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

ANONYMOUS AUTHOR(S)

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, and was evaluated on several case studies (4k LOC).

Additional Key Words and Phrases: Deductive verification, Rust, Prophecies

ACM Reference Format:
Anonymous Author(s). 2025. Using a Prophecy-Based Encoding of Rust Borrows in a Realistic Verification Tool. 1, 1 (March 2025), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Rust programming language [Matsakis and Klock II 2014] 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 because it rules out whole classes of programs that are difficult to reason about. Thus, unsurprisingly, a wide range of approaches have been recently developed to tackle the verification of Rust programs [Astrauskas et al. 2022; Ho and Protzenko 2022; Jung et al. 2017; Lattuada et al. 2023; Matsushita et al. 2021]. Among those, RustHorn [Matsushita et al. 2021] and Aeneas [Ho and Protzenko 2022] come up with a similar insight: *it is possible to reason on a well-typed Rust program* (e.g., *to verify its functional correctness) almost as if it were a purely functional program*. Intuitively, because the Rust compiler imposes strict aliasing rules on borrows, a verification tool can simply focus on the flow of values through the program and does not need to track pointers or aliasing.

RustHorn makes this intuition precise by defining 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 [Matsushita et al. 2022], establishing the soundness of the translation. On a more practical side, it constitutes the logical foundation behind the Creusot [Denis et al. 2022] deductive verification tool for Rust programs. Thanks to this prophecy-based translation of borrows, Creusot is the only deductive verification

ACM XXXX-XXXX/2025/3-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
1 let mut x: i32 = 0;
2 let y: &mut i32 = &mut x;
3 // x = 42;   // forbidden
4   // x is frozen while the lifetime
5   // of y is active
6 *y = 1;                    lifetime of y
7
8 assert!(x == 1);
```

$x \mapsto 0$

$x \mapsto \alpha,\ y \mapsto (0, \alpha)$

$x \mapsto \alpha,\ y \mapsto (1, \alpha)$

// resolve y $x \mapsto \alpha,\ y \mapsto (1, \alpha),\ \alpha = 1$

Fig. 1. Reasoning on the creation and resolution of mutable borrows.

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.

Yet, despite the foundational work of RustHorn and RustHornBelt, actually using their prophecy-based encoding of borrows to build a realistic verification tool raises a number of new challenges. *In this paper, we identify four open problems with applying the prophecy-based encoding to a realistic setting. We design novel approaches for addressing each of these problems, and implement them as extensions of Creusot.* Our solutions improve on existing work (mainly RustHornBelt and Creusot) which either implement unsatisfactory (and sometimes unsound) approaches, or none at all. In some sense, these additional challenges "simply" come from the fact that we are outgrowing the core calculus studied in earlier works; yet, actually solving these challenges requires not only deep insight but also *fundamental modifications* of the prophecy encoding, as we shall see.

In the rest of this section, we first recall the essential aspects of RustHorn's prophecy-based translation of borrows (§1.1), then we describe the four challenges that we tackle (§1.2—§1.5).

1.1 Reasoning on Mutable Borrows Using Prophecies

Logical interpretation of borrows. Let us consider a simple example, shown in Fig. 1, that illustrates the use of mutable borrows and how to reason about them when using RustHorn's prophetic encoding. On the left, we show some Rust code. On the right, to explain how one may *reason* about this code, we show what would be the logical view of the program maintained by a verification tool at each program point: a map of each program variable to its *logical model*.

On line 1, a mutable variable `x` is declared initialized to 0. The logical view of the program at this point is thus that `x` is mapped to 0. Then, `x` is *borrowed* on line 2 with `&mut x`, and the result is bound to `y`. Here, the variable `y` is a mutable borrow: a pointer that points to the data stored by `x`. While `y` is in use, the borrowed variable `x` is "frozen" and cannot be accessed. The Rust compiler ensures this by statically computing the *lifetime* of borrows: the span of the program during which the borrow is considered "in use". On our example, the dotted box visually represents the lifetime of `y` as computed by the Rust compiler.

What should be the model of a borrow in our logical view of the program? Mutable borrows are challenging because mutating a borrow modifies a value held in a *different* location, 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? 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. The current value tracks the value of the data the borrow has (temporary) exclusive access to. The 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  #[requires(true)]
2  #[ensures(if take_first { result == p.0 && ^(p.1) == *(p.1) }
3                     else { result == p.1 && ^(p.0) == *(p.0) } )]
4  fn pair_bor_mut<T>(p: (&mut T, &mut T), take_first: bool) -> &mut T {
5    if take_first { p.0 } else { p.1 } }
```

Fig. 2. Specification of `pair_bor_mut`, which takes and returns mutable borrows.

We thus model the effect of line 2 by creating a *new prophecy* α representing the final value of `y`. At this point, α is just a fresh name: we know nothing about this final value. The value pointed by `y` is currently equal to 0, so its model is the pair $(0, \alpha)$. The model of `x` is set to α: whenever `y` ends, `x` will be able to observe the mutations made by `y` during its lifetime.

On line 6, we use `y` to modify the value stored by `x` (indeed, `y` is a *mutable* borrow). On the logical side, this corresponds to updating the "current value" of the model of `y`.

