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

YUSUKE MATSUSHITA, The University of Tokyo, Japan
XAVIER DENIS, Université Paris-Saclay, CNRS, ENS Paris-Saclay, INRIA, Laboratoire Méthodes Formelles, France
JACQUES-HENRI JOURDAN, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles, France
DEREK DREYER, MPI-SWS, Germany

Rust is a systems programming language offering both low-level memory operations and high-level safety guarantees. To achieve this, Rust employs a strong ownership-based type system that prohibits mutation of aliased state, but it also allows that core typing discipline to be relaxed via safe APIs implemented internally with unsafe features. Prior work by Matsushita et al. developed a promising technique for functional verification of Rust code called RustHorn: they leverage the strong invariants of Rust types to give purely functional specifications to imperative Rust code, which in turn reduce the verification problem to one that is more amenable to off-the-shelf automated techniques. Their key idea for making this possible was to use a novel form of *prophecies* to model mutations performed on mutable references in a functional way. Unfortunately, however, their approach was limited to the *safe* subset of Rust, and it has remained unclear how to soundly extend it to support APIs that encapsulate *unsafe* code.

In this paper, we present **RustHornBelt**, the first machine-checked proof of soundness for RustHorn-style verification which supports giving purely functional specifications to safe APIs implemented with unsafe code. As the name suggests, RustHornBelt employs the same approach of *semantic typing* used in Jung et al.'s work on RustBelt, but extends RustBelt's model of types with predicate transformers that describe the behavior of Rust features in purely functional terms. Developing RustHornBelt required us to extend Iris with a more flexible treatment of step indexing based on time receipts, as well as a novel account of RustHorn-style prophecies in separation logic.

1 INTRODUCTION

The Rust programming language [Rust Community 2021a; Matsakis and Klock 2014; Klabnik et al. 2018] has shown that high-level safety is not fundamentally at odds with low-level control. Drawing from decades of academic research, Rust employs an "ownership" type system, in which the aliasing of pointers is tracked statically and direct mutation of aliased state is prohibited. This strong aliasing discipline serves to guarantee memory safety and data race freedom even in the presence of lower-level mechanisms like interior pointers and manual deallocation. Unsurprisingly, the arrival of a language with the same low-level control as C/C++, but with stronger safety guarantees than many mainstream high-level languages, has been met with a great deal of interest by programming language theoreticians and industrial software developers alike [Jung et al. 2021, 2018a; Mozilla 2021; Dropbox 2020; npm 2019; Rust Community 2021b].

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. Traditional approaches to software systems verification require constructing an explicit representation of the machine state, but this leads to a state explosion problem in C-like languages with pointers and shared-memory concurrency. Consequently, much work has focused on making verification more tractable by using *separation logic* [O'Hearn et al. 2001] (and related methods like *implicit dynamic frames* [Smans et al. 2009]) to support modular reasoning in such languages.

More recently, several projects have explored how the verification problem for Rust programs can be simplified even further by exploiting the safety guarantees already provided by the Rust type system. Prusti [Astrauskas et al. 2019] uses information from the Rust compiler to automatically

synthesize separation-logic-style proofs of memory safety for Rust programs in the Viper framework [Müller et al. 2016]; the user can then verify functional correctness on top by instrumenting the source code with additional annotations. RustHorn [Matsushita et al. 2020] goes even further by eliminating separation logic from the picture entirely: they show that, due to the strong aliasing guarantees enforced by Rust types, the behavior of (well-typed) Rust programs can in fact be expressed using first-order logical formulas, without requiring an explicit representation of memory or separation-logic analogues like points-to assertions. This encoding is compatible with existing logical solvers, and their paper demonstrates this by using a constrained Horn clauses (CHC) solver to prove correctness of Rust programs involving complex data structures *automatically*.

Unfortunately, both Prusti and RustHorn share a common limitation: they bake in the assumption that the program being verified is written entirely in the *safe* fragment of Rust. In reality, however, many Rust programs make use of *unsafe* features of Rust, such as raw pointer accesses and unchecked type casts. These unsafe features provide essential "escape hatches" for those cases where the prohibition against mutation of aliased state is simply too restrictive. For example, the widely-used `Vec` library, which provides a growable array type, manages its underlying buffer manually and relies on unchecked memory accesses for efficiency. 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`. Given that mutating shared state flies in the face of Rust's core aliasing discipline, it should come as no surprise that `Cell`'s implementation fundamentally depends on unsafe code.

In this paper, we show how a verification system in the style of RustHorn can be soundly extended to verify clients of APIs, such as `Vec` and `Cell`, which rely internally on unsafe features. Specifically, we present a new system—called **RustHornBelt**—which provides a *semantic* foundation for RustHorn based on RustBelt [Jung et al. 2018a], and we verify the soundness of this new foundation mechanically in the Coq proof assistant. Before we present our contributions in more detail, though, let us first explain a bit more about the prior work on RustHorn and why its existing foundation does not easily scale to account for unsafe code.

1.1 RustHorn: Prophetic Verification of Rust

One of the greatest challenges in automatically verifying 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, through which it can be accessed—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 tackles this challenge by heavily 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 it can be mutable, but it cannot be both aliased and mutable at the same time. The AXM principle is enforced, in turn, 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 *borrows*. A borrow represents temporary ownership of an object: when a borrow of an object is created, it is associated with a *lifetime*, a period of time during which the program can safely access the object through the borrow. Once the lifetime is over, ownership is restored to the original owner of the object. Corresponding to the two mutually exclusive options of AXM, there are two varieties of borrows: *mutable* (aka *unique*) and *immutable* (aka *shared*). Mutable borrows can be used to mutate the referent object, but they must

be unique: if a mutable borrow of object `x` exists, no other borrows of `x` may be active at the same time. In contrast, immutable borrows may be freely shared throughout the program, but cannot be used to directly mutate the object.

The key insight of RustHorn [Matsushita et al. 2020] is that, by eliminating mutation of aliased state, Rust's AXM discipline makes it possible to give *purely functional specifications* to stateful code. For the cases of shared borrows and fully owned objects, this is not too surprising: the former are immutable, so there is no need to model state change, and the latter can be modeled in a standard state-passing style à la traditional linear types [Wadler 1990; Charguéraud and Pottier 2008; Bernardy et al. 2018]. The interesting case is *mutable borrows*, for here the question is how to communicate the result of state changes through the mutable borrow back to the original owner (lender) of the object.

To answer this question, RustHorn models each mutable borrow (in functional specs) not as a memory location but rather as a pair of the *current* and *final* values of the borrowed object (where "final value" means the value the object will have when the lifetime of the borrow ends). Of course we do not know the final value in advance, so this second component of the pair acts as a kind of *prophecy*. The prophecy lets the original owner know the updated value of the borrowed object at the end of the lifetime.

To better understand this model, which in our experience is initially somewhat mindbending to those who have not seen it before, consider the following example:

```
fn rand_proj<'a, T>(mp: &'a mut (T, T)) -> &'a mut T {
    let (ma, mb) = mp;
    if rand () { ma } else { mb }
}
```

The function `rand_proj` takes a mutable borrow of a pair of `T`'s (with lifetime `'a`), and randomly returns a mutable borrow to one component of the pair. Under the RustHorn model, this means that the argument `mp` is modeled as a pair (p, p'), where p and p' are themselves both pairs of `T`'s.[1] Here, p represents the "current" pair of `T`'s stored at `mp` at the point `rand_proj(mp)` is called, and p' represents the "final" pair of `T`'s stored at `mp` at the point the lifetime `'a` ends.

Clearly, since `rand_proj` either returns a reference to the left- or right-hand component of its argument, the result of `rand_proj(mp)` will be either $(p.0, p'.0)$ or $(p.1, p'.1)$, respectively. What's more interesting is that, depending on which result is returned, we learn something about the *other* component of the pair. Without loss of generality, suppose the left-hand component is chosen, in which case the result of `rand_proj(mp)` is $(p.0, p'.0)$. In that case, the caller of `rand_proj(mp)` loses access to the right-hand component of `mp`—and thus there is no way for the right-hand component to be mutated—until the lifetime `'a` ends. Consequently, we know that $p.1$ (the "current" value of that second component) must be equal to $p'.1$ (the "final" value of that component in lifetime `'a`). Phrased in the lingo of prophecy variables, this means we can *resolve* the "prophecy" $p'.1$ to $p.1$, although formally prophecy resolution here simply means that we can deduce an equality between two values passed into the function.

The reasoning described above is encapsulated formally by the following *purely functional* specification for `rand_proj`. It is expressed as a *predicate transformer*, taking as input the post-condition Ψ and the RustHorn model of `mp`, and returning the weakest precondition for executing `rand_proj(mp)`:

$$\lambda\Psi, [(p, p')].\ \big(p'.1 = p.1 \to \Psi\,(p.0, p'.0)\big)\ \wedge\ \big(p'.0 = p.0 \to \Psi\,(p.1, p'.1)\big)$$

[1] Actually, p and p' have the type $\lfloor \mathtt{T} \rfloor \times \lfloor \mathtt{T} \rfloor$, where $\lfloor \mathtt{T} \rfloor$ is RustHorn's representation type for `T`. See §2 for details.

1.2 Proving Soundness of RustHorn-Style Verification in the Presence of Unsafe Code

To establish the soundness of RustHorn, Matsushita et al. [2020, 2021] gave a *syntactic* proof of their approach, which supports a large portion 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, the syntactic approach breaks down.

This limitation of syntactic proofs of soundness was previously articulated by Jung et al. [2018a] in their work on RustBelt. In that work, Jung et al. developed a *semantic* model for a significant subset of Rust, called λ_{Rust}. At the most basic level, the RustBelt semantic model serves to establish the soundness of the λ_{Rust} type system, as soundness of type systems is traditionally understood: well-typed programs don't "go wrong" (*i.e.,* have undefined behavior). On top of that, RustBelt explains what it means for libraries that use unsafe code to be considered semantically safe: given the interface of a library, the model computes a verification condition that the implementation of the library must satisfy in order to ensure that (well-typed) clients of the library never encounter undefined behavior. These verification conditions in turn are expressed in the Iris framework for higher-order concurrent separation logic (implemented in the Coq proof assistant), which provides an expressive logical language in which to reason modularly about ownership and state [Jung et al. 2015; Krebbers et al. 2017; Jung et al. 2018c; Jung 2020]. Jung et al. demonstrated the utility of Iris and the RustBelt model by using them to verify the safety of several representative Rust libraries that make use of unsafe features (including `Cell`).

Contributions. In this paper, we show how to bring RustHorn and RustBelt together to develop **RustHornBelt**, the first formulation of a RustHorn-style system that is capable of verifying Rust programs with unsafe code. Like the original RustHorn, RustHornBelt exploits Rust's AXM discipline to reduce the work of verifying imperative pointer programs to a much simpler purely functional verification problem, to which a variety of automated techniques are readily applicable. Unlike the original RustHorn, RustHornBelt employs a *semantic* model of types inspired by RustBelt, which allows one to prove purely functional specifications also for libraries built atop unsafe code, such as `Vec` and `Cell`. Although the correctness of these library specifications must be proven by hand in Iris, they can (once proven) be safely used in subsequent RustHorn-style automatic verification of client code. RustHornBelt is fully mechanized in the Coq proof assistant: as such, RustHornBelt does not directly provide good automation for proving programs, but it gives a formal grounds to tools such as RustHorn for supporting libraries using unsafe code such as `Vec` and `Cell`.

In RustHornBelt, the λ_{Rust} typing rule for each instruction I is extended with a RustHorn-style predicate transformer (as seen at the end of §1.1), which specifies the functional behavior of I. For program fragments written in safe Rust, these extended typing rules thus automatically construct a RustHorn-style representation of their behavior. For a library that employs *unsafe* code, we must supply a RustHorn-style predicate transformer specification of its behavior by hand, but after doing so, the RustHornBelt model of the (extended) λ_{Rust} typing judgment tells us how to translate that specification into Iris, so that we can verify that the library's implementation satisfies it. We have verified RustHorn-style specifications (in Iris/Coq) for key Rust libraries that use unsafe code, including `Vec` (a growable array), `&mut [T]` (a mutable slice), `slice::IterMut<T>` (a mutable iterator), `MaybeUninit` (a possibly uninitialized object), `mem::swap` (swap through mutable references), `Cell` (a shared mutable cell), `thread::spawn/join` (thread spawning and joining), and `Mutex` (a mutex synchronization wrapper for sharing data across threads). The supplementary material of this paper contains the Coq development of RustHornBelt.

The rest of the paper is structured as follows:

- In §2, we present a high-level overview of RustHornBelt's verification system.

- In §3, we present a simplified version of the semantic model of Rust types.
- In §4, we present a new approach to modeling RustHorn-style prophecies in Iris. Unlike the existing mechanism for prophecy variables in Iris [Jung et al. 2020; de Vilhena et al. 2020], our RustHorn-style prophecies enable one to *partially resolve* prophecies to a value which depends on further prophecies, a feature that RustHornBelt critically relies on.
- In §5, we present an improvement of Iris's program logic to improve the flexibility of step indexing, which we need to model *nested mutable borrows* in Rust.
- In §6, building on the technical contributions of §4 and §5, we present our full semantic model of Rust types and illustrate our soundness proof of RustHorn-style translations.
- Finally, in §7, we compare with related work and discuss future work.

