RustHornBelt: A Semantic Foundation for Functional Verification of Rust Programs with Unsafe Code

Yusuke Matsushita
The University of Tokyo
Tokyo, Japan
yskm24t@is.s.u-tokyo.ac.jp

Xavier Denis
Université Paris-Saclay, CNRS, ENS Paris-Saclay, INRIA,
Laboratoire Méthodes Formelles
Gif-sur-Yvette, France
xldenis@lri.fr

Jacques-Henri Jourdan
Université Paris-Saclay, CNRS, ENS Paris-Saclay, INRIA,
Laboratoire Méthodes Formelles
Gif-sur-Yvette, France
jacques-henri.jourdan@lri.fr

Derek Dreyer
MPI-SWS
Saarbrücken, Germany
dreyer@mpi-sws.org

Abstract

Rust is a systems programming language that offers both low-level memory operations and high-level safety guarantees, via a strong ownership type system that prohibits mutation of aliased state. In prior work, Matsushita et al. developed RustHorn, a promising technique for functional verification of Rust code: it leverages the strong invariants of Rust types to express the behavior of stateful Rust code with first-order logic (FOL) formulas, whose verification is amenable to off-the-shelf automated techniques. RustHorn's key idea is to use *prophecies* to describe the behavior of mutable borrows. However, the soundness of RustHorn was only established for a *safe* subset of Rust, and it has remained unclear how to extend it to support various safe APIs that encapsulate *unsafe* code (*i.e.,* code where Rust's aliasing discipline is relaxed).

In this paper, we present **RustHornBelt**, the first machine-checked proof of soundness for RustHorn-style verification which supports giving FOL specs to safe APIs implemented with unsafe code. RustHornBelt employs the approach of *semantic typing* used in Jung et al.'s RustBelt framework, but it extends RustBelt's model to reason not only about safety but also functional correctness. The key challenge in RustHornBelt is to develop a semantic model of RustHorn-style prophecies, which we achieve via a new separation-logic mechanism we call *parametric prophecies*.

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

PLDI '22, June 13–17, 2022, San Diego, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523704

Keywords: Rust, separation logic, verification, type systems, prophecy variables, Iris

ACM Reference Format:
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 *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523704

1 Introduction

The Rust programming language [39, 30, 27] has shown that high-level safety is not fundamentally at odds with low-level control. Drawing from decades of academic research [45, 11], Rust employs an *ownership* type system, where aliasing of pointers is tracked statically and direct mutation of aliased state is prohibited. This serves to guarantee memory safety and data-race freedom even in the presence of low-level features like interior pointers and manual deallocation. Unsurprisingly, the arrival of a language with low-level control as in C/C++, *as well as* stronger safety guarantees than in most existing languages, has been met with great interest by academic researchers and industrial software developers alike [22, 21, 19, 35, 17, 37, 40].

However, as Rust gets deployed in ever more critical positions in the software stack, the need to go beyond the mere safety guarantees of the language grows more pressing. Recently, several projects have developed tools for *functional verification* of Rust programs, with a focus on how the safety guarantees provided by the Rust type system can be exploited to simplify the verification problem.

Prusti [6] uses information from the Rust compiler to automatically synthesize *separation logic* [38] proofs of memory safety for Rust programs in the Viper framework [36]; the user can then verify functional correctness on top by instrumenting the source code with additional annotations.

RustHorn by Matsushita et al. [32, 33] goes even further, eliminating separation logic from the picture entirely: they show that, using Rust's strong aliasing guarantees, the behavior of (well-typed) Rust programs can be described in first-order logic (FOL) formulas, without any explicit representation of memory or separation logic analogues like points-to assertions. This encoding is amenable to off-the-shelf logic solvers, as they demonstrated with fully automated verification using CHC (constrained Horn clause) solvers [9].

However, both Prusti and RustHorn share a common limitation: they assume that the program being verified is written entirely in the *safe* fragment of Rust. In reality, however, many Rust programs depend on APIs that, while observably safe, are implemented internally with features that are *unsafe* (*i.e.,* not guarded by Rust's static ownership checking), such as raw pointer accesses and unchecked type casts. For example, the widely used `Vec` API (for growable arrays) manages its underlying buffer and performs memory access using raw pointers. The `Cell` API provides a restricted (*e.g.,* single-threaded) form of shared mutable state, allowing the contents of a `Cell` to be mutated through a shared (immutable) reference to the `Cell`. It has remained unclear how to soundly extend the formal foundations of Prusti and RustHorn to account for APIs like these that safely encapsulate uses of unsafe code.

In this paper, we present **RustHornBelt**, a *semantic* foundation for proving soundness of RustHorn-style verification, which is compatible with safe APIs built from unsafe code, like `Vec` and `Cell`, and is mechanized in Coq [12]. RustHornBelt builds on the RustBelt soundness proof for Rust [21], extending it with a model of types based on RustHorn—an extension that required us to develop several novel techniques in separation logic. Before we get there, though, let us briefly review the prior work on RustHorn and RustBelt.

RustHorn: Leveraging Rust types to verify stateful programs in first-order logic. One of the greatest challenges in automatically checking the safety of stateful programs is dealing with *mutation of aliased (or shared) state.* When an object can be aliased between multiple parts of a program—*i.e.,* there exist multiple references to it—and one alias is used to mutate or potentially deallocate the object, it can be difficult to reason modularly about the result of that mutation from the perspective of the other aliases.

Rust's type system tackles this challenge by restricting the mutation of aliased state. In particular, the design of Rust is centered around the principle of *Aliasing XOR Mutability* (AXM), which says that an object can either be aliased or be mutable, but cannot be both at the same time. The AXM principle is enforced through the concept of *ownership.* By default, objects are exclusively "owned", meaning that whichever piece of code can refer to the object can freely mutate and/or deallocate it but has unique access: there can be no aliases.

However, exclusive ownership *per se* would be too restrictive to account for common C++-style programming idioms. Thus, to enable objects to be passed by reference or shared between multiple parts of the program, Rust introduces the concept of *borrowing*. Given an object `x`, one can create either a *mutable borrow* (`&mut x`) or a *shared borrow* (`&x`) of it. The former has type &α `mut T`, which represents the *unique* right to both read and mutate the object, but only during the *lifetime* α.[1] The latter has type &α `T`, which represents the *freely duplicable* right to read the object during α, but not to write it. In either case, during the lifetime α of the borrow, the original owner of the object loses both read and write access to the object, regaining them only once α is over.

The key insight of **RustHorn** by Matsushita et al. [32, 33] is that, by severely restricting mutation of shared state, Rust's AXM discipline makes it possible to give a *pure, first-order logic (FOL) formulation* of the behavior of stateful Rust code, which is more amenable to fully automatic verification than approaches based on separation logic. For the cases of shared borrows and fully owned objects, this is not too surprising: the former temporarily prohibit state change, and the latter can be described in a standard state-passing style [45, 10, 8].

The interesting case is *mutable borrows*, for which the key question is how to "communicate" the result of state changes through the mutable borrow back to the original owner (lender) of the object, *without* relying on stateful reasoning à la separation logic. RustHorn solves this challenge by using *prophecies*. Prophecies are a classic technique in program verification [1, 44, 24], through which, when verifying a program, one can make proof decisions based on peeking into its future execution. RustHorn uses prophecies to express mutable borrows in functional style: as a pair of the *current* value of the object and the *final* (prophesied) value the object *will* have when the borrow ends.

RustHorn's approach to simplifying the Rust verification problem has already been influential, giving rise to a semi-automated Rust verifier Creusot [15], which uses RustHorn-style prophecy-based translation. RustHorn also motivated recent work on CHC solving [26].

RustBelt: Tackling Rust's type soundness semantically. Matsushita et al. [32, 33] established the soundness of RustHorn via a *syntactic* proof, which supports a significant subset of the *safe* fragment of the Rust language. However, this approach fundamentally bakes in the assumption that all code in the program adheres to the AXM discipline enforced by the Rust type system. As soon as any code in the program violates this discipline by using *unsafe* features of Rust, the syntactic approach breaks down.

This limitation of syntactic proofs of soundness was previously articulated and tackled by Jung et al. [21] in their work on **RustBelt**. That work developed a *semantic* model for a λ-calculus representing a substantial subset of Rust (called

[1] We use Greek letters α, β for lifetimes, instead of `'a`, `'b` as Rust does.

λ_{Rust}) and used the model to prove an *extensible* type soundness theorem for Rust: not only does it verify that *safe* Rust code has well-defined behavior, it also stipulates what verification condition a Rust API that uses *unsafe* features must satisfy in order to be deemed an observably safe *extension* to the language.

The RustBelt soundness proof is formalized in the higher-order concurrent separation logic Iris [25, 28, 23, 20], which is mechanized in the Coq proof assistant [12] and provides an expressive logical language for modular reasoning about ownership and state. Using Iris, RustBelt verified type soundness of λ_{Rust}, along with several representative Rust APIs built from unsafe code, including `Cell` and `Mutex`.

Contributions. In this paper, we present **RustHornBelt**, the first approach to formal verification of Rust programs that accounts soundly for the presence of (safely encapsulated) unsafe code. As the name suggests, RustHornBelt marries the benefits of RustHorn and RustBelt, providing a semantic, RustBelt-style foundation for the soundness of RustHorn-style verification. Specifically, RustHornBelt extends RustBelt's higher-order separation-logic model of Rust types to include a RustHorn-style FOL representation of types. It also extends RustBelt's typing judgment to include a *specification*—in the form of a *predicate transformer*—describing the functional behavior of the typed Rust term with respect to the RustHorn-style representation of types.

RustHornBelt's semantic foundation—presented at a high level in §2 and in more detail in §3—achieves two objectives. First of all, it provides the first machine-checked soundness proof for RustHorn: for *safe* Rust code, it verifies (in Coq) that it satisfies a RustHorn-style FOL specification. Second, like RustBelt, it is *extensible*: for a Rust API `R`, whose implementation uses *unsafe* code, we cannot automatically derive a RustHorn-style spec of its behavior—but we can choose a particular RustHorn-style spec $\mathbf{\Phi}$ that we wish to give to `R`, and RustHornBelt will tell us what verification conditions we must discharge (manually, in Iris) in order to prove that `R` satisfies $\mathbf{\Phi}$. Like RustBelt, RustHornBelt is fully mechanized in the Coq proof assistant, using the Iris framework.

In particular, we have verified RustHorn-style specs for key Rust APIs implemented with unsafe code, including `Vec` (growable array), `SmallVec` (`Vec`-like array that stores elements inline when the length is small), `&α (mut) [T]` (shared/mutable slices), `Iter(Mut)<α,T>` (shared/mutable iterators), `MaybeUninit` (possibly uninitialized object), `swap` (swap via mutable references), `Cell` (shared mutable cell), `spawn`/`join` (thread spawning and joining), and `Mutex` (mutex synchronization wrapper for sharing data across threads). We present several of these API specs in §2.3. As one can see in our paper's artifact [31], we evaluated our approach by fully mechanizing our proofs of soundness of these specs in Coq (§4.1) and also confirmed that our API specs are useful for (semi)-automated verification using Creusot (§4.2).

The key technical challenge we faced in RustHornBelt—which we explore in depth in § 3—was determining how to integrate RustHorn's prophecy-based representation of Rust's mutable borrows into RustBelt. Although Jung et al. [24] have developed an account of prophecy variables in Iris, it is not a good fit for RustHorn-style prophecies for several reasons: (a) it requires the program to be explicitly annotated with prophecy-related ghost instructions; and (b) it treats a prophecy variable name as distinct from the value it resolves to, thus making it seemingly impossible to *partially resolve* a prophecy to a value that mentions other prophesied values (a feature we need for modeling, *e.g., nested borrows* and *borrow subdivision*). We thus instead model RustHorn's prophecies via a new mechanism (encoded in Iris) we call *parametric prophecies*. It alleviates all the above problems with Jung et al.'s prophecies, so long as we embed all our RustHorn-style specs within a reader monad, ensuring that we only make observations about prophecy variables that hold under all possible resolutions of those variables.

Limitations. Although RustHornBelt provides a machine-checked semantic foundation for RustHorn-style verification of Rust programs with unsafe code, it does not constitute an automated verification framework in itself. One must link it with a separate RustHorn-style verifier for safe Rust code (*e.g.,* Creusot), and the implementation of that verifier remains part of the trusted computing base (TCB).