The lifetime of `y` ends on line 7. Nothing happens in the program (lifetimes are a purely static discipline), but from a logical standpoint, we need to account for this fact. We do this by performing **a logical step dubbed *resolution***. This step is specific to a RustHorn-style encoding of borrows. We "resolve `y`" *by adding a logical fact to our context*: that its final value α is equal to its current value 1. Afterwards, on line 8, we regain access to `x` and assert that `x` has been successfully modified. On the logical side, since we set the model of `x` to be α when borrowing it, and we have since learned "$\alpha = 1$" by resolving `y`, we can statically prove that the assertion always holds.

Specification language. The prophetic interpretation of borrows is *compositional*: one can always specify functions by describing the current and final values of borrows passed as arguments and return value. Concretely, we reuse Creusot syntax for the specification language used in functions specifications, proof assertions and loop invariants, and write `*x` for the current value of a mutable borrow `x` and `^x` for its final value. These correspond respectively to the first and second component of the *logical model* of `x` according to the prophetic encoding.

As an example, let us illustrate in Fig. 2 how we can specify `pair_bor_mut`, a function which takes a pair of borrows and returns one or the other depending on a boolean parameter. (This somewhat artificial example can be seen as a simplified version of the `index_mut` function on vectors.)

The `requires` and `ensures` clauses specify pre- and postconditions in the verifier's specification language. (Here, we could omit the precondition as it is trivial.) One can think of them as formulas in first-order logic with Rust-like syntax and syntactic sugar (*e.g.*, `if`/`else` expressions). They can refer to the function's arguments and return value (named `result`). Following Rust syntax, `p.0` and `p.1` denote the first and second component of a pair `p`.

Here, the postcondition specifies that the `result` borrow is either the first or second component of the pair `p`, depending on `take_first`. Additionally, for the other component of `p` (also a borrow), it specifies that its "final value" is equal to its "current value". Note that, in the `ensures` clause, `p` denotes the *logical model* of the function argument as an immutable value. Thus, the "current value" `*(p.0)` is the value of the borrow `p.0` *at the entrance of the function*! Then, when the postcondition specifies that *e.g.*, `^(p.0) == *(p.0)` (when `take_first` is `false`), this means that the borrow `p.0` was not modified by the function. This can be proved because the body of `pair_bor_mut` simply consumes the borrow and drops it: therefore, it must be *resolved* at the end of the function according to the prophetic translation, and resolution gives us the required equality.

1.2 Challenge: Choosing the Location of Resolution

The specification and reasoning scheme presented in §1.1 leaves an important choice undetermined: the location where the verification tool performs the resolution step. Obviously, it needs to happen after the last mutation through the borrow. But as the example of Fig. 1 illustrates, performing resolution too late may lead to rejecting correct programs[1]. Indeed, if resolution is performed after line 8, then it would not be possible to prove the assertion.

Existing literature [Denis et al. 2022, §3.2] suggests that a borrow should be "resolved at the moment it is dropped", which corresponds to the point of code where the borrow leaves its scope, according to Rust's rules for implicitly inserting `drop` instructions. However, the simple example of Fig. 1 shows that it is not satisfying, because that would lead to resolving `y` when it leaves its scope, *i.e.*, after line 8. Another option would be to resolve borrows when their lifetime ends. That would work on this example, but in general, there exists more complex cases where a borrow disappears and needs to be resolved before the end of its lifetime.

Our conclusion is that determining resolution location cannot directly depend on a pre-existing algorithm. Instead, **we design a dedicated algorithm for locating resolution points, based on a dataflow analysis**. We describe it in §5. It is subtle, because it needs to take into account several features of the Rust language and of the prophecy-based encoding itself, including various analyses performed by the borrow checker and type invariants.

1.3 Challenge: Sound Modular Resolution

In the encoding of borrows detailed in §1.1, a borrow that gets discarded must be *resolved*, which means assuming in the logic that its "final value" is equal to its "current value". This works well for single borrows that are *e.g.*, bound to program variables. However, real-world code also involves data structures containing borrows, which can themselves get discarded, thus discarding all the borrows they contain. For instance, if a vector of borrows gets discarded, how should we logically account for this fact? In other words, how should we *resolve* all the borrows that it contains?

```
1  let y: &mut i32 = ...;
2  // resolve y: assume ^y == *y
```

```
let y: Vec<&mut i32> = ...;
// resolve y: assume ???
```

Earlier work on Creusot [Denis et al. 2022, §4.2] partially addresses this question by allowing the implementor of a data type to specify a custom resolution predicate that is *assumed* to hold at resolution points. However, *nothing prevents users to define unsound resolution predicates*. This opens the door to particularly subtle soundness bugs, since resolution assumptions are silently inserted by the backend of verifier and are not visible in the source code.

In order to support both *sound* and *modular* resolution of borrows stored in containers and user-defined data types, **we define verification conditions for custom resolution predicates**. We require that any custom resolution predicate must be proved sound with respect to systematic "structural" resolution rules, as we detail in §4.

