Using the Prophecy-Based Encoding of Rust Borrows in a Realistic Verification Tool

Anonymous authors

Abstract. Verification tools targeting Rust programs need to handle a salient feature of this language: *mutable borrows*. The elegant approach of using *prophecies* to model mutable borrows is used in practice in the Creusot deductive verification tool—yet, so far, this encoding has only been described in the idealized setting of a core calculus. In this work, after observing that "scaling up" this approach into a usable verification tool is non-trivial, we show how to integrate this encoding with two common features of deductive verification systems: *ghost code* and *type invariants*. Additionally, we provide concrete implementation strategies for key aspects of the encoding that were unspecified but turn out to be crucial when considering realistic programs. All of our work has been implemented as an extension of Creusot.

Keywords: Deductive verification · Rust · Prophecies

1 Introduction

The Rust programming language [13] offers an enticing programming paradigm: programmers have access to fine-grained control over resources and data layout while enjoying the benefits of a strong type system that ensures type safety, memory safety and data race freedom. One of Rust's key features are *borrows*. A borrow is a kind of pointer: a reference that can be used to access—and possibly modify—a piece of memory held by another part of the program. Crucially, the use of borrows is restricted by the Rust type system, which statically enforces that a given piece of memory is never aliased and mutated at the same time.

In the context of program verification, Rust's static control over aliasing and mutation is a huge boon. Over the last few years, a wide range of approaches has been developed to tackle verification of Rust programs [1,12,11,9,16]. Among those, RustHorn [16] and Aeneas [9] came up with a similar key insight: it is possible to reason on a well-typed Rust program (*e.g.*, to formally verify its correctness) *almost* as if it were a purely functional program. In particular, one does not need to reason explicitly about pointers nor aliasing. In other words, Rust borrows look like pointers, but reasoning about them turns out to be much easier than reasoning about unrestricted pointers.

Specifically, RustHorn's key contribution is a lightweight encoding of Rust borrows as functional values, based on *prophecies*, a logical mechanism by which one can refer to the "final" value of a borrow. This idea was further formalized in RustHornBelt [15], establishing the soundness of the translation—a non-trivial proof. On the practical side, this prophecy-based translation of borrows is the foundation of Creusot [6], a deductive verification tool for Rust programs. It allows Creusot to be the only deductive verification tool able to handle all uses of borrows found in Rust programs (functions returning borrows, nested borrows, borrows stored in data structures...) while producing verification conditions that can be efficiently offloaded to SMT solvers.

RustHorn's prophetic translation of borrows, as formalized in RustHornBelt and implemented in Creusot, has only been studied on a core calculus. However, building a usable verification tool requires going further than a core calculus: one typically wants to support as many language features as possible, as well as providing the user with powerful logical reasoning principles to aid verification. This is in line with recent work done on Creusot to add support for iterators [5] and use so-called "traits" (originally a Rust feature) for logical reasoning [6].

In this work, we make progress towards building a realistic verification system based on a RustHorn-style encoding of borrows. We build on top of Creusot, and make contributions along the two following axes:

– We show how to integrate the prophetic encoding of borrows with new logical reasoning features: *ghost code* and *type invariants*. Both features are generally useful in performing complex verification tasks and appear in many existing deductive verification systems, such as Dafny [18], Why3 [19] or VCC [4]. Perhaps surprisingly, integrating these seemingly unrelated features presents significant challenges. In particular, we found out that it required *modifying the prophetic encoding of borrows*, in addition to some non-trivial design work to make the resulting system both sound and expressive enough.
– We show how to apply the prophetic encoding of borrows to *realistic programs*. Specifically, practical aspects of the "resolution" step of the translation (see §1.2) were left somewhat unspecified in previous work, namely *where* to perform resolution and how to do it *modularly* (and soundly). These questions become important when building a practical tool and moving from a core calculus to actual Rust programs. We provide answers to both of them.

Before going into the details of our contributions (§1.3), let us first look more closely at how *mutable borrows* are used in Rust (§1.1), and how RustHorn's prophecy-based encoding allows reasoning about them (§1.2).

1.1 Rust's Borrows

Rust provides two flavors of borrows: mutable borrows and shared borrows. Mutable borrows allow modifying the data they point to but cannot be aliased, while shared borrows allow aliasing but disallow mutation. Mutable borrows are the ones we are interested in here, and they are the ones which are represented using a prophecy-based encoding. (Shared borrows are much easier to handle as they can be directly represented as their underlying value, so we will not talk much about them.)

A representative example of the use of mutable borrows is the `index_mut` function on Rust vectors. A vector is a contiguous, extensible array, and `index_mut` can be used to get a pointer to one of its elements, given its index. It has the following signature: [1]

```
fn index_mut<'a, T>(v: &'a mut Vec<T>, i: usize) -> &'a mut T
```

Here, `T` is a type of elements and `Vec<T>` the type of vectors storing `T`s. A call to `index_mut` takes as arguments a mutable borrow `v` on a vector, an integer index `i`, and returns a mutable borrow pointing to the element stored at that index. The *lifetime* `'a` relates the borrows of the vector and of the element. This means that we are locked out from using the initial borrow of the vector until the element's borrow expires.

The following piece of code illustrates the use of this function:

```
fn f(v: &mut Vec<i32>) {
  let bor = index_mut(v, 2);
  // v.push(42); // Forbidden: v is frozen because lifetime is active
  *bor = 3;
  v.push(4); // Allowed: bor is no longer used, so v is unfrozen
}
```

This function takes as parameter `v` a mutable borrow to a vector. It first calls `index_mut` on `v` to obtain a mutable borrow inside this vector. Importantly, performing this call does not move the full ownership of `v` to `index_mut`: this creates implicitly a *reborrow* of `v` (sometimes noted `&mut *v`). This new borrow, passed to `index_mut`, points to the same memory locations, but has a *shorter* lifetime `'a` than that of `v`: once this shorter lifetime ends, we recover the right to use `v`, allowing, for instance to perform the last call `v.push(4)`. However, `'a` has to be active when `bor`, the return value of `index_mut` is still in use. When `'a` is active, `v` is *frozen*: this explains why the call `v.push(42)` would be forbidden. This is fortunate, because this call could reallocate the vector and make `bor` point to freed memory.

1.2 Reasoning on Mutable Borrows Using Prophecies

When reasoning about a program in a verification tool, we typically track the *logical model* of program value through the execution. The challenge with borrows is that mutating a borrow modifies a value held in a *different* part of the program, in a way that becomes observable *as soon as the borrow ends*. How can we express, at the logical level, the relationship between a borrow and the value it borrows from?

Logical interpretation of borrows The key idea from RustHorn is to take the logical model of a borrow to be a pair of values: its *current* value, and its prophesized *final* value. This final value is called a *prophecy* because it logically represents a value that is only known later in the program, at the time its corresponding borrow ends.

[1] In the Rust standard library, `index_mut` has a more general signature, parameterized over a general notion of index; we show here one of its most standard instances.

Let us illustrate this idea on a simple example, shown below. We look at the logical interpretation of three key operations on borrows: creating a borrow, writing to a borrow, and ending a borrow. We show Rust code on the left, and the logical context associating each variable with its model on the right.

1	`let mut x = 0;`	$x \mapsto 0$
2	`let y = &mut x;`	$x \mapsto \alpha,\ y \mapsto (0, \alpha)$
3	`*y = 1;`	$x \mapsto \alpha,\ y \mapsto (1, \alpha)$
4	`// resolve y`	$x \mapsto \alpha,\ y \mapsto (1, \alpha),\ \alpha = 1$
5	`assert!(x == 1);`	

On line 2, we create a borrow `y` from the variable `x`. We model this operation by creating a *new prophecy* α representing the final value of `y`. While `y` is active, it has exclusive access to the value it borrows and holds it as its current value. The model of `x` is set to α: thus, whenever `y` ends, `x` will be able to observe the mutations made through `y`'s lifetime. On line 3, we update the borrow with a new value, and its model is updated consequently. Finally, the borrow `y` ends on line 4, at which point we *resolve* its prophecy: we learn that the final value of the borrow is equal to its current value. This allows us to propagate the changes back to the model of `x`, and prove the assertion on line 5.