We used Creusot to confirm that our RustHorn-style specs for internally-unsafe Rust APIs are useful for automated verification of client programs; however, there remains a formal gap between RustHornBelt and Creusot. First, Creusot targets surface Rust, whereas RustHornBelt only models λ_{Rust}. Second, as it is built atop Why3 [18], Creusot represents RustHorn-style specs as purely functional WhyML functions rather than (as in RustHornBelt) predicate transformers; that said, there is a close correspondence between the two, which we expect could be formalized in future work.

Although we have formally verified RustHorn-style specs for various APIs, we do not cover all the APIs that were verified safe in RustBelt. Notably, we do not provide any specs for the APIs `Rc`, `Arc`, `RefCell`, and `RwLock`, which implement reference and access counting. This is largely due to a technical issue related to step-indexing, which we discuss in §3.5 but leave as an open problem for future work.

2 Overview of RustHornBelt

In this section, we give a high-level overview of RustHornBelt. We first review RustHorn's prophecy-based translation from Rust to FOL (§2.1), and then show how we formalize that as the *type-spec system*, Rust's type system extended with RustHorn-style specs (§2.2). We also present RustHorn-style specs for various Rust APIs implemented with unsafe code, which we have verified in RustHornBelt (§2.3).

2.1 Key Idea of RustHorn: Mutable Borrows Expressed in FOL via Prophecies

Consider the following program:[2]

```
fn max_mut<α>(ma: &α mut int, mb: &α mut int)
  -> &α mut int { if *ma >= *mb { ma } else { mb } }
fn test(mut a: Box<int>, mut b: Box<int>) {
  let mc = max_mut(&mut a, &mut b);
  *mc += 7; /* end */ assert!(abs(*a - *b) >= 7); }
```

What's going on in test? First, a and b, boxed (fully owned) pointers to integers, are taken as input. Then their integer objects are *mutably borrowed* (&mut a and &mut b) under the same *lifetime* (*i.e.,* time limit of ownership rental) α. This creates *mutable references* of type &α mut int, pointers that borrow ownership from a or b. They are passed to max_mut, which returns the one with the larger target value mc and *drops* the other. Then the target of mc is increased by 7. The mutable borrows temporarily deprive a and b of access to their integer object. When the lifetime α ends, mc's ownership gets expired, and a and b regain access. Finally, the values of a and b are asserted to be different by 7 or more.

Suppose we want to verify that the assertion at the end always succeeds. To do so, we must analyze the effect of the update *mc += 7, where mc's address is determined *dynamically*. Intuitively, this is tricky because whichever of the mutable borrows of a and b stores the smaller value will be dropped (and no longer modified) before its lifetime is over, but this does not involve any explicit communication between the borrower and the original owner. So how can we reason about this communication formally?

RustHorn solved this problem using *prophecies*. When we start a new mutable borrow &mut a, we *prophesy* the *final* state of the borrow a', peeking into the future. Importantly, we *don't decide anything* on the value a' when we create a borrow. Instead, we represent a mutable reference ma as the *pair* (a, a') of the current state a and the final state a'. When a mutable reference (a, a') is dropped (*i.e.,* its ownership is given up), we *learn* $a' = a$, *i.e.,* that the final state a' is now set to the state at that point a (which is called *prophecy resolution*). That information can be in effect "communicated" to the original owner, telling what is the state of the returned object just after the end of the lifetime. This is the key idea of RustHorn, which enables one to give FOL descriptions to Rust programs in an elegant way.

For example, max_mut's *postcondition* (input-output relation) $\mathit{MaxMut}(ma, mb, res)$ can be written in FOL as follows:[3]

$$\begin{aligned}\text{if } ma.1 \geq mb.1 \ \text{ then } & mb.2 = mb.1 \land res = ma \\ \text{else } & ma.2 = ma.1 \land res = mb\end{aligned}$$

Note that the final state of the *dropped* reference (*e.g.,* mb for the first branch) is determined (*e.g.,* $mb.2 = mb.1$).

Now the verification condition for test's assertion can be written as the following FOL formula:

$$\begin{aligned}\forall a, b.\ \forall a', b'.\ \forall mc.\ \mathit{MaxMut}\big((a, a'), (b, b'), mc\big) \ \to \\ mc.2 = mc.1 + 7 \ \to \ |a' - b'| \geq 7\end{aligned}$$

For the mutable borrow &mut a, we *prophesy* the final state a', and represent the mutable references as a pair (a, a'), and similarly for &mut b. Since mc is dropped just after *mc += 7, we set $mc.2 = mc.1 + 7$. Finally, the assertion after the lifetime's end can be described using the prophesied final states a', b' (namely, $|a' - b'| \geq 7$). Indeed, the logic formula above is true. You can check that a' and b' are always set to the actual final state of the borrows.

Though the above example involves borrows of mere integers, RustHorn's prophecy-based representation of mutable borrows can be naturally extended to various use cases of mutable borrows in Rust, such as nested borrows (*e.g.,* &mut &mut int) and borrow subdivision (*e.g.,* getting &α mut int out of &α mut List<int>), as well.

2.2 Type-Spec System: Our Formalization of RustHorn-Style Verification

RustHornBelt provides a solid, mechanized foundation for the soundness of RustHorn-style verification. Toward that end, it formalizes RustHorn-style verification by means of a *type-spec system*, which extends Rust's type system (or rather, the type system of λ_{Rust} developed in RustBelt) with generation of RustHorn-style FOL specifications. As we describe later in §3, RustHornBelt gives a *semantic* proof of soundness for this type-spec system.

Overview of the "type-spec system". The basic judgment of our type-spec system[4] is the *type-spec judgment*, which extends RustBelt's typing judgment with a specification $\mathbf{\Phi}$ of the instruction I's behavior:

$$\mathbf{L} \mid \mathbf{T} \vdash I \dashv \mathtt{r}.\ \mathbf{L}' \mid \mathbf{T}' \rightsquigarrow \mathbf{\Phi}$$

Here, we have two *type contexts* $\mathbf{T}$ and $\mathbf{T}'$, which respectively represent the state before and after executing I. The result of I is bound to the variable r (we omit this part when we ignore the result), which $\mathbf{T}'$ may refer to. A type context is a sequence of items of form either $\mathtt{a}\colon \mathtt{T}$ or $\mathtt{a}\colon^{\dagger\alpha}\ \mathtt{T}$. The former simply means we own an object a of the type T. The latter is more unique to Rust: it means that an object a of type T is *borrowed* under the lifetime α, and thus access to that object *via* a is temporarily *frozen* until α is over.

We also have the input and output *(local) lifetime contexts* $\mathbf{L}$ and $\mathbf{L}'$, which are a set of local lifetimes ($\alpha, \beta, \cdots$) that are alive before and after the execution of I, respectively. (When a lifetime context is empty, we omit it.)

[2] For simplicity, we consider an idealized *unbounded* integer type int. By managing extra preconditions for avoiding overflows, we can easily handle realistic bounded integer types like i32 (32-bit).

[3] The logic is multi-sorted. We often let a variable's sort be implicit.

[4] The type-spec system we present in the paper is a simplified version of the actual one used for Coq mechanization.

After $\rightsquigarrow$ comes what is new, the *specification*. Formally, it is a *(backward) predicate transformer* $\mathbf{\Phi}$, of the sort $(\lfloor \mathbf{T}' \rfloor \to \mathrm{Prop}) \to \lfloor \mathbf{T} \rfloor \to \mathrm{Prop}$, which calculates a precondition (of sort $\lfloor \mathbf{T} \rfloor \to \mathrm{Prop}$) for safe execution of I, given a postcondition (of sort $\lfloor \mathbf{T}' \rfloor \to \mathrm{Prop}$) that must hold after I is executed.

The sort of a type context $\lfloor \mathbf{T} \rfloor$, in turn, is defined as the product (heterogeneous list sort) of the sorts $\lfloor \mathtt{T} \rfloor$ of the items $\mathtt{a}\colon^{?}\mathtt{T}$ in $\mathbf{T}$, where the sort $\lfloor \mathtt{T} \rfloor$ of a type $\mathtt{T}$ is defined as follows:

$$\lfloor \mathtt{int} \rfloor \triangleq \mathbb{Z} \qquad \lfloor \mathtt{Box<T>} \rfloor \triangleq \lfloor \mathtt{T} \rfloor$$
$$\lfloor \mathtt{\&}\alpha\ \mathtt{T} \rfloor \triangleq \lfloor \mathtt{T} \rfloor \qquad \lfloor \mathtt{\&}\alpha\ \mathtt{mut}\ \mathtt{T} \rfloor \triangleq \lfloor \mathtt{T} \rfloor \times \lfloor \mathtt{T} \rfloor$$

Here, $\lfloor \mathtt{T} \rfloor$ describes the RustHorn-style representation of the Rust type $\mathtt{T}$. For $\lfloor \mathtt{int} \rfloor$ it is just an integer $\mathbb{Z}$, and for $\lfloor \mathtt{Box<T>} \rfloor$ and $\lfloor \mathtt{\&}\alpha\ \mathtt{T} \rfloor$ it refers to the representation of the pointer's target. The case of a *mutable reference* $\lfloor \mathtt{\&}\alpha\ \mathtt{mut}\ \mathtt{T} \rfloor$ is the interesting "prophetic" one: its representation is a pair of the *current* value stored there and the *final* value stored there when the lifetime of the borrow ends.

Note that, although the above definition of $\lfloor \mathbf{T} \rfloor$ represents a frozen object in $\mathbf{T}$ with the same sort as an active object, the meaning is quite different. For an active object $\mathtt{a}\colon\mathtt{T}$, its representation $\lfloor \mathtt{T} \rfloor$ is simply $\mathtt{a}$'s current value. For a *frozen* object $\mathtt{a}\colon^{\dagger\alpha}\mathtt{T}$, its representation $\lfloor \mathtt{T} \rfloor$ is the *prophesied* value that $\mathtt{a}$ *will* have at the lifetime α's end.

For a simple example, we can give the following type-spec judgment to integer addition:[5]

$$\mathtt{a}\colon\mathtt{int}, \mathtt{b}\colon\mathtt{int} \vdash \mathtt{a\ +\ b} \dashv \mathtt{c}.\ \mathtt{c}\colon\mathtt{int} \rightsquigarrow \lambda\Psi, [a, b].\ \Psi[a+b]$$

It takes two integers and returns an integer. The *predicate transformer* passes the output $a+b$ to the *postcondition* Ψ; intuitively, it is like a CPS program with the continuation Ψ.

Semantically, a type-spec judgment with spec $\mathbf{\Phi}$ is modeled as a Hoare triple over I, which is universally quantified over its postcondition Ψ and uses $\mathbf{\Phi}\ \Psi$ (roughly) as its precondition. For formal details, see §3.1's (tysp-sem-0) or §3.3's (tysp-sem-1).

Operations for mutable borrows. Let's type *and specify* basic operations for the *mutable borrowing* machinery.

Creation of a mutable borrow is described as follows:

MUTBOR
$$\mathtt{a}\colon\mathtt{Box<T>} \vdash \mathtt{\&mut\ a} \dashv \mathtt{b}.$$
$$\mathtt{a}\colon^{\dagger\alpha}\mathtt{Box<T>}, \mathtt{b}\colon\mathtt{\&}\alpha\ \mathtt{mut}\ \mathtt{T} \rightsquigarrow \lambda\Psi,[a].\,\forall a'.\,\Psi[a',(a,a')]$$

The *final state of the borrow* is *prophesied* as a value a', about which we know nothing now (hence the universal quantifier). The spec says that (1) the first argument to the postcondition Ψ, corresponding to the *frozen lender* $\mathtt{a}\colon^{\dagger\alpha}\mathtt{Box<T>}$ in the output typing context, is the final prophesied value a' that $\mathtt{a}$ will have when the borrow ends; and (2) the second argument to Ψ, corresponding to the *borrower* $\mathtt{b}\colon\mathtt{\&}\alpha\ \mathtt{mut}\ \mathtt{T}$, is the pair of the current state a of the borrowed object $\mathtt{a}$ and the prophesied final state a'.

[5] In a binder like "$\lambda\Psi, [a, b]$." in the spec, the bracket pattern $[a, b]$ simply destructs the (heterogeneous) list of input values.