2 OVERVIEW: RUSTHORNBELT AT A GLANCE

In this section, we present the high-level interface of our *verification system*, formally proved sound by RustHornBelt, showing how we can generate *RustHorn-style verification conditions* of Rust programs, including those using Rust libraries with unsafe code, *e.g.,* `Vec` and `Cell`. RustHornBelt's verification system is developed as an *extension of RustBelt's type system*: it takes the *typing rules* of RustBelt and adds functional specifications to each, so that typing a program *also* generates verification conditions. This conceptually mimics behaviors of RustHorn, which generates verification conditions for Rust programs in a type-directed manner. Being a proof system, our system can also discharge verification conditions in the course, so it can serve as a proof-of-concept, self-contained Rust verifier on its own. After some setup (§2.1), we first present an example *without* mutable borrows (§2.2), then illustrate how we can verify programs *with* mutable borrows using prophecies (§2.3), and finally show how we can support libraries with *unsafe* code (§2.4).

2.1 Setup

Before diving into RustHornBelt's verification, let's establish a formal setup.

First, we use the core calculus of RustBelt, λ_{Rust}. It is designed to model the mid-level intermediate representation (MIR) language used by the Rust compiler, which drastically simplifies the syntax of Rust programs. The calculus λ_{Rust} is semi-structured and uses continuation-passing style; programs are composed from linear sequences of instructions combined with (mutually recursive) continuations for control flow. Since λ_{Rust} is not the focus of our paper, we will present our examples in Rust-style syntax and explain concepts as they occur. A complete description of λ_{Rust} can be found in the RustBelt paper [Jung et al. 2018a,b].

In a RustHorn-style translation, each Rust object is given a *representation value*, in a form of *refinement* with peculiar handling of *pointers*. Remarkably, a *mutable reference* `&'a mut T` (written $\&^{\kappa}_{\mathbf{mut}}\,\tau$ in λ_{Rust}) is represented as a *pair* of the current and final values of its target object, erasing address information. On the other hand, a *owned pointer* `Box<T>` (**box** τ) and a *shared reference* `&'a T` ($\&^{\kappa}_{\mathbf{shr}}\,\tau$) are simply represented by the value of their target object.

The domain of representation values for objects of type τ called the the *representation type* $\lfloor\tau\rfloor$, which is defined as follows:

$$\lfloor\mathbf{int}\rfloor \triangleq \mathbb{Z} \qquad \lfloor\mathbf{bool}\rfloor \triangleq \mathbb{B} \qquad \lfloor\tau \times \tau'\rfloor \triangleq \lfloor\tau\rfloor \times \lfloor\tau'\rfloor \qquad \lfloor\tau + \tau'\rfloor \triangleq \lfloor\tau\rfloor + \lfloor\tau'\rfloor$$
$$\lfloor\mathbf{box}\,\tau\rfloor \triangleq \lfloor\tau\rfloor \qquad \lfloor\&^{\kappa}_{\mathbf{shr}}\,\tau\rfloor \triangleq \lfloor\tau\rfloor \qquad \lfloor\&^{\kappa}_{\mathbf{mut}}\,\tau\rfloor \triangleq \lfloor\tau\rfloor \times \lfloor\tau\rfloor$$

To specify a Rust program, we extend RustBelt's typing judgments with a *predicate transformer* (aka *weakest-precondition transformer*), which describes the functional behavior of the program fragment. Representing a program using predicate transformers is a common approach to functional

verification, especially in automated verifiers like F* [Swamy et al. 2016]. Predicate transformers subsume various logical representations of a program, including constrained Horn clauses (CHCs) [Bjørner et al. 2015] used by RustHorn. In RustHornBelt, we chose predicate transformers for the target logical representation so as to elegantly formulate RustHorn-style verification. CHC generation is syntactically a bit heavy, and RustHorn-style verification is not limited to CHC-based verification as already discussed by Matsushita et al. [2020].

In λ_{Rust}, we have two forms of typing judgments. First, let's focus on the basic one, the typing judgment for an *instruction* I:

$$\mathbf{E}; \mathbf{L} \mid \mathbf{T}_0 \vdash I \dashv x. \mathbf{T}_1 \rightsquigarrow \Phi$$

It says that the instruction I transforms the input type context $\mathbf{T}_0$ to the output type context $\mathbf{T}_1$, generating the predicate transformer Φ. An *instruction* I performs a simple operation like addition $x + y$. A *type context* $\mathbf{T}$ is an ordered list of items of the form either $p \lhd \tau$ or $p \lhd^{\dagger\kappa} \tau$. In this paper, we can understand a *path* p simply as a program variable in Rust. In a type context, we mostly use the former $p \lhd \tau$, which simply means that we have an object of type τ with the name p. The latter $p \lhd^{\dagger\kappa} \tau$ is a more *Rust-y* concept: it means that an object of type τ assigned to p is *borrowed* (or blocked) until the *lifetime* κ ends. Later, §2.3 has an example (Fig. 6) that shows how this works. The *external* and *local lifetime contexts* $\mathbf{E}$ and $\mathbf{L}$ manage lifetime information; later we use the judgment $\mathbf{E}; \mathbf{L} \vdash \kappa$ alive, saying that the lifetime κ is alive (*i.e.,* has not ended). The new item added in RustHornBelt is the *predicate transformer* Φ, the specification of the program fragment. For this judgment, it has the following type:

$$\Phi : (\lfloor \mathbf{T}_1 \rfloor \to \mathsf{Prop}) \to \lfloor \mathbf{T}_0 \rfloor \to \mathsf{Prop}$$

It transforms a postcondition over the output $\mathbf{T}_1$ into a precondition over the input $\mathbf{T}_0$. Here, a type context's representation type $\lfloor \mathbf{T} \rfloor$ for $\mathbf{T} = \overline{p \lhd^? \tau}$ (each $p \lhd^? \tau$ is $p \lhd \tau$ or $p \lhd^{\dagger\kappa} \tau$) is defined as the ordered, heterogeneous list type $[\overline{\lfloor \tau \rfloor}]$, consisting of the representation type of each element type τ in the type context.

Now let's focus on the other one, the typing judgment for a *function body* F:

$$\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F \rightsquigarrow \Phi$$

As the name indicates, a function body is used for the body or control flow of a function and ends by returning from the function. This is why this judgment does not have the output type context. For this judgment, the predicate transformer Φ is typed as follows, where τ is the return type of the function that F belongs to:

$$\Phi : (\lfloor \tau \rfloor \to \mathsf{Prop}) \to \lfloor \mathbf{T} \rfloor \to \mathsf{Prop}$$

In λ_{Rust}, we can introduce new *continuations* to form complex control flows. The *continuation context* $\mathbf{K}$ holds all continuations accessible when we enter F. By default, we only have one continuation, the *return continuation*, which only inputs the return value. In RustHornBelt, each continuation is represented as a predicate transformer, which transforms the function's postcondition into a precondition for the continuation. The return continuation is represented as $\lambda\Psi, [a]. \Psi\, a$, asserting that the function's postcondition Ψ has to hold when applied to the return value a.

2.2 Exclusive Ownership

We start with a simple function which nevertheless showcases several important aspects of RustHornBelt's verification system. The function `inc_pair` (Fig. 1) takes as parameters a pair of integers and a Boolean and returns the value of one of the components incremented by five.

```
fn inc_pair(
  p: (u32, u32), b: bool
) -> u32 {
  if b {
    return p.0 + 5
  } else {
    return p.1 + 5
  }
}
```

```
fn inc_pair(
  p: (u32, u32), b: bool
) -> u32 {
  if b {
    let (i, j) = p; let c5 = 5;
    let r = i + c5; return r;
  } else {
    ...
  }
}
```

Fig. 1. Incrementing an owned pair, side-by-side with 'explicit' Rust version.

Since Rust has a lot of syntactic sugar, we present a desugared version to the right of Fig. 1, which we will use to discuss verification. While this is not λ_{Rust}, it is sufficiently close to convey the *ideas* involved in verifying λ_{Rust} programs.

As an aid for explanation, we've annotated Fig. 1 with predicate transformers in Fig. 2. One can read Fig. 2 as a series of Hoare triples: assuming the postcondition of the function is Ψ, each pair of curly braces contains the precondition of each program point.

Before we can verify `inc_pair` we must give it a specification:

$$\lambda\Psi, [p, b].\ \left(b \to \Psi\,(p.0 + 5)\right) \wedge \left(\neg b \to \Psi\,(p.1 + 5)\right)$$

This takes a postcondition Ψ, and the inputs `p` and `b` as arguments, and gives the corresponding precondition. It states that we either return the incremented first or second projections depending on the value of `b`.

To verify `inc_pair` in RustHornBelt's system, we *type-check* it, which requires us to simultaneously prove the specification. Thus, we start by applying the rule F-IF (Fig. 3). Here, the predicate transformer matches *syntactically*, and we can apply the rule without issue. Often, though, we use *logical consequence* to transform a specification into the correct form to apply a rule. Entering the `true` branch the specification becomes:

$$\lambda\Psi, [p, b].\ \Psi\,(p.0 + 5)$$

For the entire function to be correct, this branch must type-check and have the specification above. In Rust, ownership is *sub-structural*, we take control of only *part* of a variable. So, when we wish to read the first component of `p`, we first *split* `p` so we can individually reason on the ownership of both halves of the pair.

Once we've obtained the first projection `i`, we store the constant 5 in a separate variable `c5`. To do this, we use the F-LET rule to create a new variable and then store a number inside it using S-NUM. On the next line we perform an increment using S-NAT-OP and store the result in `r`. Finally, we **`return`**, this value by *jumping* to the *return continuation*. This requires us to show that we satisfy the continuation's specification, which for returning is just the postcondition Ψ applied to the return value r, which is exactly what we have. The **`else`** branch is checked in a symmetric manner, and just like that we've verified our first Rust program!

2.3 Manipulating References

Exclusive ownership allows for a simple form of mutation: we can only mutate objects that we own. This is too limited for many uses: there is no way to recover control of an object if we pass it

```
1  fn inc_pair(
2    p: (u32, u32), b: bool
3  ) -> u32 {
4    {(b → Ψ (p.0 + 5)) ∧ (¬b → Ψ (p.1 + 5))}
5    if b {
6      {Ψ (p.0 + 5)}
7      let (i, j) = p;
8      {Ψ (i + 5)}
9      let c5 = 5;
10     {Ψ (i + c5)}
11     let r = i + c5;
12     {Ψ r}
13     return r;
14   } else {
15     {Ψ (p.1 + 5)}
16     ...
17   }
18 }
```

Fig. 2. Explicit version of Figure 1 with specifications.

F-LET
$$\frac{\mathbf{E};\mathbf{L} \mid \mathbf{T}_0 \vdash I \dashv x.\, \mathbf{T}_1 \rightsquigarrow \Phi \qquad \mathbf{E};\mathbf{L} \mid \mathbf{K};\, \mathbf{T}_1, \mathbf{T}' \vdash F \rightsquigarrow \Phi'}{\mathbf{E};\mathbf{L} \mid \mathbf{K};\, \mathbf{T}_0, \mathbf{T}' \vdash \mathbf{let}\ x = I\ \mathbf{in}\ F \rightsquigarrow \lambda\Psi, (\overline{a} +\!\!+ \overline{c}).\ \Phi\ (\lambda \overline{b}.\ \Phi'\ \Psi\ (\overline{b} +\!\!+ \overline{c}))\ \overline{a}}$$

F-IF
$$\frac{\mathbf{E};\mathbf{L} \mid \mathbf{K};\mathbf{T} \vdash F_0 \rightsquigarrow \Phi \qquad \mathbf{E};\mathbf{L} \mid \mathbf{K};\mathbf{T} \vdash F_1 \rightsquigarrow \Phi'}{\mathbf{E};\mathbf{L} \mid \mathbf{K};\, p \lhd \mathbf{bool},\, \mathbf{T} \vdash \mathbf{if}\ p\ \mathbf{then}\ F_0\ \mathbf{else}\ F_1 \rightsquigarrow \lambda\Psi, (b :: \overline{a}).\ (b \to \Phi\ \Psi\ \overline{a}) \wedge (\neg b \to \Phi'\ \Psi\ \overline{a})}$$

S-NAT-OP
$$\mathbf{E};\mathbf{L} \mid p_0 \lhd \mathbf{int},\, p_1 \lhd \mathbf{int} \vdash p_0\ \{+,-\}\ p_1 \dashv x.\ x \lhd \mathbf{int} \rightsquigarrow \lambda\Psi, [n_0, n_1].\ \Psi\ [n_0\ \{+,-\}\ n_1]$$

S-NUM
$$\mathbf{E};\mathbf{L} \mid \varnothing \vdash n \dashv x.\ x \lhd \mathbf{int} \rightsquigarrow \lambda\Psi, [].\ \Psi\ [n]$$

D-ASSGN-BOX
$$\mathbf{E};\mathbf{L} \mid p \lhd \mathbf{box}\,\tau,\, p' \lhd \tau' \vdash p \coloneqq p' \dashv p \lhd \mathbf{box}\,\tau' \rightsquigarrow \lambda\Psi, [a, b].\ \Psi\ [b]$$

D-ASSGN-BOR-MUT-COPY
$$\frac{\mathbf{E};\mathbf{L} \vdash \kappa\ \mathsf{alive} \qquad \tau\ \mathsf{copy}}{\mathbf{E};\mathbf{L} \mid p \lhd \&^{\kappa}_{\mathbf{mut}}\,\tau,\, p' \lhd \tau \vdash p \coloneqq p' \dashv p \lhd \&^{\kappa}_{\mathbf{mut}}\,\tau,\, p' \lhd \tau \rightsquigarrow \lambda\Psi, [(a, a'), b].\ \Psi\ [(b, a'), b]}$$

