A Hybrid Approach to Semi-automated Rust Verification

SACHA-ÉLIE AYOUN, Imperial College London, UK
XAVIER DENIS, ETH Zurich, Switzerland
PETAR MAKSIMOVIĆ, Nethermind, UK and Imperial College London, UK
PHILIPPA GARDNER, Imperial College London, UK

We propose a hybrid approach to end-to-end Rust verification where the proof effort is split into powerful automated verification of safe Rust and targeted semi-automated verification of unsafe Rust. To this end, we present Gillian-Rust, a proof-of-concept semi-automated verification tool built on top of the Gillian platform that can reason about type safety and functional correctness of unsafe code. Gillian-Rust automates a rich separation logic for real-world Rust, embedding the lifetime logic of RustBelt and the parametric prophecies of RustHornBelt, and is able to verify real-world Rust standard library code with only minor annotations and with verification times orders of magnitude faster than those of comparable tools. We link Gillian-Rust with Creusot, a state-of-the-art verifier for safe Rust, by providing a systematic encoding of unsafe code specifications that Creusot can use but cannot verify, demonstrating the feasibility of our hybrid approach.

CCS Concepts: • **Theory of computation** → *Separation logic*; **Automated reasoning**; **Program verification**.

Additional Key Words and Phrases: Rust, semi-automatic verification, symbolic execution, compositionality

1 Introduction

Rust [26, 33] has seen rapid adoption in recent years, particularly in the field of *systems programming*. Its success primarily stems from its rejection of false dichotomies between safety and performance: its *ownership type system* and *borrow checker* preserve memory safety while not needing garbage collection. With this success, however, also comes the need for stronger formal guarantees about the *behaviour* of Rust programs, resulting in the development of tools such as Aeneas [12], Creusot [8] and Prusti [2]. These tools all leverage the properties of the Rust type system to simplify verification, but all also share a common limitation: they can only verify *safe* Rust code.

Real-world Rust code, however, commonly relies on *unsafe* code to interface with the underlying operating system or provide low-level abstractions. Unsafe code gives the programmer 'superpowers', such as the ability to dereference raw pointers, cast between types, and manipulate potentially uninitialised memory. It is an essential part of Rust's design, allowing for new *safe* abstractions, such as `LinkedList<T>` (the type of doubly-linked lists), to be implemented efficiently in libraries. However, unsafe code also comes with greater responsibility: the onus is now on the programmer to ensure that their code does not exhibit undefined behaviour (UB) and that the corresponding APIs remain observationally safe. In addition, despite representing a fraction of the total codebase,

Authors' Contact Information: Sacha-Élie Ayoun, s.ayoun17@imperial.ac.uk, Imperial College London, London, UK; Xavier Denis, research@xav.io, ETH Zurich, Zurich, Switzerland; Petar Maksimović, p.maksimovic@imperial.ac.uk, Nethermind, London, UK and Imperial College London, London, UK; Philippa Gardner, p.gardner@imperial.ac.uk, Imperial College London, London, UK.

unsafe code is often the most complex and error-prone part of a Rust program, making it the most important one to formally verify, which none of the above-mentioned tools is able to accomplish.

We propose a **hybrid** approach to end-to-end Rust verification which, mirroring the differences between safe and unsafe code, leverages Creusot for verification of safe code and a novel tool, *Gillian-Rust*, for verification of unsafe code, which can be specified but not verified by Creusot.

Understanding the substantial challenges that Gillian-Rust had to overcome requires in-depth knowledge of the related foundational work. In 2018, Jung et al. published RustBelt [17], a theoretical framework that allows for semantic interpretation of Rust ownership types using Iris [18] and that can reason about type safety (TS). In 2022, RustHornBelt [27] extended RustBelt with the ability to reason about functional correctness (FC), allowing for safe functions implemented with unsafe code to be given first-order specifications and providing the meta-theory that now underpins Creusot. Both RustBelt and RustHornBelt, however, work on λ_{Rust}, a model that makes simplifying assumptions expected of a foundational formalisation and does not capture the intricacies of real Rust. Moreover, RustHornBelt proofs are done manually in Rocq [31], on code ported by hand from Rust to λ_{Rust}, with little automation provided. More recently, RefinedRust [10] demonstrated how advanced automation techniques from Refined-C [30] can be adapted to RustBelt to reason about FC of Rust programs. However, RefinedRust remains embedded in Rocq, which inherently limits its automation and performance. We argue that more efficient and scalable tooling is needed in order for verification to tackle the volume of existing and future unsafe Rust code.

Challenge 1: Tractable Automated Reasoning about the Real-World Rust Heap. Real-world Rust comes with numerous systems-related complications, some known from C (e.g., low-level data representation and byte-level value manipulation) and some new ones (e.g., zero-sized types, compiler-chosen layouts (C has a standardised layout), and polymorphism). While these aspects remain invisible when using safe Rust, they become a proper concern when working with unsafe code. For example, a verifier must reason generically over all possible memory layouts of programs so that it could detect any potentially disallowed memory operations. This makes reuse of existing memory models from C verification difficult and requires development of new techniques to reason automatically and efficiently about real-world Rust and the way it represents objects in memory.

Challenge 2: Type Safety (TS), Borrows, and Raw Pointers. The notion of TS is much stricter in Rust than in languages like C. Specifically, the responsibility of a safe function, even an internally unsafe one, goes beyond its own body: it must ensure that no fully-safe program calling it may trigger UBs. This substantially increases the complexity of integrating unsafe code into a Rust program.

The way Rust guarantees TS is through its strict and static ownership discipline, wherein each value must always have *an exclusive owner*. While this alone would be too restrictive, Rust also provides mutable references ($\&_{\mathsf{mut}}^{\kappa} \mathsf{T}$) and shared references ($\&^{\kappa}\mathsf{T}$) which may *borrow* ownership for a *lifetime κ*. However, even when equipped with references, safe Rust is sometimes too restrictive and prevents the implementation of types such as *doubly-linked lists*, where each node is referenced by two pointers at any time (cf. Fig. 1, bottom left), breaking exclusive ownership. In such cases, developers must resort to unsafe code in order to manipulate raw pointers ($*\mathsf{mut}\ \mathsf{T}$) which, unlike references, allow for unrestricted aliasing and do not provide any safety guarantees. This mixed use of raw pointers and safe references even further complicates the task of verifying TS of unsafe code, as it requires reasoning about lifetime-dependent safety invariants.

Challenge 3: Scaling Safe and Unsafe Rust Verification, Together. While unsafe code is used to perform some of the most complex and primitive operations of Rust programs, it still comprises a small fraction of the total codebase [1]. Furthermore, safe Rust often uses many advanced features, such as higher-order functions, which are eschewed in unsafe code. For this reason, we believe that it would be highly challenging to build a tool that both has the required expressivity for reasoning

about unsafe code, which makes extensive unrestricted use of raw pointers, and can, at the same time, reason *efficiently* and *automatically* about higher-level features used in safe Rust.

On the other hand, tools such as Creusot [7, 8] have demonstrated that verification of *safe Rust only* can be performed with impressive automation and simplicity. Ideally, one would use such a tool for the safe part of a codebase, and another, more adapted tool, for analysing the unsafe part, dividing the proof effort appropriately. This approach, however, requires both tools to agree on the semantics of *specifications* given to Rust functions. For example, if Creusot is used for safe code, the other tool has to provide a faithful interpretation of Creusot's specifications, which use a simple-to-write yet complex-to-interpret prophetic assertion language.

Contributions and Paper Outline. We present a *hybrid* approach to Rust verification, which leverages the strengths of specialised tools operating in unison to verify both safe and unsafe Rust code, illustrated in the diagram on the right-hand side.

In particular, we combine Creusot, an existing tool for safe Rust verification, with Gillian-Rust, a novel proof-of-concept semi-automated verification tool for unsafe Rust, built on top of the Gillian compositional symbolic analysis platform [25]. We manage the boundary between the tools through a shared specification language that can easily be interpreted into either Creusot or Gillian-Rust specifications. To make this possible, Gillian-Rust implements and automates the reasoning of RustBelt and RustHornBelt, which allows it to reason about the *prophetic specifications* of Creusot.