Writing to a mutable reference is type-spec'ed as follows:

MUTREF-WRITE
$$\alpha \mid \mathtt{b}\colon\mathtt{\&}\alpha\ \mathtt{mut}\ \mathtt{T}, \mathtt{c}\colon\mathtt{T} \vdash \mathtt{*b\ =\ c} \dashv \alpha \mid$$
$$\mathtt{b}\colon\mathtt{\&}\alpha\ \mathtt{mut}\ \mathtt{T} \rightsquigarrow \lambda\Psi, [b, c].\ \Psi[(c, b.2)]$$

The mutable reference's current state is updated to c, but its final state $b.2$ is preserved. The lifetime α should be active.

Dropping a mutable reference is type-spec'ed as follows (we leave the instruction empty since it is a ghost instruction that does not appear in the Rust source program):

MUTREF-BYE
$$\alpha \mid \mathtt{b}\colon\mathtt{\&}\alpha\ \mathtt{mut}\ \mathtt{T} \vdash \ \dashv \alpha \mid \ \rightsquigarrow \lambda\Psi, [b].\ b.2 = b.1 \to \Psi[]$$

Here, since we are dropping $\mathtt{b}$, we know that it will not be updated any further until the lifetime α ends, so we *learn* that the final state $b.2$ is *equal to* the current state $b.1$.[6]

Expiration of a local lifetime α, with objects borrowed under α getting unfrozen, is type-spec'ed as follows:

ENDLFT
$$\alpha \mid \overline{\mathtt{a}\colon^{\dagger\alpha}\mathtt{T}} \vdash \ \dashv \overline{\mathtt{a}\colon\mathtt{T}} \rightsquigarrow \lambda\Psi, \bar{a}.\ \Psi\,\bar{a}$$

This removes α from the lifetime context and changes each $\mathtt{a}\colon^{\dagger\alpha}\mathtt{T}$ into $\mathtt{a}\colon\mathtt{T}$, simply retaining their values.

Composing specs. As seen above, our type-spec system associates each fragment of safe Rust code with a spec in the form of a *predicate transformer*. We can then compose such specs to *verify* the functional behavior of a program.

For example, suppose we want to verify that the assertion of §2.1's `test` always succeeds. For that, we find the overall *precondition* ♠ of `test` and prove that ♠$[a, b]$ holds for any inputs a, b. We can calculate ♠ *backward*, iteratively applying predicate transformers to the final postcondition, just like Dijkstra [16]'s *weakest precondition calculus*. For `test`, we start with the assertion `assert!(abs(*a - *b) >= 7)`'s condition: ♧ $\triangleq \lambda[a, b].\ |a - b| \geq 7$.

First, just before the end of the lifetime, the condition stays the same (ENDLFT). When we go back to just before the update `*mc += 7` (MUTREF-WRITE), which is followed by dropping `mc` (MUTREF-BYE), we get the following new condition:

$$\lambda[a, b, mc].\ mc.2 = mc.1 + 7 \ \to\ |a - b| \geq 7 \qquad (\diamondsuit)$$

Compared to ♧, this is *weakened* by $mc.2 = mc.1 + 7$.

Let's go back more. First, the spec of `max_mut` can be described as follows in predicate-transformer style:

$$\lambda\Psi, [ma, mb].\ \text{if } ma.1 \geq mb.1 \text{ then } mb.2 = mb.1 \to \Psi[ma]$$
$$\text{else } ma.2 = ma.1 \to \Psi[mb]$$

Let's name this spec MaxMut_*. Now we can deduce that the condition just before the call of `max_mut` is as follows (binding the result of `&mut a` to ma and `&mut b` to mb):

$$\lambda[a, b, ma, mb].\ \mathit{MaxMut}_*$$
$$(\lambda mc.\ mc.2 = mc.1 + 7 \to |a - b| \geq 7)\ [ma, mb] \qquad (\heartsuit)$$

[6] If you wonder why implication $\to$ appears here, recall that the predicate transformer outputs the *precondition*. Given a precondition $b.2 = b.1 \to \Psi[]$, after we *learn* the equality $b.2 = b.1$ by prophecy resolution, we can combine the two to get the desired *postcondition* $\Psi[]$.

This is obtained simply by passing the condition (♢) to the predicate transformer $MaxMut_*$.

Finally, we can derive the overall *precondition* of `test`:

$$\lambda[a,b].\ \forall a',b'.\ MaxMut_*\,(\lambda mc.\ mc.2 = mc.1 + 7 \rightarrow |a'-b'| \ge 7)\,[(a,a'),(b,b')] \qquad (\spadesuit)$$

This is calculated from the previous condition (♡) as follows (by MUTBOR): $\spadesuit[a,b] \triangleq \forall a',b'.\,\heartsuit[a',b',(a,a'),(b,b')]$.

This precondition simplifies to the following:

$$\text{if } a \ge b \text{ then } |(a+7)-b| \ge 7 \text{ else } |a-(b+7)| \ge 7$$

Logic solvers can fairly easily prove that this condition holds for all a and b by case analysis on $a \ge b$, thereby establishing that the assertion of §2.1's `test` always succeeds.

2.3 Rust APIs with Unsafe Code

So far we discussed Rust's *safe* features. Now we present RustHorn-style specs for various Rust APIs with *unsafe* implementations, which we have verified in RustHornBelt.

`Vec` API. One common use of unsafe code in Rust APIs is to provide a more efficient implementation than Rust's safe typing rules allow. A canonical example of this is the ubiquitous *vector* (or growable array) type `Vec<T>`. The `Vec` API manages a dynamically allocated memory block to store and provide access to an unbounded number of objects of the type `T`, which it achieves through effective use of *raw pointers*. Raw pointers are Rust pointers whose aliasing is untracked by the type system and which are therefore potentially unsafe to use. The `Vec` API supports a variety of operations; for RustHorn-style verification, we are particularly interested in those that perform destructive *state mutation*.

First, let's consider the following operations:

```
fn push<T>(v: &mut Vec<T>, a: T)
fn pop<T>(v: &mut Vec<T>) -> Option<T>
```

They both destructively update a vector through a *mutable reference* `v: &mut Vec<T>` to it. The operation `push` adds an element `a: T` to the end of the vector (and returns nothing), and `pop` removes the last element `a` from the vector, returning `Some(a)` (and `None` if the vector is empty).

Before we can describe the behavior of these operations, we must first choose a representation for the type `Vec<T>`. Naturally, we represent a vector as a list of its contents: $\lfloor$`Vec<T>`$\rfloor \triangleq List\,\lfloor$`T`$\rfloor$. Correspondingly, the `push` and `pop` operations get the following specs:

$$v.2 = v.1 +\!\!+ [a] \;\longrightarrow\; \Psi[]$$

$$\text{if } v.1 = [] \text{ then } v.2 = [] \;\longrightarrow\; \Psi[\text{None}] \quad \text{else } v.2 = \text{last}\ v.1 \;\longrightarrow\; \Psi[\text{Some}(\text{init}\, v.1)]$$

where last w is the last element of the list w, and init w is w without its last item. In the case of both functions, $v.1$ represents the initial state of the mutable reference `v`; and since `v` is dropped before the function returns, we also learn that the prophesied "final" value of `v` (*i.e.*, $v.2$) is precisely the state of `v` at the end of the function. Thus, so far, $v.1$ and $v.2$ act pretty much like just an input and output.

Things get more interesting when an operation not only inputs but also *outputs* a mutable reference. Let's consider the following operation for random access:

```
fn index_mut<α,T>(v: &α mut Vec<T>, i: int)
                                        -> &α mut T
```

Physically, it is just address calculation: get the head address of the buffer of a vector and add the offset of i blocks. In Rust, however, such addresses are linked with *ownership*. In `index_mut`, the *mutable borrow* over a vector is *subdivided* into a smaller borrow over a specific element of the vector, inheriting the lifetime α.

We give to `index_mut` the following RustHorn-style spec:

$$0 \le i < |v.1| \ \wedge\ \forall a'.\ v.2 = v.1\{i \coloneqq a'\} \longrightarrow \Psi[(v.1[i], a')]$$

The precondition $0 \le i < |v.1|$ is for the bounds check. In addition, we *prophesy* the final state a' of the new, *subdivided* borrow for the output. Now the old borrow's prophesied final state $v.2$ is *partially* determined with respect to a' (an example of *partial prophecy resolution*). It is set to $v.1\{i \coloneqq a'\}$, which can be read as $v.1$ with the i-th element's determination left to the prophesied value a'.

`IterMut` API. Rust's `IterMut` API for *mutable iterators*—though implemented with unsafe code—exemplifies how Rust's type system actually provides stronger guarantees than those of "safe" languages like Java, leveraging ownership to eliminate common pitfalls like *iterator invalidation*.

With `iter_mut`, you can create a mutable iterator out of a mutable reference to a vector:

```
fn iter_mut<α,T>(v: &α mut Vec<T>) -> IterMut<α,T>
```

As the lifetime parameter α of `IterMut` indicates, a mutable iterator is an advanced form of mutable borrow, having temporary ownership of some memory sequence. Rust's type system ensures that, while the iterator `IterMut<α,T>` is active, the ownership of the iterated vector is frozen, preventing the vector from being modified while it is being iterated over—a phenomenon known as iterator invalidation.

With `next`, you can perform one step of mutable iteration:

```
fn next<α,T>(it: &mut IterMut<α,T>)
                              -> Option<&α mut T>
```

This yields a mutable reference to the head element `a: &α mut T`, moving the focus to the next element and returning `Some(a)` (or `None` if the iterator has reached the end).

With iterated application of `next`, it is possible to convert the mutable iterator into a bunch of mutable references to the individual elements of the vector, which can all be used simultaneously—*i.e.*, one need not give up the mutable reference to one element to obtain a mutable reference to the next. Hence, in RustHornBelt, we naturally represent a mutable iterator as a *list of mutable references* to each element of the iterated container, setting $\lfloor \texttt{IterMut<}\alpha\texttt{,T>} \rfloor \triangleq \mathit{List}(\lfloor \texttt{T} \rfloor \times \lfloor \texttt{T} \rfloor)$. This leads to the following straightforward spec for `next`:

$$\begin{aligned} &\text{if } \mathit{it}.1 = [] \text{ then } \mathit{it}.2 = [] \rightarrow \Psi[\text{None}] \\ &\qquad\qquad\;\; \text{else } \mathit{it}.2 = \text{tail } \mathit{it}.1 \rightarrow \Psi[\text{Some}(\text{head } \mathit{it}.1)] \end{aligned}$$

We can also give the following spec to `iter_mut`, which might look tricky at first:

$$|v.2| = |v.1| \rightarrow \Psi[\text{zip } v.1 \; v.2]$$

Essentially, what we are doing is an *elementwise split* of the mutable borrow over the vector (one example of *borrow subdivision*, like `Vec`'s `index_mut`). The borrow's final state $v.2$ is split elementwise into a list of prophesied values $v.2[0]$, $v.2[1], \cdots, v.2[|v.1| - 1]$, and the length $(|v.2| = |v.1|)$ is guaranteed to stay constant. The output iterator works as if it were a list of mutable references to each element of the vector. The function zip works like zip $[a, b, c]$ $[a', b', c']$ = $[(a, a'), (b, b'), (c, c')]$.

Combining `iter_mut` and `next`, we can write and functionally verify various programs that iteratively mutate vectors. For example, let's consider the following function:

```
fn inc_vec(v: &mut Vec<int>)
  { for a in v.iter_mut() { *a += 7; } }
```

This uses a mutable iterator, `v.iter_mut()`, to increment each element of the vector `*v` by 7. The **`for`** statement is syntactic sugar for repeatedly calling the `next` method and unwrapping the result to get `a: &mut int` until `None` is returned. Using the specs of `iter_mut` and `next`, we can derive the following spec on `inc_vec`: $v.2 = \text{map}\,(+7)\; v.1 \rightarrow \Psi[]$.

`SmallVec` API. The *small-vector* type `SmallVec<T,`n`>`[7] acts like a vector `Vec<T>` but uses a trickier memory layout for performance. When the number of the elements is no more than n, it stores all the elements inline, behaving like an *array* `[T;`k`]` (*array mode*). When the number of the elements gets larger, it spills out all the elements into the heap, just like a *vector* `Vec<T>` (*vector mode*).