D-DEREF-BOR-MUT-COPY
$$\frac{\mathbf{E};\mathbf{L} \vdash \kappa\ \mathsf{alive} \qquad \tau\ \mathsf{copy}}{\mathbf{E};\mathbf{L} \mid p \lhd \&^{\kappa}_{\mathbf{mut}}\,\tau \vdash {}^{*}p \dashv x.\ x \lhd \tau,\, p \lhd \&^{\kappa}_{\mathbf{mut}}\,\tau \rightsquigarrow \lambda\Psi, [(a, a')].\ \Psi\ [a, (a, a')]}$$

D-RESOLVE-BOR-MUT
$$\mathbf{E};\mathbf{L} \mid p \lhd \&^{\kappa}_{\mathbf{mut}}\,\tau \vdash \mathbf{drop}(p) \dashv \varnothing \rightsquigarrow \lambda\Psi, [(a, a')].\ a' = a \to \Psi\ []$$

D-SPLIT-BOR-MUT-PAIR
$$\frac{\mathbf{E};\mathbf{L} \vdash \kappa\ \mathsf{alive} \qquad \mathbf{E};\mathbf{L} \mid \mathbf{K};\, p_0 \lhd \&^{\kappa}_{\mathbf{mut}}\,\tau_0,\, p_1 \lhd \&^{\kappa}_{\mathbf{mut}}\,\tau_1,\, \mathbf{T} \vdash F \rightsquigarrow \Phi}{\begin{array}{l}\mathbf{E};\mathbf{L} \mid \mathbf{K};\, p \lhd \&^{\kappa}_{\mathbf{mut}}\,(\tau_0 \times \tau_1),\, \mathbf{T} \vdash \mathbf{let}\ (p_0, p_1) = p\ \mathbf{in}\ F \rightsquigarrow \\ \qquad \lambda\Psi, ((p, p') :: \overline{a}).\ \Phi\ \Psi\ ((p.0, p'.0) :: (p.1, p'.1) :: \overline{a})\end{array}}$$

D-BORROW
$$\frac{\mathbf{E};\mathbf{L} \vdash \kappa\ \mathsf{alive} \qquad \mathbf{E};\mathbf{L} \mid \mathbf{K};\, p \lhd \&^{\kappa}_{\mathbf{mut}}\,\tau,\, p' \lhd {}^{\dagger\kappa}\,\mathbf{box}\,\tau,\, \mathbf{T} \vdash F \rightsquigarrow \Phi}{\mathbf{E};\mathbf{L} \mid \mathbf{K};\, p' \lhd \mathbf{box}\,\tau,\, \mathbf{T} \vdash \mathbf{let}\ p = \&\mathbf{mut}\ {}^{*}p'\ \mathbf{in}\ F \rightsquigarrow \lambda\Psi, (a :: \overline{c}).\ \forall a'.\ \Phi\ \Psi\ ((a, a') :: a' :: \overline{c})}$$

Fig. 3. Selected (derived) rules of RustHornBelt's verification system.

```rust
fn inc_pair_mut (
  mp: &mut (u32, u32), b: bool
) -> &mut u32 {
  if b {
    *mp.0 += 5;
    return mp.0;
  } else {
    *mp.1 += 5;
    return mp.1;
  }
}
```

```rust
fn inc_pair_mut (
  mp: &mut (u32, u32), b: bool
) -> &mut u32 {
  if b {
    let (mi, mj) = mp;  drop(mj);
    let i1 = *mi;  let c5 = 5;
    let i2 = i1 + c5;  *mi = i2;
    return mi;
  } else {
    ...
  }
```

Fig. 4. Mutating a pair by reference, along with 'explicit' expanded version.

to a function. When someone calls `inc_pair` and gives up ownership of the pair, she has no way to recover the *entire* pair and observe the mutation. Using mutable references we can extend the expressiveness of Rust dramatically, by allowing owners to lend ownership for a bounded lifetime. By modifying `inc_pair` to accept a mutable reference as parameter, we enable the owner to make multiple successive changes to the same pair. Similarly to the previous example, we break down the operations which happen implicitly in Rust in Fig. 4.

In RustHornBelt, as discussed in §1.1, we integrate mutable references using the same approach as RustHorn: each reference `mx` is represented by a *pair of the current and final values* (x, x'). As we mutate the reference, the *current* value is updated, until the moment it is released or *dropped* in Rust parlance, where we *resolve* the final value to the current value. The specification of `inc_pair_mut` must explain what happens to the current and final values, we should know that one component will have been incremented and the other will stay the same. We can describe this with the following transformer:

$$\big(b \to p'.1 = p.1 \to \Psi\,(p.0 + 5,\ p'.0)\big) \ \wedge\ \big(\neg b \to p.0 = p'.0 \to \Psi\,(p.1 + 5,\ p'.1)\big)$$

If we consider the first conjunct $p'.1 = p.1 \to \Psi\,(p.0 + 5,\ p'.0)$, it expresses both aspects. When `b` is true we return the first component of `mp`, and we also *resolve* the second component to its final value. We take the second component of both `mp`'s current value (p) and its final value (p') which gives us the hypothesis $p'.1 = p.1$. The conclusion then returns the other half of the pair $\Psi\,(p.0 + 5,\ p'.0)$ with the current value incremented by 5.

Verifying `inc_pair_mut`. Verification of Fig. 5 starts just like before, using F-IF. We focus on typing and proving the first branch. The first thing we do is split the reference `mp` using D-SPLIT-BOR-MUT-PAIR. This rule has more subtle behavior than the owned equivalent. By definition a reference does not *own* the object it points to, we use a prophecy to communicate the mutation of the reference to the original owner. To project a field from a reference, we first *subdivide* it, in the process creating fresh references for each field. We link these references with the owner by also dividing the prophecy among them. When we divide a reference to a pair into two references respectively to the left and right element, we have the specification:

$$\lambda\big((p, p') :: \overline{a}\big).\ \Psi\,\big((p.0, p'.0) :: (p.1, p'.1) :: \overline{a}\big)$$

To create each output reference, we use a field from the current value p and from the prophecy p', this ensures that mutations are still tracked, and linked.

```
fn inc_pair_mut (
  mp: &mut (u32, u32), b: bool
) -> &mut u32 {
  { (b → p'.1 = p.1 → Ψ (p.0 + 5, p'.0)) ∧
    (¬b → p'.0 = p.0 → Ψ (p.1 + 5, p'.1)) }
  if b {
    {p'.1 = p.1 → Ψ (p.0 + 5, p'.0)}
    let (mi, mj) = mp;
    {j' = j → Ψ (i + 5, i')}
    drop(mj);
    {Ψ (i + 5, i')}
    let i1 = *mi;
    {Ψ (i1 + 5, i')}
    let c5 = 5;
    {Ψ (i1 + c5, i')}
    let i2 = i1 + c5;
    {Ψ (i2, i')}
    *mi = i2;
    {Ψ (i, i')}
    return mi;
  } else {
    {p'.0 = p.0 → Ψ (p.1 + 5, p'.1)}
    ...
  }
}
```

Fig. 5. Explicit Rust code for `inc_pair_mut`.

```
{True}
{∀p'. p'.1 = 6 → p'.0 = 5 + 5 → p' = (10, 6)}
let mut p = (5, 6);
{∀p'. p'.1 = p.1 → p'.0 = p.0 + 5 → p' = (10, 6)}
let mp = &mut p;
{p'.1 = p.1 → p'.0 = p.0 + 5 → p' = (10, 6)}
let mk = inc_pair_mut(mp, true);
{k' = k → p' = (10, 6)}
drop(mk);
{p' = (10, 6)}
```

Fig. 6. Client of `inc_pair_mut`.

The second, important part of RustHornBelt's handling of references is *resolution*. When we are finished with a reference, we must dispose of it and resolve its prophecy, informing the owner of the final value that was obtained. This event naturally coincides with the notion of *deallocation* or *dropping* in Rust parlance. Since we will only use the first projection of `mp`, we drop the second projection `mj`. By applying D-RESOLVE-BOR-MUT, we release the loaned ownership `mj`, and resolve the associated prophecy, thus establishing $j' = j$. (Note the naming convention; we introduce local variables j, j' for the mutable reference `mj`, and similarly i, i' for `mi`.) The rule we present here is a special case of *structural resolution*, which releases all references contained within an object, potentially recursively.

Then, we dereference `mi`, using the derived rule D-DEREF-BOR-MUT-COPY. In Rust, dereferencing a pointer takes ownership of the contents. Since a reference must be able to return the ownership it borrowed, we can only dereference types tagged `Copy` such as integers or Booleans, which can be trivially duplicated.

After performing arithmetic, the result is written back into `mi`, using D-ASSGN-BOR-MUT-COPY, which updates the *current* value of the reference in the specification. Finally, we hand off the reference by returning.

Using `inc_pair_mut`. Let's consider in Fig. 6 an example where we create a mutable borrow to call `inc_pair_mut`. We wish to show as a postcondition that `p` contains `(10, 6)`.

Intuitively, here is what happens: when we create a borrow for `mp`, we prophecise its final value p' and assign it to p in the program logic. When we call `inc_pair_mut`, we partially resolve p' (we learn that $p'.1 = 6$), and get back a reference `mk`, whose prophecy is equal to $p'.0$. When we drop `mk`, we finish resolving p' and learn that $p'.0 = 10$. At the end of `mp`'s lifetime, the value of `p` is p', which we've determined to be $(10, 6)$.

Now, we can walk through the actual, formal proof. Since it is guided by a weakest-precondition calculus, we will follow its backwards direction. The last instruction drops the reference `mk` obtained from the call to `inc_pair_mut`. This tells us that k and k' are equal, hence this becomes a hypothesis in the precondition. To reason about the function call, we use the specification we proved in Fig. 5.

The novel part of this code occurs now, when we create the borrow for `mp`: we use the rule D-BORROW, which may seem daunting. It introduces a prophecy p', which is *universally quantified* in the precondition. This prophecy is used to construct the value (p, p') of the reference `mp` and uses the prophecy p' as the new value for the borrowed location `p`. Note, we don't actually predict a concrete value for p', the precondition must hold for all possible values.

Finally, we can substitute `p` for its value when it is first assigned and we get as a precondition the following tautology:

$$\forall p'.\ \ p'.1 = 6 \ \to\ p'.0 = 5 + 5 \ \to\ p' = (10, 6)$$

2.4 Modeling Unsafe Code

The previous examples show how RustHornBelt's system subsumes RustHorn: we can verify purely safe code, including uses of mutable references. But our work goes further and also allows to address the challenges of unsafe code. Let's see how we can model types and functions, which have safe APIs but unsafe, tricky implementations.

The `Vec` type. Growable arrays—also called *vectors*—are a key data structure which finds use everywhere. In Rust the `Vec` type has a rich API which allows random mutable access using the `index_mut` function, or stack operations using `push` and `pop`. Internally, `Vec` makes use of unsafe code to allocate a large buffer of memory and to move data when a resizing occurs. These operations occur using *raw machine pointers*, which cannot be modeled using the RustHorn approach.

`Vec` illustrates a key challenge of Rust verification. Functions like `push` and `pop` have simple specifications, but since they are implemented using *unsafe* code, RustHorn can't verify or even *express* them. RustHornBelt allows us to write these specifications, by abstracting over the internals of `Vec` with a representation: the list type. We show how representations are declared in §3 and §6.

With `Vec` represented as a list, it is simple to specify operations like `push`, `pop` or `index_mut`. In Fig. 7 the functions `push` and `pop` become their equivalent operations on lists. Since the functions take mutable references as input the modification occurs by updating the prophetic value v'. Slightly more complex, `index_mut` returns a mutable reference directly to the indexed element. In the specification, this is solved by creating a prophecy a' for the returned element and using it to update the vector's prophecy. Eventually when the returned reference is discarded it will be resolved to the concrete value for this element.

Proving these specifications might require powerful tools, but *using* them shouldn't. RustHornBelt's approach means that from safe clients, we preserve the RustHorn-style first-order reasoning, regardless of implementation.

`IterMut`. The complete encapsulation of unsafe code in a safe interface opens the door to further *composition* and *abstraction*. This is exemplified by Rust's *iterators*. Internally, they make heavy use of unsafe code but provide a flexible and composable interface to safely traverse and

$\{\lambda\Psi, [(v, v'), a].\ v' = v +\!\!+ [a] \to \Psi\,()\}$
`fn push<T>(v: &mut Vec<T>, a: T)`

$\left\{\lambda\Psi, [(v, v')].\ \begin{array}{l}(v' = v = [] \to \Psi\ \mathrm{None}) \wedge \\ (\forall a.\ v = v' +\!\!+ [a] \to \Psi\,(\mathrm{Some}\ a))\end{array}\right\}$
`fn pop<T>(v: &mut Vec<T>) -> Option<T>`

$\{\lambda\Psi, [(v, v'), i].\ \ i < |v| \wedge (\forall a'.\ v' = v\{i := a'\} \to \Psi\,(v[i], a'))\}$
`fn index_mut<T>(v: &mut Vec<T>, i: usize) -> &mut T`

Fig. 7. Specification of the `Vec` API.

$\{\lambda\Psi, [(v, v')].\ |v'| = |v| \to \Psi\,(\mathrm{zip}\ v\ v')\}$
`fn iter_mut<'a, T>(v: &'a mut Vec<T>)`
`  -> IterMut<'a, T>`

$\left\{\lambda\Psi, [(w, w')].\ \begin{array}{l}(w' = w = [] \to \Psi\ \mathrm{None}) \wedge \\ (\forall p.\ w = p :: w' \to \Psi\,(\mathrm{Some}\ p))\end{array}\right\}$
`fn next<'a, T>(it: &mut IterMut<'a, T>)`
`  -> Option<&'a mut T>`

$\{\lambda\Psi, [(v, v')].\ v' = \mathrm{map}\ (+5)\ v \to \Psi\,()\}$
```
fn inc_vec(v: &mut Vec<u32>) {
  for a in v.iter_mut() {
    *a += 5;
  }
}
```

Fig. 8. API of mutable iterators and example of client.

mutate collections like `Vec`. Iterators also show how Rust goes further than languages like Java, by leveraging ownership to eliminate common source of errors like *iterator invalidation*.