1.4 Challenge: Sound Support for Ghost Code

"Ghost code" is a popular feature of deductive verifications systems; it can be found in Dafny [The Dafny development team 2025], Why3 [The Why3 development team 2025] or VCC [Cohen et al. 2009]. It is a mechanism by which the user may write code that will be erased during the normal compilation of the program, but helps verification of the non-ghost part of the program. Because

[1]The problem is even worse in our implementation of Creusot: in some cases, resolving too late may also lead to unsoundness, because, as explained in §3.3, we check type invariants at resolution.

ghost code is designed to aid verification, it is both common and convenient to allow ghost code to use *specification-level operations* on top of ordinary program instructions.

The interaction of a prophecy-based translation of borrows with ghost code has seen little study. RustHorn or RustHornBelt do not feature ghost code; meanwhile, its first appearance in Creusot [Denis and Jourdan 2023; Denis et al. 2022] is unsound. Here is one of the most straightforward examples of unsoundness:

```
1  let mut x: Snapshot<bool> = snapshot!( true );
2  let bor: &mut Snapshot<bool> = &mut x;
3  *bor = snapshot!( ! ^bor );
4
5
6  proof_assert!( false );
```

$x \mapsto \top$
$x \mapsto \alpha,\ bor \mapsto (\top, \alpha)$
$x \mapsto \alpha,\ bor \mapsto (\neg\alpha, \alpha)$
`// resolve bor`
$x \mapsto \alpha,\ bor \mapsto (\neg\alpha, \alpha),\ \alpha = \neg\alpha$

This program is accepted by Creusot, despite being unsound. This example only relies on a small set of "ghost code" features. In the code, "snapshot" should be read as "ghost"[2]; one may write `snapshot!{e}` to denote a ghost expression `e`, where `e` may read but not modify memory or control flow, and may use specification-level operations. If `e` has type `T` then `snapshot!{e}` has type `Snapshot<T>`; the Rust compiler then knows that values of the `Snapshot` type can be erased during compilation (because it is declared as a "zero-sized type"). Finally, one can only access snapshot values inside `snapshot!{...}` blocks; elsewhere they are considered as completely opaque.

In this example, we first create a boolean in ghost code, and store it in the variable `x`. Then, we take a mutable borrow `bor` 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 6).

In other words, the unrestricted combination of ghost code and prophecies allows creating *causality loops*, where a prophecy is instantiated with a value that depends on itself.

The challenge, that we tackle in §2, is to impose restrictions on what ghost code is allowed to do with borrows, carefully chosen to restore soundness without overly restricting its expressivity. For instance, it is easy enough to reject the example above by disallowing the use of "^" in ghost code; but we will show that this is not enough to regain soundness. Our solution **changes the prophetic encoding of borrows to add a "logical identity" to their model**, while allowing different borrows to share the same logical identity in specific cases determined by a static analysis.

1.5 Challenge: Support for Type Invariants

Type invariants are a common feature in deductive verification tools. They allow a library to implicitly assume that any value of a type satisfy some property that is needed for the correctness of the library. Conversely, the library must ensure that all the values it produces (or modify) satisfy the invariant. Importantly, a client of such library does not need to know that such a type invariant exists, because all the values that the library produces satisfy this property: in the clients eyes, the fact that a value satisfies the invariant is part of the type of the value.

The implementation of type invariants that we choose is to implicitly add pre- and postconditions to all functions that manipulate values of a type with an invariant. In between function calls, values

[2]The aforementioned publications on Creusot and earlier versions of the Creusot implementation used `ghost`/`Ghost` instead of `snapshot`/`Snapshot`, with "ghost" being renamed to "snapshot" later on. Here, we match what the latest version of Creusot uses, and thus use the "snapshot" terminology.

need not satisfy their type invariant: the invariant can be temporarily *opened* to allow for non-atomic updates of data structures.[3] While this approach may seem simple, it raises a number of challenges when combined with the prophecy-based encoding of borrows. The main issue can be summarized as follows: because a function may produce values by writing to borrows that it receives as argument, the tool needs to check that values returned *via* prophecies of borrows satisfy their type invariants. That interaction has never been properly studied in previous work: in previous work [Denis and Jourdan 2023], Creusot developers mention type invariants, but their implementation did not support them, so they had to manually add explicit pre- and postconditions.

More concretely, recall function `pair_bor_mut` in Fig. 2. Our challenge is to make the encoding of this function both sound and usable in practice, in particular when the type `T` stored in borrows has a type invariant. Because this function takes mutable borrows in an argument, Creusot needs to check that, when the function returns and the lifetimes of borrows end, the value stored in the borrow satisfies the invariant of `T`.[4]

A naive solution would be for Creusot to add the implicit postcondition `#[ensures(inv(^p.0) && inv(^p.1))]`, stating that, at function return, the invariants of both prophecies hold. However, when verifying `pair_bor_mut`, this postcondition would not be provable, because the value of either `^p.0` or `^p.1` is determined by the last value the caller will write to returned borrow: these check must be somehow *deferred* to the point where the returned borrow is resolved, in the caller's code.

Another issue we have is that assertions such as `#[inv(^p.0) && inv(^p.1)]` need to be automatically generated by the tool. This is non-trivial when mutable borrows are contained in complex data structures, possibly by instantiating a generic type variable: how would such a postcondition be generated for hash tables of mutable borrows?