Using this prophetic encoding, one can reason about memory in a purely local way: mutating a borrow only affects the borrow's value. The resolution of a borrow introduces an equality that can be propagated back to its lender, without requiring the knowledge of *who* the lender is to perform resolution.

Function specifications The prophetic interpretation of borrows is *compositional*: the specification of a borrow-manipulating function can be expressed solely in terms of the current and final values of its arguments and its result.

For instance, let us look at how the `index_mut` function on vectors is specified in Creusot (we omit calls to Creusot's "shallow model" operator `@` for clarity):

```
1  #[requires(0 <= i && i < v.len())]
2  #[ensures(*result == (*v)[i] && ^result == (^v)[i])]
3  #[ensures(forall<j: Int> 0 <= j && j < v.len() && j != i ==>
4                              (*v)[j] == (^v)[j] )]
5  fn index_mut<T>(v: &mut Vec<T>, i: usize) -> &mut T
```

The `requires` and `ensures` keywords specify pre- and post-conditions that can refer to the function's arguments and return value (named `result`). Here, `v` and `result` are borrows: Creusot allows the specification to refer not only to their *current* value using "`*v`" and "`*result`" but also their *final* value using "`^v`" and "`^result`".

The specification requires that `i` is a valid index (line 1), and provides two guarantees. First, the `result` borrow is responsible for the value at index `i` of the vector (line 2). Second, the contents of the vector at any other index `j` (line 3) is not changed by the call to `index_mut` (line 4). Together, these two properties provide an elegant and complete specification of `index_mut`.

1.3 Contributions

The central observation behind this work is that, while RustHorn's prophetic translation provides a clean and general foundation, putting it into practice requires finding solutions to many additional design problems.

In this work, we tackle several challenges we faced when scaling up the prophetic translation into a practical verification tool. In each case, our solution aims to strike a fine balance between *soundness* and *expressivity*. Creusot must be sound, and its soundness argument should ideally be easy to understand. However, soundness of the prophetic translation is non-trivial: it relies on the fact that prophecies cannot be used to create "causality cycles", a property that is hard to reason about and uphold when extending the system. But expressivity is also important: borrows are a ubiquitous feature of Rust, and even small restrictions on the reasoning rules for borrows can have a drastic impact on the ability to reason about realistic Rust programs. Each of the proposals below aims to be sound and have good expressivity, while minimizing the overall complexity of the system.

Ghost Code is a feature found in deductive verification tools which adds the ability to intersperse a verified program with code that is not executed but helps verification. Ghost code can also typically refer to values that exist only at the logical level: this is important for expressivity. This poses an important challenge when combining ghost code with prophecies (§2.1): one must prevent ghost code from using prophecies in nonsensical ways (*e.g.*, instantiating a prophecy with a value that depends on itself) without compromising its expressivity.

Our solution involves stratifying our logic into "prophetic" and "non-prophetic" layers (§2.2), and *extending the prophetic translation* to introduce a notion of "logical identity" of borrows (§2.3). This allows us to control the behavior of logical equality on borrows and is necessary to restore soundness, but it can also affect expressivity if one assigns identities that distinguish "too many" borrows. For that purpose, we propose a new program analysis (§2.4) identifying so-called "final borrows", which are cases where a borrow identity can be soundly "reused".

Type Invariants in deductive verification systems allow the user to specify a logical predicate associated with all values of a chosen type. This is a convenient feature for verification: it allows threading correctness invariants in a systematic fashion while typically also allowing them to be temporarily broken as long as they are restored afterward. In practice, Creusot enforces invariants at function boundaries: it automatically inserts pre- and post-conditions ensuring that every function's parameters and its return value satisfy their type invariants (§3.1). This poses a unique challenge (§3.2): it is difficult to modularly generate post-conditions asserting that all borrows received by a function satisfy their type invariants when the function returns. In particular, complications arise when borrows are passed inside arbitrary data structures or returned by the function. Therefore, we chose to *assume* that the final values of borrows satisfy their type invariant from the start, and check that this assumption is valid at resolution time (§3.3).

Modularity of Borrow Resolution. In the prophetic encoding of borrows, *resolution* is the final logical step where one "learns" the prophesized final value of a borrow. In larger programs, resolution applies not only to single borrows, but also to *data structures* containing borrows. For instance, if a vector containing borrows goes out of scope, then one must resolve all the borrows it contains at once. This is a challenge for modular verification: how does one know how to resolve borrows stored in an abstract datatype verified separately? We describe how we express modular resolution predicates (§4), which are defined in terms of the public logical model of a data structure, but are proven sound with respect to its private concrete implementation.

Location of Borrow Resolution. The prophetic encoding of borrows leaves us the choice to perform resolution anywhere between the last use of a borrow and the end of its syntactic scope. Designing an adequate placement strategy is not completely obvious (§5.1). On the one hand, borrow resolution adds a new logical fact to the context, so it should ideally happen as soon as possible. On the other hand, when considering a borrow on a value whose invariant is temporarily broken, resolving the borrow too early may not leave enough time for the user to repair the invariant. We propose a strategy for determining resolve points by combining information from three simple data-flow analyses (§5.2).

Implementation. We have implemented and evaluated our contributions by extending Creusot, which we include as an artifact. In Section 6, we further discuss implementation choices, limitations, and evaluation on case studies.

2 Interaction of Prophecies with Ghost Code

Many existing deductive verification systems provide a mechanism to annotate the verified program with so-called "ghost code", program instructions that are not executed but instead help verification. Typically, one may use ghost code to construct mathematical objects that model the concrete values the executable code is working with. Ghost instructions thus typically have access to "logical" operations, on top of "ordinary" program instructions.

In this work, we consider a relatively limited form of ghost code for Creusot. In a given program, one may write `ghost!{e}` to denote a ghost expression `e`; here, `e` may read but not modify memory. If `e` has type `T` then `ghost!{e}` has type `Ghost<T>`; the Rust compiler then knows that values of the `Ghost` type can be erased during compilation (because it is declared as a "zero-sized type"). Finally, one can only access ghost values inside `ghost!{...}` blocks; elsewhere they are considered as completely opaque.

2.1 Challenges

The fact that ghost code manipulates the logical representation of data has a drastic consequence for prophecies: using ghost code one can resolve a prophecy to a value that depends on itself, creating a "causality loop":

```
1 let mut x: Ghost<bool> = ghost! { true };  // x ↦ ⊤
2 let bor: &mut Ghost<bool> = &mut x;        // x ↦ α, bor ↦ (⊤, α)
3 *bor = ghost! { ! ^bor };                  // x ↦ α, bor ↦ (¬α, α)
4 // resolve bor: assume ( ^bor == *bor )    // We deduce α = ¬α
5 proof_assert!( false );
```

Here, we first create a boolean in ghost code, and store it in the variable `x`. Then, we take a mutable borrow to this value. On line 3, we access the prophecy of the borrow to create a contradiction: the current value of the borrow is updated to contain the *opposite* of the prophecy.

This means that on the next line, we resolve the borrow, so we have both `*bor == ^bor` and `*bor != ^bor`, and we are able to prove `false` (line 5).

As we explain later in §2.2, it is easy to disallow the `^` operator inside ghost code. However, this is not enough: it is also possible to *indirectly* observe prophecies via logical equality on borrows, as shown below.

```
1 let mut x = ghost! { true };        // x ↦ ⊤
2 let b1: &mut Ghost<bool> = &mut x;  // x ↦ α, b1 ↦ (⊤, α)
3 // resolve b1                      // x ↦ α, b1 ↦ (⊤, α),  α = ⊤
4 let b2: &mut Ghost<bool> = &mut x;  // x ↦ β, b1 ↦ (⊤, ⊤), b2 ↦ (⊤, β)
5 *b2 = ghost! { ! (b2 == b1) };      // x ↦ β, b1 ↦ (⊤, ⊤),
                                                b2 ↦ (((⊤, β) ≠ (⊤, ⊤)), β)
6 // equivalent to:                   // x ↦ β, b1 ↦ (⊤, ⊤), b2 ↦ (¬β, β)
7 // resolve b2                       // We deduce β = ¬β
8 proof_assert!( false );
```

This code uses the same idea as the previous example, but leverages equality on borrows to get the value of the prophecy. Before the execution of line 5, we know two key facts: first, `*b1` and `*b2` are both equal to `Ghost(true)`, since we have not to written anything yet into `b1` or `b2`. Second, `b1` is never written to, so `^b1 == Ghost(true)`. Then, `b2 == b1` is exactly equivalent to `(^b2) == true`, hence we can deduce `false` in the same way we did in the first example.