The function `iter_mut` converts a `Vec` into a mutable iterator on its contents. Shown in Fig. 8, the returned iterator is parameterized by the reference's lifetime `'a`. This ensures that so long as the iterator is active, the ownership of the iterated vector cannot be restored, ensuring no client can *invalidate* the iterator. The spec RustHornBelt gives to `iter_mut` is a bit tricky. We observe that the length of the prophecy $|v'|$ is the same as the current length of the vector $|v|$, We *split* the *List* $\lfloor \mathtt{T} \rfloor$-valued prophecy v' into $\lfloor \mathtt{T} \rfloor$-valued prophecies on each element of the vector. So we get the postcondition $|v'| = |v|$. In the representation of the iterator, each prophecy is coupled with the current value of the element. So finally we get a *list of pairs* $\mathrm{zip}\ v\ v'$ (*e.g.,* $\mathrm{zip}\ [1, 2, 3]\ [a', b', c'] = [(1, a'), (2, b'), (3, c')]$).

To efficiently traverse the vector, `IterMut` keeps a raw pointer into the memory backing the vector, and steps through it manually using unsafe code. Clients can do this safely by calling the `next` method, which returns a mutable reference to the next element, if there is one. Just like with vectors, specifying `next` first requires *representing* iterators logically, for which we use a *list of mutable references*. Then the `next` function can be specified as optionally returning the head of this list (see Fig. 8).

Proving that `next` satisfies its specification is challenging, but can be done using RustHornBelt (see §6). Once this is done, we can easily integrate it into safe clients. Consider Fig. 8, which includes the `inc_vec` function. We wish to prove that after calling it, all the elements of the input vector have been incremented by 5. The implementation relies on a mutable iterator through a for-loop. In Rust for-loops are sugar for calling `next` on an iterator until it returns `None`. Again, we can prove this example by successively applying RustHornBelt's verification rules. We never observe the unsafe code inside `next`, only the specification it satisfies. By manually choosing the correct loop invariant during typing, the proof can be done without particular challenge.

$\{\lambda\Psi, [\Phi].\ \forall a.\ \Phi\, a \to \Psi\, a\}$
`fn get<T>(c: &Cell<T>) -> T`

$\{\lambda\Psi, [\Phi, a].\ \Phi\, a \wedge \Psi\,()\}$
`fn set<T>(c: &Cell<T>, a: T)`

$\forall\Phi.$ $\{\lambda\Psi, [a].\ \Phi\, a \wedge \Psi\,\Phi\}$
`fn new<T>(a: T) -> Cell<T>`

Fig. 9. API of `Cell`.

The `Cell` type. When implementing cyclic data structures like mutable graphs, the AXM discipline of Rust is too restrictive. In those situations, Rust developers can use the `Cell` type, which provides *interior mutability*, mutation through a shared reference. It may seem that `Cell` undoes all the guarantees of Rust, but not so: it comes with a carefully restricted API, and cannot be shared across threads.

To represent `Cell` in RustHornBelt, we draw from separation logic the idea of *invariants*: a shared resource is represented by a *proposition*. A value of type `Cell<T>` is represented by a predicate $\lfloor \mathtt{T} \rfloor \to \mathrm{Prop}$, which must be preserved by mutations. Then, when a cell is accessed with `get`, we know the invariant must hold for the returned value, and when we `set` a new value, we must establish that the invariant also holds for it. Readers familiar with concurrent separation logic will recognize this technique, as it is often used to model mutexes, another form of *interior mutability*.

The representation of cells in RustHornBelt means that our specifications are not always first-order like they are in RustHorn. However, they remain in a standard logic, and further illustrate a key idea of RustHornBelt's translation: verification should only pay for the features used. If a program makes no use of cells, their higher-order representation won't appear in specifications or proofs. Even if a program *internally* uses them if they are *encapsulated* then external clients will not be affected.

Verifying clients of `Cell`. In a certain sense, by proving `Cell`'s API, we extend the language with new typing judgments we can use to manipulate cells. As we'll see later in §6.4 this is more than informal intuition. For the moment, let's see how we can use this new API to prove a client program. Consider the following function `inc_cell`, which increments the value stored in a cell by `i`. Recall, we represent a cell as an *invariant* Φ. As a precondition we require that the cell's invariant holds inductively.

$\{\lambda\Psi, [\Phi, i].\ (\forall n.\ \Phi\, n \to \Phi\,(n + i)) \wedge \Psi\,()\}$
```
fn inc_cell(c: &Cell<u32>, i: u32) {
  let t = c.get();  c.set(t + i);
}
```

Like the previous examples, we use logical consequence to change the specification:

$$(\forall n.\ \Phi\, n \to \Phi\,(n + i)) \wedge \Psi\,() \ \vdash\ \forall t.\ \Phi\, t \to (\Phi\,(t + i) \wedge \Psi\,())$$

We can now apply the rule for `get`, obtaining a value `t`, which satisfies $\Phi\, t$. Using S-NAT-OP we add `i` to it. Now, using `set`, we store the value `t + i` into the cell. This requires establishing $\Phi\,(t + i)$, which we get from our hypothesis combined with the fact $\Phi\, t$ we obtained from `get`. With that, we have verified our simple function on cells.

The API we use for cells is fairly flexible: the invariants we associate with cells can depend on *dynamic information* (*i.e.,* runtime values). For example, we could call `inc_cell` with the invariant $\lambda n.\ n \bmod i = 0$, where i is the representation of another program variable. As a side note, for technical reasons, the dynamic information available here is restricted to non-prophetic values.

3 FIRST STEP: A SIMPLIFIED MODEL OF TYPES

RustHornBelt is formulated as an extension of RustBelt and as such uses the same technique of *semantic typing* in its proof of soundness. By working in the higher-order separation logic *Iris* [Jung et al. 2015], we can reuse rich libraries to model various aspects of Rust.

The full semantic interpretation we give to Rust types can be challenging for readers to understand at a glance, because it uses both our formulation of prophecies (§4) and time receipts (§5). So as a first step to understanding our proof, we present a *simplified* model of RustHornBelt without prophecies or time receipts. Then in §6 we will present the complete model of types. In this section we first present a simplified semantic domain of types (§3.1) and then showcase interpretations of various Rust types, including `Cell` (§3.2).

Our formulation inherits many technical tools from RustBelt, including the core calculus λ_{Rust} and the *lifetime logic*. We explain these notions on the fly. Interested readers can get further information in the RustBelt paper [Jung et al. 2018a] and Jung's Ph.D. thesis (2020).

3.1 Simplified Semantic Domain of Types

In RustBelt, each Rust type τ is associated with its *ownership predicate* $[\![\tau]\!].\text{own} : \mathit{TId} \times \mathit{List}\,\mathit{Val} \to \mathit{iProp}$, which models *what it means to own an object of type* τ, including the *ownership of resources* granted to the object. The predicate returns an *Iris* proposition $P \in \mathit{iProp}$, which can be viewed as a 'proposition that owns resources'. The predicate is parametrized over the list of *low-level values* $\overline{v} \in \mathit{List}\,\mathit{Val}$, describing the sequence of values stored in the memory for the object, and the thread identifier $\mathfrak{t} \in \mathit{TId}$, used when modeling concurrency.

In Rust, a type τ can behave differently when shared. For example, the cell type `Cell<T>` is equivalent to `T` when fully owned, but it becomes a *shared mutable state* when put under a shared reference. In order to model such behaviors, RustBelt associates with each type a *sharing predicate* $[\![\tau]\!].\text{shr} : \mathit{Lft} \times \mathit{TId} \times \mathit{Loc} \to \mathit{iProp}$. The predicate $[\![\tau]\!].\text{shr}(\kappa, \mathfrak{t}, \ell)$ is meant to model $[\![\&^{\kappa}_{\mathbf{shr}}\,\tau]\!].\text{own}(\mathfrak{t}, [\ell])$, the ownership predicate of a shared reference to τ. Here, the *location* (aka address) $\ell \in \mathit{Loc}$ represents where the object is stored.

In our simplified model, we add a parameter for the *representation value* $a \in \lfloor\tau\rfloor$ to both the ownership and sharing predicates, hence the following signatures:

$$[\![\tau]\!].\text{own} : \lfloor\tau\rfloor \times \mathit{TId} \times \mathit{List}\,\mathit{Val} \to \mathit{iProp} \qquad [\![\tau]\!].\text{shr} : \lfloor\tau\rfloor \times \mathit{Lft} \times \mathit{TId} \times \mathit{Loc} \to \mathit{iProp}$$

We also require the ownership and sharing predicates to satisfy some conditions, like in RustBelt. The ownership predicate $[\![\tau]\!].\text{own}$ should only accept value lists $\overline{v}$ whose length is equal to the type's *size* $\text{size}(\tau)$. The sharing predicate $[\![\tau]\!].\text{shr}$ should be persistent for any argument and monotone over the lifetime parameter. (An Iris proposition P being *persistent* means it can be freely duplicated.) Since shared references `&T` are copyable, the sharing predicate should be persistent.

Following RustBelt, we also impose the following rule to convert a reference to an object satisfying τ's ownership predicate to a resource satisfying τ's sharing predicate.

$$\textsc{ty-share}\quad \&^{\kappa}_{\text{full}}\big(\exists\overline{v}.\ \ell \mapsto \overline{v} * [\![\tau]\!].\text{own}(a, \mathfrak{t}, \overline{v})\big) * [\kappa]_q \ \Rrightarrow\ [\![\tau]\!].\text{shr}(a, \kappa, \mathfrak{t}, \ell) * [\kappa]_q$$

In this rule, we transform a full borrow $\&^{\kappa}_{\text{full}}(\cdots)$ of a heap fragment $\ell \mapsto \cdots$ storing an object $[\![\tau]\!].\text{own}(\cdots)$ into a resource satisfying τ's sharing predicate.

3.1.1 Background on Iris and the Lifetime Logic. Let's take a second to cover some background on Iris and the lifetime logic.

A *view shift* $P \Rrightarrow Q$ is a connective in Iris, meaning that we can turn a resource satisfying P into a resource satisfying Q updating the internal states.[2]

The *lifetime logic* is an Iris library designed by Jung et al. [2018a] for modeling lifetime-based resource control in Iris. We have used some notions from the lifetime logic. A *lifetime token* $[\kappa]_q$ ensures that κ is still alive with fraction q. A *full borrow* $\&^\kappa_{\mathrm{full}} P$ (a notion from the lifetime logic) is a (non-persistent) resource from which we can temporarily take out a resource satisfying P while the lifetime κ is active. The full borrow was introduced to model the mutable reference type in RustBelt. The following are basic proof rules on the full borrow:

LftL-borrow
$$\triangleright P \;\Rrightarrow\; \&^\kappa_{\mathrm{full}} P \;*\; \left([\dagger\kappa] \Rrightarrow\!\!\!\!\ast \; \triangleright P \right)$$

LftL-bor-acc
$$\&^\kappa_{\mathrm{full}} P \;*\; [\kappa]_q \;\Rrightarrow\; \triangleright P \;*\; \left(\triangleright P \Rrightarrow\!\!\!\!\ast \; \&^\kappa_{\mathrm{full}} P * [\kappa]_q \right)$$

Here, a *view shift wand* $P \Rrightarrow\!\!\!\!\ast \; Q$ is an Iris connective that combines a magic wand with a view shift; it turns P into Q while also updating internal states. A full borrow $\&^\kappa_{\mathrm{full}} P$ can be created from $\triangleright P$ (LftL-borrow), along with a view shift wand that returns $\triangleright P$ when we provide a *dead-lifetime token* $[\dagger\kappa]$, which observes that κ has ended. From a full borrow $\&^\kappa_{\mathrm{full}} P$, we can *temporarily* take out $\triangleright P$ (LftL-bor-acc), with the help of a lifetime token $[\kappa]_q$. Using the closing wand $\triangleright P \Rrightarrow\!\!\!\!\ast \; \&^\kappa_{\mathrm{full}} P * [\kappa]_q$ we can retrieve $\&^\kappa_{\mathrm{full}} P$ and $[\kappa]_q$ when we return $\triangleright P$. Importantly, these rules use $\triangleright P$, P guarded under the *later modality* $\triangleright$, instead of just P.

The lifetime logic uses the later modality like this to achieve *impredicativity*. This is a typical trick in the *step-indexed* logic Iris. In Iris, the notions of resources and propositions are circular, and to avoid unsoundness Iris requires any circular reference to be *guarded* under the later modality $\triangleright$. Here in the lifetime logic, because the full borrow $\&^\kappa_{\mathrm{full}} P$ uses an Iris proposition P to characterize a resource, each reference to P is guarded under the later modality. Note that this motivates our new technique on step indexing, explained later in §5.

3.2 Simplified Interpretations of Types

Using the domain of types we just defined, we can explore the interpretations of Rust types.

Booleans. The Boolean type `bool`, written **bool** in λ_{Rust}, is one of the simplest types in Rust. Its ownership predicate is defined as follows:

$$[\![\mathbf{bool}]\!].\mathrm{own}(b, \_, \overline{\mathfrak{v}}) \;\triangleq\; \overline{\mathfrak{v}} = [b]$$

It is just the equality $\overline{\mathfrak{v}} = [b]$, which links the low-level value with the representation value. The sharing predicate $[\![\mathbf{bool}]\!].\mathrm{shr}$ is derived from $[\![\mathbf{bool}]\!].\mathrm{own}$, just as in RustBelt. The integer type **int** is also interpreted in a similar way.