We demonstrate the viability of our approach by verifying *actual* Rust standard library code (specifically, the `LinkedList` and `Vec` types), along with several other case studies. Our approach performs verification at least two orders of magnitude faster than prior works, made possible by the use of symbolic execution and the efficient memory model of Gillian-Rust.

The paper is structured as follows. In §2, we give an overview of our hybrid approach. In §3, we propose a novel symbolic memory model for Rust compatible with Gillian, capable of both layout-independent reasoning about Rust memory and performing pointer arithmetic and bit-level operations. In §4, we demonstrate how to leverage Gillian's unique extensibility to encode concepts from the *lifetime logic* of RustBelt and obtain a substantial degree of automation, enabling Gillian-Rust to reason about TS of mutable references. In §5, we show how to embed within Gillian-Rust the ability to reason about parametric prophecies as proposed by RustHornBelt, enabling FC verification. In §6, we describe end-to-end verification of a safe-unsafe Rust program, elaborating on the interpretation of hybrid specifications into Creusot and Gillian-Rust specifications and the details of Gillian-Rust automation. In §7, we evaluate Gillian-Rust by verifying TS and FC of several Rust standard library types and their safe clients, demonstrating the efficiency and scalability of our hybrid approach. Finally, we discuss the current limitations of Gillian-Rust in detail and provide a pathway towards overcoming these limitations (§8), place Gillian-Rust in the context of overall related work (§9), and give concluding remarks (§10).
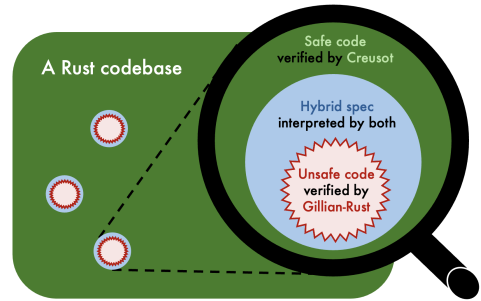
2 Overview

We present our hybrid approach in more detail, show how Gillian-Rust can be used for proving a Creusot specification, and describe the structure of Gillian-Rust as an instantiation of Gillian.
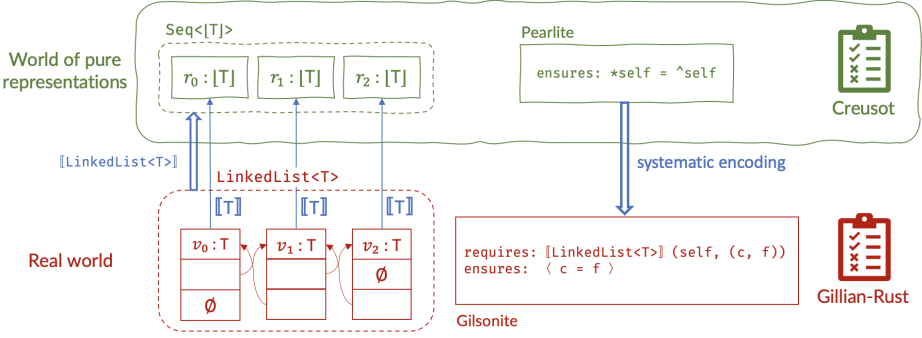
Fig. 1. A high-level illustration of the differences and connections between the world of pure representations, observed by Creusot, and the world of real representations, observed by RustHornBelt and Gillian-Rust.

2.1 A Hybrid Approach: Creusot + Gillian-Rust

The unmatched simplicity of Creusot specifications and the extent of its proof automation come from the fact that its proofs do not manipulate the real representation of objects, but a pure abstraction instead. Take, for example, doubly-linked lists, which are infamously difficult both to implement in Rust and to specify without separation logic (SL). Creusot, when performing the proof for code that uses the Rust LinkedList module, does not see its low-level representation but instead models the linked list as a sequence of values. This approach, made possible by the guarantees provided by safe Rust, sacrifices the ability to reason about the LinkedList implementation in exchange for an efficient encoding into SMT, a high degree of automation, and no need for SL.

RustHornBelt provides a foundational argument for the validity of this approach by connecting the real world to Creusot's world of pure representations. This is done by providing *ownership predicates* for each type T, which describe the safety invariant that the values of this type must uphold and connect it to the associated pure representation of type $\lfloor T \rfloor$ (cf. Fig. 1 (left)).

To verify real-world Rust, we propose a hybrid approach where Creusot verifies all proof obligations within its reach and delegates unsafe code verification to a tool dedicated for that purpose. As such a tool does not yet exist, we develop a proof-of-concept called Gillian-Rust, which has the ability to perform SL reasoning required for the verification of unsafe code, breaking the abstraction and manipulating ownership predicates directly.

A keystone to this approach is the ability to systematically encode Creusot specification, written in an assertion language called *Pearlite*, into the assertion language of Gillian-Rust, which we dub *Gilsonite*, as represented in Fig. 1 (right), and detailed in §6.

2.2 Example Usage of Gillian-Rust

Doubly-linked lists are notoriously difficult to implement in Rust: the presence of back edges violates the strict ownership discipline imposed by the use of mutable references. Instead, one must use mutable *raw pointers*, as per the code below, making doubly-linked lists a canonical example of a data structure requiring an unsafe implementation. On top, the non-trivial invariant that the list can be integrally traversed in both directions without cycles must be upheld, as otherwise the function in charge of disposing the list would visit a node twice, thereby performing a double-free.

```
struct Node<T>     { elem: T, next: Option<NonNull<Node<T>>>, prev: Option<NonNull<Node<T>>> }
struct LinkedList<T> { head: Option<NonNull<Node<T>>>, tail: Option<NonNull<Node<T>>>, len: usize }
```

We show the process of using Gillian-Rust to prove a Pearlite specification for the push_front function of the Rust standard library, which in-place adds an element to the front of a LinkedList.

Implementing Ownable. First, we connect the real Rust structure to its pure representation used by Creusot by implementing the `Ownable` *trait*[1] and defining: the type of its representation, `ReprTy` (denoted by $\lfloor \cdot \rfloor$ in mathematics); and the ownership predicate, `fn own`, which takes two parameters: the

```
impl<T : Ownable> Ownable for LinkedList<T> {
  type ReprTy = Seq<T::ReprTy>;
  #[predicate]
  fn own(self, repr: Self::ReprTy) {
    dllSeg(self.head, None, self.tail, None, repr) *
    (self.len == repr.len()) }
}
```

structure itself (`self`) and the representation. The implementation of `Ownable` for `LinkedList<T>` is given on the right-hand side (for the `dllSeg` predicate, see [4]): its representation type is a sequence of elements of type `T::ReprTy`. Note that, in order for this type to be properly defined, `T` itself must implement `Ownable`, a constraint specified using a trait bound (the ': `Ownable`' part in `<T : Ownable>`).

Type Safety (TS). Reasoning modularly about TS of unsafe code is challenging, and involves non-trivial implicit proof obligations. This is due to the fact that, by nature, type safety of a library is a global property, as "unsafety" may escape the scope of a single unsafe function [14]. Thankfully, RustBelt provides a way to reason about TS of a function in isolation. Specifically, ownership predicates capture the invariant that must be upheld by the structure to ensure TS. In Gillian-Rust, once the ownership predicate for `LinkedList<T>` has been defined by the user, we can verify TS of a function by simply adding the `#[show_safety]` attribute on top, as follows:

```
#[show_safety]
// Expands to: #[specification( requires { self.own(_) * e.own(_) } ensures { result.own(_) })]
fn push_front(&mut self, e : T) { ...implementation... }
```

This attribute expands to a Gilsonite specification which requires the ownership predicate of all input parameters to hold when entering the function, and ensures that the ownership predicate of the return value holds on function return. Here, the function `push_front` receives a mutable reference to a `LinkedList<T>` as an argument, and the ownership predicate of mutable references, initially formalised in RustBelt (and later extended by RustHornBelt), is automatically derived by Gillian-Rust. The rules that allow for manipulating the ownership invariant of mutable references are challenging to automate, and we detail our approach to this in §4. In addition, as the function is implicitly parametrised by the lifetime of the mutable reference, a *lifetime token* is added automatically by the Gillian-Rust compiled (cf. Fig. 3). Gillian-Rust is able to prove this specification fully automatically.

