Reasoning about Interior Mutability in Rust using Library-Defined Capabilities

FEDERICO POLI, ETH Zurich, Switzerland

XAVIER DENIS, ETH Zurich, Switzerland

PETER MÜLLER, ETH Zurich, Switzerland

ALEXANDER J. SUMMERS, University of British Columbia, Canada

Existing automated verification techniques for safe Rust code rely on the strong type-system properties to reason about programs, especially to deduce which memory locations do not change (i.e., are framed) across function calls. However, these type guarantees do not hold in the presence of *interior mutability* (e.g., when interacting with any concurrent data structure). As a consequence, existing verification techniques for safe code such as Prusti [Astrauskas et al. 2019] and Creusot [Denis et al. 2022] are either unsound or fundamentally incomplete if applied to this setting. In this work, we present the first technique capable of automatically verifying safe clients of existing interiorly mutable types. At the core of our approach, we identify a novel notion of *implicit capabilities*: library-defined properties that cannot be expressed using Rust's types. We propose new annotations to specify these capabilities and a first-order logic encoding suitable for program verification. We have implemented our technique in a verifier called Mendel and used it to prove absence of panics in Rust programs that make use of popular standard-library types with interior mutability, including Rc, Arc, Cell, RefCell, AtomicI32, Mutex and RwLock. Our evaluation shows that these library annotations are *useful* for verifying usages of real-world libraries, and *powerful* enough to require zero client-side annotations in many of the verified programs.

1 INTRODUCTION

Rust's ownership type system offers strong guarantees, such as memory safety, absence of data races, absence of dangling pointers, and, in general, absence of undefined behavior (UB). In the safe language fragment of Rust, these properties are statically guaranteed by the compiler, making it possible for verification techniques and tools to build upon them [Astrauskas et al. 2019; Denis et al. 2022; Ho and Protzenko 2022; Lattuada et al. 2023]. This is achieved by associating an *exclusive capability* [Boyland et al. 2001] to all mutable references and non-borrowed types and a *shared capability* to all immutable references. The former capability guarantees write access and non-aliasing, while the latter read access and immutability.

The *explicit* capabilities prescribed by Rust types are too restrictive to implement certain behaviors: cyclic data structures such as doubly linked lists, concurrent data structures, shared mutable state such as a global cache, etc. To overcome this limitation, Rust libraries may use unsafe Rust in their implementation to mutate state reached via shared references. Such libraries exhibit *interior mutability* in their API.

However, the additional expressivity for such library developers comes at the cost of losing static guarantees. In the presence of interior mutability, the explicit capabilities of an API no longer describe all mutations potentially performed by a library and, thus, no longer provide the static information needed to verify clients. This is an inherent limitation of existing modular verification tools, none of which can reason about basic usages of standard library types with interior mutability.

The program in Fig. 1 shows a simple usage of the Cell type of Rust's standard library, a container that uses interior mutability to provide an aliasable mutable cell in Rust, and thus can be

Authors' addresses: Federico Poli, Department of Computer Science, ETH Zurich, Switzerland, federico.poli@inf.ethz.ch; Xavier Denis, Department of Computer Science, ETH Zurich, Switzerland, denis.xavier@inf.ethz.ch; Peter Müller, Department of Computer Science, ETH Zurich, Switzerland, peter.mueller@inf.ethz.ch; Alexander J. Summers, Department of Computer Science, University of British Columbia, Canada, alex.summers@ubc.ca.

mutated via calls made on shared references. The `cell_client` function receives a shared reference `c` pointing to an instance of `Cell`. We use `c.set(..)` to increment the cell's content. The two `c.get()` calls around the increment return copies of the cell's content. At the end, with an `assert!(..)` statement, the function checks at runtime whether the value read after the increment is precisely one more than the value read before.

The assertion never fails at runtime because the design of the `Cell` library guarantees that its content cannot be modified concurrently by another thread as long as there is a `&Cell` instance like `c`, so the only modification happens as a result of the `set` call. However, existing verification techniques for Rust do not use this information. To soundly model the interior mutability of `Cell`, they model it as an unknown value that changes unpredictably and conservatively, assume that other threads might interfere at any moment and modify the cell's content. Consequently, existing verifiers cannot prove the last assertion, for which they report a spurious verification error.

```rust
fn cell_client(c: &Cell<i32>) {
  let before = c.get(); c.set(before + 1); let after = c.get();
  assert!(before + 1 == after); // Goal: prove that this never fails
}
```

Fig. 1. Example of a client of the `Cell` library.

This work presents the first verification technique that enables automated and precise reasoning about clients of interiorly mutable types. At the core of our work, we identify a novel notion of *implicit capabilities*, which are properties of library types that express guarantees provided by the library design rather than Rust's types. By leveraging these capabilities, our technique can effectively frame and reason about interior mutability. We implemented our technique in a verifier called **Mendel**[1], the first tool capable of automatically verifying code such as Fig. 1.

Note that our technique focuses on modularly verifying *clients* of interiorly mutable libraries using trusted library annotations. Verifying the libraries is orthogonal and requires different techniques to handle their unsafe code.

Contributions. The main contributions of our work are:
  (1) We identify the notion of implicit capabilities, define a rich family of these and propose new annotations to specify them on library types.
  (2) We present a new verification technique that uses capabilities to reason about safe Rust code, even in the presence of interior mutability.
  (3) We define an encoding of our reasoning technique to first-order logic, suitable for automation using an SMT-based verification toolchain.
  (4) We implement our reasoning technique in a deductive verifier for Rust called Mendel, showing with an evaluation that our technique is (1) *useful*, as it supports popular types with interior mutability defined in the standard library and (2) *lightweight* because it requires near-zero annotations on the client side of the simple programs in our evaluation.

Outline. The rest of this paper is structured as follows. Section 2 presents the verification challenges of our setting and identifies our novel notion of implicit capability. Section 3 gives an overview of our technique and tool from a user's perspective. In Section 4, we present our implicit capabilities, introducing library annotations to specify them and semantic rules to reason about

---

[1]Gregor Mendel, the father of modern genetics, opened the way to the study of *capabilities* and *mutations* of DNA.

them. We illustrate our technique using examples in Section 5. Section 6 introduces our novel validation technique to derive properties of sound Rust abstractions from well-established properties of safe Rust. Section 7 presents a first-order logic encoding of our capability-based reasoning technique. Section 8 discusses the implementation of our technique in a deductive verification tool and its evaluation on clients of popular standard-library types. Section 9 discusses related techniques, and we conclude in Section 10.

2   VERIFICATION CHALLENGES

In this section, we present the key problems of verifying clients of libraries exposing interior mutability.

2.1   Shared Mutable State

Interior mutability provides an essential escape hatch from Rust's ownership discipline. It plays a central role in synchronization primitives such as Mutex and for memoization in types like RefCell. At the same time, the aliased mutable state often implied by interior mutability complicates verification. Neither verification techniques that view (safe) Rust programs as essentially functional code [Denis et al. 2022; Ho and Protzenko 2022; Lattuada et al. 2023] nor techniques based on separation logic [Astrauskas et al. 2019] have a built-in support for shared mutable state. In principle, these techniques could be extended to perform some form of rely-guarantee reasoning [Dinsdale-Young et al. 2010; Dodds et al. 2009], which would require defining an invariant and protocol for each mutably aliased region and verifying that all reads and writes to that region maintain the protocol.