There are several ways to solve this particular problem, but they are not all equally satisfying: we want to retain the expressivity we already have, allowing comparisons of mutable borrows while still being ergonomic and composable. The solution we chose involves changing the model of borrows, so that testing the equality of borrows does not simply boil down to testing the equality of their current and final values.

2.2 Prophetic Logical Functions

As shown in the first example, the most straightforward way to observe the prophecy value is to use the "final" operator `^` of Creusot. In order to soundly mix ghost code and prophecies, the first step is thus to forbid using this operator in ghost code. Therefore, we stratify the ghost code language: on the first hand, the purely logical level, only used in specifications, does not interact with executable code and can use the final operator. On the other hand, the ghost level can create runtime (ghost) objects but cannot use the operator. We call the first level "prophetic logic" (annotated with `#[logic(prophetic)]`), and the second "logic" (or `#[logic]`). Then, `#[logic(prophetic)]` functions are allowed to call `#[logic]` functions, but the converse is forbidden.

In order to correctly implement this idea, one must be careful with traits: if a trait method is declared with `#[logic]`, we must forbid implementations to be annotated with `#[logic(prophetic)]`.

2.3 Non-Extensional Borrows

We have seen in §2.1 that logical equality can also be used in ghost code to observe prophecies and create paradoxes. Indeed, per RustHorn's encoding, logical equality on borrows boils down to the equality of their current and final values.

One possible solution could be to *forbid logical equality on mutable borrows*. But this would come at a great cost to ergonomics: functions that use logical equality on a polymorphic type `T` would have to forbid instantiating `T` with a mutable borrow, which does not compose well.

We instead chose to make borrows *non-extensional*. We change the model of borrows to use three fields instead of two: their current and final value and a new *identity* field. This third field can be of any type, as long as the type contains an infinite number of values (*e.g.*, `int`). When creating a new borrow, we pick a fresh identity for it, making it logically distinct from existing borrows. In fact, we have the following key property: if two borrows have the same identity, then they have the same prophecy value. This property means that we cannot extract information about the prophecy by using logical equality on borrows.

This change is enough to restore soundness of ghost code. For instance, on the last example of §2.1, `b1` and `b2` each get a unique identity so comparing them always yields `false`, regardless of the prophecy of `b2`.

There is one issue, however: implementing this change as-is results in severe limitations in the expressivity of our specification language. We outline two problematic scenarios below. In both cases, one can work around the issue by replacing equalities between borrows into equality of their respective current and final value; but the result is extremely cumbersome.

- The Rust compiler regularly introduces reborrows, in places that can be surprising. For example, consider the identity function on mutable borrows.

```rust
#[ensures(result == bor)]
fn id<T>(bor: &mut T) -> &mut T  { bor }
```

  We cannot prove its intuitive function specification because the compiler inserts a hidden reborrow when returning `bor` (*i.e.*, it is replaced with `&mut *bor`), meaning that `result` and `bor` end up with different identities.
- To ease the writing of specifications using mutable borrows, Creusot allows using the `&mut x.field` syntax in specifications. For example:

```rust
#[ensures(result == &mut x.0)]
fn first_field<T>(x: &mut (T, T)) -> &mut T  { &mut (*x).0 }
```

Before we introduce the borrow identity field, `&mut x.0` in the specification is syntactic sugar for `((*x).0, (^x).0)`, allowing us to easily prove this function specification. Even better, Rust allows us to directly write `&mut x.0` instead of `&mut (*x).0` here, meaning the code and the specification read the same. Unfortunately, with an identity field, `&mut x.0` in the specification cannot be meaningfully compared with any program borrow, because the expression `&mut (*x).0` in Rust program code generates a fresh identity.

2.4 Sharing Identity for Final Reborrows

To restore the soundness of ghost code, all we need is to forbid observing prophecies. By adding a borrow identity field, we reach this goal because whenever the prophecies of two borrows differ, their identity also differs necessarily. However, this comes at a great cost of ergonomics. Our goal in this section is to refine the notion of borrow identity, allowing more borrows to have the same identity while preserving this key property. This refinement is what we call *final reborrows*.

A first observation is that if we perform a simple reborrow (*e.g.*, `&mut *bor`) and never write to the original borrow, then the prophecy of the original borrow and the reborrow are the same. We can thus safely assign them the same identity. Let us take a look at the identity function again, with the reborrow made explicit:

```
1  #[ensures(result == bor)]
2  fn id<T>(bor: &mut T) -> &mut T {  // $bor \mapsto (x, \alpha, id)$
3      let result = &mut *bor;         // $bor \mapsto (\beta, \alpha, id), result \mapsto (x, \beta, id)$
4      result
5  }                                   // We deduce $\alpha = \beta$
```

After line 3, `bor` is not modified, and is subsequently resolved when the function ends. We thus get the equality of the two prophecies α and β, so we can inherit the identity field *id*. This allows us to prove the specification.

We also want to be able to prove the `first_field` function defined earlier: we want `&mut (*x).0` to be final, but it cannot reuse the identity of `x` directly, as it could be used to mix `&mut (*x).0` and `&mut (*x).1`. Instead, when we want to mark such a reborrow as final, we introduce a special logical function, deterministic but opaque, and call it on the identity of the original borrow. In this case, the identity of `&mut x.0` would be `field0(x.id)`.

Note that in the specification, `&mut x.0` always translates to `((*x).0, (^x).0, field0(x.id))`. Since we reuse the prophecy of `x`, we can also reuse its identity.

2.5 Detecting Final Reborrows

The remaining question is to determine which reborrows are "final", *i.e.*, when we can soundly reuse the identity field when creating a borrow from another. To get this information, we developed a static analysis based on MIR.

MIR is an intermediate language of the Rust compiler, based on a control-flow graph. It only performs basic operations on values: all types are explicit, and expressions are decomposed into elementary statements (read variable, add,

v	$\in$	Ident	Variable
f	$\in$	Ident	Field
l	$\in$	$\mathbb{Z}$	Block label
func	$\in$	Ident	Function label
pl	$::=$	$v \mid *\mathit{pl} \mid \mathit{pl}.f$	Place
stmt	$::=$	$\mathit{pl} \leftarrow \mathit{pl} \mid \mathit{pl} \leftarrow \mathit{func}(\mathit{pl}\star) \mid \mathit{pl} \leftarrow \texttt{borrow}(\mathit{pl})$	Statement
term	$::=$	$\texttt{return} \mid \texttt{goto}\ l \mid \texttt{switch}\ \mathit{pl}\ l\star$	Terminator
bb	$::=$	$l : \mathit{stmt}\star\ \ \mathit{term}$	Basic block
body	$::=$	$\mathit{func} : \mathit{bb}\star$	Function body

Fig. 1. Simplified description of the MIR intermediate language

call function, *etc.*). It is used as an intermediate step to generate executable code and to perform various program analyses.

Description of MIR To describe the analysis, we use a simplified version of MIR, whose grammar is given in Fig. 1. MIR is made of functions having a body (body) identified by labels func, consisting of a control flow graph whose nodes are called basic blocks (bb), identified by labels l. In each basic block, a succession of statements (stmt) execute in order, until we reach a terminator (term). The terminator can end the function execution (`return`), jump unconditionally to another block (`goto`) or make a conditional jump (`switch`). There are special local variables that contain the parameters and the return value of the function.

Statements act on places (pl): a place is an expression that identifies a location in memory. A place corresponds either to a local variable v, the dereference of mutable borrow stored in another place ($*\mathit{pl}$)[2], or a subfield of another place ($\mathit{pl}.f$). For example, if `x` has type `(i32, i32)`, the place `x.0` refers to the first integer of the pair.