Products. In Rust, we can make new types out of existing types in many ways. A simple example of that is the product type $\tau_0 \times \tau_1$ (written `(T0, T1)` in Rust). The ownership predicate of the product type can be derived from τ_0 and τ_1's ownership predicates as follows:

$$[\![\tau_0 \times \tau_1]\!].\mathrm{own}(p, \mathfrak{t}, \overline{\mathfrak{v}}) \;\triangleq\; \exists\, \overline{\mathfrak{v}_0}, \overline{\mathfrak{v}_1} \text{ s.t. } \overline{\mathfrak{v}} = \overline{\mathfrak{v}_0} +\!\!+ \overline{\mathfrak{v}_1}.\; [\![\tau_0]\!].\mathrm{own}(p.0, \mathfrak{t}, \overline{\mathfrak{v}_0}) \;*\; [\![\tau_1]\!].\mathrm{own}(p.1, \mathfrak{t}, \overline{\mathfrak{v}_1})$$

The representation value is set to the pair of those of the two components, and the list of low-level values is set to the concatenation of those of the components. The sharing predicate of $\tau_0 \times \tau_1$ is similarly derived from τ_0 and τ_1's sharing predicates.

[2] In Iris, a view shift $\Rrightarrow$ actually has the *mask* parameter $\mathcal{N}$, which is related to Iris invariants. In this paper, however, we omit masks since they are not relevant to the high-level exposition.

Owned pointers. Now, let's consider how to model an owned pointer **box** τ (written `Box<T>` in Rust). In RustHorn-style verification, an owned pointer is represented simply as the value of its target object. As a result, the ownership predicate is defined as follows:

$$[\![\mathbf{box}\,\tau]\!].\mathrm{own}(a, \mathtt{t}, \overline{\mathfrak{v}}) \triangleq \exists\, \ell \text{ s.t. } \overline{\mathfrak{v}} = [\ell].\ \exists \overline{\mathfrak{w}}.\ \ell \mapsto \overline{\mathfrak{w}} * \triangleright [\![\tau]\!].\mathrm{own}(a, \mathtt{t}, \overline{\mathfrak{w}}) * \mathrm{Dealloc}(\ell, \mathrm{size}(\tau))$$

As a pointer, the low-level value should be a location ℓ. We own the points-to token $\ell \mapsto \overline{\mathfrak{w}}$ and the target object $[\![\tau]\!].\mathrm{own}(\cdots)$ as well as the right to deallocate $\mathrm{Dealloc}(\cdots)$. The representation value a of an owned pointer is the same as that of the target object. The sharing predicate for owned pointers is tricky in this model (due to *delayed sharing* [Jung 2020, §12.2]), so we omit it.

Shared references. The shared reference type $\&^{\kappa}_{\mathbf{shr}}\,\tau$ (written `&'a T` in Rust) is modeled using the sharing predicate of the target type τ. First, the ownership predicate is defined as follows:

$$[\![\&^{\kappa}_{\mathbf{shr}}\,\tau]\!].\mathrm{own}(a, \mathtt{t}, \overline{\mathfrak{v}}) \triangleq \exists\, \ell \text{ s.t. } \overline{\mathfrak{v}} = [\ell].\ [\![\tau]\!].\mathrm{shr}(a, \kappa, \mathtt{t}, \ell)$$

We require the low-level value to be a location ℓ, which is passed to $[\![\tau]\!].\mathrm{shr}$ as an argument. Note that the points-to tokens are included in the sharing predicate $[\![\tau]\!].\mathrm{shr}$.

Cells. In RustHornBelt, we can model various libraries by extending the approach of RustBelt. Among them, the type `Cell<T>` is an interesting library that features *interior mutability*.

At a low level the type `Cell<T>` looks exactly like `T`, but allows *interior mutability*: its contents can be *updated through a shared reference* `&Cell<T>` using methods like `set` (shown in Fig. 9). Another tricky detail of `Cell<T>` is that `Cell`'s methods are *not thread-safe*. These aspects make it challenging to model this type.

RustBelt managed this by designing a special Iris connective in the lifetime logic, the *non-atomic persistent borrow* $\&^{\kappa/\mathtt{t}}_{\mathbf{na}}\,P$. It is similar to the full borrow $\&^{\kappa}_{\mathbf{full}}\,P$ introduced earlier, but is persistent unlike the full borrow. It satisfies the following proof rules:

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

LftL-na-acc
$$\&^{\kappa/\mathtt{t}}_{\mathbf{na}}\,P * [\kappa]_q * [\mathrm{Na} : \mathtt{t}] \Rrightarrow \triangleright P * \left(\triangleright P \Rrightarrow\!\!\!\!\ast\ [\kappa]_q * [\mathrm{Na} : \mathtt{t}]\right)$$

A non-atomic persistent borrow can be obtained from a full borrow. It allows temporarily taking out a resource satisfying P like a full borrow, but it requires not only the lifetime token $[\kappa]_q$ but also a thread-local token $[\mathrm{Na} : \mathtt{t}]$, which is the key to persistence of the borrow. Using this borrow, RustBelt defined the sharing predicate for cells $[\![\mathbf{cell}\langle\tau\rangle]\!].\mathrm{shr}(\kappa, \mathtt{t}, \ell)$ a: $\&^{\kappa/\mathtt{t}}_{\mathbf{na}}\big(\exists \overline{\mathfrak{v}}.\ \ell \mapsto \overline{\mathfrak{v}} * [\![\tau]\!].\mathrm{own}(\mathtt{t}, \overline{\mathfrak{v}})\big)$. The ownership predicate $[\![\mathbf{cell}\langle\tau\rangle]\!].\mathrm{own}(\mathtt{t}, \overline{\mathfrak{v}})$ is just the same as $[\![\tau]\!].\mathrm{own}(\mathtt{t}, \overline{\mathfrak{v}})$.

In RustHornBelt, as shown in Fig. 9, we can represent $\mathbf{cell}\langle\tau\rangle$ as a pure *invariant predicate* $\Phi : \lfloor\tau\rfloor \to \mathrm{Prop}$ over the value of the cell's content. We can define the sharing predicate of the cell type as follows, taking the invariant predicate into account:

$$[\![\mathbf{cell}\langle\tau\rangle]\!].\mathrm{shr}(\Phi, \kappa, \mathtt{t}, \ell) \triangleq \&^{\kappa/\mathtt{t}}_{\mathbf{na}}\big(\exists\, \overline{\mathfrak{v}},\ a \text{ s.t. } \Phi\, a.\ \ell \mapsto \overline{\mathfrak{v}} * [\![\tau]\!].\mathrm{own}(a, \mathtt{t}, \overline{\mathfrak{v}})\big)$$

We use a non-atomic persistent borrow like RustBelt, but the content of the borrow is richer: it is now existentially quantified over the representation value a and asserts that it satisfies the invariant Φ, which justifies the cell's representation value. The ownership predicate $[\![\mathbf{cell}\langle\tau\rangle]\!].\mathrm{own}(\Phi, \mathtt{t}, \overline{\mathfrak{v}})$ is defined as: $\exists a$ s.t. $\Phi\, a$. $[\![\mathbf{cell}\langle\tau\rangle]\!].\mathrm{own}(a, \mathtt{t}, \overline{\mathfrak{v}})$. We can prove ty-share over $\mathbf{cell}\langle\tau\rangle$ using LftL-bor-na. Methods like `Cell::get` and `Cell::set` can be proved using LftL-na-acc.

4 ALL FUTURES IN HAND: A NEW APPROACH TO PROPHECIES

As explained in the introduction, a key highlight of RustHorn-style verification is to model each mutable borrow in Rust using a pair of the *current* value of the borrowed object and the *final* value the object will have when the borrow ends. Following RustHorn, we refer to the latter as a *prophecy*.

The moment a borrow is created, the prophecy reveals to the lender *knowledge of the future*, giving it the final value of the borrowed object. That prophecy can later be *resolved* when the borrow is dropped, at which point we know that the current and final value of the object are equal. In effect, prophecies afford a form of "time travel", and it is well known from science fiction that naively engaging in time travel can lead to so-called *causal loops*—where a future event can affect the present, which can in turn affect the future event—and that to preserve consistency of the universe, such causal loops are to be avoided at all costs.

Existing approaches to prophecies in separation logic [Vafeiadis 2008; Zhang et al. 2012; Jung et al. 2020] ensure absence of causal loops by requiring that prophecy resolution must be performed by physical program instructions. In particular, the user must change the program, annotating it with actual prophecy resolution instructions in appropriate places.[3] This approach ensures that the values to which prophecies are resolved cannot depend circularly on any reasoning within the program's proof: rather, they are given "ground truth" from the trace of the program's execution.

There are several reasons, however, why such an approach is ill-suited to our purposes. First of all, it would make for a rather heavyweight method to require explicit prophecy resolution instructions to be inserted into Rust code everywhere mutable borrows are dropped. Second, RustHorn prophecies are typically resolved to *logical* representations of types, which are not runtime objects of the program.

Third, and most importantly, in RustHorn we frequently resolve a prophecy to a value which itself depends on another prophecy—what we call *partial prophecy resolution*. This occurs for example when resolving a borrow of a borrow (a *nested borrow*), in which case the resolution value is the current value of the inner borrow, which itself contains a prophecy. It also occurs when we *subdivide a mutable borrow* into potentially several other new borrows, in which case the prophecy of the subdivided borrow is resolved to a value that depends on the prophecies of the new borrows.

To see a concrete example of this, consider the following instance of subdivision.

```
/* enum Option<T> { None, Some(T) } */
let mo: &mut Option<i32> = ...;
if let Some(mi) = mo { /* `mi: &mut i32` */ *mi = ...; }
```

First we take a mutable reference `mo: &mut Option<i32>` to an integer option. Then we perform pattern matching and *subdivide* `mo`. We check if the value `*mo` has the tag `Some` and, if so, we take out a mutable reference to the inner integer `mi: &mut i32` and update the integer through it. Let's see how this is modeled in RustHorn. First, `mo` is modeled as $(\mathrm{Some}\, i, o')$ for some integer i and a prophetic option value o', given that `*mo` has the tag `Some`. When we create the inner mutable reference `mi`, it is modeled as (i, i'), where i' is a newly created integer prophecy for this subdivided borrow. Next, we *observe* that o' is equal to $\mathrm{Some}\, i'$, which *partially resolves* o' to a value that still depends on another new prophecy i'.

This additional flexibility of partial prophecy resolution goes beyond what existing separation-logic approaches to prophecies support. Hence, in RustHornBelt, we take a different tack, and develop a novel separation-logic account of prophetic reasoning that is a better fit for RustHorn-style verification and supports partial prophecy resolution. Unlike the existing approaches, we do not require the program to be annotated with any physical prophecy resolution instructions. Rather, prophecies and prophecy reasoning remain completely *logical*.

To achieve this, the basic idea of our prophecies is that we only permit the user to make assertions about prophecies that hold *for all possible future prophecy resolutions simultaneously*. Furthermore,

[3] Separately, an erasure theorem is proven to establish that the prophecy resolution instructions do not change the observable behavior of the original, unannotated program.

$$\mathsf{persistent}(\langle \pi.\, \hat{\phi}\, \pi \rangle)$$

$$\textsc{Proph-intro}\quad \frac{A \text{ is non-empty}}{\mathsf{True} \;\Rrightarrow\; \exists \xi \text{ s.t. } \xi.\mathsf{type} = A.\; [\xi]_1}$$

$$\textsc{Proph-frac}\quad [\xi]_{q+q'} \;\dashv\vdash\; [\xi]_q * [\xi]_{q'}$$

$$\textsc{Proph-resolve}\quad \frac{\hat{a} \colon \mathit{ProphAsn} \to \xi.\mathsf{type} \qquad \mathsf{dep}(\hat{a}, Y)}{[\xi]_1 * [Y]_q \;\Rrightarrow\; \langle \pi.\, \pi\, \xi = \hat{a}\, \pi \rangle * [Y]_q}$$

$$\textsc{Proph-obs-merge}\quad \frac{\langle \pi.\, \hat{\phi}\, \pi \rangle \qquad \langle \pi.\, \hat{\psi}\, \pi \rangle}{\langle \pi.\, \hat{\phi}\, \pi \wedge \hat{\psi}\, \pi \rangle}$$

$$\textsc{Proph-obs-impl}\quad \frac{\langle \pi.\, \hat{\phi}\, \pi \rangle \qquad \forall \pi.\, \hat{\phi}\, \pi \to \hat{\psi}\, \pi}{\langle \pi.\, \hat{\psi}\, \pi \rangle}$$

$$\textsc{Proph-obs-true}\quad \frac{\forall \pi.\, \hat{\phi}\, \pi}{\langle \pi.\, \hat{\phi}\, \pi \rangle}$$

$$\textsc{Proph-obs-sat}\quad \frac{\langle \pi.\, \hat{\phi}\, \pi \rangle}{\mathsf{True} \;\Rrightarrow\; \exists \pi_0.\, \hat{\phi}\, \pi_0}$$

Fig. 10. Selected proof rules of the prophecy library.

we avoid causal loops by requiring that when a prophecy is (partially) resolved to a value v, that value v can only mention *unresolved* prophecies. As we will see in §6, we can ensure that the latter condition always holds by building it into the semantic model of Rust types.

4.1 Our Approach

We realize our approach to prophecies in Iris as a stand-alone, language-independent library, making it easy to integrate into other developments. The Iris proof rules for our new RustHorn-style prophecies are shown in Fig. 10.