In order to address this challenge, **we propose in §3 to assume the type invariants of all the final values of mutable borrows.** For soundness, this requires checking that the last value written to a borrow satisfy its type invariants at the moment of resolution, but, in exchange, the creator of a borrow can assume that the prophecy it creates satisfies it.

The rest of the paper

In the following sections, we describe our solutions to each of the challenges above. We describe how to soundly support ghost code (§2) and type invariants (§3), handle "resolve" for arbitrary collections in a sound and modular fashion (§4), and how to decide the location of "resolve" points (§5). We have implemented all of these contributions as extensions of Creusot. We adapted Creusot's existing repository of case studies to use our changes, and evaluate their impact (§6). We finally go over remaining limitations and future work (§7).

2 INTERACTION OF PROPHECIES WITH GHOST CODE

As we saw in §1.4, 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".

It is important to note that these causality loops may be created without using the `^` operator at all: the prophecy can also be retrieved *indirectly* using logical equality on borrows, which compares their logical model, i.e. the pair of their current and final value.

[3] Alternatively, type invariants could be implemented using subset types, as in Why3. But the question of when the type invariants can be opened would lead to similar problems: the main fundamental question here is when should we assume and check type invariants.

[4] Of course, when `pair_bor_mut` is annotated with a complete postcondition as described in §1.1, these checks are pointless, because the caller can establish type invariants by knowing these equalities. So we assume here that the function is not annotated with any explicit specification.

```
1  let mut x = snapshot!( true );
2  let b1: &mut Snapshot<bool> = &mut x;
3
4
5  let b2: &mut Snapshot<bool> = &mut x;
6  *b2 = snapshot!( ! (b2 == b1) );
7
8
9
10 proof_assert!( false );
```

```
  x ↦ ⊤
  x ↦ α, b1 ↦ (⊤, α)
// resolve b1
  x ↦ α, b1 ↦ (⊤, α), α = ⊤
  x ↦ β, b1 ↦ (⊤, ⊤), b2 ↦ (⊤, β)
  x ↦ β, b1 ↦ (⊤, ⊤), b2 ↦ (((⊤, β) ≠ (⊤, ⊤)), β)
  x ↦ β, b1 ↦ (⊤, ⊤), b2 ↦ (¬β, β)
// resolve b2
  x ↦ β, b1 ↦ (⊤, ⊤), b2 ↦ (¬β, β), β = ¬β
```

Fig. 3. Deriving `false` by combining ghost code and equality

The code in Fig. 3 uses the same idea as the previous example in §1.4, but leverages equality on borrows to get the value of the prophecy and deduce a logical contradiction. Before the execution of line 6, we know two key facts: first, `*b1` and `*b2` are both equal to `true`, since we have not to written anything yet into `b1` or `b2`. Second, `b1` is never written to, so we can resolve `b1` and obtain `^b1 == 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 problem, but they are not all equally satisfying. Because logical equality is used pervasively in specifications, restricting its use is quite tricky. In particular, we want to continue to allow comparing mutable borrows at least in some cases (see §2.2).

To prevent these issues, we introduce two mechanisms: first, in §2.1, we introduce a separation between ghost code and specifications, by layering the specification language. Then, we change the logical model used for borrows, so that testing the equality of borrows does not simply boil down to testing the equality of their current and final values.

2.1 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 specification language: we have the purely logical part, that is only used in specifications, does not interact with executable code and can use the final operator. And we have the part for ghost code, that can create snapshot objects from runtime values, but cannot use the final operator. We call the first part "prophetic logic" (annotated with `#[logic(prophetic)]`), and the second "logic" (or `#[logic]`). Then, `#[logic(prophetic)]` functions are allowed to call `#[logic]` functions, but the reverse is forbidden.

In order to correctly implement this idea, one must be careful with logic functions defined in Rust traits [Denis et al. 2022, §4]: if a trait method is declared with `#[logic]`, we must forbid implementations to be annotated with `#[logic(prophetic)]`.

2.2 Non-Extensional Borrows

We have seen in Fig. 3 that logical equality can also be used in ghost code to observe prophecies and create paradoxes. Indeed, as 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. In the example of Fig. 3, `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.

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

  We cannot prove its intuitive specification because the compiler implicitly replaces the expression `bor` 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:

```
#[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.

2.3 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 ↦ (x, α, id)
3     let result = &mut *bor;         // bor ↦ (β, α, id), result ↦ (x, β, id)
4     result
5 }                                   // We deduce α = β
```

After line 3, `bor` is not modified, and is subsequently resolved when the function ends. We thus get

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

Fig. 4. Simplified description of the MIR intermediate language.

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.

Note, however, that the identity *cannot* always be shared between reborrows, as that would violate the key property presented in 2.2: that if two borrows share the same identity, then their prophecy is the same. Indeed, the example in Fig. 3 can be modified to use reborrows:

```
let mut x: bool = ...;
let b = &mut x;
let b1: &mut Snapshot<bool> = &mut *b;
let b2: &mut Snapshot<bool> = &mut *b;
*b2 = snapshot!( !(b2 == b1) );
proof_assert!(false);
```

Here `b1` and `b2` might not have the same prophecy, so they cannot share the same identity.

We thus need a more fined grained approach: one that makes reborrows non-final in the general case, while still allowing functions like the identity or projections.