A statement can be either an assignment $\mathit{pl} \leftarrow \mathit{pl}$, which copies the content of the right-hand side into the left-hand side; a function call[3] $\mathit{pl} \leftarrow \mathit{func}(\mathit{pl}\star)$; or a borrow $\mathit{pl} \leftarrow \texttt{borrow}(\mathit{pl})$, which creates a borrow of the right-hand side place, and writes the result into the left-hand side.

Program locations λ describe a point in a function body: there is a program location before each statement and before each terminator. The label of a basic block is the first program location of the block.

The Final Borrows Analysis For each program location λ, we compute a set of places Φ_λ. A place pl is in Φ_λ if it contains a least one dereference, and we

[2] In real MIR, places can dereference other kinds of pointers, but we ignore this here.
[3] In real MIR, function calls are terminators because they have several return points. We ignore this complication, because Creusot doesn't support exceptional returns.

$$
\begin{array}{ll}
{}_{\lambda_1}\ pl_1 \leftarrow pl_2\ {}_{\lambda_2} & \vdash \Phi_{\lambda_1} = (\Phi_{\lambda_2} \setminus (C(pl_1) \cup C(pl_2))) \cup D(pl_1) \\
{}_{\lambda_1}\ pl \leftarrow \mathit{func}(pl_1, \ldots, pl_n)\ {}_{\lambda_2} & \vdash \Phi_{\lambda_1} = (\Phi_{\lambda_2} \setminus (C(pl) \cup \bigcup_{i=1}^{n} C(pl_i))) \cup D(pl) \\
{}_{\lambda_1}\ pl_1 \leftarrow \texttt{borrow}(pl_2)\ {}_{\lambda_2} & \vdash \Phi_{\lambda_1} = (\Phi_{\lambda_2} \setminus (C(pl_1) \cup C(pl_2))) \cup D(pl_1) \\
{}_{\lambda_1}\ \texttt{return} & \vdash \Phi_{\lambda_1} = \bigcup_{v \in vars} D(v) \\
{}_{\lambda_1}\ \texttt{goto}\ l & \vdash \Phi_{\lambda_1} = \Phi_l \\
{}_{\lambda_1}\ \texttt{switch}\ pl\ l_1 \cdots l_n & \vdash \Phi_{\lambda_1} = \bigcap_{i=1}^{n} \Phi_{l_i}
\end{array}
$$

Fig. 2. Rules of the final borrows analysis

statically know that the mutable borrow corresponding to the last dereference has its current value equal to its prophecy. If pl contains only one dereference, this means that it is in Φ_λ if the value of the corresponding borrow does not change until it expires. A reborrow of place pl immediately before program location λ is considered final if $pl \in \Phi_\lambda$.

The analysis finds the greatest solution of the system of set equations described in Fig. 2, by computing a greatest fixed point iteratively. This is a backward data-flow analysis: when a borrow is resolved, we know its current value is equal to its prophecy, hence the corresponding places belong to Φ for these program locations. This information is propagated backward through statements that preserve this property: in "${}_{\lambda_1}\ \mathit{stmt}\ {}_{\lambda_2}$", λ_1 denotes the program location just before *stmt*, and λ_2 the location just after.

The analysis depends on two auxiliary sets of places associated with a given place pl:

- The set $C(pl)$ of places that conflict with pl. Two places conflict if they have the same root variable, and the memory they give access to overlap.
- The set $D(pl)$ of places pl', such that pl' is a subplace of pl, and pl' contains one more dereference $(*)$ than pl. That is always a subset of $C(pl)$.

We now focus on the effects of the assignment statement. The effects of $pl_1 \leftarrow pl_2$ on the set are to remove $C(pl_1)$, remove $C(pl_2)$, and add $D(pl_1)$. The set $C(pl_1)$ is removed, because the current value of pl_1 just changed, so we cannot statically guarantee that it is equal to its prophecy. Any place in conflict with pl_1 might also have had their value change, which is why we remove the whole $C(pl_1)$ set. Now we might wonder why we also do the same for pl_2, which is not written to by this statement. The issue here is that we cannot statically track the current value of borrows inside pl_2 anymore: writes to $*pl_1$ might affect the value in $*pl_2$. For this reason, we remove the set $C(pl_2)$ as well.

Finally, note that writing into pl_1 destroys its previous content, resolving the borrows it contains. In particular, writes into $*pl_1$ after the statement do not affect the value of $*pl_1$ before the statement. So we can add $*pl_1$ to the set, and in fact we can add $D(pl_1)$.

In order to better understand what is happening here, we come back to the example `first_field` that we already mentioned above. When annotated with sets of locations Φ, its MIR code is (we omit places that do not appear in the program):

```
// Φ = {}
result ← borrow (*x).0
// Φ = {*x, (*x).0}
return
```

At the point right after the borrow, the place `(*x).0` is in the set, so the reborrow is final, and we can prove the specification of the function.

3 Interaction of Prophecies with Type Invariants

In Rust, some types have a semantic interpretation dictating that a certain predicate holds true for all publically visible values. A typical example is the type `HashMap` of hash tables, implicitly using as an internal invariant the fact that each mapping is stored in a bucket corresponding to its hash value. This can be specified using type invariants, a mechanism to associate a logical predicate to a type. We give a brief description of the support of this feature in our version of Creusot (§3.1). While type invariants exists in many deductive verification tools, when used in Creusot, there are unique challenges related to the prophetic encoding of mutable borrows (§3.2). The notion of *prophetic invariants* addresses these challenges (§3.3). We finally give an illustrative example in §3.4.

3.1 Type Invariants in Creusot

Our version of Creusot allows users to attach type invariants to their types by implementing the `Invariant` trait and defining the predicate `invariant`. For example, we can define the type `Even` of even integers as follows:

```
struct Even(u64);
impl Invariant for Even {
    #[logic] fn invariant(self) -> bool { self.0 % 2 == 0 }
}
```

To determine the invariant of a particular type, we do not only take into account implementations of the `Invariant` trait but also the structure of the type itself. We see why this is necessary by considering the type `Option<Even>`: intuitively, values such as `None` and `Some(Even(4))` should be allowed, but not `Some(Even(3))`. This type should have the expected invariant regardless of whether there is an explicit trait implementation. Hence, we define the invariant of a type as the conjunction of a user-defined part and a structural part, which is derived automatically based on the type's definition. Here are some examples of types and their corresponding invariants[4]:

Type of `x`	Invariant
`Option<Even>`	`match x { None => true, Some(y) => invariant(y) }`
`(Even, Even)`	`invariant(x.0) && invariant(x.1)`
`Box<Even>`	`invariant(*x)`

For the expressivity of our approach, it is important to allow a type invariant to be temporarily broken. For example, if a hash table maintains in a separate field the number of entries in the table, it is natural to use a type invariant to guarantee that this field contains the right value. When an element is added to the table, we first need to update the table itself, and then increment the counter: in between, the invariant is broken.

Hence, we only enforce type invariants at *function boundaries* by automatically adding pre- and post-conditions to every function based on its parameter and return types. Each parameter corresponds to a pre-condition stating that the type invariant of the argument holds, and the return type corresponds to a post-condition for the invariant of the return value. This scheme ensures that invariants hold across function boundaries, complementing the principle of modular verification.

3.2 Challenge

While type invariants are well understood in principle and supported by many verification systems, their implementation in Creusot presents a unique challenge in their interaction with prophecies. In particular, we have to consider what it means to take a mutable borrow to a value with a type invariant, as demonstrated by the following example:

```
1  let mut x = Even(2);
2  let bx = &mut x;       // bx ↦ (Even(2), α),  x ↦ α
3  take_even_bor(bx);
4  assert!(invariant(x));
```

On line 2, we create a mutable borrow `bx` borrowing `x`, which has a type invariant stating that its value must be even. The borrow is subsequently passed to the function `take_even_bor`. Following the general principle that type invariants are enforced at function boundaries, we should be able to prove the assertion on line 4 stating that the new value contained in variable `x` satisfies its type invariant.