Rely-guarantee reasoning is powerful, but overly complicated for the usage patterns of shared mutable state in many commercially used Rust libraries. In contrast to traditional imperative languages, Rust's library types typically constrain their usage of interior mutability; they provides a safe interface with specific guarantees about the aliasing and mutability. These *implicit capabilities* cannot be checked by the Rust compiler, but they provide powerful informal reasoning principles to Rust programmers. For example, given an object c : &Cell, calling c.get() twice in a row is guaranteed to return the same result, a consequence of the single-threaded nature of Cell. Existing verifiers cannot leverage these capabilities and thus struggle to specify and verify this property. This brings us to our first challenge: **how could a verifier leverage the implicit capabilities of Rust types to reason about interior mutability?**

2.2   Usability

All existing verification techniques for safe Rust enable programmers to reason at the abstraction level of the Rust program without exposing them to complex program logic. Our goal is to retain this abstraction also in the presence of interior mutability. To achieve this goal, the guarantees provided by Rust libraries should be expressed via simple annotations rather than complex logical formulas. The challenge, thus, is **how to specify implicit capabilities in a way that is simple for programmers but provides enough information to enable precise reasoning?**

2.3   Conditional Library Properties

Not every usage of interior mutability is as simple as Cell or Mutex: the Arc type of atomic reference counted pointers uses interior mutability to update its reference count. When exactly one copy of the Arc is around, it becomes possible to safely assume the value is unchanged between two consecutive reads; no other thread could have modified this unique Arc. This capability is

```
#[extern_spec]
#[capable(&self => local(self.as_ptr()))] Ⓐ
impl<T : Copy> Cell<T> {
  #[pure_memory]
  pub fn as_ptr(&self) -> *mut T;

  #[pure_unstable]
  #[ensures(result ==== deref(self.as_ptr()))] Ⓑ₁
  pub fn get(&self) -> T;

  #[ensures(deref(self.as_ptr()) ==== value)] Ⓑ₂
  pub fn set(&self, value: T);
}
```

```
fn cell_client(c: &Cell<i32>) {
  let before = c.get();
  c.set(before + 1);
  let after = c.get();
  // Ok!
  assert!(before + 1 == after);
}
```

Fig. 2. The motivating example from Figure 1, together with an annotated version of the Cell API. The annotations express implicit capabilities of the type, as well as postconditions and purity for its functions.

conditional on the runtime value of the reference counter, demonstrating a need for first-class support for capabilities in specifications: **we should be able to condition capabilities on arbitrary runtime expressions**.

3 A WHIRLWIND TOUR OF MENDEL

We implemented our technique in a deductive verifier called Mendel. Reasoning about interior mutability is based on a novel lightweight *capability annotation*. These capabilities describe guarantees of individual types that are not captured by Rust's type system. Mendel can leverage capabilities to automatically reason about the correctness of subtle Rust programs, as we illustrate in this section.

3.1 Motivating Example

We start with the motivating examples from Figure 1. We restated the code and the required Mendel specification in Section 3. On the left side we show the (trusted) annotations of the Cell API, provided as part of the Mendel tool. Line Ⓐ attaches a *local* capability to &Cell<T>. The **local** capability ensures that as long as a &Cell<T> reference exists, all reads must occur from the same thread. We also provide specifications for get and set on lines Ⓑ₁ and Ⓑ₂ which describe their functional behavior. The annotations pure_memory and pure_unstable denote different levels of 'purity' which are available for functions in Mendel. All pure functions are side-effect free and deterministic; he different purity annotations express which parts of the heap they may depend on, as we explain later.

Given these specifications, Mendel verifies cell_client automatically. Because c is of type &Cell<i32>, Mendel can use the **local** capability to conclude that only the code of cell_client can modify the cell's value while the function executes and, thus, that the value of after is exactly the one provided to set. By combining this local reasoning with the functional specifications for get and set, Mendel proves that the assertion holds.

As this example shows, verification for clients of Cell is lightweight and requires no annotations on the client code. As usual in deductive verifiers, the annotations on the library are trusted and provided by the tool developers.

```
#[capable(&mut self => writeRef(self.as_ptr()))]
impl<T> RefCell<T> {}

#[capable(&self => readRef(self.refcell().as_ptr()))]
impl<'b, T> Ref<'b, T> {}

fn use_refcell(x: &RefCell<i32>) { /* omitted */ }

fn refcell_client(x: &RefCell<i32>, y: RefMut<i32>) {
  let Ok(a /*: Ref */) = x.try_borrow() else { return; };
  let before: i32 = *a;
  use_refcell(x);
  let Ok(b /*: Ref */) = x.try_borrow() else { return; };
  let after: i32 = *b;
  assert!(before == after);  // Both assertions succeed
  assert!(x.as_ptr() as *const _ != y.deref() as *const _);
}
```

Fig. 3. Example of a client of the RefCell library. The full annotations on the RefCell library can be seen in the test suite of the evaluation.

3.2 Lightweight Non-aliasing

Rust's RefCell type provides mutable memory locations with dynamically checked borrow rules. The capability annotations in Fig. 3 state two essential guarantees of the RefCell and Ref libraries: (1) Any &mut RefCell instance is capable of obtaining a mutable reference to its content. That is, mutable instances of RefCell hold the unique capability for mutating their content. This annotation uses the existing RefCell::as_ptr API method to identify the content of the RefCell by address. (2) Any &Ref instance is capable of obtaining a shared reference to the content of the associated RefCell. To formalize this property, the annotation uses an auxiliary specification method Ref::refcell (not shown in the figure) that models the RefCell instance associated with a Ref.

Using these two annotations, Mendel verifies panic-freedom of the client function refcell_client as follows. First, we observe that a is alive across the use_refcell(x) call. From the **readRef** capability of a, we can obtain an **immutable** capability, which allows us to establish that a is unchanged across calls. This, combined with standard functional specifications for the other methods in the program, makes it possible to verify the first assertion.

Second, before the last assertion, the instances a and y are both alive. The instance a still holds an **immutable** capability that implies immutability of x's content. According to the first capability annotation in Fig. 3, the instance y of type RefCell holds an implicit capability that implies exclusive mutable access to y's content. Since these two capabilities are incompatible, it follows that the content of x *must* be at a different location than the content of y, which proves the second and last assertion. Without the library capability annotations, none of the two assertions could be verified.

```
fn client(mut x: Arc<i32>, y: Arc<i32>) -> i32 {
  if x.strong_count() == 1 { // All 3 succeed
    assert!(x.strong_count() == 1);
    assert!(x.as_ptr() != y.as_ptr());                        #[extern_spec]
    Arc::into_inner(x).unwrap()                               #[capable(&self
  } else { // All 3 assertions fail                             if self.strong_count() == 1, Ⓐ
    assert!(x.strong_count() != 1);                             local(self.as_ptr())
    assert!(x.as_ptr() != y.as_ptr());                        )]
    assert!(x.into_inner().is_none());                        impl<T> Arc<T> {
    0                                                           // Functional specs for Arc<T>
  }                                                           }
}
```

Fig. 4. Example of a client of the Arc library. The then-branch is correct, while all assertions in the else-branch might fail. Mendel verifies the then-branch successfully, using the conditional capability of the Arc type.

3.3 Conditional Capabilities

Types like the atomic reference counter Arc use interior mutability to modify their counters. The current count affects the result of different API functions like Arc::into_inner[2] which succeeds only if there is a single outstanding reference. The example in Figure 4 tests the count of x to determine whether it is safe to call into_inner. If the strong count is exactly one, then we know that this count is stable, that x and y do not alias (since x must be the only copy of this reference-counted pointer), and that the call to into_inner will succeed (for the same reason). However, if our test determined that the count was not 1, none of these assertions hold: the reference count could have been decremented by another thread to 1 after the test, and x could alias y.

Mendel can perform these reasoning steps based on the annotation on line Ⓐ which grants a **local** capability *only if* the count is exactly 1. This capability allows Mendel to verify the then-branch in client by ruling out interference from other threads (but not the else-branch, where the condition of the capability does not hold.

4 IMPLICIT CAPABILITIES AND FRAMING

In this section, we present the syntax of our capabilities, illustrate their usage on the example from Fig. 3, and present derived properties of our capabilities that enable additional reasoning steps.

4.1 Syntax of Capabilities

Our technique supports the following capabilities (the implementation provides additional syntactic constructs, which are not essential here).

---

[2]https://doc.rust-lang.org/std/sync/struct.Arc.html#method.into_inner

$$\text{cap} \ni c ::= \textbf{readRef}(p) \mid \textbf{writeRef}(p) \mid \textbf{read}(p)$$
$$\mid \textbf{write}(p) \mid \textbf{immutable}(p)$$
$$\mid \textbf{unique}(p) \mid \textbf{local}(p)$$
$$\mid \textbf{noReadRef}(p) \mid \textbf{noWriteRef}(p) \mid c_1 \wedge c_2$$

$$\text{place} \ni p ::= x \mid * p \mid p.f$$
$$\text{type} \ni \tau ::= \textbf{int} \mid \textbf{bool} \mid \textbf{\&mut } \tau$$
$$\mid \textbf{\&} \ \tau \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2$$
$$\text{contract} ::= \textbf{capable}(\tau \textbf{ if } e \Rightarrow c)$$
$$\text{expr} \ni e ::= \textbf{true} \mid \textbf{false} \mid x \mid e_1 \wedge e_2 \mid ...$$

Each capability c falls into one of the following categories:

(1) **Core capabilities**, corresponding to the explicit capabilities of Rust references. **readRef** corresponds to the capability of shared references. It provides *shared* read-only access, with the guarantee that the target memory location is not modified via aliases. **writeRef** corresponds to the capability of mutable references and of fully initialized non-borrowed types. It provides *exclusive* read and write access.