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

2.4.1 Description of MIR. To describe the analysis, we use a simplified version of MIR, whose grammar is given in Fig. 4. 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 ($*pl$)[5], or a subfield of another place ($pl.f$). 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 $pl \leftarrow pl$, which copies the content of the right-hand side into the left-hand side; a function call[6] $pl \leftarrow func(pl\star)$; or a borrow $pl \leftarrow \texttt{borrow}(pl)$, 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.

2.4.2 The final borrows analysis. For each program location λ, we compute a set of places Φ_λ: a reborrow of place *pl* immediately before program location λ will be considered final if $pl \in \Phi_\lambda$. Since the analysis only considers syntactic and typing information, the computed sets are under-approximations, but we believe that this is enough to prove most programs in practice.

A place is in Φ_λ if it contains at least one dereference, and we 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.

The analysis finds the greatest solution of the system of set equations described in Fig. 5, by computing the 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 "$_{\lambda_1}$ *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. $C(pl)$ is removed from Φ_λ whenever *pl* is accessed (read or write).
- The set $D(pl)$ of places pl', such that pl' is a subplace of *pl*, and pl' contains one more *dereference* ($*$) than *pl*. This is always a subset of $C(pl)$. $D(pl)$ is added to Φ_λ whenever *pl* is written to.

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 changed, 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

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

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

Fig. 5. Rules of the final borrows analysis.

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 present a small example, together with the corresponding MIR code annotated with the sets of locations Φ, as well as the sets C and D:

```
1  let b: &mut T = ...;
2  let b1 = &mut *b;
3  let b2 = &mut *b;
4  return;
```

```
1  // Φ = {...}
2  b ← ...
3  // Φ = {*b1, *b2}
4  b1 ← borrow (*b)
5        // C(*b) = {b, *b}, C(b1) = {b1, *b1}, D(b1) = {*b1}
6  // Φ = {*b1, *b2}
7  b2 ← borrow (*b)
8        // C(*b) = {b, *b}, C(b2) = {b2, *b2}, D(b2) = {*b2}
9  // Φ = {*b, *b1, *b2}
10 return
```

Recall that the analysis is backward. At each step, the C sets are removed, and the D sets are added. At the point right after the first borrow, the place `(*b)` is not in the set, so the reborrow `b1 = &mut *b` isn't final. However, at the point after the second borrow, the place `(*b)` is in the set, so the reborrow `b2 = &mut *b` is final.

3 INTERACTION OF PROPHECIES WITH TYPE INVARIANTS

In Rust, some types have a semantic interpretation dictating that a certain predicate holds 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 exist in many deductive verification tools, there are unique challenges when used together with 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:

```
1 struct Even(u64);
2 impl Invariant for Even {
3     #[logic] fn invariant(self) -> bool { self.0 % 2 == 0 }
4 }
```

To determine the invariant of a particular type, we 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 trait `Invariant` is explicitly implemented for `Option`. Hence, we define the invariant of a type as the conjunction of the 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 structural invariants:

Type of `x`	Invariant
`Option<Even>`	`match x { None => true, Some(y) => inv(y) }`
`(Even, Even)`	`inv(x.0) && inv(x.1)`
`Box<Even>`	`inv(*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 postconditions to every function based on its parameter and return types. Each parameter corresponds to a precondition stating that the type invariant of the argument holds, and the return type corresponds to a postcondition for the invariant of the return value. This scheme ensures that invariants hold across function boundaries, complementing the principle of modular verification.

3.2 How Should Type Invariants of Prophecies Be Enforced?

Implementation of type invariants 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!(inv(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 some function `take_even_bor` defined somewhere else. 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 like function outputs, one conceivable solution is adding postconditions stating that the invariants of the parameters' prophecies hold. Thus, naively, the function `take_even_bor` that is used in example above would have an implicit postcondition like `#[ensures(inv(^bx))]`.

However, this is unsatisfactory. Recall the `pair_bor_mut` function defined in Fig. 2. What should Creusot add as implicit pre- and postconditions for `pair_bor_mut`? This function should require that the type invariant holds at function entry (*i.e.*, `#[requires(inv(*p.0) && inv(*p.1))]`, after unfolding structural invariants) and ensure them for the return value (*i.e.*, `#[ensures(inv(*result))]`).

But, following the general principle stated above, the caller may rely on the fact that invariants are reestablished for the values pointed to by the borrows. In other words, it may rely on `inv(^p.0)` and `inv(^p.1)`. But `pair_bor_mut` cannot ensure these, because one of the borrows is moved as the return value, and therefore this fact depends on the *caller* reestablishing the invariant on the return value. Instead, it could ensure something like `inv(^result) ==> inv(^p.0) && inv(^p.1)`: if the caller fails reestablishing the invariant, `inv(^result)` would not hold after resolution, and the caller cannot rely on `inv(^p.0) && inv(^p.1)`.

We now have a modularity problem: while it is relatively easy to see that the caller may need `inv(^p.0) && inv(^p.1)` in this case, this assertion is more complex if the function takes a triple of borrows, or even an arbitrary data structure containing borrows. The left-hand side of the implication, `inv(^result)`, would need to be adapted similarly if the functions returns borrows within a data structure. The situation worsen even more when considering functions taking borrows of borrows as parameters, or when generic parameters of polymorphic functions are instantiated with borrows.