As mutable borrows in function parameters act similarly to function outputs, one conceivable solution is adding post-conditions stating that the invariants of the parameters' prophecies hold. However, this is unsatisfactory as demonstrated by the following function:

[4] The definition of the structural type invariant of mutable borrows is somewhat surprising: we delay its definition in §3.3, where we discuss it more in detail.

```
fn pair_bor_mut<T>(p: (&mut T, &mut T), take_first: bool) -> &mut T {
    if take_first { v.0 } else { v.1 }
}
```

Finding a post-condition for the prophecies contained in the parameter `p` presents several challenges:

– Structural invariant derivation must now distinguish two kinds of invariants, one for pre-conditions using the current value of mutable borrows, and one for post-conditions using final values. For example, the parameter `p` would get the pre-condition `invariant(*p.0) && invariant(*p.1)` and the post-condition `invariant(^p.0) && invariant(^p.1)`. This distinction is less intuitive for borrows of borrows. It is not clear in general whether the post-condition for `x: &mut &mut T` should be `invariant(^(*x))` or `invariant(^(^x))`.
– In our example, one of the borrows will not be resolved inside the function, and thus we cannot prove its prophecy invariant. Here, the post-condition we would want is something like `invariant(^result) ==> invariant(^p.0) && invariant(^p.1)`.

In conclusion, the naive approach of adding pre- and post-conditions to functions is insufficient to check that type invariants are maintained when using mutable borrows in Creusot. We need another technique to generate these verification conditions.

3.3 Prophetic Type Invariants for Mutable Borrows

How should we specify that `take_even_bor` reestablishes the type invariant of the value pointed to by `x` before returning? This is important for the caller of `take_even_bor`, which needs this information to continue using the value contained in the borrowed place.

To overcome this issue, we introduce the notion of “prophetic invariants”: prophecies now additionally carry a promise that the eventual final value satisfies its type invariant. We achieve this by *assuming* the invariant of the prophecy when creating a mutable borrow (*i.e.*, adding an axiom for the invariant). Furthermore, we define the type invariant of a type `x: &mut T` as `invariant(*x) && invariant(^x)`, combining the invariants of both the current and the final value of a mutable borrow. Using this trick, the creator of the borrow knows that the new value of the borrowed place satisfies its invariant from the fact we *assumed*.

Of course, this assumption comes at a cost: we need to check that, indeed, the final value of the borrow satisfies its type invariant. This check occurs when we know the actual value of the final value, *i.e.*, at resolution time: just before resolving a borrow, we create a new verification condition stating that the current value of the borrow satisfies its type invariant. This new verification condition acts as the hypothetical post-condition we cannot automatically generate for `pair_bor_mut`.

There is one final problem with this approach: when defining a type invariant, we are not required to prove that the invariant is *inhabited* (*i.e.*, there exist values satisfying the type invariant). However, when creating a borrow, we prophecize a value satisfying the type invariant, hence assuming that the type invariant is inhabited. Thus, we could declare an uninhabited invariant, create a temporary local value (which does not need to satisfy the type invariant), create a borrow of this value and thus unsoundly deduce falsity because the prophecy should satisfy the type invariant.

Let us search for a solution to this problem. The naive approach of requiring inhabitedness of type invariants is not an option, as, for parameterized types, inhabitedness may depend on properties of type parameters. We chose instead to reestablish the type invariant of a place before borrowing it. This way, we know that the type invariant is inhabited since the borrowed place satisfies it.

3.4 Example

The following example illustrates the concept of prophetic invariants:

```
1 fn call_pair_bor_mut(mut x: Even, mut y: Even) {
2     let bx = &mut x; // $bx \mapsto (x,\alpha)$, $x \mapsto \alpha$, assume invariant($\alpha$)
3     let by = &mut y; // $by \mapsto (y,\beta)$, $y \mapsto \beta$, assume invariant($\beta$)
4     let b = pair_bor_mut((bx, by), false); // $b \mapsto (y,\beta)$
5     // check the invariant of $y$, resolve $b$
6     assert!(invariant(x) && invariant(y)); // provable
7 }
```

For each mutable borrow bx and by, we assume the invariants of the prophecies α and β, respectively. These assumptions let us immediately prove the assertion on line 6, as the borrowed variables are assigned the prophetic values. To ensure that the prophecies actually satisfy the invariants, additional proof obligations are inserted whenever a mutable borrow is resolved. In the example, one of the borrows passed to `pair_bor_mut` is resolved inside the called function and the other borrow is resolved in the caller, depending on the boolean parameter `take_first`. As `take_first` is `false`, the borrow bx is resolved in `pair_bor_mut` while by is resolved in `call_pair_bor_mut` under its new name b. When b is resolved on line 5, we thus have to prove the invariant of its current value y before we learn $\beta = y$. This fact is trivially obtained from the post-condition generated for the invariant of the return value of `pair_bor_mut`. Importantly, this reasoning does not depend on the specification of `pair_bor_mut`: Creusot does not know that the value of b is (y, β), it simply knows this is a mutable borrow satisfying its type invariant.

There is an interesting consequence of prophetic invariants: because invariants of prophecies are checked at the points where they are resolved, resolution becomes crucial for the soundness of prophetic invariants. We must ensure that final values of mutable borrows always satisfy their invariants. This is guaranteed by the algorithm determining at which point to resolve each prophecy (§5).

4 Modular and Sound Resolution of Borrows

When using the prophetic encoding of borrows, *resolution* plays a central role, effecting the transfer of ownership back to the lending variables. It is natural to extend the notion of resolution beyond merely borrow types (`&mut T`) to *any* type, allowing us to recover the prophecies in a value regardless of how deeply nested they are. For example, we could define the resolution of a vector `v` as:

```
forall<i : Int> 0 <= i && i < v.len() ==> v[i].resolve()
```

Thanks to this definition, when a vector of borrows `Vec<&mut i32>` (or even a vector of vectors of borrows, *etc.*) is resolved, we recover the information of the equality of each prophecy with the current value of the corresponding borrow.

When modularly verifying code from different libraries, library authors should thus be able to declare what the resolution predicate is for types they define (especially if their implementation is private). In prior work in Creusot [6], this was achieved through a *trusted* trait `Resolve` that library authors could implement.

Challenge A major limitation of this approach is that nothing prevents a user from providing an unsound implementation of `Resolve` (*e.g.*, `false`). At resolve points, Creusot simply assumed user-defined `Resolve` predicates to hold. In other words, library authors routinely extended the trusted computing base of the tool. To make things worse, an unsoundness in resolution is particularly pernicious: it can appear anywhere in client functions and is not visible in the source since the assumptions of resolutions are automatically inserted by the tool.

We solve this by introducing a verification condition for user definitions of `Resolve`, related to an automatically generated structural definition of `Resolve`. This new condition ensures users may only use custom instances to perform information hiding, or reformulation in more easily usable terms.

Structural vs User-defined Resolution. Resolution is a structural property, so one may wonder if user-defined predicates are necessary at all. First, they are needed for modularity reasons: a resolution predicate should only refer to the public interface of a type (*e.g.*, its logical model) while a structural implementation would, by definition, expose details of its implementation. Second, defining resolution through direct recursion will often produce a predicate which is suboptimal for verification. Consider a binary search tree: we could write `resolve` recursively, or we could rely on abstractions already established for other parts of the proof, such as a logical `get` function. We could then define `resolve` as:

```
  forall<k:_, v:_> tree.get(k) == Some(v) ==> k.resolve() && v.resolve()
```

This definition makes it easy for provers to reason about the resolution of key-value, while keeping the concrete definition of trees entirely abstract.

Sound User-defined Resolution. We still need to ensure the soundness of user-defined resolution predicates. Our solution is to allow such predicates, but, at definition time, require the user to prove them sound *with respect to a built-in* *structural resolution predicate*. This proof must be discharged once by the implementor of a custom predicate, as part of the implementation of the corresponding type. Then, clients of the type can simply use the user-defined predicate.