The `SmallVec` API supports all the key methods of the `Vec` API—including `push`, `pop`, `index_mut` and `iter_mut`. Interestingly, the functional *specs* for these `SmallVec` methods are *exactly the same* as the specs for their `Vec` counterparts. A small-vector is represented as a list of values ($\lfloor \texttt{SmallVec<T,}n\texttt{>} \rfloor \triangleq \mathit{List}\lfloor \texttt{T} \rfloor$), regardless of the internal memory layout (array mode or vector mode). As we can see here, RustHorn-style verification can abstract away representation details and focus on observable functional properties.

`Cell` API. Though useful for avoiding memory safety bugs and data races, Rust's prohibition of aliased mutable state is too restrictive in many situations, such as implementing cyclic data structures. To meet such needs, Rust also provides a number of APIs with *interior mutability*, meaning that they allow mutation even through a *shared* reference, albeit in carefully controlled ways.

Arguably the simplest such API is `Cell`, whose safety is guaranteed by various restrictions (*e.g.*, it can only be used within a single thread). It provides the following operations:

```
fn new<T>(a: T) -> Cell<T>
fn get<T: Copy>(c: &Cell<T>) -> T
fn set<T>(c: &Cell<T>, a: T)
```

You can convert a `T` to a cell `Cell<T>` by calling `new`. Then, using a *shared* reference to a cell `&Cell<T>` with *copyable* content, you can both read from the cell by `get` *and* write a new value to the cell by `set`.

Such interior mutability is useful for writing code but makes functional verification (especially in the RustHorn style) more challenging. RustHornBelt proposes one simple approach to solve this problem: *invariants*.

Concretely, we represent `Cell<T>` as an invariant predicate, with $\lfloor \texttt{Cell<T>} \rfloor \triangleq \lfloor \texttt{T} \rfloor \rightarrow \text{Prop}$. For `get`, we know that the read value a satisfies the invariant, which amounts to the following spec: $\forall a.\; c(a) \rightarrow \Psi[a]$, where c is the invariant representing the cell, of sort $\lfloor \texttt{T} \rfloor \rightarrow \text{Prop}$. For `set`, we promise that writing to the cell will preserve the invariant, hence the following spec: $c(a) \wedge \Psi[]$. For `new`, we can choose the cell's invariant Φ, which should be satisfied by the initial value of the cell. Thus, we give `new` the following spec, for any Φ: $\Phi(a) \wedge \Psi[\Phi]$.

Using these specs, we can do some functional verification. For example, let's consider the following function:

```
fn inc_cell(c: &Cell<int>, i: int)
  { c.set(c.get() + i); }
```

We should ensure that the update by `set` does not invalidate the cell's invariant. That is satisfied by giving the following spec to `inc_cell`: $\big(\forall n.\; c(n) \rightarrow c(n+i)\big) \wedge \Psi[]$. What comes before $\wedge$ is the main precondition, which is satisfied if, for example, $c = \lambda n.\,(n \text{ is odd})$ and $i = 4$.

RustHornBelt allows the invariant Φ for a cell to depend on runtime values. For example, we can call `inc_cell` with the invariant $\lambda n.\; n \bmod k = 1$, where k represents another program variable `k: int`. For technical reasons, we restrict this dependency to *non-prophesied* values: we cannot choose an invariant that depends on the prophecy of a mutable borrow. This is not a strong limitation: as we explain in §4, one practical use case of `Cell` is *memoization*, which does not require to lift this restriction.

Finally, we have also proven sound similar invariant-based specs for the `Mutex` API, a thread-safe variant of `Cell` which uses a lock to control mutable access to the shared cell.

3 Proving Semantic Soundness of the Type-Spec System

We now proceed to explain how we prove *semantic soundness* of our type-spec system. After outlining our semantic

[7] The actual notation used by the Rust library is `SmallVec<[T;`n`]>`.

approach (§3.1), we present our a new prophecy framework *parametric prophecies* (§ 3.2) and show how we use it to solve to the key challenge, modeling *mutable borrows* in the RustHorn style (§3.3). We then sketch the soundness proofs of several type-spec rules (§3.4), before concluding by briefly discussing a technical issue concerning step-indexing (§3.5).

3.1 Basics of Our Semantic Approach

RustBelt's approach. Inspired by earlier work on Foundational Proof-Carrying Code [4, 5, 2, 3], RustBelt proved soundness of Rust (or, to be precise, a simplified variant of Rust called λ_{Rust}) by building a *semantic model* of its type system. In particular, this semantic model modeled Rust types as predicates in Iris, an expressive separation logic. We begin by reviewing the big picture of how this works.

First, each Rust *type* $\mathtt{T}$ is associated with an *ownership predicate* $[\![\mathtt{T}]\!](t, \overline{v})$, a separation-logic predicate in Iris that *semantically* models what it means to own an object of the type $\mathtt{T}$.[8] The predicate takes low-level data $\overline{v} \in \mathit{List\,LowVal}$ (a sequence of values like location/address ℓ, integer n, etc.) as well as a thread identifier $t \in \mathit{ThrId}$ (for modeling concurrency). Then, using ownership predicates, RustBelt gives a semantic model to judgments of Rust's type system.

Finally, RustBelt semantically interprets Rust's syntactic *typing rules* and proves each of them as a separate lemma in Iris. This amounts to Rust's *semantic type soundness* (called the *fundamental theorem of logical relations*). The proof is *extensible*: when you add a new typing rule, all you need is to prove the new rule's semantic interpretation. Notably, this approach can flexibly support various safe Rust APIs implemented with unsafe code (like those discussed in §2.3), by formulating each safe API as a set of new typing rules.

The semantics is validated by the *adequacy theorem*, which says that a complete (*i.e.*, closed) and semantically well-typed Rust program will never encounter undefined behavior (formalized as a "stuck state") under any execution trace.

RustHornBelt's first step. Our work, RustHornBelt, extends RustBelt's approach to tackle soundness of the type-*spec* system, proving functional correctness beyond mere safety. Before diving into our full model, we first show a *simplified* model, which we evolve further in §3.3 and §3.5.

First, the ownership predicate has the form $[\![\mathtt{T}]\!](a, t, \overline{v})$, extending the original RustBelt ownership predicate with a *representation value* $a \in \lfloor \mathtt{T} \rfloor$, which is used for RustHorn-style verification. In the (trivial) case of the integer type `int`, this representation value precisely matches the underlying physical value:[9]

$$[\![\mathtt{int}]\!](n, \_, [m]) \;\triangleq\; n = m$$

[8] Each Rust type $\mathtt{T}$ is also given a *sharing predicate*, which amounts to the ownership predicate of $\&\alpha\ \mathtt{T}$, but we omit the details here for space reasons.
[9] It is defined to be False for cases where the low-level data is not of the form $[m]$ (*e.g.*, $[\ell, \ell']$). The same reading applies to other definitions.

More interesting is the case of boxed pointers (`Box<T>`):

$$[\![\mathtt{Box<T>}]\!](a, t, [\ell]) \;\triangleq\; \exists \overline{v}.\; \ell \mapsto \overline{v} * \mathrm{Dealloc}(\ell, |\mathtt{T}|) * \triangleright [\![\mathtt{T}]\!](a, t, \overline{v})$$

A boxed pointer fully owns the memory block at ℓ (along with the right to deallocate it; $|\mathtt{T}|$ is the size of the low-level data for $\mathtt{T}$, equal to $|\overline{v}|$), and also owns the target object (via the target type $\mathtt{T}$'s ownership predicate). The target object ownership is protected by a *later modality* $\triangleright$—discussed more in § 3.5—which acts as a kind of "guard" so that one can soundly define general *recursive types* such as:

```
enum List<T>{ Cons(T, Box<List<T>>), Nil }
```

The high-level point to take away concerning the above model of `Box<T>` is that it is the same as in RustBelt but for the threading through of the parameter a.

We are now ready to define a simplified version of the semantics of our type-spec judgment, as follows:

$$[\![\mathbf{L} \mid \mathbf{T} \vdash I \dashv \mathtt{r}.\, \mathbf{L}' \mid \mathbf{T}' \rightsquigarrow \Phi]\!] \;\triangleq$$
$$\forall \Psi, t.\; \{\exists \overline{a}.\; \Phi\, \Psi\, \overline{a} * [\![\mathbf{L}]\!] * [\![\mathbf{T}]\!](\overline{a}, t)\}\; I\; \{\mathtt{r}.\, \exists \overline{b}.\; \Psi\, \overline{b} * [\![\mathbf{L}']\!] * [\![\mathbf{T}']\!](\overline{b}, t)\} \qquad \text{(tysp-sem-0)}$$

It is basically a Hoare triple over the instruction I. We quantify over an *arbitrary postcondition* Ψ. The output objects' values $\overline{b}$ should satisfy Ψ, and the input objects' values $\overline{a}$ should satisfy $\Phi\,\Psi$, the precondition calculated by the predicate transformer Φ. The semantics of a type context $[\![\mathbf{T}]\!](\overline{a}, t)$ is simply the separating conjunction of each object's semantics $[\![\mathtt{a} :^{?} \mathtt{T}]\!](a, t)$. For an active object $\mathtt{a} : \mathtt{T}$, the semantics is $[\![\mathtt{T}]\!](a, t, \mathtt{a})$. We give the semantics of frozen objects in §3.3.

3.2 Our Key Innovation: Parametric Prophecies

Background. One of the major challenges tackled by RustBelt was building a semantic model of Rust's mutable and shared borrows. Toward this end, RustBelt relied on a new "logical API" (which was derived within Iris) called the *lifetime logic*, which made it possible to define the model of Rust borrows at a much higher level of abstraction. Inheriting the infrastructure of RustBelt, we too rely on the lifetime logic in our model of borrows (as we will explain in §3.3).

But RustHornBelt faces an additional challenge in modeling *mutable borrows* in particular, namely figuring out how to account semantically for RustHorn-style *prophecies*. To understand the challenge, consider what happens when we try to prove soundness of rule MUTBOR for creation of a mutable borrow. We will see the proof in detail later in §3.3, but roughly, following the structure of (tysp-sem-0), the proof goal will look something like

$$\{(\forall a'.\, \Psi[a', (a, a')]) * [\![\mathtt{Box<T>}]\!](a, t, \mathtt{a})\}$$
$$\&\mathbf{mut}\ \mathtt{a}\ \{\mathtt{b}.\ \exists a'.\ \Psi[a', (a, a')] * [\![\mathtt{a} :^{\dagger\alpha} \mathtt{Box<T>}]\!](a', t) * [\![\&\alpha\ \mathbf{mut}\ \mathtt{T}]\!]((a, a'), t, \mathtt{b})\}$$

where a and a' represent the *current* and (prophesied) *final* states of the borrow, respectively. The problem here is that, to establish the post, we need to pick *some* instantiation for the (existentially quantified) final state a'. How can we do this? We clearly cannot just pick some random value: that would mean committing up front to what the final state will be, which would prevent us from later resolving the prophecy to a potentially different value when the borrow is eventually dropped (rule MUTREF-BYE). Disaster!

Our solution. To solve this, we have developed a novel prophecy framework in Iris, called *parametric prophecies*. Its key idea is to *consider all possible futures simultaneously*. This is achieved through the *clairvoyant monad* $\mathit{Clair}\,A \triangleq \mathit{ProphAsn} \to A$, a reader monad over a *prophecy assignment* $\pi \in \mathit{ProphAsn}$ modeling one possible future (*i.e.,* mapping of prophecy variables to values). By embedding our reasoning about prophecies (especially the spec Φ) within this monad—*i.e.,* parameterizing over *every future* π—we can *refer to* prophesied values while staying parametric w.r.t. what they actually are until we are ready to resolve them. In particular, returning to the proof of MUTBOR, parametric prophecies will enable us to instantiate a' with a freshly chosen prophecy variable in the domain of π, without having to commit to how it is resolved until the borrow is dropped.