In conclusion, the naive approach of adding pre- and postconditions 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

Let's come back to our initial, simple, problem: how to specify that `take_even_bor` reestablishes the type invariant of the value pointed to by `x`? This is important for the caller of `take_even_bor`, which needs this information to continue using the value contained in the borrowed place.

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). 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 actually know the final value: 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 postcondition that 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 prophesize 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. As a solution, we choose 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, take_first: bool) {
2     let bx = &mut x;                                  // bx ↦ (x, α), x ↦ α, assume inv(α)
3     let by = &mut y;                                  // by ↦ (y, β), y ↦ β, assume inv(β)
4     let b = pair_bor_mut((bx, by), take_first); // b ↦ (y, β) or b ↦ (x, α)
5     b += 2;                                           // b ↦ (y + 2, β) or b ↦ (x + 2, α)
6     // check that *b is even, resolve b
7     assert!(inv(x) && inv(y)); // provable
8 }
```

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 7, 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`. For simplicity, sake, assume `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 6, we thus have to prove the invariant of its current value y before we learn $\beta = y + 2$. This fact is simply obtained from the implicit postcondition generated for the invariant of the return value of `pair_bor_mut`. Importantly, this reasoning does not depend on the explicit specification of `pair_bor_mut`, which could be left empty: Creusot does not need to know that the value of b is (y, β) or (x, α), it simply knows this is a mutable borrow satisfying its type invariant.

4 MODULAR AND SOUND RESOLUTION OF BORROWS

When using the prophetic encoding of borrows, the *resolution* step connects the logical model of a borrow and the value it borrowed from. As we described in §1.3, resolution applies not only to naked mutable borrows, but *any* value that may itself contain borrows. This is necessary to recover logical information about all the borrows that it contains. For example, we could define the resolution clause of a vector `v` as:

```
1 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 that, for each borrow in the vector, its prophesized final value is equal to its current.

More generally, when modularly verifying code from different libraries, library authors should be able to declare what the resolution predicate is for the types that they define (especially if their implementation is private). In prior work in Creusot [Denis et al. 2022], this was achieved through a *trusted* trait `Resolve` that library authors could implement.

Challenge. A major limitation of the approach in Creusot 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 that 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, that is 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)]
  fn resolve(self) -> bool;

                                  #[logic(prophetic)]
                                  #[requires(inv(self))]
                                  #[requires(structural_resolve(self))]
                                  #[ensures((*self).resolve())]
                                  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 encoding of prophecies crucially relies on the insertion of *resolution* assumptions. As we mention in §1.2, choosing *when* to insert these resolution points it is not obvious.

Literature on the prophetic encoding [Denis et al. 2022; Matsushita et al. 2022, 2021] does not prescribe the precise program point where resolution happens: the only requirement of the existing meta-theory is that it should happen after we are guaranteed the borrow is no longer mutated. This leaves a lot of freedom in the placement of borrows. As sketched in the introduction §1, it is not correct to resolve variables when they leave their scope, as suggested by Creusot's authors [Denis et al. 2022, §3.2] ("borrow is resolved at the moment it is dropped").

We consider the question of the precise position of these assumptions in the program to be an open problem. Indeed, it is a delicate balance between soundness and expressivity of the verification tool: wrong placement may both be unsound or prevent proving valid programs (i.e., incompleteness of the tool). This is difficult, because this problem interacts with many features of the Rust language and of Creusot:

- It interacts with lifetimes of borrows, both because resolution of a borrow must happen before its lifetime ends, and because resolving a variable which is still borrowed may lead to incompleteness by lack of knowledge about its value.
- It interacts with initializedness of variables, because we should not resolve uninitialized variables.
- It interacts with the notion of *places* (*i.e.*, portions of local variables), because a variable may be partially initialized or borrowed.
- It interacts with types invariants, because it involves checking the type invariant for the final value. It is *unsound* to skip resolution of a borrow, but resolving early weakens the context in which the invariant is proved, and may lead to failure to prove a valid program.

In §5.1, we detail the constraints that need to be tackled to solve this 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.

5.1.1 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.

5.1.2 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.[7]

