs With Unsafe Code. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. ACM, 841–856. doi:10.1145/3519939.3523704

[29] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2020. RustHorn: CHC-Based Verification for Rust Programs. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (LNCS, Vol. 12075)*. Springer, 484–514. doi:10.1007/978-3-030-44914-8_18

[30] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the foundational verification of C code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 158–174. doi:10.1145/3453483.3454036

[31] Pratap Singh, Joshua Gancher, and Bryan Parno. 2025. OwlC: compiling security protocols to verified, secure, high-performance libraries. In *Proceedings of the 34th USENIX Conference on Security Symposium* (Seattle, WA, USA) *(SEC '25)*. USENIX Association, USA, Article 261, 20 pages.

[32] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. 2024. Anvil: Verifying Liveness of Cluster Management Controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 649–666. https://www.usenix.org/conference/osdi24/presentation/sun-xudong

[33] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* 71, 6, Article 40 (Nov. 2024), 75 pages. doi:10.1145/3676954

[34] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. 2025. Tree Borrows. *Proc. ACM Program. Lang.* 9, PLDI, Article 188 (June 2025), 24 pages. doi:10.1145/3735592

[35] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. GhostCell: Separating Permissions from Data in Rust. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. doi:10.1145/3473597

[36] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. 2024. VERISMO: a verified security module for confidential VMs. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 32, 16 pages.

Received 2025-11-14; accepted 2026-04-03