Functional Correctness. Next, our goal is to specify that the function actually performs the desired operation. This can be elegantly done in Pearlite by describing the update performed on the sequence which represents the `LinkedList`: *when the mutable reference expires*, the representation of the mutable reference will be its representation when the function is entered with the element prepended. Representations are accessed using the postfix operator `@`, the current value of a mutable reference is accessed using the dereference prefix operator `*`, and the value of a mutable reference at the time it expires is accessed using the prophecy prefix operator `^`:

```
#[requires((*self)@.len() < usize::MAX@)]
#[ensures((^self)@ == (*self)@.prepend(e))]
fn push_front(&mut self, e : T) { /* Implementation ... */ mutref_auto_resolve!(self) }
```

Pearlite, inspired by RustHorn [28], uses prophecy variables and the final value operator `^` in order to specify such a property. RustHornBelt provides the theory underpinning this, and we provide a high-level description of the corresponding proof techniques as well as their implementations and automation strategies in Gillian-Rust in §5. Using our systematic encoding, we can translate this Pearlite specification into a Gilsonite specification: this particular translation is given in §6, together with further explanations. Finally, after adding a single line which triggers a semi-automatic tactic during verification, Gillian-Rust is able to prove this specification.

---

[1]A trait is, akin to a Haskell typeclass, a form of interface describing a list of items that can be implemented for a type.

2.3 Building Gillian-Rust on top of Gillian

Gillian-Rust is an instantiation of Gillian [9, 25], a multi-language compositional symbolic execution platform. To instantiate Gillian to a target language (TL), one must implement a symbolic state model of the TL in OCaml, exposing: a representation of the symbolic TL state, as an OCaml type; *actions*, which are primitive operations for manipulating the state; and *core predicates*, which are the building blocks of an SL assertion language for describing states, and which allow one to write function specifications, user-defined predicates (e.g., for describing data structures), loop invariants, and proof tactics (e.g., predicate folding/unfolding). One must also implement a compiler from the TL to Gillian's intermediate language (which is a simple goto-based intermediate language parametric on the above-mentioned actions), and from TL assertions to Gillian assertions (which are parametric on the above-mentioned core predicates).

In Gillian-Rust, symbolic states have the form $\sigma = (h, \xi, \gamma, \phi, \chi)$, comprising: a symbolic heap h (§3); a lifetime context ξ (§4.1); a guarded predicate context γ (§4.2); an observation context ϕ (§5.2); and a prophecy context χ (§5.3).

Gillian action execution is described using judgements of the form $(\sigma, \pi).act(\vec{v}) \rightsquigarrow ((\sigma', v_o), \pi')$, the meaning of which is that: in the symbolic execution configuration (σ, π) where σ is a symbolic state and π is a path condition (i.e. a first-order formula constraining the symbolic variables), executing action *act* with arguments $\vec{v}$ yields a state σ', value v_o, and path condition π'. Expectedly, symbolic execution may branch, that is, executing an action may produce several outcomes.

For each core predicate ρ, Gillian requires two actions: the *consumer*, $cons_\rho$, which removes the resource corresponding to ρ from a given symbolic state; and the *producer*, $prod_\rho$, which does the opposite. On top, Gillian extends consumption and production to entire assertions, enabling compositionality (through reuse of function specifications) and predicate folding and unfolding. This is what makes Gillian uniquely extensible in the space of semi-automated compositional verification tools, as it allows one to automate the basic rules of their custom SL. Under the hood, consumption and production are powered by an assertion matching mechanism that enforces predictable, backtrack-free proof search [23–25]. Further, the predicate folding is almost fully automated, while predicate unfolding is performed heuristically. The strong performance of Gillian is evidenced by the verification times obtained for real-world JavaScript, C, and now Rust code.

3 Reasoning about the Real Rust Heap

While RustBelt provides the theoretical framework on which our work is founded, it intentionally avoids the challenge of reasoning about the real Rust heap by instead defining an operational semantics and type system for λ_{Rust}, a small lambda-calculus with a simplified memory model. For example, in λ_{Rust}, all integers are unbounded and take one cell in memory, ignoring the 12 different primitive machine integer types offered by Rust, which take between 1 and 16 bytes in memory.

The literature, from previous work on other systems programming languages such as C, already has ways of reasoning about machine integers, but Rust also comes with challenges currently undealt with. In particular, while C comes with a specific algorithm that describes and decides on the layout of structures in memory and allows for arbitrary pointer arithmetic to access structure fields, the Rust compiler provides fewer guarantees, reserving the right to re-order fields and adjust padding between them. Rust also has features that do not exist in C, such as enums (tagged unions), which offer even fewer guarantees, as Rust may manipulate fields arbitrarily to reduce the overall size of the structure without affecting expressivity, in a process called niche optimization.

So far, Rust symbolic execution tools have been working around these issues. For example, Prusti encodes structures using the object-oriented memory model of Viper, allowing efficient field access but preventing pointer arithmetic reasoning, and Kani compiles Rust to a C-like representation

by choosing a specific layout for each structure, dropping the guarantee that a verified program would be correct had the compiler made different layout choices authorised by the language [11].

In this section, we describe the solution provided by Gillian-Rust, which does a best-effort attempt at *maintaining abstraction*—hence preserving field-access efficiency—while still allowing for pointer arithmetic by leveraging Gillian's ability to implement custom heap models directly in OCaml. We show how to encode addresses so that they are layout-independent, describe a novel representation of objects in the heap that allows for efficient automated reasoning, and present the points-to core predicate, which allows for specifying the Rust heap in Gillian-Rust.

3.1 Layout-Independent Memory Addresses

The representation of addresses in Rust constitutes a challenge on its own. Ideally, one would prefer to reuse the one used by Gillian-C, inspired by CompCert [22] and also used in RustBelt, where an address is a pair $(l, o) \in Loc \times \mathbb{N}$ of an object location (identifying a unique allocation) and an offset. However, because of the above-mentioned challenges, this representation is insufficient, as structure field access may correspond to different offsets depending on the compiler-chosen layout.

To overcome this issue, Gillian-Rust modifies the encoding of offsets by using sequences of *projection elements* forming a *projection* (we reuse the compiler's internal terminology) instead of a natural number. Specifically, a projection element

$$l \in Loc \quad e \in \widehat{\mathbb{Z}} \quad i, j \in \mathbb{N}$$
$$\text{pr} \in \text{ProjE} \quad ::= \quad +^{\mathsf{T}}e \mid .^{\mathsf{T}}i \mid .^{\mathsf{T} \cdot j}i$$
$$a \in Addr \quad ::= \quad (l, \vec{\text{pr}})$$

represents either: an offset of e times the size of the type T, where e is a symbolic integer, denoted by $+^{\mathsf{T}}e$; or the offset of the i-th field of a structure (relative w.r.t. the beginning of the structure), denoted by $.^{\mathsf{T}}i$; or the relative offset of the i-th field of the j-th variant of an enum, denoted by $.^{\mathsf{T} \cdot j}i$.

This representation makes the interpretation of a symbolic address effectively parametric on the layout chosen by the compiler: given a layout which provides a concrete offset for each field of a structure or an enum, and a size to every type, each projection element can be interpreted as a symbolic natural number, and each projection as the sum of the interpretations of its elements.