Basics. Formally, let a *prophecy (variable)* $x \in \mathit{ProphVar}\,A$ be simply a wrapper around a natural number $n \in \mathbb{N}$. As $\mathit{ProphVar}\,A$ is infinite, we can at any point create a *prophecy token* $[x]_1$ for a *fresh* prophecy x. This token signifies that x has not yet been resolved. Ownership of prophecy tokens can be fractionally split and merged:

PROPH-INTRO
$$\mathsf{True} \Rrightarrow \exists x.\, [x]_1$$

PROPH-FRAC
$$[x]_{q+q'} \dashv\vdash [x]_q * [x]_{q'}$$

A *prophecy assignment* $\pi \in \mathit{ProphAsn}$, modeling one possible future, is a map that assigns a value $\pi\, x \in A$ to every prophecy $x \in \mathit{ProphVar}\,A$ for any sort A. Now we have the *clairvoyant monad* $\mathit{Clair}\,A \triangleq \mathit{ProphAsn} \to A$, parameterization over *every future* π. A prophecy $x \in \mathit{ProphVar}\,A$ lifts to a clairvoyant value $\uparrow x \triangleq \lambda\pi.\, \pi\, x$ ($\in \mathit{Clair}\,A$).

We mark clairvoyant values (*i.e.,* values of sort $\mathit{Clair}\,A$) with a hat $\hat{}$ (*e.g.,* $\hat{a}$). Also, we use the following *functorial* notations with a star $\star$: $\hat{\phi} \overset{\star}{\land} \hat{\psi} \triangleq \lambda\pi.\, \hat{\phi}\,\pi \land \hat{\psi}\,\pi$, $\hat{a} \overset{\star}{=} \hat{b} \triangleq \lambda\pi.\, \hat{a}\,\pi = \hat{b}\,\pi$, $\hat{p}\overset{\star}{.}1 \triangleq \lambda\pi.\, (\hat{p}\,\pi).1$ (similarly for $\overset{\star}{.}2$), ${}^\star(\hat{a}, \hat{b}) \triangleq \lambda\pi.\, (\hat{a}\,\pi, \hat{b}\,\pi)$, and ${}^\star[\hat{a}_1, \ldots, \hat{a}_n] \triangleq \lambda\pi.\, [\hat{a}_1\,\pi, \ldots, \hat{a}_n\,\pi]$.

For prophetic reasoning, we introduce a *prophecy observation* $\langle\hat{\phi}\rangle$ (where $\hat{\phi} \in \mathit{Clair}\,\mathsf{Prop}$), which asserts that a pure proposition $\hat{\phi}\,\pi$ holds for *every valid future* π (*i.e.,* for every π that respects the prophecy resolutions that have occurred so far). The rules for reasoning about observations are fairly straightforward:

PROPH-IMPL
$$\frac{\forall\pi.\, \hat{\phi}\,\pi \to \hat{\psi}\,\pi}{\langle\hat{\phi}\rangle \vdash \langle\hat{\psi}\rangle}$$

PROPH-MERGE
$$\langle\hat{\phi}\rangle * \langle\hat{\psi}\rangle \vdash \langle\hat{\phi} \overset{\star}{\land} \hat{\psi}\rangle$$

PROPH-TRUE
$$\frac{\forall\pi.\, \hat{\phi}\,\pi}{\langle\hat{\phi}\rangle}$$

Prophecy resolution. For each prophecy x, we can *resolve* it exactly once:[10]

PROPH-RESOLVE
$$\frac{\mathsf{dep}(\hat{a}, Y)}{[x]_1 * [Y]_q \Rrightarrow \langle \uparrow x \overset{\star}{=} \hat{a}\rangle * [Y]_q}$$

Consuming the full token $[x]_1$, we can finally fix the value of the prophecy x to an arbitrary clairvoyant value $\hat{a}$, getting an *observational* equality: $\langle \uparrow x \overset{\star}{=} \hat{a}\rangle$. Internally, we *prune away* all the futures in which x is not equal to $\hat{a}$.

Notably, the rule PROPH-RESOLVE allows the clairvoyant value $\hat{a}$ to depend on *other* prophecies (the ones in the set Y). This is essential in RustHornBelt for modeling *borrow subdivision*. For example, let's consider `Vec`'s `index_mut` (§2.3). It *subdivides* the input mutable reference to the vector `v: &α mut Vec<T>` into the output mutable reference to the i-th element `&mut T`. For this subdivision, the input's prophecy x should be *partially resolved* to a value depending on the newly created prophecy y for the output, observing $\langle \uparrow x \overset{\star}{=} {}^\star[\ldots, \hat{a}_{i-1}, \uparrow y, \hat{a}_{i+1}, \ldots]\rangle$, where $\hat{a}_k$ is the current value of the vector's k-th element.

Crucially, however, PROPH-RESOLVE also imposes the condition that the prophecies in the finite set Y (*i.e.,* the ones $\hat{a}$ depends on) must all be *unresolved*.[11] This is ensured by consuming (and then immediately returning) fractional tokens for the prophecies in Y—*i.e.,* $[Y]_q \triangleq \mathop{\Large *}_{y \in Y} [y]_q$. The reason we need this condition is to prevent prophecy resolution from causing a paradox where there are no valid futures.

To see how this might happen, suppose we have $[x]_1$ and $[y]_1$; if PROPH-RESOLVE did not impose the "$[Y]_q$" condition, we could use it to first resolve x to $\uparrow y$, and then resolve y to $\lambda\pi.\, \uparrow x\,\pi + 1$, which put together would yield the impossible observation $\langle \uparrow x \overset{\star}{=} \lambda\pi.\, \uparrow x\,\pi + 1\rangle$. Thanks to the "$[Y]_q$" condition, however, such a paradox is ruled out.

One important consequence of this paradox avoidance is that we are able to additionally prove the following rule, which establishes that reasoning in the clairvoyant monad remains consistent (*i.e.,* there always exists *some* valid π under which our observations hold):

PROPH-SAT
$$\langle\hat{\phi}\rangle \Rrightarrow \exists\pi.\, \hat{\phi}\,\pi$$

In essence, PROPH-SAT says that we can escape the clairvoyant monad and convert our prophetic observation into a "ground" assertion (of type Prop) when needed. Concretely, one key place where this rule is fundamentally needed is in proving that certain branches of a proof or program are impossible (*e.g.,* to prove that `assert!(false)` or `panic!` are dead code). In such cases, our prophetic reasoning will get us to a point where we have proven a contradiction *within*

[10] The *view shift* $P \Rrightarrow Q$ means that a resource of P turns into a resource of Q by updating the internal state. It actually takes a "mask" parameter $\mathcal{E}$, but we elide this detail in the paper to reduce noise.

[11] Here, the predicate finding the dependencies $\mathsf{dep}(\hat{a}, Y)$ is defined as $\mathsf{dep}(\hat{a}, Y) \triangleq \forall\pi, \pi'.\, (\forall z \in Y.\, \pi\, z = \pi'\, z) \to \hat{a}\,\pi = \hat{a}\,\pi'$.

the clairvoyant monad (*i.e.,* $\langle \lambda\_.\ \mathrm{False} \rangle$), but in order to complete the proof we need to obtain a *bona fide* contradiction (*i.e.,* to prove False outside the monad); PROPH-SAT lets us do precisely that by converting $\langle \lambda\_.\ \mathrm{False} \rangle$ to False.

3.3 RustHornBelt's Model of Mutable Borrows

Review of RustBelt's "lifetime logic". As mentioned in §3.2, in RustHornBelt we inherit RustBelt's use of the *lifetime logic* for modeling Rust borrows. Let's briefly review the lifetime logic (see the RustBelt's paper for details; we didn't change the lifetime logic itself). The central mechanism of the lifetime logic is the *borrow proposition* $\&^{\alpha} P$ (specifically called a *full borrow*), which reflects *temporary* ownership of the Iris proposition P but only *during* the lifetime α. Two selected lemmas about $\&^{\alpha} P$:[12]

LFTL-BORROW
$$\triangleright P \ \Rrightarrow\ \&^{\alpha} P \ast \big([\dagger\alpha] \mathrel{\Rrightarrow\!\!\ast} \triangleright P\big)$$

LFTL-BOR-ACC
$$\&^{\alpha} P \ast [\alpha]_q \ \Rrightarrow\ \triangleright P \ast \big(\triangleright P \mathrel{\Rrightarrow\!\!\ast} \&^{\alpha} P \ast [\alpha]_q\big)$$

Depositing $\triangleright P$, we can create a borrow proposition $\&^{\alpha} P$ along with the "inheritance" $[\dagger\alpha] \mathrel{\Rrightarrow\!\!\ast} \triangleright P$, which says that we can retrieve $\triangleright P$ once α has ended (where the death of α is signaled by a *dead lifetime token* $[\dagger\alpha]$). With $\&^{\alpha} P$ in hand, we can get temporary access to its content $\triangleright P$ by trading in a fractional *lifetime token* $[\alpha]_q$, which ensures that α is still alive (in the same way the prophecy token $[x]_q$ ensures that x has not yet been resolved). Remarkably, these lemmas work for *any Iris proposition P*, which exemplifies Iris's *higher-order* expressivity and is crucial for modeling Rust types. The cost of this expressivity, however, is that P must be put under a *later* $\triangleright$, which creates technical difficulties (see §3.5).

Given the lifetime logic, RustBelt models the *mutable reference* type `&α mut T` simply as follows: $[\![\&\alpha\ \mathtt{mut}\ \mathtt{T}]\!](t, [\ell]) \triangleq \&^{\alpha}\big(\exists \bar{v}.\ \ell \mapsto \bar{v} \ast [\![\mathtt{T}]\!](t, \bar{v})\big)$. It is a borrow proposition whose content describes ownership of *some* low-level data $\bar{v}$ stored at ℓ. Correspondingly, a *frozen* object $\mathtt{a} :^{\dagger\alpha} \mathtt{T}$ is modeled as the object's inheritance: $[\![\mathtt{a} :^{\dagger\alpha} \mathtt{T}]\!](t) \triangleq [\dagger\alpha] \mathrel{\Rrightarrow\!\!\ast} [\![\mathtt{T}]\!](t, \mathtt{a})$. This enables the lender of a borrow (*i.e.,* owner of the frozen object) to unfreeze the object once it can prove α is dead.

Lastly, the lifetime context's semantics $[\![\mathbf{L}]\!]$ is defined as the iterated separating conjunction of a token $[\alpha]_1$ (knowledge that α is alive) and the proposition $[\alpha]_1 \mathrel{\Rrightarrow\!\!\ast} [\dagger\alpha]$ (the ability to end α) over all α in $\mathbf{L}$.[13]

RustHornBelt's model of mutable borrows. With the lifetime logic and parametric prophecies in hand, we are now ready to model mutable borrows in the RustHorn style.

First, we update the RustHornBelt ownership predicate into the form $[\![\mathtt{T}]\!](\hat{a}, t, \bar{v})$, where the first parameter is now a *clairvoyant* representation value $\hat{a} \in \mathit{Clair}\,\lfloor \mathtt{T} \rfloor$ rather than an inhabitant of $\lfloor \mathtt{T} \rfloor$. The ownership predicate for `Box<T>` doesn't change (just $\hat{a}$ is used instead of a). For `int`, we have a slight update: $[\![\mathtt{int}]\!](\hat{n}, \_, [m]) \triangleq \hat{n} = \lambda\_.\ m$.

We also update the semantics of the type-spec judgment $[\![\, \mathbf{L} \mid \mathbf{T} \vdash I \dashv \mathtt{r}.\ \mathbf{L}' \mid \mathbf{T}' \rightsquigarrow \mathbf{\Phi} \,]\!]$ as follows, using *clairvoyant* values $\bar{\hat{a}}, \bar{\hat{b}}$ and *prophecy observations* $\langle \lambda\pi.\ \cdots \rangle$:

$$\begin{aligned} &\forall \hat{\Psi}, t.\ \big\{ \exists \bar{\hat{a}}.\ \langle \lambda\pi.\, \mathbf{\Phi}\,(\hat{\Psi}\,\pi)\,(\overline{\hat{a}\,\pi}) \rangle \ast [\![\mathbf{L}]\!] \ast [\![\mathbf{T}]\!](\bar{\hat{a}}, t) \big\} \\ &\qquad I\ \big\{ \mathtt{r}.\ \exists \bar{\hat{b}}.\ \langle \lambda\pi.\, (\hat{\Psi}\,\pi)\,(\overline{\hat{b}\,\pi}) \rangle \ast [\![\mathbf{L}']\!] \ast [\![\mathbf{T}']\!](\bar{\hat{b}}, t) \big\} \end{aligned} \tag{tysp-sem-1}$$

Now for the *pièce de résistance*, we model `&α mut T`, the type of *mutable references*, as follows:

$$\begin{aligned} [\![\&\alpha\ \mathtt{mut}\ \mathtt{T}]\!](\hat{p}, t, [\ell]) \ &\triangleq\ \exists x \text{ s.t. } \hat{p}\overset{\star}{.}2 = {\uparrow}x. \\ &\mathsf{VO}_x(\hat{p}\overset{\star}{.}1) \ast \&^{\alpha}\big(\exists \hat{a}, \bar{v}.\ \ell \mapsto \bar{v} \ast [\![\mathtt{T}]\!](\hat{a}, t, \bar{v}) \ast \mathsf{PC}_x(\hat{a})\big) \end{aligned}$$

There is a lot going on here. First of all, as expected, the RustHorn-style representation $\hat{p}$ of a mutable reference is a clairvoyant pair of the current and final states of the borrow, where the latter is some prophecy x (hence, $\hat{p}\overset{\star}{.}2 = {\uparrow}x$).