(2) **Fine-grained capabilities** to read, write, express immutability, unique access, or thread-local ownership (**read**, **write**, **immutable**, **unique**, **local**). These are fundamental properties used across many program reasoning techniques, such as the type system of Pony [Clebsch et al. 2015], experimental type systems for C# [Gordon et al. 2012] and, in general, separation logic [Reynolds 2002]. Compared to the core capabilities, **read**and **write**are weaker definitions that lack the immutability and non-aliasing guarantees of shared and mutable references. Likewise, **unique** is weaker than **writeRef**, but for a subtle reason that we will discuss in Sec. 6: **writeRef** implies that it is possible to obtain a mutable reference through the library API, while **unique** does not. As the name implies, **immutable** ensures the target does not change its value across any statement.

(3) **Deny capabilities noReadRef** and **noWriteRef**express that there *cannot* exist references to a certain memory location. This makes it possible to deduce (1) non-aliasing properties between existing references used in safe Rust and memory locations managed by libraries and (2) immutability of memory locations across assignments to mutable references, as done to verify the example in Fig. 3.

Each capability is indexed by a place p, which is a syntactic representation of a memory location. Capabilities are attached to types through **capable**(..) annotations, which can optionally include a condition e that must hold for the capability to be valid. When the condition is *true*, we write **capable**(c) instead.

4.2 Reasoning with Capabilities

During verification, our technique manipulates capabilities in three major ways: (1) It derives from the capabilities of a composite type the capabilities of its components, for instance, the capabilities of a field $p.f$ from the capabilities of a reference p. (2) It derives capabilities from others for the same place, for instance, a **readRef** capability from a **writeRef** capability. (3) It exploits incompatibilities between capabilities; for instance, **writeRef**(p) and **writeRef**(q) imply that p and q are not aliased. In the following, we formalize these important properties of capabilities.

The structural rules in Fig. 5 allow us transform a capability of a reference type into a capability of the pointee type, a capability of a product type into capabilities of its components, and so on. We present here the rules for **readRef**(p) and **writeRef**(p), but analogous rules exist for other capabilities.

The implications and incompatibilities between capabilities for the same place are shown in Fig. 6 by arrows and dashed edges, respectively. Implications are transitive, but incompatibilities

$$\textbf{writeRef}(p) \vdash \textbf{writeRef}(p.f) \qquad\qquad \textbf{readRef}(p) \vdash \textbf{readRef}(p.f)$$

$$\frac{p : \textbf{\&mut } T}{\textbf{writeRef}(p) \vdash \textbf{writeRef}(*p)} \qquad\qquad \frac{p : \textbf{\& } T \ \lor \ p : \textbf{\&mut } T}{\textbf{readRef}(p) \vdash \textbf{readRef}(*p)}$$

Fig. 5. Structural properties of capabilities.

$$\textbf{writeRef}(p) \vdash \textbf{readRef}(p)$$
$$\textbf{readRef}(p) \vdash \textbf{immutable}(p)$$
$$\textbf{immutable}(p) \vdash \textbf{read}(p)$$
...
$$\textbf{immutable}(p) \land \textbf{write}(p) \vdash \bot$$
$$\textbf{unique}(p) \land \textbf{read}(p) \vdash \bot$$

Fig. 6. Diagram (on the left) of the extended capability implications (arrow edges) and incompatibilities (dashed edges), with their mathematical interpretation on the right (implications, then incompatibilities).

are not. Note that $\textbf{read}(p)$ and $\textbf{write}(p)$ are not incompatible; in fact, types like Cell rely on having both capabilities for the same memory location.

Framing Properties. As we illustrated with the examples in the previous section, verification uses capabilities to derive non-aliasing properties and framing properties. The former follows from the incompatibilities discussed above. To derive the latter, we apply the capability reasoning *across* each statement as we explain next.

We say a capability is held *across* a statement if it is available both before and after a statement. We can now define the framing properties that Mendel uses to automate reasoning:

- An **immutable**(p) capability guarantees that p can be framed out of any statement S.
- If p does not occur in a statement S, then a **unique**(p) capability allows excluding p from the frame.
- For any assignment $q = e$, the combination of **local**(p) and **noWriteRef**(p) ensure that p can be framed around the assignment. The **local** capability ensures other threads cannot modify p while the **noWriteRef** ensures the current thread cannot either.
- Across call statements $q = f(\overline{x})$ of functions that are marked by a purity annotation (presented in Sec. 4.3), a **local**(p) and **noWriteRef**p capability allows framing p across the call.

4.3 Purity Annotations and Framing of Function Values

Libraries typically offer methods to inspect data structures, for instance, to obtain the length of a vector. When such functions are deterministic, side-effect free, and always terminate, we call them *pure* and allow them to be used in specifications. The result of a pure method typically depends on the state of their arguments. For instance, Vec::len returns the length of a vector by reading its value from a private field.

Reasoning about the results of pure methods, particularly the framing properties of their results in the presence of heap modifications, requires information about the memory locations on which a pure method's result depends. In the absence of unsafe code, framing is simple: calling a pure

method on equal arguments will lead to equal results. However, this conclusion might be invalid when pure methods depend on memory locations subject to interior mutability because these locations could be modified through an alias.

Among libraries with interior mutability, `Cell::as_ptr` is a pure method that returns a raw pointer whose target address is computed as a fixed offset from the address where the `Cell` instance is stored. Consequently, framing is not compromised by interior mutability. In the same API, the `Cell::get` method is pure as well because it returns the value contained in the `Cell` instance by internally dereferencing the result of `Cell::as_ptr`. As discussed earlier, mutations via aliases might affect the result of `Cell::get`, but only within the same thread. However, the result of `Vec::len` is only affected by the vector's contents; the same vector can be moved in memory without affecting its result.

These three methods are all pure but provide different framing guarantees. In the interest of lightweight specifications, we abstain from equipping pure methods with precise specifications of their read effects. Instead, we define three classes of pure functions — #[pure], #[pure_memory], and #[pure_unstable] — corresponding to different footprint definitions, ordered from the more to the less restrictive.

Methods declared #[pure] (such as `Vec::len`) may depend only on the *values* reachable from its arguments. These values might be reached by following fields, dereferencing references, or calling other #[pure] functions. Crucially, pure-value methods must *not* depend on the content of types with interior mutability (i.e., `UnsafeCell`). Moreover, pure-value methods may depend on the address of the target of a raw pointer but not on the value of its target. Conversely, they may depend on the value of the target of a reference but not on the address of their target. This follows the intuition that the value of a raw pointer is its target address, while the value of a reference is the value of its target. These rules provide strong framing properties; interior mutability does not affect pure-value methods.

Methods declared #[pure_memory] (such as `Cell::as_ptr`) may depend on the values reachable from its arguments (like pure-value methods) and, additionally, on their memory addresses.

The difference between #[pure] and #[pure_memory] functions is that the result of the former is guaranteed to be the same no matter how values are moved in memory, while the latter can observe (and return) the different memory addresses reachable from its parameters.

Methods declared #[pure_unstable] (such as `Cell::get`) may depend on any memory value, including the content of types with interior mutability. Consequently, concurrent modifications might affect the result of such methods. This purity level does not enable any form of framing; programmers must provide postconditions that effectively restrict the footprint. For example, as shown in Fig. 7 below, the postcondition of `Cell::get` states that the result is equal to the dereference of **self**.as_ptr(), where ==== expresses structural equality that considers both values and addresses (e.g., the target address of references). In contrast, `deref` is a built-in pure-unstable function that models dereferences. The capabilities that `Cell` declares for the memory location **self**.as_ptr() make it possible to reason about the results of multiple `Cell::get` calls. In pure contexts, the results of pure-unstable functions with the same arguments are guaranteed to be evaluated to the same value.

The purity of method implementations is checked syntactically. These checks ensure, among other rules, that pure methods have copy-type parameters, call only pure methods of the appropriate purity level, contain assignments only to local variables, and follow the rules of the different purity levels explained above.