Specifically, we require that a user-defined `resolve` must be implied by a new `structural_resolve` predicate, and defined automatically as follows:

Type of `x`	Structural Resolution
`(T1, T2)`	`x.0.resolve() && x.1.resolve()`
`&mut T`	`^x == *x`
`Box<T>`	`(*x).resolve()`

Note that this definition is *shallow*: `structural_resolve` is not recursively defined in terms of itself, but instead uses the resolution predicates for individual components of the type.

Using `structural_resolve`, we can state the correctness criterion for a custom `resolve` by adding a lemma function `resolve_coherence` to our `Resolve` trait:

```
pub trait Resolve {
                                  #[logic(prophetic)]
                                  #[requires(inv(self))]
                                  #[requires(structural_resolve(self))]
  #[logic(prophetic)]             #[ensures((*self).resolve())]
  fn resolve(self) -> bool;       fn resolve_coherence(&self);
}
```

Implementations of `Resolve` are required to demonstrate that this lemma holds, ensuring that no unsound definition can be admitted. User definitions can thus only be used for information hiding: for instance, a bogus definition of `resolve` as `false` would require proving `true ==> false`, and thus be rejected.

5 Timely Resolution of Borrows

The prophetic encoding of prophecies crucially relies on the insertion of the *resolution* assumption, for each borrow, as soon as we are certain that the borrow will not be used.

Given a value, there may be several valid "*resolve points*" (*e.g.*, immediately after its last use, or later when it goes out of scope). In our earlier examples, we have each time indicated informally the resolve point we consider. But of course, in practice, a verification tool must compute and choose valid resolve points automatically, without relying on any user annotations. How should resolve points be determined? The existing literature on RustHorn-style prophecies does not yield a satisfactory answer! RustHornBelt [15] formalizes borrow resolution but does not prescribe when it should happen. RustHorn [16] and Creusot's authors [6, §3.2] say that a "borrow is resolved at the moment it is dropped".

However, as we explain in §5.1, this is not satisfying because it would lead to unprovable verification conditions for legitimate programs. Choosing *where* in the code we should insert resolution assumptions is in fact a subtle problem. Next, in §5.2, we detail the algorithm we use to solve this problem.

5.1 Challenges

Resolve points need to satisfy several constraints, which we detail next. Some of them are important for soundness, while others (of lower priority) are needed to make some verification conditions provable.

Resolution of a variable must happen when the value it contains will no longer be used. Otherwise, a future use could change the value of a borrow contained in the variable, and we would assume that the prophecy of the borrow equals a value which is not the final value of the borrow. As a result, provers would assume a wrong new value for the borrowed place, which is unsound.

A borrow should be resolved before its lifetime ends. The *lifetime* of a borrow is a time span during the execution of the program during which the borrow is accessible, but the borrowed variable is temporarily inaccessible. It corresponds to a code span automatically inferred by the compiler in order to check for safety, and it is known by Creusot. It is important that a borrow is resolved before its lifetime ends for two reasons: first, if a borrow is not resolved at the end of its lifetime, we do not have any information about the new value of the borrowed place (which is now accessible), a loss in precision that can lead to unprovable verification conditions. Second (and more importantly), as explained in §3, we assert the validity of the type invariant of borrows before resolving them, and this assertion is important for the *soundness* of the tool. If we miss the resolution of a borrow, we may have broken the type invariant of the value contained in the borrow while, as we explain in §3, we do assume it holds.[5]

The lifetime of a borrow can be shorter than its lexical scope. Consider the following *valid* Rust function:

```
fn f() {
  let mut x = 1;
  let b = &mut x;
  // [b] is still in scope, but its lifetime has ended.
  // Hence [x] can be accessed.
  x += 1;
```

[5] The reader may argue that Rust's type system does not enforce the absence of memory leak, and that *e.g.*, `std::mem::forget` or cycles of reference-counted pointers may leak mutable borrow without giving us the opportunity to resolve it. This is true: our system may miss resolutions (and hence precision) because of memory leaks, but this cannot cause unsoundness. The reason is that when a borrow is leaked by using one of these mechanisms, it has to be passed to a library function (*e.g.*, `std::mem::forget` or `std::rc::Rc::new`) which does enforce the type invariant.

```
  proof_assert!(x == 2);
}
```

In this function, we create a borrow `b` of local variable `x`, and then we write to local variable `x`. This write is allowed because the lifetime of the borrow has ended when the write happens. However, note that at this point of code, the variable `b` which contains the borrow is still in scope: this illustrates a feature of Rust called "non-lexical lifetimes", where the lifetime of a borrow can be shorter than its lexical scope [14].

This is challenging for determining resolution points, because a natural candidate for the resolve point of a variable is when it leaves its scope (*i.e.*, when it is dropped, as RustHorn [16] and Creusot's authors [6, §3.2] suggest). This is not a valid choice in general because resolution needs to happen before the end of the lifetime, which is sometimes before the value is dropped. In the example above, this would mean that the final `proof_assert` would not be provable, because it depends on the resolution of `b`.

We should not resolve variables when they are uninitialized. In Rust, local variables can be declared uninitialized and later initialized. One can also "un-initialize" a local variable by moving the ownership of its content somewhere else (*e.g.*, by passing it as a parameter to another function). The end result is that, at some program points, some local variables may not be initialized. Of course, we should never resolve such variables, which contain nonsensical data.

This is made more complicated by the fact that we may not know statically whether a variable is initialized or not:

```
fn f(x: Box<&mut i32>) {
  if <some boolean value> {
    g(x) // Move content of x to function f
  }
  // x is initialized or not, depending on whether we took the branch
}
```

Hence, in this case, we should resolve `x` immediately before the join point, in the implicit `else` branch. Fortunately, the Rust compiler rejects any use of a variable that is not statically known to be initialized: if, at a control flow join point, we lose the track of the initializedness, then it is sound to resolve the variable immediately before the join point because it is also necessarilly dead.

There is, however, a subtlety: Rust tracks initializedness by *places* rather than local variables. That is, it is possible to move the value out of the second component of a pair, leaving the other component initialized and available for any kind of use. In this case, we are not able to establish the type invariant for the local variable (part of its value is undefined), and thus resolving the whole variable is impossible. For example, consider the following piece of Rust code:

```
#[ensures(*x.0 == ^x.0 && result == x.1)]
fn h(x: (&mut i32, Box<&mut i32>)) -> Box<&mut i32> {
  return x.1
}
```

Here, the function `h` returns the second component of the parameter `x` by *moving* it to the caller. Hence, when the function returns, `x.1` is undefined, but `x.0` still needs to be resolved.

Literature on prophetic models for mutable borrows [16,15,6] only consider resolving local variables at once. But this observation calls for a mechanism for determining resolve points for each *place* (in the sense of the MIR intermediate language, see Fig. 1) instead of individual local variables. All the constraints we evoked above in this section for resolving variables also applies to places.

As much as possible, we should not resolve borrows if they are frozen. To understand this constraint, consider the following example (the type `Even` and its type invariant are defined in §3.1):

```
fn f(e: &mut Even) {
  let b = &mut e.x;
  *b = 2;
}
```

In function `f`, where should we resolve `e`? A simple answer might be that we resolve `e` as soon as we know it will never be used again (*i.e.*, it is *dead*). That is, we could resolve `e` just after `b` is created. This is not a good idea: at this program point, the final value of `b` is not known, hence the current value of `e.x` (which is equal to the final value of `b`) is not known either. Therefore, we cannot establish at this program point that the type invariant holds for `e`. This is problematic because (recall §3) we must prove the type invariant before resolution.

The problem here is that we are trying to resolve `e` when it is still *frozen* (*i.e.*, `e` is borrowed, with a lifetime which is still active): in general, we should refrain from resolving places which are frozen, because we may not know anything about their value. In the case of the above example, this means that `e` should be resolved at the end of the function, after `b` is resolved. At this program point, we know the value of the prophecy of `b`, so we know the value of `e`, and have what is needed to reestablish the type invariant.

Note, however, that this constraint cannot always be satisfied because the lifetime at which a borrow is frozen can exceed its scope. As an example, consider the following Rust function:

```
fn g(e: &mut Even) -> &mut i32 {
  &mut e.x
}
```

This function *returns* a borrow inside its parameter `e`. The lifetime of the returned borrow depends on the calling code, and exceeds the body of the function: as a result, the parameter `e` is frozen even after it leaves its scope, and there is no program point where we could resolve it while satisfying this constraint.