[7]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

5.1.3 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 the branch was taken
}
```

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 necessarily 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 [Denis et al. 2022; Matsushita et al. 2022, 2021] 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. 4) instead of individual local variables. All the constraints we evoked above in this section for resolving variables also applies to places.

5.1.4 Resolving frozen borrows may prevent proving valid programs and should be avoided if possible. 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 that 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

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.

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 from 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`.[8]

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 §2.4, 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.

[8] In order to allow verification of such functions, we could take inspiration from Prusti [Astrauskas et al. 2022, 2019] and support *pledges*, assertions that need to be valid only when a lifetime ends. This is out of the scope of this paper.

Places containing a dereference of a mutable borrow correspond to memory that is given back to the lender when the lifetime of the borrow ends. The lender will then handle resolution of this memory: the only situation where we resolve this place is right before it is overwritten. In particular, initializedness, liveness or frozenness play no role in the placement of these resolutions. Therefore, these places are not considered by these analyses: we only consider places which do not contain a dereference of a mutable borrow[9].

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

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

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 $N(\lambda) \setminus N(\lambda')$.

We can easily prove that constraints 5.1.1, 5.1.2 and 5.1.3 are satisfied by this algorithm.

6 EVALUATION

We presented the design of several extensions to RustHorn's prophetic translation, adding support for features often found in other deductive verification tools. We show that these extensions are *useful* and *practical*.

We proceed by implementing them in Creusot and evaluating their impact on Creusot's "test suite". The Creusot "test suite" is a comprehensive repository (10k LOC) that contains many of the case studies formalized using Creusot since its inception (on top of regression test cases). It includes a number of larger library-sized examples focusing mainly on textbook data structures and algorithms, as well as all the case studies from earlier publications related to Creusot.

Fortunately, showing that our new extensions are *useful* is not hard. Prior to this work, many of the existing case studies *already* used ghost code, type invariants, and modular resolution through various *ad-hoc* (and sometimes unsound) encodings! For instance, previous work [Denis et al. 2022] crucially relied on a form of ghost code—that we now know was unsound—to verify `gnome_sort`,

[9]Places corresponding to array indexing (not mentioned in Fig. 4, 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.

case study \ features	Snapshots (§2)	First-class snapshot values (§2)	Invariants (§3)	Structural resolve (§4)	Final borrows (§2.3)
Red-black trees	✓		✓	✓	
Hashmap	✓		✓	✓	
Binary decision diagrams		✓	✓		
Sparse arrays			✓	✓	
Knuth shuffle	✓				
Iterators	✓	✓	✓	✓	✓
Sorting algorithms	✓				

Table 1. Description of the case studies and the features they use.
Red-black trees implement mappings as balanced trees. *Hashmap* implement hash tables with buckets. *Binary decision diagrams* use hash-consing and arena-based allocation. *Sparse arrays* corresponds to the sparse array challenge from the VACID0 benchmark. *Knuth shuffle* implements the Fisher-Yates shuffling method. *Iterators* include reimplementation of various iterators and combinators of the standard library. *Sorting algorithms* include insertion sort, selection sort, heap sort and gnome sort.

and also axiomatizes modular resolution predicates. The present work aims at providing a principled account of these features, exactly because they were so useful.

We also wish to show that our new extensions are *practical*: they can be used in realistic contexts. To do that, we implemented our extensions in Creusot and checked whether they can be used in Creusot's test suite. This meant replacing ad-hoc encodings by their principled equivalent. In the end, we were able to *update all the case studies* to use our extensions. The resulting code was in many cases unchanged, shorter or more generic.

More concretely, we show in Table 1 which of Creusot's larger, pre-existing, case studies make use of our new extensions, as we detail just next.

Snapshots. One common use of ghost code is to take *snapshots* that remember the previous value of a variable at a specific program point. This is useful in sorting algorithms for loop invariant of the form "the contents of the array is a permutation of its initial contents". Red-black trees, hashmap, and Knuth's shuffle use snapshots similarly. When desugaring a `for`-loop, Creusot maintains a snapshot of the sequence of elements produced so far by the iterator [Denis and Jourdan 2023].

More interestingly, it is also useful to *store "first-class" snapshot values inside program data structures*. The "binary decision diagrams" data structure stores a snapshot value mapping node identifiers to node contents, tied by a type invariant to actual in-memory nodes. The `Map` iterator's structure also stores a snapshot field to track the sequence of elements produced by its underlying iterator. In both cases, one must deal with mutable borrows over these data structures, and thus *mutable borrows of the snapshot values they contain*, which get updated using values computed from other borrows of snapshot values. This shows that "first-class" snapshot values, although a part of the problematic examples of §2, are also useful in legitimate programs and thus worth supporting.

Type invariants. Before this work, manually-written pre- and postconditions were needed to express that each of the functions of a data structure requires the invariant to hold at function entry, and re-establishes it when returning. Even worse, clients of these data structures needed to maintain these type invariants, through other manually written pre- and postconditions.

The case of iterators were particularly problematic. Recall that type invariants are used by iterators combinators `Map` and `Filter` to guarantee that the precondition of the closure holds at every call. The method `next` of these iterators thus had a precondition for these type invariants. Now,

if iterator `Map` is bundled in an iterator *combinator*, say `Skip<Map<...>>`, then, when calling `Map`'s implementation of `next`, the code of `Skip` needs to discharge this precondition. Being written generically, `Skip` itself needed a type invariant to maintain the type invariant of its underlying iterator. The end result was that *every* iterator combinator needed to manually thread type invariants.