5 CASE STUDIES

By using the capability and purity annotations presented so far, it is now possible to explain the main proof steps needed to verify the examples in Fig. 10, and Fig. 4, which make use several types of the standard library. The full details of the encoding and memory model will be presented in Sec. 7, while the full annotation of the library types is available in the artifact of the evaluation (Sec. 8).

```
#[extern_spec]
#[capable(&self => local(self.as_ptr()))]
#[capable(&self => noReadRef(self.as_ptr()))]
#[capable(&self => noWriteRef(self.as_ptr()))]
#[capable(&mut self => writeRef(self.as_ptr()))]
impl<T> Cell<T> {
  #[pure_memory]
  pub fn as_ptr(&self) -> *mut T;

  #[ensures(deref(result.as_ptr()) ==== value)]
  pub fn new(value: T) -> Cell<T>;

  #[ensures(deref(self.as_ptr()) ==== value)]
  pub fn set(&self, value: T);
}

#[extern_spec]
impl<T: Copy> Cell<T> {
  #[pure_unstable]
  #[ensures(result ==== deref(self.as_ptr()))]
  pub fn get(&self) -> T;
}
```

Fig. 7. Example of the capability and contract annotations on the Cell type of the standard library. The figure contains two **impl** blocks because in the second the T parameter has an additional Copy trait restriction; this is a design choice of the library API. All #[capable(..)] annotations are trusted by the verifier. The method as_ptr acts as a model of the memory location of the cell's content, so that other contracts can refer to it.

5.0.1 Usage of Cell. The program on the right of Figure 10 contains two assertions, only one of which should verify. Proving or disproving the assertions by using the library annotations provided in Fig. 7 requires first reasoning about the value of x.as_ptr(), which is used in the contracts of the Cell library. Since the Cell::as_ptr method is marked as pure-memory, the call x.as_ptr() is known to depend only on the values and corresponding addresses reachable from x. These values and addresses are constant throughout the execution of the cell_client function, because across each statement, a writeRef(x) capability remains held by the client, implying an immutable(x) capability that in turn guarantees immutability of x.

Since across each statement writeRef(x) implies readRef(*x), where *x is of type Cell, by the capability annotations of the library, we can then deduce local(x.as_ptr()) and noWriteRef(x.as_ptr()). These two capabilities guarantee immutability of the target of

x.as_ptr() across each statement of the function, except for use_cell(x) because it is a call of a non-pure function. This immutability property is sufficient to prove the last assertion in the program, as expected.

Regarding the first assertion, checking a == b, our technique does not generate any deduction that proves it, so the assertion is reported as potentially failing. This is the correct outcome because to falsify a == b it would be enough to implement use_cell with the single statement x.set(x.get() + 1).

5.0.2 Usage of RefCell. The program in Figure 10 contains two assertions, the first of which can be verified by deducing an immutability property for the content of the RefCell across the use_refcell(x) call, the second by proving a non-aliasing property between the data contained in x and y. To verify the absence of panics it would be necessary to also prove that the cell is not mutably borrowed when executing the borrow method, because in that case the library would panic. The RefCell type, in fact, contains two internal states mutable via shared references: one is used to store what is commonly called the content of the cell, while the second holds a reference counter, called borrow flag, that tracks how the content is borrowed. For simplicity, here we only present the main proof steps necessary to verify the two assertions, omitting the proof steps related to the borrow flag.

A simplified version of the annotated library is in Fig. 8. Similarly to the annotations of Cell, the method RefCell::as_ptr is marked as pure-memory, thus the expression x.as_ptr() can be deduced to be constant throughout the execution of refcell_client because a readRef(x) capability is available across each statement, and **readRef** implies **immutable**.

Initially, the content of the cell is copied into the variable before, because *a is desugared to *a.deref() and the contract of the Ref::deref function ensures that the result is a reference pointing to the memory location of a.as_ptr(), which in turn is known to be equal to x.as_ref() because of the postcondition of try_borrow. Across the initialization of before and all the following statements the place a remains *immutably* available (i.e., available, but immutably borrowed), implying for each statement a readRef(a) capability that, by the annotations of Ref, in turn implies readRef(a.as_ptr()). The immutability properties of the latter capability ensure that the content of the cell does not change across these statements, which in particular means that after use_refcell(x), the value of before is still equal to the content of the cell. Next, the *x.borrow() expression returns a Ref instance whose deref method returns a reference to the content of the cell, which is then used to initialize the variable after. Since the value of before was already equal to the content of the cell, it thus follows that before = after, as required to verify the first assertion.

For the second assertion, it is enough to notice that a is available before the assertion, generating a writeRef(a) capability that implies read(a.as_ref()) where a.as_ref() = x.as_ref(). Also y is available, generating a writeRef(y) that, by the annotations of RefMut and implication properties, leads to unique(y.as_ref()). Because of the non-aliasing property between the implied **unique** and **read** capabilities[3], it follows that x.as_ref() ≠ y.as_ref(), which, considering the postcondition of RefMut::deref, is then enough to prove that the second assertion always holds.

5.0.3 Usage of Arc. The program in Fig. 4 requires reasoning about the values and properties of the reference counter of the Arc type, whose API is annotated in Fig. 9. Like with RefCell, the type definition of Arc actually contains (at least) two internal states mutable via shared references: one is used to store what is commonly called the content of the Arc, while the second holds its reference

---

[3]An alternative is to use the non-aliasing property between the implied **immutable** and **write** capabilities.

```
#[capable(&self => readRef(self.as_ptr()))]
impl<'b, T> Ref<'b, T> {}

#[capable(&self => readRef(self.as_ptr()))]
#[capable(&mut self => writeRef(self.as_ptr()))]
impl<'b, T> RefMut<'b, T> {}

#[extern_spec]
impl<T> RefCell<T> {
  #[pure_memory]
  pub fn as_ptr(&self) -> *mut T;

  #[ensures(if let Ok(ref actual_ref) = result {
    actual_ref.as_ptr() == self.as_ptr() && ...
  } else { ... })]
  pub fn try_borrow<'b>(&'b self) -> Result<Ref<'b, T>, ...>;

  #[requires(...)]
  #[ensures(result.as_ptr() == self.as_ptr() && ...)]
  pub fn borrow<'b>(&'b self) -> Ref<'b, T>;
}

#[extern_spec]
impl<'b, T> Deref for Ref<'b, T> {
  #[pure_memory]
  #[ensures(result as *const _ == self.as_ptr())]
  fn deref<'a>(&'a self) -> &'a T;
}

#[extern_spec]
impl<'b, T> DerefMut for RefMut<'b, T> {
  #[ensures(result as *mut _ == self.data_ptr() && ...)]
  fn deref_mut<'a>(&'a mut self) -> &'a mut T;
}
```

Fig. 8. A simplified portion of the library specification of the RefCell standard library module. Methods such as borrow_flag_ptr are not part of the API of RefCell, but clients of the library can model them by introducing new traits. The omitted code (...) corresponds to library properties that are not necessary to prove the examples in this paper.

counter, also called *strong* counter[4]. Since the API does not expose the memory location of this counter, the specification models it with a pure method, marked as ghost to make sure that it can

---

[4]For simplicity, in this example we assume that the *weak* counter is always zero. Reasoning about weak pointers in concurrent code is challenging, but we support that kind of reasoning in thread-unsafe libraries. For example, the Rc library used in our evaluation.

be called only from specifications[5]. The key capability annotation of the library is the following, expressing that if the value of the reference counter is 1, then its value cannot be modified by other threads:

```
#[capable(&self if Arc::strong_count(self) == 1 => local(Arc::strong_ptr(self)))]
```

Thanks to the conditionally **local** capability of Arc capability, in the first branch of arc_client, the reference counter is known to be local. Combined with the **noWriteRef** capability of the Arc library, it ensures that the counter remains exactly 1 for the whole first branch, making it possible to verify the first assertion. Knowing that the counter is 1, the second assertion can then be verified: before the assertion the explicit capability of x implies writeRef(x.as_ptr()), while the explicit capability of y implies readRef(y.as_ptr()). The non-aliasing property between these two capabilities ensures that x.as_ptr() ≠ y.as_ptr(). The assertion in the **else** branch of the program checks a second time the same condition that was necessary to enter the branch. However, the evaluation of the condition of the **if** and the evaluation of the last assertion are done at different program points. When the reference counter is not 1, nothing in the library contracts guarantees its value remains the same. Other threads might drop all the existing clones of x, suddenly bringing the reference counter to 1 just before executing the last assertion. Without any information about the counter, a verifier will then conservatively report a verification error, as expected.

```
#[extern_spec]
#[capable(&self => readRef(Arc::as_ptr(self)))]
#[capable(&self => noReadRef(Arc::strong_ptr(self)))]
#[capable(&self => noWriteRef(Arc::strong_ptr(self)))]
#[capable(&self if Arc::strong_count(self) == 1 => local(Arc::strong_ptr(self)))]
#[capable(&mut self if Arc::strong_count(self) == 1 => writeRef(Arc::as_ptr(self)))]
#[capable(&mut self if Arc::strong_count(self) == 1 => unique(Arc::strong_ptr(self)))]
impl<T> Arc<T> {
  #[pure]
  fn as_ptr(this: &Self) -> *const T;

  #[pure_unstable]
  #[ensures((result == 1) == (deref(Arc::strong_ptr(this)) == 1))]
  fn strong_count(this: &Self) -> usize;

  #[pure] #[ghost_fn]
  fn strong_ptr(this: &Self) -> *mut usize;
}
```

Fig. 9. A simplified portion of the specification of the Arc type of the standard library. The strong_ptr is not part of the API of Arc, but it can be introduced using new traits. Here we show it as part of Arc for simplicity.