As a first step, we fix *the infinite set of prophecies* (or prophecy variables) $\mathit{ProphVar} \ni \xi, \eta, \cdots$. We can create a fresh prophecy ξ by choosing an unused prophecy from this set. Each prophecy ξ is associated with the *type* $\xi.\mathsf{type}$, which is a non-empty type that specifies the range of values that can be assigned to the prophecy. We can then construct a *prophecy assignment* $\pi \in \mathit{ProphAsn}$, as a map from each prophecy $\xi \in \mathit{ProphVar}$ to any value in $\xi.\mathsf{type}$. Conceptually, a prophecy assignment models *one possible future*. The type $\mathit{ProphAsn}$ is non-empty because $\xi.\mathsf{type}$ is non-empty for any ξ.

Our prophetic reasoning is then *parametrized over prophecy assignments* (called *π-parametrized* for short), which ensures we reason over *all possible futures* (*i.e.,* assignments) for our prophecies in parallel. Later in modeling types for RustHornBelt (§6), we characterize each Rust object by a *π-parametrized representation value* (*i.e.,* a map from $\mathit{ProphAsn}$ to a representation value). As a convention in this paper, we use a variable with a hat (like $\hat{a}$) to denote a π-parametrized value. For example, a mutable reference to an integer `&mut i32` is modeled as a π-parametrized integer pair $\hat{p} = \lambda\pi.\, (n, \pi\, \xi)$, where n is the target integer value and ξ is the prophecy of the mutable borrow. Recall that $\pi\, \xi$ is conceptually the value finally assigned to ξ in one 'possible future' π. Generally, a π-parametrized value like that $\hat{p}$ can be viewed as a *value that depends on prophecies*.

To manage unresolved prophecies, we introduce the notion of a *prophecy token* $[\xi]_q \in \mathit{iProp}$. A prophecy token is a (non-persistent) Iris proposition that claims a fraction $q \in (0, 1]_{\mathbb{Q}}$ of ownership for a prophecy ξ. By holding $[\xi]_q$, we ensure that ξ is not yet resolved. When we *create a new prophecy* ξ using Proph-intro, we get the full token $[\xi]_1$ for it. Here, we can specify the prophecy ξ's type A. We use the *view shift* $\Rrightarrow$ to switch the prophecy's state from 'not yet used' to 'in use (and unresolved)'. Prophecy tokens can then be split and merged according to the fraction (Proph-frac).

As reasoning advances, we resolve more prophecies and have fewer possible futures, which means that we get more information from the prophecy assignments π that remain valid. Such information is modeled as a *prophecy observation* $\langle \pi.\, \hat{\phi}\, \pi \rangle \in \mathit{iProp}$, which is a *persistent* Iris proposition asserting

$$\text{persistent}(\boxtimes\, n)$$

$$\textsc{Time-receipt-mono}\quad \frac{n \le m}{\boxtimes\, m \vdash \boxtimes\, n}$$

$$\textsc{Wp-time-receipt-incr}\quad \frac{\boxtimes\, n \qquad \text{wp}\, e\, \{\Phi\} \qquad e \text{ is not a value}}{\text{wp}\, e\, \{\, v.\ \boxtimes(n+1) \ast \Phi\, v \,\}}$$

$$\textsc{Wp-laters-time-receipt}\quad \frac{\boxtimes\, n \qquad \text{wp}\, e\, \{\Phi\} \qquad e \text{ is not a value}}{\Rightarrow\!\!\!\!\Rightarrow^{n+1} P \ \Rightarrow\ \text{wp}\, e\, \{\, v.\ P \ast \Phi\, v \,\}}$$

Fig. 11. Selected proof rules of the flexible step indexing library.

that the pure proposition $\hat{\phi}\, \pi \in \text{Prop}$ holds for any valid prophecy assignment π. Here, 'π.' is a variable binder. We can reason on prophecy observations: we can conjoin two observations (Proph-obs-merge), weaken an observation (Proph-obs-impl), and get a trivial observation (Proph-obs-true).

The remarkable rule is Proph-resolve, *resolution* of a prophecy ξ (*i.e.,* assigning the final value to ξ). Let's understand this through the 'subdivision' example presented earlier in §4. We have the prophecy ξ_o for the mutable reference `mo`, which we wish to (partially) resolve to the *π-parametrized* value $\hat{a} = (\lambda\pi.\ \text{Some}\,(\pi\, \eta_i))$, which *depends on* η_i, the newly taken prophecy for `mi`. To resolve ξ_o, we must own the full token $[\xi_o]_1$. And also we should ensure that all the *dependencies* Y (just $\{\eta_i\}$ here) of $\hat{a}$ are *not yet resolved*. As a witness of that, we use a prophecy token $[Y]_q \triangleq \ast_{\eta \in Y} [\eta]_q$, which is just $[\eta_i]_q$ here. In RustHornBelt, we can acquire such tokens from an object of any type τ (ty-own-proph, §6.3). As mentioned earlier in §4, in this framework we prevent causal loops by limiting dependencies of the assigned value $\hat{a}$ only to unresolved prophecies. At a high level, we guarantee dependencies between prophecies to form a *partial order* by making them respect the dynamic, chronological order of prophecy resolution.

The prophecy framework is designed to ensure that at any time *there is at least one possible future*, *i.e.,* one valid prophecy assignment, which amounts to *soundness* of the prophetic reasoning. From this property, we obtain the rule Proph-obs-sat to leave the prophetic world and return to the *real world*: whenever we have a prophecy observation $\langle \pi.\ \hat{\phi}\, \pi \rangle$, we can obtain a *pure proposition* $\exists \pi_0.\ \hat{\phi}\, \pi_0$, *i.e.,* know that $\hat{\phi}$ holds for some prophecy assignment.

Notably, this formulation is language-independent, *i.e.,* it makes no assumptions about the target language. In existing approaches, as mentioned above, explicit prophecy resolution annotations were needed to acquire information about the future of prophecies from the program world. In our approach, we avoid that by managing all possibilities about the future inside the logical world, which has *no need for information on the future of the execution of the program.*

5 TIME RECEIPTS FOR MORE FLEXIBLE STEP INDEXING

The Iris logic we use to model types is based on the technique of *step indexing*. In particular, because of how the lifetime logic [Jung et al. 2018a, §5] is modeled in Iris, when we access a resource P guarded by a borrow, we don't get P; instead, we get P guarded by a *later modality*: $\triangleright\, P$ (see LftL-bor-acc in §3.1.1). Typically, to obtain P from $\triangleright\, P$ we must perform a step of computation. If we have $\triangleright\, P$ as a precondition, then we can frame it in the specification of a non-reduced program fragment to get P as a postcondition.

However, in RustHornBelt, we sometimes need to access resources which are stored in a stack of borrows with a statically unbounded depth (*e.g.,* ty-own-proph in §6.3). To access such a resource, we must successively open each borrow, obtain the nested borrow under a later modality, strip it,

and repeat with the nested borrow. In standard Iris, accessing a resource behind d layers of borrows thus requires d steps of computation. Since d is not statically bounded, this is unacceptable.

As a solution, we propose a slight change to the definition of the weakest precondition connective at the core of Iris's program logic. Normally, in Iris, framing allows us to strip at most one later modality for each step of computation. Instead, we propose that each step of computation can be used to strip a *variable* number of later modalities: in this model, the i-th step of computation of the program can strip i later modalities. This is relevant to our problem, since the level of nesting of borrows in RustHornBelt is linked to the depth of in-memory data structures, which is itself bounded by the number of steps of computation executed since the start of the program.

One final ingredient is missing: with the new definition of weakest preconditions, if one wants to strip $n+1$ layers of later modalities, they must provide a proof that the program has executed at least n steps of computation. But how can we state the assertion "at least n steps of computation have been executed" in our logic? To answer this question, we use the notion of *time receipts*, introduced by Mével et al. [2019].

Time receipts exist in two flavors: *persistent time receipts* and *exclusive time receipts*. Although our model actually uses both kinds of time receipts, for the sake of simplicity, we only explain persistent time receipts here. Intuitively, a persistent time receipt $⧖ n$ witnesses that n steps of computation have been performed, and therefore, that n laters can be stripped from assertions in each subsequent step of computation. The basic rules for the time receipt are shown in Fig. 11.

First, we note that persistent time receipts are, well, persistent: they can be duplicated at will. Then, rule Time-receipt-mono shows that persistent time receipts are monotonic: if one knows that at least m steps of computation have been performed, then so have n steps where $n \leq m$. Rule Wp-time-receipt-incr makes it possible to witness an additional step of computation: if we have a pre-existing time receipt $⧖ n$, then we can transform it into an incremented time receipt $⧖(n+1)$ in the postcondition of the weakest precondition connective for a program fragment which still requires computation (*i.e.,* it is not a value).

Finally, Wp-laters-time-receipt lets us use time receipts to strip laters. It requires a time receipt $⧖ n$, a proof of weakest precondition for a non-reduced program fragment and an assertion P hidden behind $n+1$ laters. With these premises, this rules concludes a weakest precondition connective for the same program fragment, but with P as an additional postcondition. Moreover, instead of requiring $\triangleright^{n+1} P$ as a premise, we require the weaker premise $\Rrightarrow^{n+1} P$: this is defined, roughly, as $n+1$ later modalities interleaved with Iris's update modality. These interleaved update modalities are important, since we may need to perform Iris ghost updates between the stripping of each later: for example, in our example of nested borrows, we need to claim the content of a layer of nested borrows, which requires a ghost update.

6 RUSTHORNBELT: RUST TYPES MODELED IN IRIS

Now that we have introduced our approach to prophecies (§4) and the time receipts for flexible step indexing (§5), we illustrate our *full model* of Rust types in Iris, achieved by extending the simplified model presented in §3. First, we present our semantic domain of types (§6.1). We present our interpretations of various types (§6.2) and then focus on our interpretation of the *mutable reference* type (§6.3). Finally, we briefly explain how we can verify soundness of RustHorn-style translations using the semantic interpretations of types (§6.4).

6.1 The Semantic Domain of Types

In the simplified domain of types in presented in §3.1, we added a parameter for the representation value $a \in \lfloor \tau \rfloor$ to the ownership and sharing predicates. In our full semantic domain of types, we instead add a parameter for the *prophetic representation value* $\hat{a} \in \mathit{ProphAsn} \to \lfloor \tau \rfloor$, which is

parametrized over the prophecy assignment $\pi \in \mathit{ProphAsn}$. We also add a new parameter, the *depth* $d \in \mathbb{N}$. So we have the following signatures in our full domain:

$$[\![\tau]\!].\mathrm{own} : (\mathit{ProphAsn} \to \lfloor \tau \rfloor) \times \mathbb{N} \times \mathit{TId} \times \mathit{List}\, \mathit{Val} \to \mathit{iProp}$$
$$[\![\tau]\!].\mathrm{shr} : (\mathit{ProphAsn} \to \lfloor \tau \rfloor) \times \mathbb{N} \times \mathit{Lft} \times \mathit{TId} \times \mathit{Loc} \to \mathit{iProp}$$

We use prophetic, π-parametrized values to model *mutable borrows*. For example (as mentioned in §4.1), a mutable reference to an integer $\&^{\kappa}_{\mathbf{mut}}\, \mathbf{int}$ is characterized by a prophetic value $\lambda\pi.\, (n, \pi\, \xi)$, where n is the target integer value and ξ is the prophecy variable for the mutable reference.

The *depth* $d \in \mathbb{N}$ is an auxiliary parameter for using our *flexible step indexing* library (§5). It acts as an upper bound on the number of *later modalities* we need to strip off to process the whole body of an object. For example, the depth of a doubly nested owned pointer to an integer $\mathbf{box}\,\mathbf{box}\,\mathbf{int}$ should be two or more, while the depth of a singly-linked integer list $\mathbf{list}\langle\mathbf{int}\rangle \triangleq \mu X.\, ()+\mathbf{int}\times\mathbf{box}\, X$ should be its length or more. In the model of the type context, each object is tagged with the time receipt $\boxtimes d$ for the object's depth d (see §6.4).

Like before, we require the ownership and sharing predicates to satisfy certain conditions. Some of these conditions are the same as in the simplified domain: the ownership predicate should follow the type's size, and the sharing predicate should be persistent and monotone over the lifetime parameter. We also require the two predicates to be monotone over the depth parameter (*i.e.,* we can freely bump up the depth). Like TY-SHARE in the simplified domain, we have a rule for converting a reference to an object satisfying the ownership predicate to the sharing predicate. We also have new rules related to *prophecy tokens* (§4.1) in this full domain (TY-OWN-PROPH and TY-SHR-PROPH), which are crucial for supporting *nested mutable borrows*. Because they are closely linked with how mutable references are modeled, we introduce them in §6.3.

6.2 Interpreting Types

Our interpretation of types is largely a straightforward extension to the interpretation in the simplified domain §3.2. Representation values are now π-parametrized in the full domain, but we can just think *pointwise* over the parameter π.

The newly introduced depth parameter needs some explanation. For example, the ownership predicate for the owned pointer type $\mathbf{box}\,\tau$ is defined as follows in the full domain:

$$[\![\mathbf{box}\,\tau]\!].\mathrm{own}(\hat{a}, d, \mathrm{t}, \overline{\mathfrak{v}}) \triangleq d > 0 \;*\; \exists\, \ell \text{ s.t. } \overline{\mathfrak{v}} = [\ell].\; \exists \overline{\mathfrak{w}}.\; \ell \mapsto \overline{\mathfrak{w}} \;*\; \triangleright\, [\![\tau]\!].\mathrm{own}(\hat{a},\, d-1,\, \mathrm{t},\, \overline{\mathfrak{w}}) \;*\; \mathrm{Dealloc}(\ell, \mathrm{size}(\tau))$$

The depth d is set to one plus the depth of the target object, because its target object is put under the later modality $\triangleright$. Recall that the depth represents (an upper bound on) the number of laters we have to strip to process the whole object.