3.2 Objects in the Rust Symbolic Heap

Our goal is to represent objects in the symbolic heap in a way that would enable us to efficiently resolve field accesses and perform only layout-independent pointer arithmetic. To this end, we propose a hybrid tree representation featuring two kinds of nodes: *structural nodes*, which represent a region of memory for which we know the structure but not necessarily the layout (such as Rust structures or enums), and on which no pointer arithmetic is allowed; and *laid-out nodes*, which are known to have an array-like layout and admit certain pointer arithmetic. For clarity of presentation, we provide a high-level description of the heap, focussing on the main functionalities and insights.

Structural Nodes. Structural nodes are annotated with their type, and may be one of the following:

- a single node containing either: the special value Uninit, representing uninitialised memory, which is illegal to read; the special value Missing, representing memory that has been framed off; or a symbolic value;
- a tree representing a structure, consisting of: a root (internal) node, which holds no information; and children nodes, which represent its fields; or
- a tree representing an enum with a concrete discriminant[2], containing: an internal node holding said discriminant; and children nodes representing the fields of the corresponding enum variant.

The types annotating the nodes must be *sized* (i.e., must have a size known at compile-time[3]), thereby providing an interpretation for each node. The *load* and *store* primitive operations are

---

[2]A symbolic enum (i.e., an enum with a symbolic discriminant) would be represented as a single node with a symbolic value.
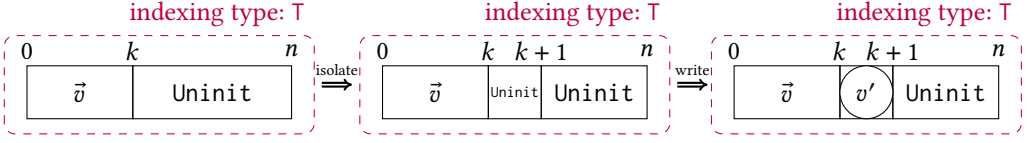[3]In contrast to unsized types, such as the slice type $[\mathsf{T}]$, for which the size is only known at run-time.

Fig. 2. Update of a laid-out node corresponding to $n * \text{size\_of::<T>()}$ bytes.

provided in the interface of the symbolic heap and must ensure that the validity invariants [15] of values written in memory are maintained (e.g., that booleans are represented by bit-patterns 0b0 and 0b1 only). They are also responsible for enforcing other important aspects of the Rust semantics, such as that loading a value from memory in the context of a *move* will deinitialise that memory.

In the diagram below, we give an example of a structure S and its structural node representation, comprising an internal node annotated with type S and two single-node children with respective values and types $(x, u32)$ and $(y, u64)$. The type of the left child, for example, indicates that it represents a region of 4 bytes in memory, and that the symbolic value x is an integer in the range $[0, 2^{32})$. We also show two potential interpretations of a structural node for S, depending on the compiler-chosen field ordering: the top interpretation is obtained when the ordering is from-largest-to-smallest, and the bottom when the ordering is from-smallest-to-largest, inserting the appropriate padding when needed. This structural node in particular can only be navigated using $.^S 0$ or $.^S 1$.

Laid-out Nodes. While structural nodes facilitate efficient resolution for a large majority of memory accesses, they are not a novel concept. The novelty of our approach lies in combining structural nodes with laid-out nodes, inspired by Gillian-C [25], which describe a region of memory with an array-like layout in the sense that it allows for basic indexing pointer arithmetic. For example, Rust arrays, which are at the core of the Rust vector type, are always laid out contiguously such that the n-th element of an array of type $[T; N]$ starts at offset $n * \text{size\_of::<T>()}$ w.r.t the beginning of the array, regardless of the layout of the element itself. Similarly, any integer type, say u32, can be seen as array-like as it is always represented by contiguous bytes in memory.
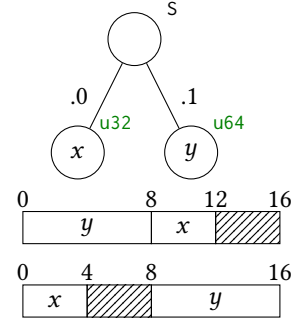
A laid-out node is a pair composed of a sized type (called indexing type) and a list of structural nodes each annotated with the range it occupies in multiples of the size of the indexing type. For example, Fig. 2 (left) shows a laid-out node with indexing type T and two structural nodes, the first carrying a symbolic list value $\vec{v}$ occupying the range $[0, k)$ (note that the k is symbolic), and the second capturing uninitialised memory occupying the range $[k, n)$, with $k < n$.

When resolving pointer arithmetic, Gillian-Rust is able to automatically destruct and reassemble laid-out nodes, allowing for arbitrary range access and manipulation. For example, Fig. 2 (middle) and (right) show the process of writing a single value of type T at the k-th offset; this corresponds to pushing at the end of a vector with sufficient capacity. Gillian-Rust achieves this by first isolating the region in which the newly added value is going to be written (Fig. 2, middle), splitting the second node into two, and then overwriting the appropriate region (in this case, from k to $k + 1$) with a structural node corresponding to the added value (Fig. 2, right), simplified for this example to be a single node. Importantly, the indexing type does not have to match the type of each individual sub-node. For example, explicit calls to the Rust allocator API will always result in a laid-out node with indexing type u8 (i.e., single bytes), but can be populated with values of arbitrary other type T.

3.3 Specifying the Rust Heap: The Typed Points-to Core Predicate

We focus on the most important core predicate used to specify heap shape with Gilsonite: the typed points-to predicate, $a \mapsto_\mathsf{T} v$, which is satisfied by a heap fragment starting from address a and containing `size_of::<T>()` bytes, which together form a valid representation of the value v. The remaining core predicates are only variations on this theme and are used for specifying, for example, slices or potentially uninitialised memory.

The separation logic induced by the core predicates can be used by the verification engineer to specify a variety of predicates, pre-conditions and post-conditions. For example, the typed points-to predicate is enough to specify the ownership predicate for the `LinkedList` type of the standard library, which is described in detail in [4].

4 Automating Reasoning about Mutable Borrows

Handling mutable borrows is one of the main challenges when trying to specify and verify Rust programs in fully-safe and unsafe contexts alike. While RustBelt [17] provides a theoretical framework for reasoning about mutable borrows within Iris and proves its correctness in Rocq, this reasoning itself is manual and slow. In this section, we show how to leverage the unique flexibility of Gillian to automate reasoning about lifetimes and basic operations on mutable borrows.

4.1 Modelling Lifetimes: Core Predicates

In Rust, a lifetime is a type-level variable representing a period of time during which a reference is valid. It is the responsibility of the borrow checker of the compiler to compute sound lifetimes for all references so that the ownership discipline of Rust is maintained.

In RustBelt, lifetimes are encoded as *tokens* in its separation logic: the token $[\kappa]_q$, with $0 < q \le 1$, represents an alive lifetime κ, while $[\dagger\kappa]$ denotes that the lifetime κ has expired. RustBelt also provides rules to reason about lifetime tokens, some of which are included below for illustrative purposes: e.g., Lft L-not-own-end states that a lifetime cannot be alive and expired at the same time; Lft L-end-persist states that an expired lifetime token is persistent (i.e. it can be duplicated); while Lft L-tok-fract states that alive lifetime tokens may be split into fractions (for $0 < q, q'$).

Lft L-not-own-end	Lft L-end-persist	Lft L-tok-fract
$[\kappa]_q * [\dagger\kappa] \Rightarrow \mathrm{False}$	$\mathrm{persistent}([\dagger\kappa])$	$[\kappa]_{q+q'} \Leftrightarrow [\kappa]_q * [\kappa]_{q'}$

A lifetime context ξ is then a partial finite map from lifetimes to either the currently owned fraction of the lifetime token (a symbolic real number in the $(0, 1]$ interval), or an indicator of expiration, $\dagger$.

$$\kappa \in Lft \approx \mathcal{P}(\mathbb{N})$$
$$\xi \in Lctx = Lft \rightharpoonup^{fin} \widehat{\mathbb{R}}^\dagger_{(0,1]}$$

In Gillian-Rust, both kinds of tokens become core predicates, and we demonstrate how the three RustBelt rules shown above are automated by providing an excerpt of the rules governing their consumers and producers in Fig. 3.[4] While simple, these rules are illustrative of the relationship between custom consumers/producers and automation. For example, the rule Lft-Produce-Alive-Add adds a fraction q of an alive token when a fraction q' is already owned, automating the right-to-left implication of Lft L-tok-fract. On the other hand, Lft-Produce-Own-End vanishes (i.e. assumes False) when producing an alive token in a context where the lifetime has expired, automating Lft L-not-own-end. Similarly, in the consumer/producer paradigm, a core predicate is made persistent when its producer is idempotent and its consumer does not modify memory. Hence, together, rules Lft-Consume-Exp and Lft-Produce-Exp-Dup automate Lft L-end-persist.