---

[5]It does not need to be pure-memory, because moving an Arc instance does not move its content or reference counter, so it is sound to abstract from the concrete memory.

```
fn use_refcell(x: &RefCell<i32>) { /* ??? */ }

fn refcell_client(x: &RefCell<i32>, y: RefMut<i32>) {
  let Ok(a /*: Ref */) = x.try_borrow() else { return; };
  let before: i32 = *a;
  use_refcell(x);
  let after: i32 = *(x.borrow());
  assert!(before == after); // Succeeds
  assert!(x.as_ptr() as *const _ != y.deref() as *const _); // Succeeds
}
```

Fig. 10. Example of a client of the RefCell library. At runtime, both assertions never fail, so a complete verifier should be able to verify them. The first line in the implementation, **let** Ok(..) = .., makes the function return in case the try_borrow method fails due to x being already borrowed.

6 SEMANTICS & SOUNDNESS

There are two aspects to the soundness of Mendel, the first is the soundness of the capabilities and Mendel's reasoning about them, and the second is the soundness of the trusted library specifications provided by Mendel. In this section, we address both aspects; we first present the intuitions behind Mendel's capabilities and then discuss a lightweight technique to prove that specific capabilities are sound. Although a formal proof of soundness does not accompany Mendel, we present the intuitions justifying its underlying reasoning (and soundness) in this section. In this section, we sketch what a formalization of Mendel semantics would look like and the key theorems required to prove the soundness of our verifier.

A primer on separation logic. Separation logic is a formalism designed to reason about *resources*, non-duplicable entities that can be shared or changed by different parts of a program. At its core is the concept of *ownership*, typically of memory locations. When we say that x owns the values $\overline{v}$ (also written $x \mapsto \overline{v}$), we mean that x is the only name through which we can access the portion of memory described by $\overline{v}$. Exclusive ownership enables mutation the memory locations of x. Ownership can be divided into arbitrary fractions (denoted $x \mapsto_q \overline{v}$), giving up the permission to mutate the memory in exchange.

Occasionally, we must reason about *shared mutable state*, whether for concurrency or to implement complex mutable data structures. Separation logic provides a tool for this: *invariants*. An invariant is like a box containing an assertion describing a resource (such as x is an even value). It can be temporarily opened for a single computation step, so long as we put the assertion back in afterward. Between each opening, the resource inside may have changed; thus, we cannot be sure that two consecutive openings will yield the same result. Invariants can be either *global*, useful for concurrent reasoning, or *thread-local*, which enables caches or other thread-local storage.

6.1 Intuitive Semantics for Capabilities

Mendel's capabilities have natural interpretations regarding separation logic: they are separation logic predicates, and their intuitions follow:

(1) **read**(p): Stores a fraction of the ownership for p inside an invariant. This allows the holder of this capability to open the invariant and read a value. By quantifying existentially over

the value of p inside the invariant, we can ensure that two successive reads do not have the guarantee of observing the same value.

(2) **write**(p): Stores the *whole* ownership for p in an invariant.

(3) **immutable**(p): Stores a fraction of the ownership for p in a global an invariant. The difference with regards to **read**, is that it quantifies existentially over the value of p *outside* the invariant. This ensures that each time the invariant is opened during a read, the same values *must* be observed.

(4) **unique**(p): Is an alias for ordinary ownership $p \mapsto \overline{v}$ of some values in memory.

(5) **local**(p): A **local**capability ensures that the invariants of all **read**(p) capabilities is thread local.

(6) **readRef**(p): Contains the ownership predicate for an immutable borrow at p.

(7) **writeRef**(p): Contains the ownership predicate for a mutable borrow at p.

(8) **noReadRef**(p): is equivalent to the negation of **readRef**

(9) **noWriteRef**(p): is equivalent to the negation of **writeRef**

6.2 Soundness of Structural Properties

Given appropriate definitions for each of the capabilities, the structural and implication properties can be modeled as ordinary separation logic implications. For example, weakening a **write**(p) into a **read**(p) can easily be performed, since **write** contains the whole ownership for p in its invariant.

This can also be extended to the non-aliasing properties: to verify that having both **write**(p) and **immutable**(p) it suffices to show that they would combine to more than whole ownership of p.

6.3 Soundness of Immutability Properties

The immutability or framing properties used by Mendel are more challenging to specify. To prove these correct with the lowest required effort in a formalization like RustBelt, we propose to prove a series of lemmas corresponding to each individual immutability property. These lemmas would rely on being able to encode the equalities obtain by a given property as Rust assertions.

Consider the property implied by **immutable**(p), which ensures that across any statement, the value of p is unchanged. Using the formalism of RustBelt [Jung et al. 2018a], the corresponding theorem for a program F would look something like the following: given $T \vdash F$ where $x \lhd \tau \in T$, and **immutable**x then $T \vdash \mathbf{let}\, x' = x \,\mathbf{in}\, F ; \mathbf{assert}(x' = x)$ also types. The proof for this would follow from the definition of **immutable** which would allow us to conclude that the value of x at the moment of the assertion must be the same as before the execution of F.

6.4 Validating Library Specifications

An essential task for any verifier is determining which values are unchanged or *framed* by specific operations. In Mendel, this is one of the essential roles performed by the **readRef** capability, which expresses the immutability of a reference to a memory location. For example, consider the code on the left of Fig. 11, which uses try_borrow to perform a runtime aliasing check on a RefCell, returning an a of type Ref if there are no outstanding RefMut instances. That Ref instance carries with it a **readRef** capability, which allows Mendel to conclude that the value of x is not modified by the call to unknown.

The capability annotations of RefCell result from a careful study of the code and documentation of the RefCell library, an ad-hoc process difficult to follow across Rust's many interiorly mutable libraries. We have developed a novel proof technique to help us specify the capabilities of Rust libraries. The core idea is to rewrite a Rust program into a semantically equivalent one, making explicit the capabilities implicit in the source program.

```
fn client(x: &RefCell<i32>) {                    fn client(x: &RefCell<i32>) {
  if let Ok(a) = x.try_borrow() {                  if let Ok(a) = x.try_borrow() {
    // a is available from here...                   let tmp: &i32 = a.deref();
    unknown(x);                                      unknown(x);
    // ...to here                                    drop(tmp); // restores a
  }                                                }
}                                                }
```

Fig. 11. Example of a rewriting (original code on the left; resulting code on the right) that introduces a conversion method call, a.deref(), using a place that was available across the unknown(x) statement.

The code on the right of Fig. 11 performs this, introducing a local variable tmp pointing to the content of x and dropping tmp after unknown(x) so that it remains alive during the call. This rewriting is semantically equivalent because deref is a *pure*, *total* function that does not use synchronization primitives and thus has no observable side effects. We call such methods *conversion methods*. At this point, we note that in Rust, it is UB's responsibility to have a shared reference whose target value is modified while the reference is alive. Assuming that unknown is sound, and given that tmp is alive during the call of unknown, its value (and thus x's) value cannot have been modified. This result holds in both the transformed and original programs as they are observationally equivalent, and unknown cannot detect the rewriting. This rewriting justifies the **readRef** capability given to Ref instances.

Concretely performing in a tool the rewriting is not necessary in order to derive, e.g., the immutability properties of a **readRef** capability across unknown(x) in the example. Instead, it is sufficient to identify that the rewriting is possible.