6.3 Interpreting Mutable References

Now let's focus on our highlight, the interpretation of *mutable references* $\&^{\kappa}_{\mathbf{mut}}\, \tau$. To better motivate the our complex model of mutable references, we will introduce it *step by step* using a series of simplified, partial models.

First, let's start with the following 'wrong' model:

$$[\![\&^{\kappa}_{\mathbf{mut}}\, \tau]\!].\mathrm{own}(\hat{p}, d, \mathrm{t}, \overline{\mathfrak{v}}) \triangleq d > 0 \;*\; \exists\, \ell \text{ s.t. } \overline{\mathfrak{v}} = [\ell].\; \exists \overline{\mathfrak{w}}.\; \&^{\kappa}_{\mathrm{full}}\big(\, \ell \mapsto \overline{\mathfrak{w}} \;*\; [\![\tau]\!].\mathrm{own}(\lambda\pi.\, (\hat{p}\, \pi).0,\, d-1,\, \mathrm{t},\, \overline{\mathfrak{w}})\, \big)$$

To make it possible for the lender to get back the resource when the lifetime ends, we use a *full borrow* $\&^\kappa_{\mathrm{full}}(\cdots)$ (explained in §3.1.1),[4] and place inside all the loaned resources: the ownership of memory cells $\ell \mapsto \overline{w}$ and the model of the inner type $[\![\tau]\!].\mathrm{own}(\cdots)$. The representation value of the inner type is set to $\lambda\pi.\,(\hat{p}\,\pi).0$, the left-hand component of the mutable reference's value $\hat{p}$. Like an owned pointer, the depth is set to one plus the depth of the target object.

However, this model doesn't work! Because the representation value, depth and low-level values are *fixed* in the full borrow, the mutable reference *can't perform any update*. A full borrow $\&^\kappa_{\mathrm{full}} P$ establishes a contract between the borrower and lender: the borrower promises to store a resource satisfying P for the lender. Here, the contract mentions fixed values of these parameters. To allow updates, we *existentially quantify* the content of the borrow over these parameters:

$$[\![\&^\kappa_{\mathbf{mut}}\,\tau]\!].\mathrm{own}(\hat{p}, d, \mathtt{t}, \overline{v}) \triangleq d > 0 * \exists \ell \text{ s.t. } \overline{v} = [\ell].\ \&^\kappa_{\mathrm{full}}\big(\exists \hat{a}', d', \overline{w}.\ \ell \mapsto \overline{w} * [\![\tau]\!].\mathrm{own}(\hat{a}', d', \mathtt{t}, \overline{w})\big)$$

Now the mutable reference can freely perform updates, but we still have problems.

First, *the lender gets no information about the values $\hat{a}', d'$* after the borrow ends! In RustHorn-style representation, we employ a *prophecy* ξ to solve this issue. When we create a new mutable borrow, we take a fresh prophecy ξ and share it between the lender and the borrower (mutable reference). We can update our model with prophecies as follows:

$$[\![\&^\kappa_{\mathbf{mut}}\,\tau]\!].\mathrm{own}(\hat{p}, d, \mathtt{t}, \overline{v}) \triangleq d > 0 * \exists \ell \text{ s.t. } \overline{v} = [\ell].\ \exists \xi \text{ s.t. } \lambda\pi.\,(\hat{p}\,\pi).1 = \lambda\pi.\,\pi\,\xi.\ \&^\kappa_{\mathrm{full}}\big(\exists \hat{a}', d', \overline{w}.\ ([\xi]_1 \vee \langle\pi.\,\pi\,\xi = \hat{a}'\pi\rangle) * \ell \mapsto \overline{w} * [\![\tau]\!].\mathrm{own}(\hat{a}', d', \mathtt{t}, \overline{w})\big)$$

This model requests that the prophecy ξ corresponds to the 'prophetic' part of the mutable reference's representation value $\lambda\pi.\,(\hat{p}\,\pi).1$. The prophecy can be either *resolved* or not. Before resolution we have a full *prophecy token* $[\xi]_1$ (obtained from Proph-intro). At the time the reference is released, we consume the prophecy token and resolve the prophecy ξ to get a *prophecy observation* $\langle\pi.\,\pi\,\xi = \hat{a}'\pi\rangle$. In the borrow, we store a *disjunction* $[\xi]_1 \vee \langle\pi.\,\pi\,\xi = \hat{a}'\pi\rangle$, to switch between these two states, allowing the lender to know the value of ξ after the borrow's lifetime ends. Interestingly, it is fine for the borrow to end before the prophecy is resolved; the lender will then obtain the prophecy token $[\xi]_1$ and perform resolution for itself by consuming the token.

This model is getting better, but has one big issue: there *is no link* between the inner values $\hat{a}', d'$ and the mutable reference's values $\hat{p}, d$! To establish this link, we use Iris *ghost state* shared between inside and outside the full borrow. This ghost state is encapsulated in two auxiliary tokens, a *value observer* $\mathrm{VO}_\xi[\hat{a}]_d$ and a *prophecy controller* $\mathrm{PC}_\xi[\hat{a}]_d$. This gives the following model:

$$[\![\&^\kappa_{\mathbf{mut}}\,\tau]\!].\mathrm{own}(\hat{p}, d, \mathtt{t}, \overline{v}) \triangleq d > 0 * \exists \ell \text{ s.t. } \overline{v} = [\ell].\ \exists \xi \text{ s.t. } \lambda\pi.\,(\hat{p}\,\pi).1 = \lambda\pi.\,\pi\,\xi.\ \mathrm{VO}_\xi[\lambda\pi.\,(\hat{p}\,\pi).0]_{d-1} * \&^\kappa_{\mathrm{full}}\big(\exists \hat{a}', d', \overline{w}.\ \mathrm{PC}_\xi[\hat{a}']_{d'} * \ell \mapsto \overline{w} * [\![\tau]\!].\mathrm{own}(\hat{a}', d', \mathtt{t}, \overline{w})\big)$$

The value observer outside the full borrow *observes* the parameters of the prophecy controller inside, constraining $\hat{a}'$ to $\lambda\pi.\,(\hat{p}\,\pi).0$ and d' to $d-1$. The prophecy controller subsumes the disjunction $[\xi]_1 \vee \langle\pi.\,\pi\,\xi = \hat{a}'\pi\rangle$ used in the previous model, and can be used by the lender to learn the prophecy's value. The mutable reference performs updates by changing the parameters of the value observer and prophecy controller simultaneously. Formally, we have the following rules:

Mut-agree
$$\mathrm{VO}_\xi[\hat{a}]_d * \mathrm{PC}_\xi[\hat{a}']_{d'} \vdash (\hat{a}, d) = (\hat{a}', d')$$

Mut-update
$$\mathrm{VO}_\xi[\hat{a}]_d * \mathrm{PC}_\xi[\hat{a}]_d \Rrightarrow \mathrm{VO}_\xi[\hat{a}']_{d'} * \mathrm{PC}_\xi[\hat{a}']_{d'}$$

[4] Note again that a full borrow $\&^\kappa_{\mathrm{full}} P$ is a semantic notion modeled in the lifetime logic in Iris, while $\&^\kappa_{\mathbf{mut}}\,\tau$ is just a (syntactic) Rust type for a mutable reference.

Importantly, we can resolve the prophecy ξ of a mutable reference with the following rule:

MUT-RESOLVE
$$\frac{\mathrm{dep}(\hat{a}, Y)}{\mathrm{VO}_\xi[\hat{a}]_d \ast \mathrm{PC}_\xi[\hat{a}]_d \ast [Y]_q \ \Rrightarrow\ \langle \pi.\, \pi\,\xi = \hat{a}\,\pi \rangle \ast \mathrm{PC}_\xi[\hat{a}]_d \ast [Y]_q}$$

This rule is an extension of PROPH-RESOLVE. Using it we obtain the prophecy observation $\langle \pi.\, \pi\,\xi = \hat{a}\,\pi \rangle$. During resolution, we lose the value observer but keep the prophecy controller, which is necessary to fulfill the borrow's contract, and allow the lender to observe the resolved value. Note, the model is still a slight simplification: the full model includes a time receipt and a modification for monotonicity over the depth parameter. Please see the Coq mechanization for the full model.

Based on the model of the mutable reference type, we can also model advanced types that use mutable borrows, such as the mutable iterator type `IterMut<'a, T>`.

Nested mutable borrows. In the rule MUT-RESOLVE, we needed a partial prophecy token $[Y]_q$ over the dependencies Y of the representation value $\hat{a}$. These dependencies are non-empty when we handle *nested mutable borrows*. For example, suppose we have a nested mutable reference $\&^\kappa_{\mathbf{mut}}\,\&^\kappa_{\mathbf{mut}}\,\mathbf{int}$ of the representation value $\lambda\pi.\,((n, \pi\,\xi), \pi\,\eta)$. When we release it, we resolve the prophecy η to the value $\lambda\pi.\,(n, \pi\,\xi)$, which depends on ξ. So we need to be able to temporarily take out a prophecy token for ξ from the inner mutable reference. To make that possible, we have the following proof rule:

MUT-PROPH-TOK
$$\mathrm{VO}_\xi[\hat{a}]_d \ast \mathrm{PC}_\xi[\hat{a}]_d \vdash \mathrm{VO}_\xi[\hat{a}]_d \ast [\xi]_1 \ast ([\xi]_1 \mathrel{-\!\!\ast} \mathrm{PC}_\xi[\hat{a}]_d)$$

When we have a value observer and a prophecy controller, we can temporarily take out $[\xi]_1$.

The tricky point is that we need to support prophecy resolution of a mutable reference $\&^\kappa_{\mathbf{mut}}\,\tau$ for *any type* τ. For example, τ can be a list of mutable references $\mathbf{list}\langle\&^\kappa_{\mathbf{mut}}\,\mathbf{int}\rangle \triangleq \mu X.\,() + \&^\kappa_{\mathbf{mut}}\,\mathbf{int} \times \mathbf{box}\,X$, which can contain an unbounded number of prophecies. To ensure that we can do the operation for any type τ, we require the following rule on the ownership predicate of any type:

TY-OWN-PROPH
$$\begin{array}{l}[\![\tau]\!].\mathrm{own}(\hat{a}, d, \mathrm{t}, \overline{\mathfrak{v}}) \ast [\mathrm{lft}(\tau)]_{q'} \ \Rrightarrow^d \\ \quad \exists\, q, Y \text{ s.t. } \mathrm{dep}(\hat{a}, Y).\ \ [Y]_q \ast \left([Y]_q \Rrightarrow\!\!\!\ast\ [\![\tau]\!].\mathrm{own}(\hat{a}, d, \mathrm{t}, \overline{\mathfrak{v}}) \ast [\mathrm{lft}(\tau)]_{q'} \right)\end{array}$$

It lets us temporarily get access to a partial prophecy token $[Y]_q$ for the dependencies Y of the representation value $\hat{a}$. We also have a similar rule TY-SHR-PROPH for the sharing predicate. Here, we make use of our flexible step indexing to perform an *n-step-taking view shift* $P \Rrightarrow^n Q$: a view shift from P to Q interleaved with n laters. To take out all the relevant prophecy tokens from the object, we need to strip many laters. For example, we have to strip n laters for a list $\mathbf{list}\langle\&^\kappa_{\mathbf{mut}}\,\mathbf{int}\rangle$ of length n. Also, we have to strip k laters for a k-fold nested mutable reference $\&^\kappa_{\mathbf{mut}}\,\&^\kappa_{\mathbf{mut}}\,\cdots\&^\kappa_{\mathbf{mut}}\,\mathbf{int}$. The depth parameter d upper-bounds this number, and our flexible step indexing machinery (§5) lets us perform this operation in a *single program step* as long as we have a *time receipt* $\boxtimes\, d$ (WP-LATERS-TIME-RECEIPT). Note that a lifetime token $[\mathrm{lft}(\tau)]_{q'}$ is necessary here. Here, $\mathrm{lft}(\tau)$ is defined as the intersection of all lifetimes in τ (*e.g.*, $\mathrm{lft}(\&^\kappa_{\mathbf{mut}}\,\&^{\kappa'}_{\mathbf{mut}}\,\mathbf{int}) = \kappa \sqcap \kappa'$). We need this lifetime token to open the full borrow of each mutable reference in the object.

6.4 Proving Soundness of RustHorn-Style Translations

Using our semantic interpretations of Rust types, we can prove the soundness of RustHornBelt. Here we briefly explain how the proof works.

First, we give the interpretation of a *type context*. As explained in §2.1, a type context $\mathbf{T}$ is a list of items of form either $p \lhd \tau$ (we *own* an object of type τ for the path p) or $p \lhd^{\dagger\kappa} \tau$ (an object of

type τ is *borrowed* until the lifetime κ ends). We define $\mathbf{T}$'s interpretation $[\![\mathbf{T}]\!](\overline{\hat{a}}, \mathfrak{t})$ as the separating conjunction of the interpretation $[\![p \lhd^? \tau]\!](\hat{a}, \mathfrak{t})$ of each element $p \lhd^? \tau$ of $\mathbf{T}$. An *active path* $p \lhd \tau$ is interpreted as follows:

$$[\![p \lhd \tau]\!](\hat{a}, \mathfrak{t}) \triangleq \exists d.\ \textstyle\bigtriangledown d * [\![\tau]\!].\mathrm{own}(\hat{a}, d, \mathfrak{t}, [[\![p]\!]])$$

It is τ's ownership predicate equipped with a time receipt $\bigtriangledown d$ for the depth. The time receipt lets us strip d laters for operations like TY-OWN-PROPH (by the rule WP-LATERS-TIME-RECEIPT). Here, a path p is interpreted as a value $[\![p]\!] \in \mathit{Val}$. The interpretation of a *blocked path* $p \lhd^{\dagger\kappa} \tau$ is trickier, but it is largely a straightforward extension of the definition in RustBelt.