---

[4]In these rules, to avoid clutter: the judgement uses only the lifetime context instead of the entire symbolic state; and the return value is elided because both actions return unit.

$$\frac{\text{LFT-PRODUCE-ALIVE-ADD}}{\xi(\kappa') = q' \qquad \pi \vdash (\kappa = \kappa' \wedge 0 < q \wedge q + q' \leq 1) \qquad \xi' = \xi\,[\kappa \leftarrow q + q']}{(\xi, \pi).\mathrm{prod}_{[\cdot]}\,(\kappa, q) \rightsquigarrow (\xi', \pi)}$$

$$\frac{\text{LFT-PRODUCE-OWN-END}}{\xi(\kappa') = \dagger \qquad \pi \vdash (\kappa = \kappa')}{(\xi, \pi).\mathrm{prod}_{[\cdot]}\,(\kappa, q) \quad \textbf{vanishes}} \qquad \frac{\text{LFT-CONSUME-EXP}}{\xi(\kappa') = \dagger \qquad \pi \vdash (\kappa = \kappa')}{(\xi, \pi).\mathrm{cons}_{[\dagger\cdot]}\,(\kappa) \rightsquigarrow (\xi, \pi)} \qquad \frac{\text{LFT-PRODUCE-EXP-DUP}}{\xi(\kappa') = \dagger \qquad \pi \vdash (\kappa = \kappa')}{(\xi, \pi).\mathrm{prod}_{[\dagger\cdot]}\,(\kappa) \rightsquigarrow (\xi, \pi)}$$

Fig. 3. Consumer and producer rules for lifetime tokens (simplified, excerpt)

4.2 Modelling Full Borrows: Guarded Predicates

In Rust, a mutable reference of a value of type T during lifetime κ, denoted by $\&_{\mathsf{mut}}^{\kappa}\mathsf{T}$, corresponds to *temporary* ownership of the reference and the value it points to. To model such a behaviour, RustBelt introduced full borrows, denoted by $\&^{\kappa}P$, which are higher-order predicates denoting that the resource described by assertion P is borrowed during lifetime κ. In RustBelt, where ownership predicates do not expose a pure representation, the ownership predicate of a mutable reference p and the key rules for manipulating mutable borrows are as follows:

$$[\![\&_{\mathsf{mut}}^{\kappa}\mathsf{T}]\!](p) \triangleq \&^{\kappa}(\exists v.\ p \mapsto v * [\![\mathsf{T}]\!](v)) \qquad \begin{array}{c}\text{LFTL-BORROW-ACC}\\ \&^{\kappa}P * [\kappa] \Rrightarrow\!\!\!\ast\ \rhd P * (\rhd P \Rrightarrow\!\!\!\ast \&^{\kappa}P * [\kappa])\end{array}$$

In particular, LFTL-BORROW-ACC states that one may *open a borrow* by temporarily giving up the corresponding lifetime token, and may later *close that borrow* after having reformed the invariant, at which point the token is recovered. Crucially, having to reform the invariant inside a borrow is what ensures that a callee function which is given a borrow may not cause undefined behaviour in the future, and every borrow must eventually be closed, as the lifetime token is required at the time it expires. In Gillian-Rust, the view shift operator present in the LFTL-BORROW-ACC rule is realised via guarded predicate unfolding, introduced shortly, whereas the later modality, $\rhd$, is omitted; in §8, we provide a justification for the soundness of this approach.

Full borrows raise two main challenges for a semi-automated tool such as Gillian: **1)** it needs to reason about higher-order predicates; and **2)** it needs to automatically understand when to open and close borrows in common proof patterns. We now present the two key insights behind the encoding and automation of reasoning about full borrows in Gillian-Rust.

Compiling Higher-Orderness Away. While program proofs do make use of higher-order rules such as LFTL-BORROW-ACC, they only use them with a specific, finite set of instantiations. For example, when proving pop_front_node, one only needs to manipulate the particular borrow predicate corresponding to the ownership predicate $[\![\&_{\mathsf{mut}}^{\kappa}\mathsf{LinkedList}\text{<}\mathsf{T}\text{>}]\!]$. When using the Gilsonite API, a user may instantiate the full borrow assertion using the #[borrow] attribute. For instance, the ownership predicate for mutable references is defined as follows in the Gilsonite library:

```
impl<T> Ownable for &mut T { #[borrow] fn own(self) -> Gilsonite { exists v. (self -> v) * v.own() } }
```

obtaining an ownership predicate for mutable references of type T. Note that such predicates can be defined parametrically, using a generic type; when required for a more specific type, such as LinkedList<T>, they will be instantiated at compilation time.

Finally, ownership predicates for type parameters are compiled to abstract predicates, that is, predicates that cannot be unfolded, a well-known trick in the world of semi-automated tools. This ensures that if a specification has been proven using a type parameter T, then this type parameter can be instantiated with any other type to obtain a new trusted specification, with the instantiation happening at the call site that requires it.

Leveraging Known Automations for Borrow Access. The key insight to automating borrow access is the understanding that borrows behave very similarly to standard predicates encoded in a semi-automated SL-based verification tool. In particular, VeriFast, Viper, and Gillian all support predicates of the form $(\delta, \vec{v}) \in (Str \times \text{List}(Val))$, where each predicate consists of a name δ (normally a string) and parameters $\vec{v}$. Predicates of this form are said to be *folded* and each of the above-mentioned tools maintains a list of predicates as part of their state.

Each of these tools also comes with two ghost commands that allow users to manipulate folded predicates: unfold and fold. In particular, unfold removes a predicate stored in its folded form from the state and produces its definition in its place, whereas fold is its dual, consuming the predicate's definition from the state and adding its folded form to the state.

One may notice the similarity between the borrow access rule and the folding and unfolding of predicates: when closed, both borrows and folded predicates act as abstract tokens that can be exchanged for the resource they contain. The only distinction is the "cost" of unfolding: none for predicates, and a lifetime token for borrows.

A guarded predicate context $\gamma \in \text{List}(Str \times Lft \times \text{List}(Val))$ is a list of predicates which are annotated with a lifetime such that its token is the cost for their opening. It exposes two actions: gunfold/gfold, which respectively behave like unfold/fold apart from the fact that they consume/produce that guarding lifetime token, and produce/consume an additional opaque *closing token*, denoted by $C_\delta(\kappa, q, \vec{x})$, which embodies the closing update $(P \Rrightarrow \&^\kappa P * [\kappa]_q)$.

The UNFOLD-GUARDED rule describes successful execution of gunfold. For clarity, we decompose symbolic states into a pair (μ, γ), where μ represents the remaining components. In addition, we write in purple elements of the rule which are novel with respect to the more classic unfold rule. Finally, this command is performed in the context of a program p, where p.predDefs maps predicates to their definitions.