In this case, it is still important (for soundness) to resolve `e` somewhere, because the type invariant of the borrowed place (of type `Even`) needs to be reestablished. Hence, we decide to resolve it as late as possible, *i.e.*, when it leaves its scope at the end of the function. This leads to an unsolvable goal; this is expected because nothing prevents the caller of this function from writing an odd integer in the returned borrow, thus breaking the invariant for `e`.[6]

5.2 Algorithm for Determining Resolve Points

We now describe the procedure we use to address the challenges described in §5.1. It works on the Rust's compiler MIR intermediate representation we already described in §2.5, and relies on three data-flow analyses:

- A per-place initializedness analysis, determining for each place and each program point whether we know statically that this place is initialized. A place can be uninitialized either because the underlying local variable has not yet been written to, or because its value has been moved elsewhere. This analysis is already implemented in the Rust compiler (it is forbidden to use the value of a place which is not initialized); we reuse it directly. We note $I(\lambda)$ the set of places which are definitely initialized at program point λ.
- A per-place liveness analysis, determining for each place and each program point whether we know statically that the value it contains will not possibly be used by the program in the future. We note $\Lambda(\lambda)$ the set of places for which we *cannot* statically guarantee they will not be used after program point λ. We say these are *live* places at program point λ.
- A per-place frozenness analysis, determining for each place and each program point whether that place is *frozen* because there is a borrow of that place (or part of it) whose lifetime is still active. This analysis reuses information computed by the *borrow checker*: the part of the Rust compiler that checks that borrows are used in a way that does not violate ownership and aliasing restrictions; determining frozenness is central to its operation. We note $F(\lambda)$ the set of places which are frozen at program point λ.

Places containing a dereference of a mutable borrow (*i.e.*, corresponding to memory accessed through a mutable borrow) correspond to memory that is given back to the lender when the lifetime dies. Hence, the only situation where we resolve them is before they are overwritten. In particular, initializedness, liveness or frozenness play no role for these places. Therefore, they are not considered by these analyses: we only consider places which do not contain the dereference of a mutable borrow[7].

Given the result of these analyses, we compute, for each program location λ, two sets of places:

$$N(\lambda) = I(\lambda) \cap (\Lambda(\lambda) \cup F(\lambda)) \qquad\qquad R(\lambda) = I(\lambda) \setminus N(\lambda)$$

[6] In order to allow verification of such functions, we could take inspiration from Prusti [1,2] and support *pledges*, assertions that need to be valid only when a lifetime ends. This is out of the scope of this paper.

[7] Places corresponding to array indexing (not mentioned in Fig. 1, but existing in the real implementation) are also ignored because we cannot statically distinguish array slots. The remaining places are called "move paths" by the Rust compiler developers, and some support is available in the compiler to perform analyses on them.

Any initialized place is either in $R(\lambda)$ or in $N(\lambda)$. The general idea of our algorithm for the insertion of resolution is to make sure that, at program location λ, places in $R(\lambda)$ have already been resolved, and that values in places of $N(\lambda)$ will either be resolved or moved somewhere else. Hence, when a place enters $R(\lambda)$, we should resolve it, and when a place leaves $N(\lambda)$, then either it has been moved, or we should resolve it. More precisely, we are inserting resolve statement at the following program points:

- At function entry, we resolve any places in $R(\lambda)$.
- When a local variable x leaves its scope (at function exit or when leaving a nested scope), we resolve any place in $N(\lambda)$ which is a subplace of x.
- We decompose statements (including function calls) into two steps: the first step does a pure computation (*e.g.*, typically it evaluates the right-hand side of an assignment), while the second step performs some side effects (*e.g.*, a place is assigned with the value computed during the first step). We note λ_1, λ_2 and λ_3 the program points before, between and after these two steps, respectively.
  Before the statement, we resolve places in $N(\lambda_1) \cap R(\lambda_2)$. Then, we consider the second step: if *pl* is a place written by the statement second step, we resolve it before the statement if $pl \in N(\lambda_2)$ and after the statement, if $pl \in R(\lambda_3)$.
- Suppose λ is the last program point of basic block jumping (*via* a `goto` or a `switch` terminator) to λ', the first program point of a basic block. Along the corresponding edge in the control flow graph, we resolve places in $N(\lambda) \setminus N(\lambda')$.

6 Discussion

6.1 Evaluation

In order to validate the ergonomics of our system, we migrated Creusot's test suite to our extension. We were able to make all the tests pass, without any significant regression, despite the additional soundness restrictions described in §2 and §4. We have not observed any example where the placement of resolution as described in §5 were unsatisfying.

Thanks to the new support of type invariants, we were able to simplify the implementation of non-trivial data structures (*e.g.*, red-black tree maps, hash tables, binary decision diagram) implemented as part of the test suite: we successfully used type invariants to remove the manual threading of invariant that was used before this work. Additionally, we used type invariants to simplify specifications of iterators in Creusot [5].

We were mostly satisfied by all these uses of type invariants. The main inconveniences we encountered was the need to guard, in specifications for iterators, some quantifiers by the corresponding type invariant; and, in the proof of red-black trees, the fact that not all invariants hold when recursively calling internal functions. In this case, we placed the whole tree type in a top-level wrapper type, which makes it possible, thanks to type invariant, to provide to the client a specification free of internal invariant.

6.2 Limitations and Future Work

We believe our extensions of Creusot and RustHorn's prophetic translation to be sound, but we have not yet established this formally. Extending RustHornBelt with our new features would be a natural way to tackle this question.

From an expressivity perspective, our type invariants are less powerful than the ones in Why3 [19] or Dafny [18]. In those systems, type invariants effectively define subset types: values satisfy their invariant even at the logical level (specifications and assertions). In our work, this is not the case: type invariants only insert conditions in pre- and post-conditions of functions. Subset types would be interesting but challenging to add to Creusot: the underlying SMT solvers used by Creusot assume all types to be inhabited, but subset types break this property (in particular, the Rust standard library includes an empty type, which Creusot sees as a type with the invariant $\bot$).

Our ghost code is also of limited expressivity, in that it does not carry ownership: a ghost value of type `Ghost<T>` is always duplicable. Allowing ghost linear values would be more expressive, following ideas from Verus [12]. This would be a non-trivial extension of our current work, but remaining challenges should be orthogonal to prophecies: we expect the present work to smoothly apply to this more expressive form of ghost code.

Our type invariants determine verification conditions that must be checked at the resolution point of a borrow to any value of the corresponding type. It would additionally be convenient to allow specifications to declare that an assertion must be satisfied at the resolution of a specific borrow, on a case-by-case basis (see *e.g.*, the last example in §5.1). This is easily expressed using Prusti's *pledge* mechanism, but currently not possible in Creusot.

7 Related Work

To our knowledge, Creusot [6] is the only deductive verification tool for Rust that uses a prophecy-based encoding of mutable borrows. Here, we also compare with work on Rust verification based on different techniques, and deductive verification tools with functionalities related to the challenges we presented.

Verification of Rust code using a prophetic encoding of borrows. Our work is based on Creusot [6] and extends it beyond its state of the art. Ghost code was briefly mentioned in earlier works as a feature of Creusot [6,5], but one can check that these works rely on an *unsound* implementation of ghost code (they incorrectly accept the second example of §2.1). Type invariants were also mentioned in the earlier work on iterators [5], but in the actual implementation these to-be "type invariants" were in fact manually threaded through the code. Modular resolution predicates written by library authors were part of the trusted computing base [6, §4.2]; in our work, they soundly lead to proof obligations that need to be discharged by the user. Finally, resolution in Creusot was triggered at the moment a variable stops being used; this does not account for partially uninitialized values and type invariants, as opposed to our approach.

RustHorn [16] first introduced a prophetic translation of borrows by translating a core subset of Rust to constrained Horn clauses. It targets the automated verification of unannotated programs and does not handle specifications, loop and type invariants, or other features for functional correctness verification.