The other key difference from the RustBelt model of mutable references is the presence of two ghost state assertions: the *value observer* $\mathsf{VO}_x(\hat{p}\overset{\star}{.}1)$ and the *prophecy controller* $\mathsf{PC}_x(\hat{a})$. The purpose of these assertions is to make it possible to refer to the *current* state of the borrow *both inside and outside of the borrow proposition.* In particular, note that, on the one hand, we need to *existentially* quantify over that current state inside the borrow proposition because otherwise the borrower would not be able to change it when they mutate ℓ; but on the other hand, we also need to be able to connect the current state to the first component of the representation value $\hat{p}\overset{\star}{.}1$. The VO and PC assertions make this possible using a fairly typical Iris-style "linked ghost state" construction, whereby two separately ownable propositions can independently assert the identity of some shared state, with the assurance that (a) their assertions must agree and (b) they can be updated, but only jointly. Formally, we have:

MUT-AGREE
$$\mathsf{VO}_x(\hat{a}) \ast \mathsf{PC}_x(\hat{a}') \ \vdash\ \hat{a} = \hat{a}'$$

MUT-UPDATE
$$\mathsf{VO}_x(\hat{a}) \ast \mathsf{PC}_x(\hat{a}) \ \Rrightarrow\ \mathsf{VO}_x(\hat{a}') \ast \mathsf{PC}_x(\hat{a}')$$

Finally, we model a *borrowed, frozen* object $\mathtt{a} :^{\dagger\alpha} \mathtt{T}$ as:

$$[\![\mathtt{a} :^{\dagger\alpha} \mathtt{T}]\!](\hat{b}, t) \ \triangleq\ [\dagger\alpha] \mathrel{\Rrightarrow\!\!\ast} \exists \hat{a}.\ \langle \hat{b} \overset{\star}{=} \hat{a} \rangle \ast [\![\mathtt{T}]\!](\hat{a}, t, \mathtt{a})$$

After the end of α, we get back ownership of the object of type `T`, whose *actual* value is $\hat{a}$, together with the knowledge that the *prophesied* value $\hat{b}$ (typically of the form ${\uparrow}x$) is equivalent to $\hat{a}$, via the prophecy observation $\langle \hat{b} \overset{\star}{=} \hat{a} \rangle$.[14]

[12] The *view-shift wand* $P \mathrel{\Rrightarrow\!\!\ast} Q$ denotes a resource R satisfying the *view shift* $R \ast P \Rrightarrow Q$. Again we elide the mask parameter $\mathcal{E}$.

[13] To be precise, the wand actually takes a later: $[\alpha]_1 \mathrel{\Rrightarrow\!\!\ast} \triangleright \mathrel{|\!\!\Rrightarrow} [\dagger\alpha]$.

[14] This model is a bit simplified for presentation. Instead of an observational equality $\langle \hat{b} \overset{\star}{=} \hat{a} \rangle$, we actually get $\triangleright \hat{b} :\approx \hat{a}$, where a *prophecy equalizer* $\hat{b} :\approx \hat{a}$ is what becomes an observational equality once we get a token of $\hat{a}$'s dependencies, *i.e.,* $\forall Y \text{ s.t. } \mathrm{dep}(\hat{a}, Y).\ \forall q.\ [Y]_q \mathrel{\Rrightarrow\!\!\ast} \langle \hat{b} \overset{\star}{=} \hat{a} \rangle \ast [Y]_q$.

3.4 Proving Soundness of Type-Spec Rules

With our model in hand, let us now sketch the proofs of a few key type-spec rules for mutable borrowing.

Borrow creation. First, let's tackle creation of a mutable borrow (MUTBOR). Unfolding the semantics of the type-spec judgment, we reach the following Hoare-triple goal (for any input value $\hat{a}$, postcondition $\hat{\Psi}$, and thread t):

$$\{\, \langle \lambda\pi.\ \forall a'.\ (\hat{\Psi}\,\pi)\,[a', (\hat{a}\,\pi, a')]\,\rangle \ast [\![\texttt{Box<T>}]\!](\hat{a}, t, \texttt{a}) \,\}$$
$$\texttt{\&mut a}\ \ \{\, \texttt{b}.\ \exists\, \hat{c}, \hat{b}.\ \langle \lambda\pi.\ (\hat{\Psi}\,\pi)\,[\hat{c}\,\pi, \hat{b}\,\pi]\,\rangle \ast$$
$$[\![\texttt{a}:^{\dagger\alpha}\texttt{Box<T>}]\!](\hat{c}, t) \ast [\![\texttt{\&}\alpha\ \texttt{mut}\ \texttt{T}]\!](\hat{b}, t, \texttt{b}) \,\}$$

The operation `&mut a` just returns the location `a`, so actually $\texttt{b} \triangleq \texttt{a}$. The proof goes as follows.

First, we create a prophecy x and get the value observer $\mathrm{VO}_x(\hat{a})$ and the prophecy controller $\mathrm{PC}_x(\hat{a})$ for x:

MUT-INTRO
$$\mathsf{True} \Rrightarrow \exists x.\ \mathrm{VO}_x(\hat{a}) \ast \mathrm{PC}_x(\hat{a})$$

Pick $\hat{c} \triangleq {\uparrow}x$ and $\hat{b} \triangleq {}^{\star}(\hat{a}, {\uparrow}x)$. From the input observation we immediately get the output observation, simply by instantiating a' into the prophecy's value ${\uparrow}x\ \pi$:

$$\langle \lambda\pi.\ \forall a'.\ (\hat{\Psi}\,\pi)\,[a', (\hat{a}\,\pi, a')]\,\rangle \vdash \langle \lambda\pi.\ (\hat{\Psi}\,\pi)\,[\hat{c}\,\pi, \hat{b}\,\pi]\,\rangle$$

We then unfold the model of $[\![\texttt{Box<T>}]\!](\hat{a}, t, \texttt{a})$ to get:

$$\texttt{a} \mapsto \bar{v} \ast \mathrm{Dealloc}(\texttt{a}, |\texttt{T}|) \ast \triangleright[\![\texttt{T}]\!](\hat{a}, t, \bar{v})$$

And let P be $\exists \hat{a}', \bar{v}.\ \texttt{a} \mapsto \bar{v} \ast [\![\texttt{T}]\!](\hat{a}', t, \bar{v}) \ast \mathrm{PC}_x(\hat{a}')$.

By LFTL-BORROW, constructing and depositing $\triangleright P$, we create a *borrow proposition* $\&^{\alpha} P$ and its *inheritance* $[\dagger\alpha] \Rrightarrow\!\!\ast\ \triangleright P$. Now we have all we need to construct the required resources for the *mutable reference* `b`:

$$\mathrm{VO}_x(\hat{a}) \ast \&^{\alpha} P \vdash [\![\texttt{\&}\alpha\ \texttt{mut}\ \texttt{T}]\!]({}^{\star}(\hat{a}, {\uparrow}x), t, \texttt{b})$$

We use the remaining resources for the *frozen* box:

$$([\dagger\alpha] \Rrightarrow\!\!\ast\ \triangleright P) \ast \mathrm{Dealloc}(\texttt{a}, |\texttt{T}|) \vdash [\![\texttt{a}:^{\dagger\alpha}\texttt{Box<T>}]\!]({\uparrow}x, t)$$

The frozen box is unfolded into $[\dagger\alpha] \Rrightarrow\!\!\ast\ \exists \hat{a}'.\ \langle {\uparrow}x \ {}^{\star}{=}\ \hat{a}' \rangle \ast [\![\texttt{Box<T>}]\!](\hat{a}', t, \texttt{a})$. To prove this, we "execute" the given view-shift wand with $[\dagger\alpha]$ to get $\triangleright P$. Take out the value $\hat{a}'$ out of P. Consuming $\mathrm{PC}_x(\hat{a}')$ inside P, we get the desired $\langle {\uparrow}x \ {}^{\star}{=}\ \hat{a}' \rangle$. Using the remaining parts of P and $\mathrm{Dealloc}(\texttt{a}, |\texttt{T}|)$, we can construct the box.

Write. To write to a mutable reference `*b = c` (MUTREF-WRITE), we get access to the borrow proposition's content by LFTL-BOR-ACC and actually update it, and accordingly renew the observed current state by MUT-UPDATE.

Borrow dropping. Consider dropping of a mutable reference (MUTREF-BYE). First, by LFTL-BOR-ACC, we get temporary access to the borrow proposition's content, which contains the prophecy controller $\mathrm{PC}_x(\hat{a})$. We then use the following ghost update rule to *resolve the prophecy* x, disposing of the value observer in the process (as we should only be able to resolve once!):

MUT-RESOLVE
$$\frac{\mathrm{dep}(\hat{a}, Y)}{\mathrm{VO}_x(\hat{a}) \ast \mathrm{PC}_x(\hat{a}) \ast [Y]_q \Rrightarrow \langle {\uparrow}x \ {}^{\star}{=}\ \hat{a} \rangle \ast \mathrm{PC}_x(\hat{a}) \ast [Y]_q}$$

Now we get an observation $\langle {\uparrow}x \ {}^{\star}{=}\ \hat{a} \rangle$, which makes the prophecy's value ${\uparrow}x$ effectively equal to the current state $\hat{a}$. We can use it for the postcondition to satisfy the rule's spec $\lambda\Psi, [b].\ b.2 = b.1 \to \Psi[]$.

Unfreezing. Unfreezing of objects at a lifetime's end (ENDLFT) can be proved easily. We first get a dead-lifetime token $[\dagger\alpha]$ by consuming $[\alpha]_1$ in the lifetime context. Using it, we "execute" the view-shift wand of each frozen object $[\dagger\alpha] \Rrightarrow\!\!\ast\ \exists \hat{b}.\ \langle \hat{b} \ {}^{\star}{=}\ \hat{a} \rangle \ast [\![\texttt{T}]\!](\hat{b}, t, \texttt{a})$, to get an active object $[\![\texttt{T}]\!](\hat{b}, t, \texttt{a})$. Thanks to the observation $\langle \hat{b} \ {}^{\star}{=}\ \hat{a} \rangle$ for each object, we can prove the rule's spec $\lambda\Psi, \bar{a}.\ \Psi\,\bar{a}$.

Verifying specs for APIs with unsafe code. We can also *semantically* verify all our RustHorn-style specs for *safe Rust APIs with unsafe implementations*.