UNFOLD-GUARDED
$$p.\text{predDefs}[\delta(\kappa, \vec{x})] = P$$
$$(\sigma, \pi).\text{cons}_{[\cdot]}(\alpha, q) \rightsquigarrow (\sigma', \pi')$$
$$\sigma' = (\mu', \gamma') \quad \delta(\alpha, \vec{v}) \in \gamma'$$
$$\gamma'' = \gamma' \setminus \delta(\alpha, \vec{v}) \quad \sigma'' = (\mu', \gamma'')$$
$$P' = P * C_\delta(\kappa, q, \vec{v})$$
$$(\sigma'', \pi').\text{prod}(P'[\vec{x}/\vec{v}]) \rightsquigarrow (\sigma''', \pi'')$$
$$\overline{p \vdash (\sigma, \pi).\text{gunfold}(\delta(\alpha, \vec{v})) \rightsquigarrow (\sigma''', \pi'')}$$

This encoding of full borrows has one important advantage: Gillian comes with years of experience in automating separation logic proofs, including heuristics that are able to decide when to automatically unfold or fold predicates as required by the analysis. By encoding borrows in the above way, we can immediately leverage those heuristics and allow for automatic opening and closing of full borrows. In particular, proving the type safety of `LinkedList::pop_front` and `LinkedList::push_front` becomes completely automatic once the safety invariants of `LinkedList` has been properly specified as in §3.3.

4.3 Proving Safety of Borrow Extraction

Unfortunately, opening and closing are not the only operations that one needs when working with full borrows. We identify several recurring patterns in unsafe Rust programs and provide ways of instantiating lemmas that allow us to analyse code that uses these patterns.

In particular, borrow extraction—the process of cutting a borrow up into a smaller borrow—is a common pattern in unsafe Rust programming, and every data-structure module of the standard library provides at least one function that uses this pattern (e.g., `LinkedList::front_mut` or `Vec::get_mut`). In fact, borrow extraction is the most idiomatic way of modifying an element of a collection. Most often, implementing such a function is unsafe, as incorrect borrow extraction could break the safety guarantees of Rust. For example, consider the case in which the `LinkedList` library implementer creates a `first_node_mut` function, which returns a mutable reference not to the first element (`&mut T`), but to the first node (`&mut Node<T>`), which contains the first element as well as next and prev pointers (Fig. 4, left). Then, **using only safe code**, a client function could modify the next pointer to point

to the node itself, creating a cycle in the list. As explained in §2.2, this would certainly lead to an undefined behaviour, although not during the execution of first_node_mut itself.

On the other hand, returning a mutable reference to the first element (&mut T), as per Fig. 4 (right), is not an issue, with the intuition being that one can *remove* the resource associated with the element and obtain a *remainder*. To that remainder one can then add any other element that satisfies the invariant of T, recovering a structure satisfying the LinkedList invariant. This principle is embodied by the BORROW-EXTRACT rule (which we have proven in Iris using RustBelt), where P is the invariant of the LinkedList, Q is the invariant of T, and $Q \twoheadrightarrow P$ is the remainder. In addition, the rule allows one to add a persistent context if it is required for performing the extraction. For example, in the case of the LinkedList, the extraction of the first node is only possible if it is not empty (i.e. if the head pointer is not None, which would be captured in that persistent context).

Fig. 4. An invalid and a valid LinkedList mutable reference.

Using the Gilsonite API, users may instantiate the ghost command that performs the view shift in the conclusion of the BORROW-EXTRACT rule by specifying the borrow predicates $\&^{\kappa} P$ and $\&^{\kappa} Q$ as well as the persistent assertion F, as illustratively done below[5]:

BORROW-EXTRACT
$$\frac{\text{persistent}(F) \qquad F * P \Rightarrow Q * (Q \twoheadrightarrow P)}{F * [\kappa]_q * \&^{\kappa} P \Rrightarrow \&^{\kappa} Q * [\kappa]_q}$$

```
#[extract_lemma( forall head, tail, len, p. assuming { head == Some(p) } // F
  from { list_ref_mut_frozen(list, head, tail, len) } // &ᵏP
  extract { Ownable::own(&mut (*p.as_ptr()).element) } // &ᵏQ
)]
fn extract_head<T: Ownable>(list: &mut LinkedList<T>); // Implicitly parametric on κ
```

Gillian itself cannot prove that BORROW-EXTRACT holds or manipulate borrows using such a rule. Instead, the Gillian-Rust compiler produces two lemmas: one corresponding to the rule conclusion, which is marked as trusted and left unproven, and one corresponding to the rule hypotheses, which needs to be proven. As the rule itself has been proven to hold in Iris, the Gillian-Rust meta-theory therefore ensures that if we prove the second lemma, the first lemma also has to hold.

To automatically prove this second kind of lemmas, we have extended Gillian with the ability to reason about magic wands, adapting the related work on Viper [6], to Gillian's parametric separation logic; the details of this extension are out of scope of this presentation.

5 Functional Correctness and Prophetic Reasoning

While the ability to manipulate full borrows is enough to verify type safety of programs that make use of mutable references, it is not enough to prove functional correctness of these programs. In particular, the rule LFTL-BORROW-ACC presented previously enforces that the **same** invariant be used to close the full borrow, effectively losing the information that the value was updated.

Specifying functional correctness of programs manipulating mutable references is, in itself, a challenge, as it requires the ability to specify properties which shall only hold *in the future*, that is, at the time when the borrow expires. Thankfully, this challenge has been addressed by previous work: Prusti [2] introduced pledges and RustHorn [28] introduced prophecy variables, later used in Creusot. However, only the latter has been given a foundational formalisation in RustHornBelt [27], an extension of RustBelt which describes how prophetic specifications interact with full borrows.
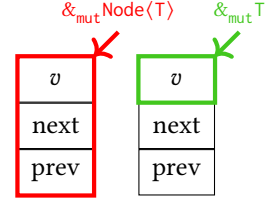
---

[5]The list_ref_mut_frozen predicate is a borrow predicate obtained from the ownership predicate of &mut LinkedList by freezing existentials corresponding to the head, tail and len fields of the structure. Freezing existentials is a common strategy for extracting borrows, supported by the Gilsonite API. For lack of space, we present this in [4].

We next recall the workings of RustHornBelt and show how its concepts are encoded in Gillian-Rust. To conclude our technical presentation, we show how Pearlite specifications are compiled to Gilsonite, explaining how unsafe proof goals can be delegated by Creusot to Gillian-Rust.

5.1 Representations, Parametric Prophecies, and Observations

In order to reason about functional correctness within the framework of RustBelt, RustHornBelt extends ownership predicates with an additional parameter corresponding to a pure mathematical *representation* of the value. Given a type `T`, the type of its representation is denoted by $\lfloor T \rfloor$. For example, a value of type `LinkedList<T>` is represented by a sequence of which each element is the representation of the element at the corresponding index in the list, i.e. $\lfloor \text{LinkedList<T>} \rfloor = \text{Seq<}\lfloor T \rfloor\text{>}$.

Mutable references, on the other hand, are represented as a pair of representations of the inner type (i.e., $\lfloor \text{\&mut T} \rfloor = \lfloor T \rfloor \times \lfloor T \rfloor$), where the first element denotes the value to which the mutable reference currently points, and the second denotes the value it will have at the time it expires.

RustHornBelt then proposes an ownership predicate for mutable references which exposes this representation, using a notion of *parametric prophecies*. A prophecy variable x is attached to the mutable reference, and the second element of the representation pair r is the future value of this prophecy, denoted by $\uparrow x$.

$$\llbracket \&^\kappa_{\text{mut}} T \rrbracket(p, r) \triangleq \exists x \text{ s.t. } r^\star 2 = \uparrow x. \, \text{VO}_x(r^\star 1) * \\ \&^\kappa(\exists v, a. \, p \mapsto v * \llbracket T \rrbracket(v, a) * \text{PC}_x(a))$$

In addition, there are two connected resources respectively called *value observer*, denoted by VO_x, and *prophecy controller*, denoted by PC_x, which together provide a solution to the problem of information loss when closing a full borrow. In particular, the observer maintains the last-observed current value and, when the borrow opens, the previously-lost value of the representation a is recovered through the

Mut-Agree
$$\text{VO}_x(a) * \text{PC}_x(a') \vdash a = a'$$

Mut-Update
$$\text{VO}_x(a) * \text{PC}_x(a) \Rightarrow \\ \text{VO}_x(a') * \text{PC}_x(a')$$

Mut-Agree rule. Before closing a borrow again, the verification engineer may use the Mut-Update rule to update the value of the prophecy variable to match the new representation.