Thanks to our principled implementation of type invariants, we can remove all these repetitive lines of specification: no mention of type invariants remain in pre- or postconditions in examples presented in Table 1. In total, we could remove about 80 lines of specification.

Structural resolve. In the examples of Table 1, there are 19 instances of the `Resolve` trait. These instances were trusted, thus increasing the trusted computing base. The technique presented in §4 allowed us to formally verify them. These verification conditions were automatically discharged by SMT solvers.

The only remaining instances whose validity is trusted are in Creusot's standard library, and correspond to abstract types.

Final borrows. We initially envisioned simpler mechanisms for handling prophecy identifiers: first without reusing them at all, then only reusing them for direct reborrows (`&mut *b`). This was however insufficient to make the "Iterators" case-studies work without major complications. Indeed, iterator combinators—which store an underlying iterator in a structure field—need final reborrows to apply to field reborrows (`&mut b.field`) to be able to specify the behavior of their `next` method in terms of the underlying iterator's `next`.

With the strategy presented in §2.4, we found that nearly all specifications and proofs could be kept as-is: one line of specification in Creusot's standard library had to be changed, and one line of specification in the `IterMut` iterator test needed to be reformulated and was replaced by 4 lines.

7 LIMITATIONS AND FUTURE WORK

Among the features we presented, many have an impact on soundness. We have not yet established this formally, but we believe that extending RustHornBelt with these new features would be a natural way to ensure soundness. In §2, we changed the representation of borrows (with a new "identity" field) to ensure that ghost values cannot depend on prophecies. We believe that adding such a field is possible in RustHornBelt, and that specification assertion can then be classified in RustHornBelt into prophetic assertions and non-prophetic assertions, to model the notion of prophetic logical functions described in §2.1. As discussed in §3, we should make it so that prophecies always respect their invariant, and the constraints for resolution location described in §5.1 could then be justified. In §4, we asserted that structural resolution was enough to ensure soundness of a particular `Resolve` implementation. This should be tackled in RustHornBelt by defining a type class for resolution.

Our ghost code is also of limited expressivity, in that it does not carry ownership: a snapshot value of type `Snapshot<T>` is always duplicable. Allowing ghost linear values would be more expressive, following ideas from Verus [Lattuada et al. 2023]. 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.

8 RELATED WORK

To our knowledge, Creusot [Denis et al. 2022] 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 [Denis et al. 2022] and extends it beyond its state of the art. Ghost code was briefly mentioned in earlier works as a feature of Creusot [Denis and Jourdan 2023; Denis et al. 2022], but one can check that these works rely on an *unsound* implementation of ghost code (they incorrectly accept the example shown in Fig. 3). Type invariants were also mentioned in the earlier work on iterators [Denis and Jourdan 2023], 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 [Denis et al. 2022, §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 [Matsushita et al. 2021] 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) [Matsushita et al. 2022] is a formalization of the RustHorn approach in Rocq. 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 [Gäher et al. 2024] 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 Rocq 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 [Lattuada et al. 2023] 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[10]: the present work should be relevant.

Aeneas [Ho and Protzenko 2022] 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 [Astrauskas et al. 2022] translates Rust programs to the Viper [Müller et al. 2016] verification language, by encoding Rust's ownership discipline (including borrows) into affine capabilities [Astrauskas et al. 2019]. 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 postconditions 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 [Dross and Kanig 2020; Jaloyan et al. 2020]. 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.

Type invariants in other deductive verification tools. Other tools like Why3 [The Why3 development team 2025] or Dafny [The Dafny development team 2025] chose a different approach to encode type invariants. 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 postconditions of executable 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 $\bot$).

DATA-AVAILABILITY STATEMENT

If the reviews are positive, we will submit for Artifact Evaluation the source code of our version of Creusot, extended with the features presented in this paper. This source code contains the benchmarks described in §6 in the `tests` directory.

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

In the case reviewers want to access the source code before the Artifact Evaluation, they can download it as supplementary material for this submission.

REFERENCES

Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods*. LNCS, Vol. 13260. Springer. https://doi.org/10.1007/978-3-031-06773-0_5

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust types for modular specification and verification. *Proc. ACM Program. Lang.* 3, OOPSLA (2019). https://doi.org/10.1145/3360573

Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *TPHOL (LNCS, Vol. 5674)*. Springer. https://doi.org/10.1007/978-3-642-03359-9_2

Xavier Denis and Jacques-Henri Jourdan. 2023. Specifying and Verifying Higher-order Rust Iterators *(LNCS, Vol. 13994)*. Springer. https://doi.org/10.1007/978-3-031-30820-8_9

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: a Foundry for the Deductive Verification of Rust Programs *(LNCS, Vol. 13478)*. Springer. https://hal.inria.fr/hal-03737878

Claire Dross and Johannes Kanig. 2020. Recursive Data Structures in SPARK. In *CAV (LNCS, Vol. 12225)*. Springer. https://doi.org/10.1007/978-3-030-53291-8_11

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

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

Georges-Axel Jaloyan, Claire Dross, Maroua Maalej, Yannick Moy, and Andrei Paskevich. 2020. Verification of Programs with Pointers in SPARK. In *ICFEM* (Singapore). Springer. https://doi.org/10.1007/978-3-030-63406-3_4

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2017). https://doi.org/10.1145/3158154

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA (2023). https://doi.org/10.1145/3586037

Nicholas D. Matsakis and Felix S. Klock II. 2014. The Rust language. *SIGAda Ada Letters* 34, 3 (2014). https://doi.org/10.1145/2692956.2663188

Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *PLDI* (2022-06-09). https://doi.org/10.1145/3519939.3523704

Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *TOPLAS* 43, 4 (2021). https://doi.org/10.1145/3462205

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (LNCS, Vol. 9583)*. Springer. https://doi.org/10.1007/978-3-662-49122-5_2

The Dafny development team. 2025. The Dafny programming and verification language. https://dafny.org/

The Why3 development team. 2025. The Why3 verification platform. https://why3.org/