As explained in §2, RustHornBelt has syntactic typing judgments $\mathbf{E}; \mathbf{L} \mid \mathbf{T} \vdash I \dashv x.\, \mathbf{T}' \rightsquigarrow \mathbf{\Phi}$ and $\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F \rightsquigarrow \mathbf{\Phi}$, respectively for an instruction I and a function body F, which each carry a specification in the form of a predicate transformer $\mathbf{\Phi}$. Now using the interpretation of type contexts, we can give *semantics* to these judgments in Iris, written as $\mathbf{E}; \mathbf{L} \mid \mathbf{T} \models I \dashv\!\models x.\, \mathbf{T}' \rightsquigarrow \mathbf{\Phi}$ and $\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \models F \rightsquigarrow \mathbf{\Phi}$ ($\vdash$ replaced with $\models$). The definition of the two judgments is basically a straightforward extension of that of RustBelt.

We proved the following theorem, which is the core of RustHornBelt:

THEOREM 6.1 (FUNDAMENTAL THEOREM OF LOGICAL RELATIONS). *For each syntactic typing rule of RustHornBelt, its semantic version (each $\vdash$ replaced with $\models$) holds in Iris.*

Theorem 6.1 says that any *syntactic* deduction rule for RustHorn-style verification of Rust programs is *semantically* justified. This lets us take a semantic approach to Rust libraries with unsafe code: for each method of a library, we just verify a *semantic* judgment that specifies typing and RustHorn-style representation of the method.

Although during execution unsafe code may violate Rust's AXM principle, these violations are *encapsulated* by *safe* APIs. The behaviors of such APIs are described by the approach of *semantic typing*, in which types are understood not by their syntax but their meaning. Unsafe code behaves like safe code when viewed from safe APIs. So long as we prove that the safe APIs are *semantically typed* with the help of Iris, we can then combine them with the semantic deduction rules given by Theorem 6.1 to derive semantic judgments for well-typed Rust programs that use the verified APIs. Remarkably, this proof strategy is *extensible*: to support new features and libraries of Rust, we just need to verify the semantic deduction rules specifically for them as a lemma in Iris.

To justify semantic judgments, we also proved the following theorem:

THEOREM 6.2 (ADEQUACY). *For any λ_{Rust} function f such that $\varnothing; \varnothing \mid \varnothing \models f \dashv\!\models x.\, x \lhd \mathbf{fn}() \to () \rightsquigarrow \lambda\Psi, [].\, \Psi\, [\lambda\Psi', [].\, \Psi'()]$ holds, no execution of f (with the trivial continuation) ends in a stuck state.*

Theorem 6.2 justifies a semantically deduced predicate transformer for any closed program. Here, $\lambda\Psi', [].\, \Psi'()$ is the expected value of the function f. In RustHornBelt, the representation value of a Rust function is a predicate transformer. The ownership predicate of Rust's function type is defined using the semantic judgment for a function body, just like in RustBelt.

7 RELATED AND FUTURE WORK

Prophecies. First introduced by Abadi and Lamport [1988] for proving refinement between state machines, *prophecies* have been studied for decades, although they still remain somewhat exotic. Vafeiadis [2008], Zhang et al. [2012] and Jung et al. [2020] modeled prophecies in separation logic, mainly to prove *linearizability* (or *logical atomicity*) of tricky concurrent data structures (particularly, Restricted Double-Compare Single-Swap). The prophecies of Jung et al. [2020] have also been used to verify an algorithm for fixed-point calculation [de Vilhena et al. 2020]. All these

works use prophecies with a single future, and require one to annotate the program with prophecy resolution instructions. We explain why this approach does not work for RustHornBelt in §4.

Toward the goal of verifying fine-grained concurrent data structures, Turon et al. [2013] and Liang and Feng [2013] employed a different technique of *speculation*. Speculation has some conceptual similarity with our RustHorn-style prophecies in that, like our prophecies, speculation considers multiple possible futures simultaneously. However, the mechanisms are very different. With speculation, possible states are first *speculated* and then later on either *committed* to or *dropped* as the proof explores further steps of the program's execution. In contrast, our model only provides the ability to make *persistent* observations that will continue to hold under all future prophecy resolutions. A key motivation for speculation was that speculative observations were more modularly composable than previous formalizations of prophecy variables. However, at least in the context where persistent prophecy observations are useful, our work overcomes this limitation, and shows how to compose prophetic observations in a natural manner.

We believe that our prophecy framework (§4) may be applicable to contexts outside RustHorn-style verification, potentially even serving as a viable replacement for annotation-based prophecies (*e.g.*, Jung et al. [2020]) in concurrency reasoning, but future work remains to demonstrate this.

RustHorn. Matsushita et al. [2020, 2021] gave a *syntactic* proof of correctness of RustHorn's translation over a *safe* subset of Rust. Their proof establishes a bisimulation between the execution of a Rust program and the resolution over the constrained Horn clauses (CHCs) RustHorn generates for the program. However, this proof is non-modular, and extension with a new unsafe feature can easily require global reconstruction of the proof, thus motivating the need for our work. Still our work does not fully subsume their proof, because while we prove only soundness they also proved *completeness*. In fact, completeness is lost when we add support for libraries with interior mutability like `Cell`, since invariants can't precisely track dynamic changes of values.

RustHorn automatically verifies the safety (the property that no assertion fails) of Rust programs by translating them to CHCs and calling automated solvers [Komuravelli et al. 2014; Champion et al. 2018]. Although the support is limited to a safe subset of Rust, it successfully verified interesting programs that mutate recursive data structures like lists and trees using mutable borrows. Our work on RustHornBelt verifies the soundness of RustHorn-style translations over various APIs with *unsafe* code, but future work remains in evaluating how well that extended translation enables automated verification. RustHorn's approach has also been applied in other Rust verification tools such as Creusot [Denis 2021], which translates to functional programs rather than CHCs.

Verification under ownership-based types. There are a number of projects that verify Rust programs focusing on unique aspects of Rust's type system.

Prusti [Astrauskas et al. 2019] analyzes type information from the Rust compiler to synthesize verification conditions in the separation logic of Viper [Müller et al. 2016], which is suited for automated verification. Rather than using a high-level abstraction like a lifetime logic, they directly represent the flow of ownership with the help of lifetime information from the Rust compiler. To model a mutable borrow, they introduced the notion of *pledges*, which model properties that are true at the end of a borrow. Still, their approach struggles with certain advanced use cases of mutable borrows. Also, they do not support unsafe code, due to limitations of Viper's expressivity.

Electrolysis [Ullrich 2016] developed a translation from Rust programs to purely functional programs which can then be verified manually in the Lean theorem prover. Using this approach the author verified challenging algorithms implemented in Rust. Electrolysis represents mutable borrows using functional lenses, and passes them to functions using a state-passing style: the borrow is given as input to a function and returned alongside the result. This approach is elegant

but does not work when the addresses of mutable references are *dynamic*. Also, the correctness of the translation has not been formally proved.

There are also a number of approaches to bounded verification of (potentially unsafe) Rust code, which do not guarantee correctness in all executions but are helpful for finding bugs. CRUST is a bounded model checker [Toman et al. 2015] that can automatically verify safety of a Rust library with unsafe code. It verifies that up to n library calls, there is no violation of Rust's aliasing and safety guarantees. They successfully re-discovered bugs that existed in an older version of a standard Rust library. However, it does not scale to usage of multiple libraries. Lindner et al. [2018] and Baranowski et al. [2018] performed bounded, automated functional verification of Rust programs with unsafe code, by symbolic execution and SMT solving, respectively.

Ownership typing has also been applied to verify programs in languages other than Rust [Clarke et al. 1998; Jim et al. 2002; Zibin et al. 2010; Tov and Pucella 2011]. Among such approaches, RefinedC [Sammler et al. 2021] uses ownership typing with refinement to perform automated functional verification of C programs entirely in Coq. It controls ownership of mutable pointers in the style of Mezzo [Balabonski et al. 2016] rather than by lifetime-based borrows. RefinedC is proven sound in Iris via the *semantic typing* approach, like RustBelt and RustHornBelt. RefinedC carefully constructs a separation logic suited for automation, Lithium, which allows powerful goal-directed proof search. Constructing a logic and proof search like that for RustHornBelt is an intriguing direction of future work. By working entirely in Coq, RefinedC enjoys a very small trusted core: it verifies C programs in the same framework used to prove the metatheory. On the other hand, RustHornBelt is meant as a *foundational proof* for *separate* RustHorn-style verification tools; this separation enables those tools to leverage existing well-engineered automated verifiers.

Verification of imperative pointer programs. Outside of Rust, much work has been done on the verification of pointer programs, especially with separation logic. Steel [Fromherz et al. 2021] is a language for the verification of imperative pointer programs, built on top of the SteelCore [Swamy et al. 2020] separation logic implemented in the F* proof assistant [Swamy et al. 2016]. Programs are verified using *Hoare quintuples* which separate *ownership* and *verification* concerns. This approach allows for ergonomic verification with strong integration with SMT solvers. The underlying separation logic, SteelCore, is inspired by the Iris logic we based our work on. Though SteelCore does not aim for the same generality as Iris, it features many of the same core abstractions like partial commutative monoids (to represent resources) and named invariants.

Outside of separation logic, Why3 applies *region typing* [Gondelman 2016] to verify pointer programs in a first-order Hoare logic. It associates a unique *region* with each mutable location and precisely tracks aliases during type checking, in order to enforce some form of ownership. The region typing of Why3 is completely hidden from users, reminiscent of how our RustHorn-style translation encapsulates the underlying separation logic. However, to accurately track regions it only supports *non-recursive* types, and data types like singly linked lists must be axiomatized.

REFERENCES

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

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

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

Marek S. Baranowski, Shaobo He, and Zvonimir Rakamaric. 2018. Verifying Rust Programs with SMACK. In *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA (Lecture Notes in Computer Science, Vol. 11138)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 528–535. https://doi.org/10.1007/978-3-030-01090-4_32

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

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

Adrien Champion, Naoki Kobayashi, and Ryosuke Sato. 2018. HoIce: An ICE-Based Non-linear Horn Clause Solver. In *Programming Languages and Systems - 16th Asian Symposium, APLAS (Lecture Notes in Computer Science, Vol. 11275)*, Sukyoung Ryu (Ed.). Springer, 146–156. https://doi.org/10.1007/978-3-030-02768-1_8

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

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

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

Xavier Denis. 2021. Mastering Program Verification using Possession and Prophecies. *32 ème Journées Francophones des Langages Applicatifs* (2021), 174.

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

Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. 2021. Steel: Proof-oriented Programming in a Dependently Typed Concurrent Separation Logic. In *26th ACM SIGPLAN International Conference on Functional Programming (ICFP 2021)*. https://www.microsoft.com/en-us/research/publication/steel-proof-oriented-programming-in-a-dependently-typed-concurrent-separation-logic/

Léon Gondelman. 2016. *Un système de types pragmatique pour la vérification déductive des programmes. (A Pragmatic Type System for Deductive Verification)*. Ph. D. Dissertation. University of Paris-Saclay, France. https://tel.archives-ouvertes.fr/tel-01533090

Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, Carla Schlatter Ellis (Ed.). USENIX, 275–288. http://www.usenix.org/publications/library/proceedings/usenix02/jim.html

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

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

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018b. RustBelt: Securing the Foundations of the Rust Programming Language — Technical Appendix. https://plv.mpi-sws.org/rustbelt/popl18/appendix.pdf

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

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018c. 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

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

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

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

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-based Model Checking for Recursive Programs. In *Computer Aided Verification - 26th International Conference, CAV (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 17–34. https://doi.org/10.1007/978-3-319-08867-9_2

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

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

Marcus Lindner, Jorge Aparicius, and Per Lindgren. 2018. No Panic! Verification of Rust Programs by Symbolic Execution. In *16th IEEE International Conference on Industrial Informatics, INDIN*. IEEE, 108–114. https://doi.org/10.1109/INDIN.2018.8471992

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

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

Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *ACM Trans. Program. Lang. Syst.* (2021). https://doi.org/10.1145/3462205 To be published in TOPLAS Special Issue on ESOP 2020.

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

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

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

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

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

Rust Community. 2021a. *Rust Programming Language*. https://www.rust-lang.org/

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

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

Jan Smans, Bart Jacobs, and Frank Piessens. 2009. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In *ECOOP - Object-Oriented Programming, 23rd European Conference (Lecture Notes in Computer Science, Vol. 5653)*, Sophia Drossopoulou (Ed.). Springer, 148–172. https://doi.org/10.1007/978-3-642-03013-0_8

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 256–270. https://doi.org/10.1145/2837614.2837655

Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: an extensible concurrent separation logic for effectful dependently typed programs. *Proc. ACM Program. Lang.* 4, ICFP (2020), 121:1–121:30. https://doi.org/10.1145/3409003

John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. CRUST: A Bounded Verifier for Rust. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 75–80. https://doi.org/10.1109/ASE.2015.77

Jesse A. Tov and Riccardo Pucella. 2011. Practical Affine Types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 447–458. https://doi.org/10.1145/1926385.1926436

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

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

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

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.

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

Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. 2010. Ownership and Immutability in Generic Java. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 598–617. https://doi.org/10.1145/1869459.1869509