Lastly, RustHornBelt introduces *observations*, denoted by $\langle \psi \rangle$, where ψ is a pure assertion containing information known about prophecy values. Observations act as a second layer of truth, preventing future information from leaking into the separation logic and creating paradoxes.

5.2 Key Idea: Parametric Prophecies and Symbolic Execution

In order to encode prophecies into Iris, RustHornBelt wraps the entire execution into a reader monad. In simple terms, execution is performed within a context which preemptively captures an assignment for the future value of each existing prophecy variable (i.e., a map $PcyVar \to Value$).

One of the key ideas presented in this work comes from noticing that symbolic execution in the Gillian meta-theory can be formalised using an environment of the same nature, of type $SVar \to Value$, which assigns a concrete interpretation to each symbolic variable. Therefore, parametric prophecies appear to be closer to symbolic variables than they are to prophecy variables formalised by Jung et al. [19]. This intuition suggests that one may use the same process to reason about prophecy variables as for symbolic variables, and ideally fit them into the same framework. In symbolic execution, each state carries a *path condition* π, a pure formula which accumulates all currently-known constraints about the existing symbolic variables, while for prophecy variables, it is the observations that play this role of constraint accumulator. The core idea behind encoding prophecy variables follows from this remark: observations can simply take the shape of a secondary path condition, implemented as a custom resource algebra in OCaml within the Gillian framework, making calls to the Gillian solver when required.

To this end, we introduce a new custom resource algebra in Gillian which consists of only one symbolic expression, called *observation context* and denoted by $\phi \in Obs$. The observation context

OBS-MERGE
$$\langle\psi\rangle * \langle\psi'\rangle \vdash \langle\psi \wedge \psi'\rangle$$

PROPH-SAT
$$\langle\psi\rangle \Rightarrow \exists\varepsilon.\ \varepsilon(\psi)$$

PROPH-TRUE
$$(\forall\varepsilon.\ \varepsilon(\psi)) \Rightarrow \langle\psi\rangle$$

OBSERVATION-PRODUCE
$$\frac{\pi \wedge \phi \wedge \phi' \text{ SAT}}{(\phi,\pi).\text{prod}_{\langle\cdot\rangle}(\phi') \rightsquigarrow (\phi \wedge \phi',\pi)}$$

OBSERVATION-CONSUME
$$\frac{(\pi \wedge \phi \Rightarrow \phi') \text{ VALID}}{(\phi,\pi).\text{cons}_{\langle\cdot\rangle}(\phi') \rightsquigarrow (\phi,\pi)}$$

Fig. 5. Excerpts: observation rules from RustHornBelt (top) and observation consumer/producer rules (bottom)

may depend on both prophecy variables and symbolic variables. Fig. 5 (top) presents some of
the rules that apply to observations in RustHornBelt, while Fig. 5 (bottom) shows Gillian-Rust
consumer and producer rules for the successful cases. Again, for clarity of presentation, we elide
the non-needed components of the state and the return values.

OBS-MERGE indicates that our model of observations as a single symbolic expression is appro-
priate, and that framing on a new observation amounts to simply conjuncting it with the current
observation. In addition, PROPH-SAT tells us that if an observation holds, then at least one prophecy
assignment must satisfy it. Together, these rules instruct us how to implement the producer for
observations: if the conjunction of the path condition, current observation, and new observation is
satisfiable, then we can add the produced observation to our current one (cf. OBSERVATION-PRODUCE).
Finally, PROPH-TRUE states that anything that is true independently of prophecy variables can be
captured as an observation, that is, anything that is true outside of the prophetic world is also true
within it. With our approach, this means that the path condition can be used seamlessly as part of
our observations when needed, embodied in the OBSERVATION-CONSUME rule: when checking if an
observation ϕ' holds, we check that it is entailed by the current path condition and observation.

5.3 Value Observers and Prophecy Controllers

Value observers and prophecy controllers provide yet another opportunity to leverage the flexibility
of Gillian and implement a custom resource algebra. In particular, we entirely automate the MUT-
AGREE rule by defining a *prophecy context* $\chi \in PcyVar \rightarrow Expr \times \mathbb{B} \times \mathbb{B}$ as a map that associates each
prophecy variable with its current value and two Booleans, which correspond to the ownership of
the value observer and of the prophecy controller in the state.

Below, we provide rules for successfully producing a value observer into the state; the production
rules for the prophecy controller are analogous and therefore elided:

VOBS-PRODUCE-WITHOUT-CONTROLLER
$$\frac{x \notin \text{dom}(\chi) \quad \chi' = \chi\,[x \leftarrow (a,\top,\bot)]}{(\chi,\pi).\text{prod}_{\text{VO}}(x,a) \rightsquigarrow (\chi',\pi)}$$

VOBS-PRODUCE-WITH-CONTROLLER
$$\frac{\chi(x) = (a',\bot,\top) \quad \chi' = \chi\,[x \leftarrow (a',\top,\top)]}{(\chi,\pi).\text{prod}_{\text{VO}}(x,a) \rightsquigarrow (\chi',\pi \wedge (a=a'))}$$

In particular, producing $\text{VO}_x(a)$ in a prophecy context which does not already contain any binding
for the prophecy variable x will bind x to the triple $(a,\top,\bot)$, thereby encoding that the current
value for the prophecy is a, that its value observer is in the context, but not its prophecy controller.
On the other hand, if the controller with value a' already exists in the current state, that is, if the
prophecy context already has the triple $(a',\bot,\top)$ bound to x, then the Boolean flag corresponding
to the presence of the corresponding value observer is set to true without modifying the current
value and we learn that $a = a'$, in the form of an additional constraint added to the path condition.

However, this does not automate the MUT-UPDATE rule: after modifying the contents of a mutable
reference p: &mut T, one still needs to apply this rule to be able to close the mutable borrow. The
current Gillian implementation does not allow full automation of this process, but we are able
to provide the MUT-AUTO-UPDATE lemma, which the verification engineer can apply by writing

```
1  #[pearlite::ensures(sorted((^l)@) && l@.permutation_of((^l)@))]
2  pub fn merge_sort(l: &mut LinkedList<i32>) { /* Standard impl. using split and merge */ }
3
4  #[pearlite::ensures(inp@.permutation_of(result.0@.concat(result.1@)))]
5  fn split(inp: &mut LinkedList<i32>) -> (LinkedList<i32>, LinkedList<i32>) {
6    let old_inp = snapshot!(inp);
7    let mut (left, right, push_left) = (LinkedList::new(), LinkedList::new(), true);
8    let mut popped = snapshot! { Seq::EMPTY };
9    #[pearlite::invariant(
10     popped.concat(inp@).ext_eq(old_inp@) && popped.permutation_of(left@.concat(right@))
11   )]
12   while let Some(i) = inp.pop_front() {
13       popped = snapshot! { popped.push(i) };
14       snapshot!({perm_right::<i32>; perm_left::<i32>});
15       if push_left { left.push_front(i); } else { right.push_front(i); };
16       push_left = !push_left;
17   }
18   (left, right)
19 }
20
21 #[pearlite::requires(sorted(l@))] #[pearlite::requires(sorted(r@))]
22 #[pearlite::ensures(sorted(result@) && result@.permutation_of(l@.concat(r@)))]
23 fn merge(l: &mut LinkedList<i32>, r: &mut LinkedList<i32>) -> LinkedList<i32> { ... }
```
Fig. 6. A fragment of our Merge Sort algorithm, implemented using doubly-linked lists

`p.prophecy_auto_update()`, and which updates the current value of the prophecy by automatically choosing the appropriate value that will allow the borrow to be closed.

Finally, Gillian-Rust also provides a manual way of *resolving* mutable references, as described by MUT-REF-RESOLVE, which, as proposed by RustHornBelt, allows us to obtain an observation of the equality between the current value of the prophecy and its future value at the time where the corresponding mutable reference expires.