For an interesting example, let's consider the `iter_mut` method for converting a mutable reference $\texttt{\&}\alpha\ \texttt{mut Vec<T>}$ into a *mutable iterator* $\texttt{IterMut<}\alpha\texttt{,T>}$ (§ 2.3). To verify the method, it suffices to prove the following Hoare triple:

$$\{\, \langle \lambda\pi.\ |{\uparrow}x\,\pi| = |\hat{v}\,\pi| \to (\hat{\Psi}\,\pi)\,[\mathrm{zip}\,(\hat{v}\,\pi)\,({\uparrow}x\,\pi)]\,\rangle \ast$$
$$[\alpha]_q \ast [\![\texttt{\&}\alpha\ \texttt{mut Vec<T>}]\!]({}^{\star}(\hat{v}, {\uparrow}x), t, \texttt{v}) \,\}$$
$$\texttt{iter\_mut(v)}\ \ \{\, \texttt{it}.\ \exists\, \hat{b}.\ \langle \lambda\pi.\ (\hat{\Psi}\,\pi)\,[\hat{b}\,\pi]\,\rangle \ast$$
$$[\alpha]_q \ast [\![\texttt{IterMut<}\alpha\texttt{,T>}]\!](\hat{b}, t, \texttt{it}) \,\}$$

Here we sketch the proof. By `Vec<T>`'s semantics, the vector's value $\hat{v}$ decomposes into ${}^{\star}[\hat{a}_1, \ldots, \hat{a}_n]$. Now we create new prophecies $y_1, \ldots, y_n$ along with a value observer $\mathrm{VO}_{y_i}(\hat{a}_i)$ and a prophecy controller $\mathrm{PC}_{y_i}(\hat{a}_i)$ for each i. We then pick $\hat{b} \triangleq {}^{\star}[{}^{\star}(\hat{a}_1, {\uparrow}y_1), \ldots, {}^{\star}(\hat{a}_n, {\uparrow}y_n)]$, and construct the mutable iterator $[\![\texttt{IterMut<}\alpha\texttt{,T>}]\!](\hat{b}, t, \texttt{it})$, which is equivalent to iterated separating conjunction of the (imaginary) mutable reference $[\![\texttt{\&}\alpha\ \texttt{mut T}]\!]({}^{\star}(\hat{a}_i, {\uparrow}y_i), t, [\ell + i \cdot |\texttt{T}|])$ to the i-th element, over $i \in 1..n$ (where ℓ is the head location). Also, we need the observation $\langle {\uparrow}x \ {}^{\star}{=}\ {}^{\star}[{\uparrow}y_1, \ldots, {\uparrow}y_n] \rangle$. To achieve this, we should *split the borrow proposition* of $\texttt{\&}\alpha\ \texttt{mut Vec<T>}$ to get the borrow propositions for $\texttt{IterMut<}\alpha\texttt{,T>}$, *partially resolving* the old prophecy x. Although we omit details here for space reasons, even *borrow subdivision* like this can be verified using our semantic model.

3.5 A Technical Problem Involving Step-Indexing

We now briefly describe a rather technical problem that we encountered in developing RustHornBelt, which we overcame by developing a more powerful model of the weakest precondition in Iris.

"Step-index hell". The problem pertains to a core feature of Iris's semantic foundation, namely *step-indexing* [5, 2]. The model of Iris propositions is parameterized by a *"step-index"*, which roughly determines the depth of definedness of the proposition (the higher the step-index, the more defined). Step-indexing is reflected into the Iris logic via the so-called *later modality* $\triangleright P$: this intuitively means "P at one lower step-index", which is weaker than P itself. We have already seen the later modality showing up in the model of `Box<T>` (§3.1), as well as in the rules for borrow propositions $\&^{\alpha} P$ such as LftL-bor-acc. It is often used as a kind of *"guard"* to ensure that recursive, higher-order constructions are well-founded. The guard can be "stripped off" at certain moments of "progress" in a proof, notably when reasoning about a physical step of computation, at which point one can strip *one later* $\triangleright$ off any proposition in the proof context.

Although step-indexing is responsible for much of Iris's expressivity (especially "higher-order ghost state" [23]), it can also lead to sticky situations. One such situation arises in RustHornBelt when proving mutref-bye. We want to resolve the borrow's prophecy to value $\hat{a}$ using mut-resolve, for which we need (for soundness, as explained in §3.2) to produce $[Y]_q$, the set of prophecy tokens for all prophecies on which $\hat{a}$ depends. However, in the case that the type of borrowed data is a *recursive type* containing *mutable references*, those prophecy tokens may be buried under *statically unbounded number of laters*. This is a typical example of the kind of "step-index hell" one often encounters when developing semantic models of rich type systems.[15]

Our solution. Our solution to this puzzle, in short, is to strengthen the model of Iris weakest pre $\mathsf{wp}\ e\ \{\Phi\}$ so that reasoning about the n-th step of a program's computation can *strip off n laters*, not just one. This works well thanks to the following observation: it takes at least d program steps to construct an object of *"pointer-nesting depth" d*. When we want to unearth prophecy tokens from nested mutable references inside an object after d steps, the tokens could be buried underneath *at most d laters*, and so our improved weakest pre can strip off all the laters we encounter.

Formally, we use a *time receipt* $⧗\, n$ [34], which persistently records the fact that n program steps have passed. We get $⧗\, 0$ for free, and $⧗\, n$ grows into $⧗(n+1)$ in one step. When we have $⧗\, n$, we can strip off $n+1$ laters in one step:[16]

wp-laters-time

$$\frac{e \text{ is not a value}}{⧗\, n \ast \ {\Rrightarrow\!\triangleright}^{n+1}\ P \ \ast\ \mathsf{wp}\ e\ \{\Phi\}\ \vdash\ \mathsf{wp}\ e\ \{v.\ P \ast \Phi\, v\}}$$

We then add a *pointer-nesting depth* parameter d to RustHornBelt's ownership predicate, which we connect to these time receipts. For example, `Box<T>`'s semantics is updated to $[\![\texttt{Box<T>}]\!](a, d+1, t, [\ell]) \triangleq \exists \bar{v}. \cdots \ast \triangleright [\![\texttt{T}]\!](a, d, t, \bar{v})$, where the box pointer's depth is set to one plus its target's. Each object is then equipped with a *time receipt* corresponding to the depth: $[\![\texttt{a}\colon \texttt{T}]\!] \triangleq \exists d.\ ⧗\, d \ast [\![\texttt{T}]\!](\hat{a}, d, t, \texttt{a})$ (we similarly update $[\![\texttt{a}\colon^{\dagger\alpha}\ \texttt{T}]\!]$). This time receipt gives us sufficient ammunition to strip off as many laters as we might need in order to access tokens buried within the object.

Remaining challenge. Unfortunately, reference-counted pointers (such as those provided by the `Rc` API) make it possible—when used in conjunction with APIs like `RefCell`—to increase pointer-nesting depth by an unbounded quantity in only one execution step (*e.g.,* by concatenating lists). This violates our key observation above (the linking of pointer-nesting depth with execution time). Handling of these APIs thus remains an intriguing technical challenge, which we leave to future work.

4 Evaluation and Case Studies

We evaluated our approach discussed in §3 by fully mechanizing the semantic soundness proof of the type-spec system in the Coq proof assistant, verifying various safe Rust APIs that encapsulate unsafe code (§4.1). We also confirmed that RustHornBelt's specs for Rust APIs (such as `IterMut` and `Cell`) are in fact usable for automated verification of Rust programs, via a RustHorn-style verifier Creusot (§4.2).

4.1 Mechanization in Coq

We built RustHornBelt's Coq development by extending that of RustBelt [21]. It has ~19kLOC of Coq code in total. We were able to reuse the key sub-components, the *lifetime logic* (~2kLOC) and the untyped core calculus (λ_{Rust}) (~3kLOC), as well as the overarching proof structure for verifying the type system. The development took two implementors ~6 months to complete, adding ~7kLOC to the final proofs.

We first modeled basic Rust types and verified type-spec rules for operations on them, extending RustBelt with functional specs. The verified basic types include: box pointer `Box<T>`, shared and mutable references `&α (mut) T`, tuple `(T₁, ..., Tₙ)`, sum $\texttt{T}_1 + \cdots + \texttt{T}_n$[17], array `[T; n]`, integer `int`, boolean `bool`, function `fn(T̄) -> T'`, and recursive types[18].

Then we also modeled advanced Rust types and verified type-spec rules for key API functions encapsulating unsafe code, including:

- Vector `Vec<T>` — `new`, `drop`, `len`, `push`, `pop`, `index(_mut)`, `as_(mut_)slice/iter(_mut)`[19]

15 The reader may wonder if we can use Transfinite Iris [41] instead of Iris to solve this problem. Unfortunately we cannot, because in Transfinite Iris we lose the ability to commute separating conjunction ($\ast$) and later ($\triangleright$), which our model (especially the *lifetime logic*) crucially relies on.

16 Here, instead of just $\triangleright^{n+1} P$ we allow ${\Rrightarrow\!\triangleright}^{n+1} P$, *i.e., P* under $n+1$ laters interleaved between *fancy updates* $\Rrightarrow$ ($\Rrightarrow Q$ is equivalent to True $\Rrightarrow\!\!\ast\ Q$).

17 This amounts to Rust's `enum` type.

18 This supports *non-covariant* recursion, *e.g.,* recursion with self reference under the mutable reference `&α mut`.

19 We equate the two methods, because we used the same model for the shared/mutable slice and iterator.

API	#Funs	LOC Type	LOC Code	LOC Proof
`Vec`	9	147	59	459
`SmallVec`	9	209	75	619
`&`α `(mut) [T]` / `Iter(Mut)`	9	253	38	428
`Cell`	8	102	20	188
`Mutex` / `MutexGuard`	7	258	30	222
`JoinHandle`	2	73	12	52
`MaybeUninit`	5	140	8	108
Misc	3	0	14	85

Figure 1. Coq mechanization of Rust APIs. #Funs: Number of the functions verified. Type: LOC of the semantic model and proof for the type(s). Code: LOC of the λ_{Rust} implementation of the functions. Proof: LOC of the verification proof of the type-spec rules.

- Small-vector `SmallVec<T,`n`>` — `new`, `drop`, `len`, `push`, `pop`, `index(_mut)`, `as_(mut_)slice/iter(_mut)`[19]
- Shared/mutable slice `&`α `(mut) [T]` — `len`, `split_at(_mut)`, `[T;`n`]::as_(mut_)slice`
- Shared/mutable iterator `Iter(Mut)<`α`,T>`[20] — `Iter(Mut)::next`, `Iter(Mut)::next_back`
- Cell `Cell<T>` — `new`, `into_inner`, `from_mut`, `get_mut`, `get`, `set`, `replace`
- Mutex `Mutex<T>` — `new`, `into_inner`, `get_mut`, `lock`
- Mutex guard `MutexGuard<`α`,T>` — `deref(_mut)`, `drop`
- Thread / `JoinHandle<T>` — `spawn`, `join`
- Maybe-uninitialized `MaybeUninit<T>` — `new`, `uninit`, `assume_uninit(_ref, _mut)`
- Misc — `swap`, `panic!`,[21] `assert!`[21]

We implemented each function in our core calculus λ_{Rust}. As in RustBelt, our λ_{Rust} implementation of each function is meant to extract the essence of the real-world Rust implementation, simplifying away uninteresting details. For example, our λ_{Rust} version of `Vec::push` uses a simpler reallocation strategy than the original Rust version.

In Fig. 1, we report the code size of the implementation and proof of a selection of Rust APIs. A function with a large implementation and involving mutable borrows tends to require a larger code size and more significant proof effort. Roughly speaking, modeling a Rust type took ~1 hour, and verifying each function took about 10 minutes–2 hours for us. We still need a large amount of boilerplate code for the proof. Further automation of this part is left to future work.

We also validated our type-spec system by (somewhat manually) verifying small Rust programs, with ~800 LOC of Coq code. The verified programs include what correspond to `inc_vec` and `inc_cell` shown in §2.3, demonstrating the Rust APIs `Vec`, `IterMut` and `Cell`.

[20] For simplicity, for the shared/mutable iterator `Iter(Mut)<`α`,T>`, we used the same model as the shared/mutable slice `&`α `(mut) [T]`.

[21] Abortion is implemented just as a stuck term.

Name	LOC Code	LOC Spec	#VCs	Time/VC
`List-Reversal`	22	10	1	0.09
`All-Zero`	12	6	2	0.05
`Go-IterMut`	14	11	1	0.23
`Even-Cell`	15	6	3	0.03
`Fib-Memo-Cell`	29	53	28	0.06
`Even-Mutex`	38	13	3	0.03
`Knights-Tour`	131	47	10	0.12

Figure 2. Creusot benchmarks. Code: LOC of the program code verified. Spec: LOC of the specs added to the program, including lemmas and definitions. #VCs: Number of the VCs generated by Why3. Time/VC: Average time (seconds) to solve each VC.