One assumption of our proof technique is to know what are the *conversion methods* of a library API, which we define as the public safe methods whose only purpose is to convert between two types in a way that (i) has no side effects, (ii) always terminates, (iii) never panics, and (iv) does not use synchronization primitives. Intuitively, this means that usages of conversion methods cannot be detected at runtime by the library by means such as incrementing a counter each time the method is executed. Examples of conversion methods in the standard library are the methods get_mut (implemented for Mutex, RwLock, Cell, RefCell, OnceCell, UnsafeCell, SyncUnsafeCell), Deref::deref (implemented for MutexGuard, RwLockReadGuard, RwLockWriteGuard, Ref, RefMut, Box), and DerefMut::deref_mut (implemented for MutexGuard, RwLockWriteGuard, RefMut, Box) [Rust 2023].

Identifying the conversion methods of a library API is a manual process that requires reading the documentation and the source code of the library. However, we believe that automatic identification of such methods is much easier than proving the immutability of a memory location handled via raw pointers in unsafe code, which is the alternative to our technique.

7 ENCODING

In this section, we present the main choices of our encoding of Rust programs and of the implicit capabilities to a first-order-logic subset of the Viper language [Müller et al. 2016]. This Viper-based approach allows us to define our Rust verification technique independently from the lower-level techniques used for verification, such as symbolic execution or verification condition generation.

7.1 Memory Model

At a high level, our encoding uses a versioning technique to model the state of the memory at different program points in a control-flow graph. Each program point is associated with an unconstrained memory. Then, while encoding each statement in the control-flow graph, the encoding progressively introduces constraints between memory versions. These constraints express the framing properties which hold across the execution of a statement. Possible thread interferences are encoded as no-op statements, over which only the values other threads cannot modify are framed. Additionally, the typing information generates non-aliasing constraints between values stored with the same memory version, similar to a separating conjunction in separation logic.

We represent the memory (both heap and stack) of a Rust program with a family of *memory* total functions $\mathcal{M}_T : \mathcal{A}_T \times \mathcal{V} \to \mathcal{S}_T$, each of which, indexed by a type T, maps an address $\mathcal{A}_T$ and a memory version $\mathcal{V}$ to a *memory snapshot* $\mathcal{S}_T$ (presented later). These functions are defined using Viper *domains* in the Viper language.

7.2 Type Instances

Rust type instances are modeled as *memory snapshots*: a mathematical representation of the accessible values and memory locations without following raw pointers or entering UnsafeCell types. In our encoding, we model the memory snapshot of a variable x as an algebraic data type (representing its reachable values and memory locations) with the following recursive definition:

- The memory snapshot of a primitive type is its mathematical value.
- The memory snapshot of a raw pointer is the target's address, represented as an integer.
- The memory snapshot of a reference comprises both the address and the memory snapshot of its target.
- The memory snapshot of UnsafeCell does *not* contain any representation of its content. It is as if the UnsafeCell were defined as an empty tuple.
- The memory snapshot of tuples is composed of the memory snapshot of its elements.
- The memory snapshot of structures is composed of the memory snapshot of its fields.
- The memory snapshot of an enumeration is composed of the discriminant represented as an integer and then, depending on the value of the discriminant, the memory snapshot of the fields of the corresponding variant.

Memory snapshots are necessary for modeling pure-memory Rust functions, encoded as a mathematical function whose arguments and return value are the memory snapshots of the corresponding Rust instances. To model pure-value functions, we rely on a weaker representation of Rust instances that abstracts over memory addresses, called *value snapshots*[6]. Since memory snapshots are always at least as descriptive as value snapshots, it is possible to convert instances of the former to the latter.

7.3 Separation in First-Order Logic

For each type, the approach described in Sec. 4 defines that a memory address parametrizes capabilities at a given program point. This is sufficient in a separation-logic formalization, where non-aliasing properties can be defined as holding only between capabilities associated with separate portions of the memory. However, for automation, we designed our encoding based on first-order logic, in which we model capabilities as uninterpreted boolean functions. In this setting, we need a new mechanism to encode whether two capabilities originate from different portions of the memory.

---

[6]For the readers who are already familiar with Astrauskas et al. [2022], the snapshots of Prusti correspond to the value snapshots of this encoding.

In our encoding, we model memory disjointness by introducing for all capabilities a new parameter that we call *root place*, which represents with a syntactic Rust place the set of memory locations that can be reached and mutated using safe Rust code. We define a (finite) set of root places for each program point so that different root places correspond to disjoint memory regions. Thanks to the design of Rust, we can compute this set just by analyzing the borrow-checking information provided by the compiler. The compiler already uses similar representations to ensure memory safety in safe Rust.

Given a type T, we define the capability functions $\langle\text{kind}\rangle_T(r, a, w)$, where $\langle\text{kind}\rangle$ is one of the capability kinds (i.e., **readRef**, **writeRef**, etc.), $w \in \mathcal{V}$ is a memory version that models the memory at a program point, $a \in \mathcal{A}_T$ is the memory address of the capability and $r \in \mathbb{N}$ identifies the root place at program point w from which the capability originates. In our encoding, each root place corresponds to an *explicit* capability assumed to hold based on the compiler information.

With syntactic checks, we also determine which subset of the root places of a program point is not used by the statement that follows immediately after. The *unused root places* computed this way are useful in our encoding to model framing properties across the statement. They represent memory locations guaranteed to be unreachable (thus unaffected) by the statement.

7.4 Properties of Capabilities

For each Rust type T used in a program, the properties of its capabilities are modeled with axioms.

7.4.1 Implication Properties. The implications represented by the arrow edges in Fig. 6 are modeled by axioms of the following shape, where $\langle\text{kind}_{\text{LHS}}\rangle$ and $\langle\text{kind}_{\text{RHS}}\rangle$ are the source and target capability kinds of the edge:

$$\forall r \in \mathbb{N}, a \in \mathcal{A}_T, w \in \mathcal{V}. \; \langle\text{kind}_{\text{LHS}}\rangle_T(r, a, w) \Rightarrow \langle\text{kind}_{\text{RHS}}\rangle_T(r, a, w)$$

As an example, one of the implications for the `i32` type is the following, expressing that for all roots, memory addresses, and versions, a **writeRef** capability always implies **unique**:

$$\forall r, a, w. \; \text{writeRef}_{\text{i32}}(r, a, w) \Rightarrow \text{unique}_{\text{i32}}(r, a, w)$$

The structural implications of Fig. 5 are encoded by a slightly more generic axiom of the following shape, where $\langle a_{\text{RHS}}\rangle$ is the expression representing the address of a field instance or of the target of a reference:

$$\forall r, a, w. \; \langle\text{kind}_{\text{LHS}}\rangle_T(r, a, w) \Rightarrow \langle\text{kind}_{\text{RHS}}\rangle_T(r, \langle a_{\text{RHS}}\rangle, w)$$

7.4.2 Non-Aliasing Properties. Similarly to the implication properties, the non-aliasing represented by the red dashed edges in Fig. 6 are encoded as axioms of the following shape:

$$\forall r_1, r_2, a, w. \; \langle\text{kind}_{\text{LHS}}\rangle_T(r_1, a, v) \wedge \langle\text{kind}_{\text{RHS}}\rangle_T(r_2, a, v) \; \Rightarrow \; r_1 \neq r_2$$

7.4.3 Capability Annotations. The encoding of any capability annotation, conditional or not, is done by generating an axiom of the following shape. In the axiom template, $\langle\text{cond}\rangle$ represents the runtime condition that guards the capability annotation ("true" if there is none), and $\langle a_{\text{RHS}}\rangle$ is the encoding of the Rust expression identifying the target memory location of the capability.

$$\forall r, a, w. \; \langle\text{kind}_{\text{LHS}}\rangle_T(r, a, w) \wedge \langle\text{cond}\rangle \; \Rightarrow \; \langle\text{kind}_{\text{RHS}}\rangle_T(r, \langle a_{\text{RHS}}\rangle, w)$$

7.4.4 Immutability Properties. The immutability properties constrain the memory versions of two program points separated by a statement, e.g., w_1 and w_2. In our encoding, we model each statement as having one extra memory version representing the transition between w_1 and w_2. The encoding assumes (based on the compiler's information) the explicit capabilities of the *unused root*

Table 1. On the left: Description of the modeled unstable memory locations. On the right: Description of the library annotations. "LOC" reports the number of lines of code needed to annotate the library (excluding empty lines, comments, imports and module declarations). "Functions" reports how many existing functions (including methods) were annotated as pure or not ("Imp."), and how many ghost functions were necessary to annotate the library. "Specifications" reports the number of library capability annotations ("Cap."), and the number of lines of code ("Contr.") occupied by the contracts (i.e., pre- and postconditions, and purity attributes).