RustHornBelt (RHB) [15] is a formalization of the RustHorn approach in Coq. It defines a core language for Rust on which it establishes the soundness of the prophetic reasoning principles. Notably, RHB's core language has no notion of "place" as they appear in Rust's MIR intermediate representation; resolution of borrows is only performed on variables. This is a major simplification compared to our work which considers the full extent of MIR. RHB allows customizing resolution predicates in a sound manner, but this requires unfolding the RHB's semantic model of Rust types, something that we wish to avoid. Unlike Creusot, in RHB resolution points are manually triggered by users as a manual reasoning step. In the context of an automated verification tool this would be extremely cumbersome. Finally, RHB does not include ghost code or type invariants.

RefinedRust [8] does not directly implement RustHorn's prophetic encoding, but takes inspiration from it and adapts it for the verification of unsafe Rust code. In RefinedRust, a borrow is interpreted as the pair of its current value and a "borrow name" reminiscent of a prophecy variable. When a borrow is resolved, a Separation Logic assertion is produced relating the borrow name to its final value. RefinedRust works on a core calculus much closer to MIR than RustHornBelt; in particular, RefinedRust works on "places", just like our work. RefinedRust does not support ghost code, but supports type invariants. Like us, it needs to account for their interaction with borrows; because they are working in Separation Logic, they are able to track borrows for which an invariant needs to be restored using a dedicated resource. In contrast to our work, RefinedRust is a foundational framework embedded into Coq using the Iris separation logic. This allows for greater expressivity and the ability to reason about unsafe code. However, RefinedRust supports a more restricted set of Rust features compared to Creusot and supports less powerful proof automation.

Deductive verification of Rust code using other approaches. Verus [12] is a deductive verification tool with a design closely related to Creusot. Verus relies on a simpler functional translation of code written in a verification language much closer to Rust. It does not use prophecies and only supports mutable borrows passed as function parameters, not in return positions (this rules out functions like `index_mut`). Verus has type invariants and expressive ghost code; their interaction with borrows is much simpler than in our case because of their restricted encoding. Interestingly, the Verus developers seem to be considering adopting a prophecy-based encoding of borrows[8]: the present work should be relevant.

[8] https://github.com/verus-lang/verus/discussions/35#discussioncomment-4925078

Aeneas [9] translates Rust programs to purely functional programs that can be loaded and verified in a proof assistant. Mutable borrows are translated using "backward functions", which appear to be closely related to prophecies. Aeneas' encoding is very general on paper, but implementation is still ongoing to handle certain more complex situations that Creusot and our work handle (*e.g.*, borrows in types, nested loops, nested borrows in signatures). Aeneas currently has no dedicated specification language or features such as ghost code or type invariants: program verification tasks are carried out using the proof assistant's usual reasoning tools.

Prusti [1] translates Rust programs to the Viper [17] verification language, by encoding Rust's ownership discipline (including borrows) into affine capabilities [3]. Compared to Creusot's prophetic encoding and its model in RHB, the soundness of Prusti's encoding is easier to justify (but puts more burden on the verification infrastructure which needs to handle a flavor of Separation Logic). We believe this is why Prusti supports type invariants without many of the challenges we faced. On the other hand, Prusti does not yet support ghost code or mutable borrows in their full generality, though we don't expect them to face the same soundness challenges performing these extensions. In Prusti, specifications involving borrows use "pledges", assertions deferred to the end of a lifetime. This is similar in spirit to Creusot post-conditions that refer to the "final" value of a borrow.

Verification using borrows in SPARK/Ada. SPARK, a verification enabled subset of Ada, has recently been extended with the ability to reason about pointers using borrows [10,7]. SPARK's logical encoding of borrows has similarities with RustHorn's prophetic encoding: a borrow is represented by its current value and a *borrow relation* which ties the borrow's value to its lender—similar to how the prophetic translation allows collecting facts relating a borrow's current and final value. Specifications about borrows are expressed using *pledges* similar to Prusti's. SPARK borrows are however much less expressive than Rust borrows; the location they alias must be statically known, and they cannot be stored in records. SPARK by design keeps clear of many of the challenges that we tackled in our work.

Acknowledgments. This research was partly supported by the Décysif project funded by the Île-de-France region and by the French government in the context of "Plan France 2030".

References

1. Astrauskas, V., Bílý, A., Fiala, J., Grannan, Z., Matheja, C., Müller, P., Poli, F., Summers, A.J.: The Prusti project: Formal verification for Rust. In: NASA Formal Methods, LNCS, vol. 13260 (2022). https://doi.org/10.1007/978-3-031-06773-0_5
2. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust types for modular specification and verification. Proc. ACM Program. Lang. **3**, 147:1–147:30 (2019). https://doi.org/10.1145/3360573

3. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging rust types for modular specification and verification. Proc. ACM Program. Lang. **3**(OOPSLA) (Oct 2019). https://doi.org/10.1145/3360573, https://doi.org/10.1145/3360573
4. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5674, pp. 23–42. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_2, https://doi.org/10.1007/978-3-642-03359-9_2
5. Denis, X., Jourdan, J.H.: Specifying and verifying higher-order Rust iterators. Lecture Notes in Computer Science, vol. 13994, pp. 93–110. Springer. https://doi.org/10.1007/978-3-031-30820-8_9, https://hal.science/hal-03827702
6. Denis, X., Jourdan, J.H., Marché, C.: Creusot: a foundry for the deductive verification of Rust programs. Lecture Notes in Computer Science, Springer Verlag, https://hal.inria.fr/hal-03737878
7. Dross, C., Kanig, J.: Recursive data structures in SPARK. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 178–189. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_11, https://doi.org/10.1007/978-3-030-53291-8_11
8. Gäher, L., Sammler, M., Jung, R., Krebbers, R., Dreyer, D.: Refinedrust: A type system for high-assurance verification of rust programs. Proc. ACM Program. Lang. **8**(PLDI) (Jun 2024). https://doi.org/10.1145/3656422, https://doi.org/10.1145/3656422
9. Ho, S., Protzenko, J.: Aeneas: Rust verification by functional translation. Proc. ACM Program. Lang. **6**(ICFP) (Aug 2022). https://doi.org/10.1145/3547647, https://doi.org/10.1145/3547647
10. Jaloyan, G.A., Dross, C., Maalej, M., Moy, Y., Paskevich, A.: Verification of programs with pointers in spark. In: Formal Methods and Software Engineering: 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1–3, 2021, Proceedings. p. 55–72. Springer-Verlag, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-63406-3_4, https://doi.org/10.1007/978-3-030-63406-3_4
11. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: Rustbelt: securing the foundations of the rust programming language. Proc. ACM Program. Lang. **2**(POPL) (Dec 2017). https://doi.org/10.1145/3158154, https://doi.org/10.1145/3158154
12. Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., Hawblitzel, C.: Verus: Verifying rust programs using linear ghost types. Proc. ACM Program. Lang. **7**(OOPSLA) (Apr 2023). https://doi.org/10.1145/3586037, https://doi.org/10.1145/3586037
13. Matsakis, N.D., Klock II, F.S.: The Rust language. SIGAda Ada Letters **34**(3), 103–104 (Oct 2014). https://doi.org/10.1145/2692956.2663188, https://doi.org/10.1145/2692956.2663188
14. Matsakis, N.: Rust RFC 2094: non-lexical lifetimes (2017), https://github.com/rust-lang/rfcs/blob/master/text/2094-nll.md
15. Matsushita, Y., Denis, X., Jourdan, J.H., Dreyer, D.: RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. pp. 841–856. ACM. https://doi.org/10.1145/3519939.3523704, https://inria.hal.science/hal-03777103

16. Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based verification for Rust programs. ACM Trans. Program. Lang. Syst. **43**(4) (Oct 2021). https://doi.org/10.1145/3462205, https://doi.org/10.1145/3462205
17. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 9583, pp. 41–62. Springer-Verlag (2016), https://doi.org/10.1007/978-3-662-49122-5_2
18. The Dafny development team: The Dafny programming and verification language, https://dafny.org
19. The Why3 development team: The Why3 verification platform, https://why3.lri.fr/