4.2 Case Studies in Creusot

We confirmed that our API specs verified by RustHornBelt are *useful* by using them to semi-automatically verify several example client Rust programs in a pre-existing RustHorn-style semi-automated verifier, Creusot [15] (available at https://github.com/xldenis/creusot/). Creusot takes as input a Rust program with spec annotations and then verifies the program by generating VCs (verification conditions) fed to SMT solvers, using Why3 [18] as a backend engine.

Benchmarks. We implemented a Rust library providing specifications for `Vec`, `IterMut`, `Cell`, and `Mutex`. Using the library, we implemented seven verification benchmarks totaling 407 lines of code, specs included. We verified the benchmarks with Creusot, using Why3 configured with a standard automated proof strategy, and using Z3 [13] 4.8.12 or CVC4 [7] 1.8.0 as the backend SMT solver. The benchmarks were executed on Ubuntu 21.04 with an Intel Core i5-10310U CPU and 16 GiB of RAM.

In Fig. 2, we present our benchmarks and evaluation results. `List-Reversal` proves in-place list reversal. `All-Zero` uses a loop to zero each element of a mutably borrowed vector. `Go-IterMut` increments each element of a vector through a mutable iterator. `Even-Cell` and `Even-Mutex` perform invariant-based verification on `Cell` and `Mutex`, respectively. `Knights-Tour` demonstrates scalability on a larger example.

Defunctionalized invariants. In our Coq formalization of RustHornBelt, each cell is represented as an invariant of the *predicate* sort $\lfloor$`T`$\rfloor \to$ Prop. Since a higher-order structure like that can't be directly handled by today's SMT solvers, we *defunctionalize* such invariants for verification in Creusot. We introduce a *trait* (analogous to type class in Haskell) `Inv<T>` for a ghost type `I` that expresses an invariant over `T`:

```
trait Inv<T>: 'static {
  #[predicate] fn inv(&self, a: T) -> bool; }
```

For example, we can make a ghost type Even that expresses the "evenness" invariant on integers (u64 in Rust):

```
struct Even {}
impl Inv<u64> for Even {
  #[predicate] fn inv(&self, a: u64) -> bool {
    a % 2 == 0 } }
```

Then we construct a wrapper type Cell<T,I>, which annotates the standard Cell<T> with the ghost object i: I for the invariant. The methods get and set on Cell<T,I> are given the invariant-based specs (which are trusted):

```
struct Cell<T,I> { c: std::cell::Cell<T>; i: I; }
impl<T: Copy, I: Inv<T>> Cell<T,I> {
  #[trusted] #[ensures(self.i.inv(result))]
  fn get(&self) -> T { self.c.get() }
  #[trusted] #[requires(self.i.inv(a))]
  fn set(&self, a: T) { self.c.set(a) } }
```

For example, we can automatically verify Even-Cell using Cell<u64,Even>.

Fib-Memo-Cell. This benchmark Fib-Memo-Cell verifies a *memoized* recursive function. It uses a vector of cells of type Vec<Cell<Option<u64>,Fib>> for memoization, with the invariant that the i-th Cell in the vector should either store None or Some(fib i), where fib i is the i-th Fibonacci number. We encode this invariant on each Cell using the ghost type Fib. Unlike Even, Fib actually wraps a (ghost) payload of type usize, which represents the index i at which the cell is stored in the vector (and which we ensure matches the actual index of the cell by placing an extra precondition on the function).

Even-Mutex. The benchmark Even-Mutex is a concurrent version of Even-Cell, proving that the *concurrently* shared mutable value is always even. We represent each mutex Mutex as a defunctionalized invariant, just like Cell. Locking a mutex returns a MutexGuard, which can be used to read from and write to the mutex. For concurrency, we use spawn/join to spawn and join several threads. In Rust, spawn takes a *closure* which will be executed. Before we call spawn, we should satisfy the closure's precondition. After we call join on the JoinHandle returned by spawn, we obtain a result that satisfies the closure's postcondition.

5 Related Work

Prophecies. First introduced to prove refinement between state machines [1], *prophecies* have been studied for decades, although they still remain a somewhat exotic technique.

Jung et al. [24] modeled prophecies in Iris (influenced by existing literature [44, 46]), mainly to prove *logical atomicity* of tricky concurrent data structures (though there have been other applications [14]). In their approach, to ensure consistency of prophecy reasoning, prophecy creation and resolution take the form of *ghost* program instructions, which provide a sort of "ground truth" for the prophecy, but also require cumbersome user annotations. Moreover, their prophecies distinguish between the *name* of a prophecy and the value it resolves to (mediated by a kind of "prophecy heap" mapping prophecy names to values). As such, they do not provide a way to resolve a prophecy to a value that mentions the values of other (as yet unresolved) prophecy variables—a feature we require in RustHornBelt to model nested borrows and borrow subdivisions.

For separation logic verification of fine-grained concurrency like Jung et al. aimed at, Turon et al. [42] and Liang and Feng [29] employed a technique of *speculation*. Their approach allows the proof to speculate about multiple possible logical states, combine them through the "speculative choice" connective $P \oplus Q$, and then cull the set of possible states once more information becomes available later in the proof. However, it does not provide an analogue of prophecy variables. In contrast, our prophecy framework provides *persistent* observations $\langle \hat{\phi} \rangle$, which can express knowledge of prophecy variables' values that holds under *all* possible futures.

Proof of RustHorn. Our work does not fully subsume the original proof of RustHorn's translation [32, 33], in that they also proved *completeness* while we prove only soundness. Completeness is in fact lost for APIs with interior mutability like Cell and Mutex in the invariant style, as invariants can't precisely track dynamic state changes in general.

Verifying Rust programs. Prusti [6] analyzes type information from the Rust compiler to synthesize verification conditions in the separation logic Viper [36], suited for automated verification. They directly reconstruct the flow of ownership with the help of lifetime information from the Rust compiler. To model a mutable borrow, they introduced *pledges*, which models a property that is true at the borrow's end. However, their approach struggles with certain advanced use cases of mutable borrows, and they do not support unsafe Rust code.

Electrolysis [43] developed a translation from a Rust program to a purely functional program, which can then be verified manually in a proof assistant. The translation works for a few specific patterns of borrowing, but fails to handle common usages like max_mut in § 2.1. The translation has also not been formally proved sound.

Acknowledgments

We thank Ralf Jung, Naoki Kobayashi, and the anonymous reviewers for their helpful comments. This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289), and in part by JSPS KAKENHI Grant Number JP21J20459 "Theory and application for robust and high-performance systems programming languages".

References

[1] Martín Abadi and Leslie Lamport. 1988. The Existence of Refinement Mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 165–175. https://doi.org/10.1109/LICS.1988.5115

[2] Amal Ahmed. 2004. *Semantics of Types for Mutable State.* Ph. D. Dissertation. Princeton University.

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

[4] Andrew W. Appel. 2001. Foundational Proof-Carrying Code. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, 247–256. https://doi.org/10.1109/LICS.2001.932501

[5] Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

[6] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 147:1–147:30. https://doi.org/10.1145/3360573

[7] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

[8] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *PACMPL* 2, POPL (2018), 5:1–5:29. https://doi.org/10.1145/3158093

[9] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday (Lecture Notes in Computer Science, Vol. 9300)*, Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte (Eds.). Springer, 24–51. https://doi.org/10.1007/978-3-319-23534-9_2

[10] Arthur Charguéraud and François Pottier. 2008. Functional translation of a calculus of capabilities. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 213–224. https://doi.org/10.1145/1411204.1411235

[11] David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*, Bjørn N. Freeman-Benson and Craig Chambers (Eds.). ACM, 48–64. https://doi.org/10.1145/286936.286947

[12] Coq Community. 2021. *The Coq Proof Assistant.* https://coq.inria.fr/

[13] Leonardo Mendonça de Moura and Nikolaj 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 (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[14] Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. 2020. Spy Game: Verifying a Local Generic Solver in Iris. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 33:1–33:28. https://doi.org/10.1145/3371101

[15] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2021. The Creusot Environment for the Deductive Verification of Rust Programs. (2021). https://hal.inria.fr/hal-03526634

[16] Edsger W. Dijkstra. 1976. *A Discipline of Programming.* Prentice-Hall. https://www.worldcat.org/oclc/01958445

[17] Dropbox. 2020. *Rewriting the Heart of Our Sync Engine - Dropbox.* https://dropbox.tech/infrastructure/rewriting-the-heart-of-our-sync-engine

[18] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP (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

[19] Google. 2021. *Rust in the Android platform.* https://security.googleblog.com/2021/04/rust-in-android-platform.html

[20] Ralf Jung. 2020. *Understanding and Evolving the Rust Programming Language.* Ph. D. Dissertation. Saarland University, Saarbrücken, Germany. https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647

[21] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. 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

[22] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust. *Commun. ACM* 64, 4 (2021), 144–152. https://doi.org/10.1145/3418295

[23] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *Journal of Functional Programing* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[24] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The Future is Ours: Prophecy Variables in Separation Logic. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

[25] 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, POPL*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. https://doi.org/10.1145/2676726.2676980

[26] Hari Govind V K, Sharon Shoham, and Arie Gurfinkel. 2022. Solving Constrained Horn Clauses Modulo Algebraic Data Types and Recursive Functions. *Proceedings of the ACM on Programming Languages* POPL (1 2022).

[27] Steve Klabnik, Carol Nichols, and Rust Community. 2018. *The Rust Programming Language.* https://doc.rust-lang.org/book/

[28] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. https://doi.org/10.1145/3009837

[29] Hongjin Liang and Xinyu Feng. 2013. Modular Verification of Linearizability with Non-Fixed Linearization Points. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 459–470. https://doi.org/10.1145/2491956.2462189

[30] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT*, Michael Feldman and S. Tucker Taft (Eds.). ACM, 103–104. https://doi.org/10.1145/2663171.2663188

[31] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. *RustHornBelt: A Semantic Foundation for Functional Verification of Rust Programs with Unsafe Code, Artifact.* https://doi.org/10.5281/zenodo.6501665 Latest version of the Coq mechanization and the Creusot benchmarks available at https://gitlab.mpi-sws.org/iris/lambda-rust/-/tree/masters/rusthornbelt and https://github.com/xldenis/rhb-specs, respectively.

[32] 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 (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 484–514. https://doi.org/10.1007/978-3-030-44914-8_18

[33] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-Based Verification for Rust Programs. *ACM Trans. Program. Lang. Syst.* 43, 4, Article 15 (October 2021), 54 pages. https://doi.org/10.1145/3462205

[34] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

[35] Mozilla. 2021. *Rust language — Mozilla Research.* https://research.mozilla.org/rust/

[36] 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 (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

[37] npm. 2019. *Rust Case Study: Community Makes Rust an Easy Choice for npm.* https://www.rust-lang.org/static/pdfs/Rust-npm-Whitepaper.pdf

[38] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL (Lecture Notes in Computer Science, Vol. 2142)*, Laurent Fribourg (Ed.). Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1

[39] Rust Community. 2021. *Rust Programming Language.* https://www.rust-lang.org/

[40] Rust Community. 2021. *Sponsors — Rust Programming Language.* https://www.rust-lang.org/sponsors

[41] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. 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, 80–95. https://doi.org/10.1145/3453483.3454031

[42] Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical Relations for Fine-Grained Concurrency. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 343–356. https://doi.org/10.1145/2429069.2429111

[43] Sebastian Ullrich. 2016. *Simple Verification of Rust Programs via Functional Purification.* Master's thesis. Karlsruhe Institute of Technology. https://pp.ipd.kit.edu/uploads/publikationen/ullrich16masterarbeit.pdf

[44] Viktor Vafeiadis. 2008. *Modular Fine-Grained Concurrency Verification.* Ph. D. Dissertation. University of Cambridge, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.612221

[45] Philip Wadler. 1990. Linear Types Can Change the World!. In *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods*, Manfred Broy (Ed.). North-Holland, 561.

[46] Zipeng Zhang, Xinyu Feng, Ming Fu, Zhong Shao, and Yong Li. 2012. A Structural Approach to Prophecy Variables. In *Theory and Applications of Models of Computation - 9th Annual Conference, TAMC (Lecture Notes in Computer Science, Vol. 7287)*, Manindra Agrawal, S. Barry Cooper, and Angsheng Li (Eds.). Springer, 61–71. https://doi.org/10.1007/978-3-642-29952-0_12