Library	Modeled unstable locations	Type	LOC	Functions			Specifications	
				Imp.	Pure	Ghost	Cap.	Contr.
UnsafeCell	Content	UnsafeCell	16	3	1	0	1	6
Rc	Content, strong and weak reference counters	Rc	89	4	4	2	10	33
		Arc	51	2	3	1	6	18
Arc	Content, strong reference counter	Cell	28	3	2	0	4	8
		RefCell	127	6	1	10	8	52
Cell	Content	Ref	34	1	1	2	3	9
RefCell	Content, borrow flag	RefMut	45	2	1	2	5	11
AtomicI32	Content	AtomicI32	23	2	0	1	3	5
Mutex	Content, poison flag, lock flag	Mutex	103	4	1	8	9	40
		MutexGuard	48	3	0	2	6	12
RwLock	Content, poison flag	RwLock	76	5	1	5	4	35
		RwLockReadGuard	23	1	0	2	1	4
Box	Target	RwLockWriteGuard	37	2	0	2	4	7
Mutex with invariant	-	Box	40	3	1	1	3	10
Verus-style cell	-	Option	33	3	2	0	0	22
Verus-style pointer	Target	Result	37	4	2	0	0	25
		ControlFlow	9	0	2	0	0	4
		Mutex with inv.	51	6	2	0	0	10
		Verus-style cell	82	5	0	3	0	22
		Verus-style pointer	74	4	1	3	2	16

places of the statement. These follow all properties described above, plus immutability properties such as the following, which models the immutability of the **immutable** capability.

$$\forall r, a, w_1, w_2. \; \text{immutable}_T(r, a, \text{across}(w_1, w_2)) \implies \mathcal{M}_T(a, w_1) = \mathcal{M}_T(a, w_2)$$

8 IMPLEMENTATION AND EVALUATION

We implemented our verification technique in *Mendel*: our new capability-based verification tool for Rust. As an engineering choice, we developed Mendel by reusing various components from the codebase of Prusti: parsing and type-checking of specifications, retrieval of borrow-checker information, and so on. Our implementation removes support for many Rust features not directly relevant to our usage of capabilities such as closures, traits, loops, quantifiers, or iterators. None of these are incompatible with our technique but would require additional engineering work to implement. We then used Mendel to show that our capability annotations are *useful* and, on the client side, *lightweight*. All measurements were averaged over 10 runs (after a JVM warm-up) on a laptop with a i7-7700HQ processor, 16 GB of RAM, and operating system Ubuntu 22.04.

Benchmark Origins. Our benchmarks consist of interiorly mutable libraries and handwritten clients, which we gathered from several sources. First, we annotated the APIs of popular types with and without interior mutability of the standard library using capability annotations, contracts, and helper ghost methods. As a second source of libraries, we ported to our specification language the core of three libraries related to interior mutability taken from the test suites of Creusot and

Table 2. Description of the verified clients, each of which tests properties of the library type reported in the "Used library type" column. "Lines of code" reports the total number of lines of code of the program (excluding empty lines, comments, empty `main` functions, imports and module declarations) and, among them, the lines of code used for contracts. "Assertions" reports the number of assertions to be verified in the program, classified by meaning: "Expected" are assertions that verify or that report a verification error as expected; "Incompl." are assertions for which the verifier reports an error due to an incompleteness. Each assertion occupies 1 line of code, counted as part of the "Total" column. "Time" reports the average verification time with standard deviations, out of 10 runs, using the Viper backend based on verification condition generation.

Client	Used library type	Lines of code		Assertions		Time (s)
		Total	Contracts	Expected	Incompl.	
arc.rs	Arc<i32>	66	0	27	0	10.0 ± 0.7
arc_rwlock.rs	Arc<RwLock<Vec<i32>>>	97	6	29	2	34.6 ± 0.8
atomic.rs	AtomicI32	35	0	9	2	5.6 ± 0.1
cell.rs	Cell<i32>	102	5	30	0	8.8 ± 0.2
mutex.rs	Mutex<i32>	47	0	18	0	12.7 ± 0.4
rc.rs	Rc<i32>	102	0	53	0	15.8 ± 0.9
refcell.rs	RefCell<i32>	71	6	25	0	13.5 ± 0.8
unsafecell.rs	UnsafeCell<i32>	35	7	7	0	4.8 ± 0.2
box.rs	Box<i32>	10	0	4	0	5.5 ± 0.1
mutex_inv.rs	Mutex<i32> with invariant	11	0	1	0	4.9 ± 0.3
verus_cell.rs	Verus-style cell	9	0	4	0	6.2 ± 0.4
verus_ptr.rs	Verus-style pointer	34	7	10	0	10.3 ± 0.5

Verus: a `Mutex` with a monitor invariant from the former; a cell-like and a pointer-like type from the latter. The annotated libraries are described in Table 1.

For each library, we built several safe clients, each including many assertions to check the functional behavior of the API interactions. These clients are simple compared to real-world code, but the goal is to test short sequences of API calls. Finally, we ran our verification tool on the clients, measuring their verification time. The client programs and their verification time are described in Table 2.

8.1 Results

We now present the results of our evaluation for both libraries and their clients.

8.1.1 Library Benchmarks. In Table 1, on the left, we listed the unstable memory locations we modeled for each library using our capability annotations. The simplest library types, such as `UnsafeCell` or `Cell`, only contain one memory location, while more complex types can have more. For example, we modeled three unstable memory locations in the `Rc` and `Mutex` libraries. When annotating the `Arc` library, we made one simplifying assumption: we assumed that the weak reference counter is always zero. This limitation does not apply to single-threaded libraries such as `Rc`, for which we can annotate its strong and weak reference counter. Regarding the types taken from the test suite of Creusot and Verus, we kept the existing assumptions: the type of a `Mutex` with invariant assumes that no mutex is ever poisoned – something that happens when a thread panics while holding a lock. This is not the case for our other annotations of the `Mutex` and `RwLock` types, for which we also model the panic flag. Regarding the Verus-style pointer type, its API is sound, assuming that its clients always respect the declared preconditions. This choice of Verus brings greater flexibility in the design of libraries. We retained this choice to evaluate our tool on these advanced specification cases.

In Table 1, on the right, we present statistics regarding our annotations of the libraries. We could reuse existing API methods for many library types by marking the methods as pure. For example, in the Rc library, we marked the `Rc::as_ptr` method as pure-value, `Deref::deref` pure-memory, `Rc::strong_count` and `Rc::weak_count` as pure-unstable. However, the existing methods were not always sufficient to specify the library. In some cases, we needed to introduce new ghost methods to model specific type properties. Consider the `Mutex` type: we added a `data_ptr` method that returns the address at which the content of the mutex is stored. We marked these additional methods as ghost, though the library implementation could provide most.

Most of our ghost methods were necessary to model aspects of the interiorly mutable values of libraries. For example, by exposing their address or by making it possible to refer to their value from specifications. Among the types with the highest number of ghost methods, `RefCell` uses them to expose the address and value of the contained data and of the borrowing flag, tracking the aliasing status of the type (non-borrowed, read-borrowed, or write-borrowed). In `Mutex`, we used the ghost methods to model the address and value of the protected data and other internal flags (locking and poisoning). In `RwLock`, we did the same as in `Mutex`, but without modeling the locking flag. In the Verus-style libraries, existing methods were already marked as ghosts.

8.1.2 Client Benchmarks. In Table 2, we report statistics regarding the clients in our evaluation, among which the used library type and the average verification time. Each function in the client programs makes a sequence of API calls, checking with many assertions or with a postcondition that the verifier can prove the expected functional behavior of the library. We also used preconditions to check only a particular scenario in a few clients. For example, in `refcell.rs`, we gave functions the precondition that their `RefCell` argument is not read- nor write-borrowed. In two cases, we also hit an incompleteness of our technique, described at the end of this section.

The verification time we measured for these programs goes from 4.8 seconds for the most straightforward libraries to 34.6 seconds for the more complex ones. The slowest client, `arc_rwlock.rs`, requires the verifier to reason about nested library types: a `Arc` containing a `RwLock`, containing a `Vec`. The second slowest client, `refcell.rs`, uses the `RefCell` library, which has the highest number of modeled unstable locations. These measurements were all made using the Viper backend, which internally translates Viper programs to Boogie. By profiling the verifier in a few cases, our preliminary results indicate that most of the verification time is spent generating and parsing Boogie code. This might result from our encoding technique that generates one axiom to model each capability property. Future work might confirm these profiling results and explore more efficient engineering solutions.

8.2 Discussion

From the evaluation results, we can conclude that our specification technique works effectively on real-world libraries with interior mutability. It makes it possible to describe the properties decided by library developers and to verify the usage of these libraries using an automated verifier. In particular:

(1) The specification language is expressive: it is possible to explicitly declare the interior mutable properties of common standard library types.
(2) The technique is compositional: in the presence of nested types such as `Arc<RwLock<Vec<i32>>>`, the capabilities automatically propagate properties across the type boundaries.
(3) The technique integrates with existing code: developers can reuse existing methods in the specifications, for example by modeling with the `as_ptr` method the address of the content of the `Arc` library. Many types require no new methods to be added to the library.

(4) The technique is lightweight: in many cases, the client programs required no proof annotations, and the verification time was reasonable.

Nevertheless, our verification technique has some limitations. Our technique uses capabilities to deduce a program's framing and non-aliasing properties. In certain situations, the capabilities we presented are not expressive enough to describe the content of a type with sufficient precision. When this occurs, we must reason conservatively, assuming that the content might be mutated by any function call or aliased by any other type instance. This leads to incompleteness when reasoning about the framing or non-aliasing properties of the program. We encountered two cases during the evaluation where the verifier reported an incompleteness error. For the Arc type, we do not have a capability that precisely describes its content when (a) the strong reference counter is not 1, or (b) the strong reference counter is 1, the weak reference counter is 0, and the Arc is immutably shared. For the AtomicI32 type, we do not have a capability that describes its content when the AtomicI32 instance is borrowed by *local references*[7]. Future work might overcome this by introducing new capabilities or combining our capability-based technique with other verification techniques for concurrent code.

9 RELATED WORK

9.1 Rust Verifiers

Prusti [Astrauskas et al. 2019] is a deductive verifier for Rust that leverages Rust's type properties to build a memory-safety proof based on separation logic automatically and — given user-written contracts — verify the functional correctness of the program. Prusti does not support reasoning about interior mutability because it lacks the necessary expressivity and completeness, even though its verification technique is sound in the presence of libraries implemented with unsafe code. Since both Mendel and Prusti are sound on their own but incomplete in different cases, combining both approaches would make it possible to reduce the incompleteness to only the cases where both techniques are simultaneously incomplete.

RustBelt [Jung et al. 2018a] is a Coq formalization of Rust in which it is possible to model and verify the soundness of libraries. This work was later extended by RustHornBelt [Matsushita et al. 2022], adding support for verifying functional correctness. While both works are based on the Iris [Jung et al. 2018b] framework and require manual proofs, our verification technique is automated and can be used by developers who do not have advanced knowledge of Coq or separation logic. The language of RustBelt and RustHornBelt is more expressive than the capability specifications of our work but also more verbose. Proving the properties of our capabilities in Iris is an interesting research question for future work. We believe our capabilities are a useful user-readable abstraction that, in Iris, correspond to lemmas that derive the properties of a capability from the semantic invariant of a library type.

Creusot [Denis et al. 2022] is a deductive verifier that leverages Rust's type properties to verify functional properties. Creusot uses a technique based on *prophecies* to encode Rust programs into first-order logic, using the Why3 language [Filliâtre and Paskevich 2013]. The technique that it uses is not based on notions of capabilities and only supports reasoning about interior mutability by wrapping the types behind an API with a monitor invariant. Our technique makes it possible to reason more precisely about mutations to the content of types with interior mutability without needing to change existing methods' signatures or to wrap the types behind a new API. Moreover, Creusot's technique does not support reasoning about memory addresses, while our work has first-class support for them.

---

[7]That is, a reference that has not been passed to any function call.

Verus [Lattuada et al. 2023] is another deductive verifier for Rust that, like Creusot, is based on a first-order logic encoding of Rust programs. One novelty of Verus is it use of the linearity and borrow checks of Rust to let the user manage separation-logic permissions by using regular (ghost) Rust variables. In our view, Mendel's capability system can be seen as a generalization of the permissions tracked by Verus. For example, Verus' PermData type might be modeled in our technique as a structure providing a **unique** capability for the target of the associated PPtr type. The approach of Verus requires defining custom libraries in which the types representing permissions show up explicitly as arguments or return types, while our technique makes it possible to annotate existing libraries without modifying their method signatures, only adding new methods. As a result, our technique requires fewer annotations on the client side and can be applied to existing Rust code.

Aeneas [Ho and Protzenko 2022] translates a subset of Rust into a pure lambda calculus, which can be verified using interactive theorem provers like F* or Lean. Their technique explicitly does not support interior mutability or unsafe code. However, we believe library annotations like ours might be used to detect usages of interior mutability that could be translated into a pure functional language.

9.2 Verification of Other Languages

RefinedC [Sammler et al. 2021] verifies the functional correctness of C code by using a type system with ownership and refinement types, carefully designed so that the Coq proof of memory safety and functional correctness is automated and syntax-directed. Compared to our specification language, their type system is more complex and does not have a notion of immutability.

VCC [Cohen et al. 2009] verifies low-level concurrent C code annotated with global invariants [Cohen et al. 2010]. Their invariants typically require each shared object to keep track of its referencing objects using a set of back-pointers. This technique could be ported to Rust by modeling a ghost set of back-pointers for each object. However, cyclic data structures are unidiomatic in Rust, and manually updating the set of back-pointers is verbose. Our technique requires neither of the two. Still, our first-order logic encoding based on memory versions is inspired by their encoding to Boogie [Barnett et al. 2005].

Pony [Clebsch et al. 2015] is a programming language that ensures data-race freedom of concurrent actor-based code using a strong type system with capabilities, among which deny and unique properties. The rich expressivity of Pony types inspired part of our work on the capabilities of Rust libraries.

10 CONCLUSION

We have presented a new technique to specify the capabilities of Rust libraries implemented with unsafe code and to verify the functional correctness of some of their safe clients. Our approach enables developers to explicitly declare the intended aliasing and mutability properties of *existing* libraries, making it easier for automated verifiers and other human developers to reason about their usage. An open-source implementation of our verification technique and our evaluation data is available on GitHub [men 2024].

REFERENCES

2024. Repository of the Mendel verifier for safe Rust clients of interior mutability. https://github.com/viperproject/mendel-verifier

Vytautas Astrauskas, Aurel Bílý, Jonás Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings (Lecture Notes in*

Computer Science, Vol. 13260), Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez (Eds.). Springer, 88–108. https://doi.org/10.1007/978-3-031-06773-0_5

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. https://doi.org/10.1145/3360573

Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (Lecture Notes in Computer Science, Vol. 4111)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, 364–387. https://doi.org/10.1007/11804192_17

John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2072)*, Jørgen Lindskov Knudsen (Ed.). Springer, 2–27. https://doi.org/10.1007/3-540-45337-7_2

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*, Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos A. Varela (Eds.). ACM, 1–12. https://doi.org/10.1145/2824815.2824816

Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 23–42. https://doi.org/10.1007/978-3-642-03359-9_2

Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. 2010. Local Verification of Global Invariants in Concurrent Programs. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6174)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.). Springer, 480–494. https://doi.org/10.1007/978-3-642-14295-6_42

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13478)*, Adrián Riesco and Min Zhang (Eds.). Springer, 90–105. https://doi.org/10.1007/978-3-031-17244-1_6

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 504–528. https://doi.org/10.1007/978-3-642-14107-2_24

Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. 2009. Deny-Guarantee Reasoning. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 363–377. https://doi.org/10.1007/978-3-642-00590-9_26

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 21–40. https://doi.org/10.1145/2384616.2384619

Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP (2022), 711–741. https://doi.org/10.1145/3547647

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

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. , 286–315 pages.

https://doi.org/10.1145/3586037

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*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 841–856. https://doi.org/10.1145/3519939.3523704

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Rust. 2023. The Rust Standard Library, version 1.69.0 (84c898d65 2023-04-16). https://doc.rust-lang.org/std/

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 *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 158–174. https://doi.org/10.1145/3453483.3454036