MUT-AUTO-UPDATE
$$\llbracket \mathsf{T} \rrbracket (v, a') * \mathrm{VO}_x(a) * \mathrm{PC}_x(a) \Rightarrow$$
$$\llbracket \mathsf{T} \rrbracket (v, a') * \mathrm{VO}_x(a') * \mathrm{PC}_x(a')$$

MUT-REF-RESOLVE
$$\llbracket \&_{\mathsf{mut}}^\kappa \mathsf{T} \rrbracket (p, (a, a')) \Rrightarrow\!\!\ast \langle a = a' \rangle$$

Borrow extraction with prophecies. When manipulating the ownership predicate of a mutable reference with prophecies in the style of RustHornBelt, the rule for extracting sub-borrows must be adapted to perform *partial resolution* of the prophecy. The corresponding rule is substantially more complex than BORROW-EXTRACT, but it yields the same level of automation and we have proven it correct in the Rocq development of RustHornBelt. For lack of space, we present it in [4].

6 Anatomy of a Hybrid Proof : Merge Sort

In this section, we present a detailed example of a hybrid proof, showing how we can use Creusot and Gillian-Rust to prove the correctness of a Merge Sort implementation that uses doubly-linked lists. We briefly cover the safe implementation and its verification in Creusot, and then explain how we interface with Gillian-Rust to prove correctness of associated unsafe operations.

Writing a hybrid proof. Following the approach outlined in §2, we divide the work as follows: (1) Creusot is responsible for verifying the safe parts (here, the Merge Sort algorithm itself), which normally constitute the great majority of the code; while (2) Gillian-Rust is responsible for verifying the unsafe parts (here, the doubly-linked list operations), which are normally more low-level and perform more complex but smaller operations such as manipulation of pointers or uninitialised memory. In Figure 6, we present a fragment of our Merge Sort implementation. For space reasons, we elide the (standard) implementations of merge_sort and merge, focusing instead on the split function, which takes a mutable borrow to a linked list and splits it into two halves.

```
pub struct LinkedList<T> { ... }

impl<T : Ownable> LinkedList<T> {
  #[hybrid::ensures(forall<x : _> result == Some(x) ==> Seq::singleton(x).concat((^self)@) == (*self)@)]
  #[hybrid::ensures(result == None ==> ^self == *self && self@.len() == 0)]
  pub fn pop_front(&mut self) -> Option<T> { ... }

  #[hybrid::requires(self@.len() < usize::MAX@)]
  #[hybrid::ensures(Seq::singleton(e).concat((*self)@) == (^self)@)]
  pub fn push_front(&mut self, e: T) { ... }

  #[hybrid::ensures((*self)@.push(e) == (^self)@)]
  pub fn push_back(&mut self, e: T) { ... }
}
```

Fig. 7. The LinkedList library used by our Merge Sort algorithm

In Creusot, unsafe types such as LinkedList<T> are treated as *opaque types*, on which no operations can be performed. To reason about them, Creusot axiomatises their representation function using a ShallowModel trait, and the Pearlite[6] specifications of their APIs are assumed as axioms. We can access this shallow model through its associated operator @. Using this model operation, we specify the postcondition of the split function as per line 4 of Figure 6, stating that the concatenation of the two resulting lists is a permutation of the input list. Operations on mutable borrows are specified using the *final* operator ^, which accesses the prophecy of a mutable reference. In line 1, we specify that the initial value ((*l)@) of the list is a permutation of its final value ((^l)@).

In Figure 7, we present the specification of the LinkedList library used by our Merge Sort. We use the hybrid::requires and hybrid::ensures attributes to specify, respectively, the pre- and post-conditions of the pop_front, push_front, and push_back functions. These attributes act as the bridge between Pearlite and Gilsonite, in that from them, using the compilation mechanism presented shortly, we are able to generate the Gilsonite specification expected by Gillian-Rust. For example, for push_front, we will end up with the following specifications:

```
// Pearlite specification
#[pearlite::requires(self@.len() < usize::MAX@)]
#[pearlite::ensures(Seq::singleton(e).concat((*self)@) == (^self)@)]
// Gilsonite specification
#[gilsonite::specification(forall s_repr, e_repr.
  requires { self.own(s_repr) * e.own(e_repr) $ s_repr.0.len() < Int::from(usize::MAX) $ }
  exists r_repr. ensures { ret.own(r_repr) * $Seq::singleton(e_repr).concat(s_repr.0) == s_repr.1$ }
)]
pub fn push_front(&mut self, e: T) { ... }
```

Verification of the complete Merge Sort and accompanying Linked List implementation is performed by successively running cargo creusot and cargo gillian to generate the proof obligations for Creusot and Gillian-Rust, respectively, which are then discharged by running the appropriate backends: Why3 for Creusot and the Gillian-Rust backend for Gillian-Rust.

Compilation of Creusot Specifications. To compile Creusot specifications to Gilsonite, we first need to interpret Creusot's types in Gillian-Rust. Recall that we interpret Rust types using their *representations*, and that LinkedList<T> is interpreted via the Ownable trait in Gillian-Rust as gillian_rust::Seq<T::ReprTy>. In addition, we must interpret the *logical* types of Creusot, which is also done by defining appropriate instances of Ownable: in particular, the creusot::Seq<T> type of Creusot, just like LinkedList<T> or Rust, is interpreted as gillian_rust::Seq<T::ReprTy>. Like Creusot and RustHornBelt, we interpret mutable borrows as a pair of the representation of the value and a prophecised value, so that &mut LinkedList<T> is interpreted as (Seq<T::ReprTy>, Seq<T::ReprTy>).

---

[6]Pearlite is a first-order logic, including the standard connectives as well as support for functions and predicate definitions.

```
1  #[gilsonite::specification( ... )]
2  pub fn push_front(&mut self, elt: T) {
3      self.push_front_node(Box::new(Node::new(elt)));
4      mutref_auto_resolve!(self); // <- Single additional annotation required
5  }
6
7  fn push_front_node(&mut self, mut node: Box<Node<T>>) { unsafe {
8      node.next = self.head; node.prev = None;
9      let node = Some(Box::leak(node).into());
10     match self.head { None => self.tail = node, Some(head) => (*head.as_ptr()).prev = node, }
11     self.head = node;
12     self.len += 1;
13 } }
```

Fig. 8. Implementation of push_front

Specification interpretation is done by *elaboration*, the general schema of which is given on the right. We require ownership of every function argument, associating each with a representation value, and in the end, we own the result, again associated with a representation value. We then place the preconditions and postconditions into prophecy observations, sub-

$$\{P\}\; \mathit{fn}\; f\langle\kappa\rangle(x_1 : \mathsf{T}_1, \ldots, x_n : \mathsf{T}_n) \to \mathsf{T}_{\mathsf{ret}}\; \{Q\}$$
$$\Longrightarrow$$
$$\{(\circledast_{i=1}^n \llbracket \mathsf{T}_i \rrbracket (x_i, m_i)) * \langle P[x_i/m_i]\rangle * [\kappa]_q\}$$
$$\mathit{fn}\; f\langle\kappa\rangle(x_1 : \mathsf{T}_1, \ldots, x_n : \mathsf{T}_n) \to \mathsf{T}_{\mathsf{ret}}$$
$$\left\{ \begin{array}{c} \exists m_{\mathsf{ret}}.\; \llbracket \mathsf{T}_{\mathsf{ret}} \rrbracket (\mathsf{ret}, m_{\mathsf{ret}}) * \\ \langle Q[x_i/m_i][\mathsf{ret}/m_{\mathsf{ret}}]\rangle * [\kappa]_q \end{array} \right\}$$

stituting occurrences of Rust variables with their corresponding representation values. Following this process, we obtain the Gilsonite specification for pop_front given earlier.

Gillian-Rust in Action: `LinkedList::push_front`. To complete our tour of hybrid verification, we explain how Gillian-Rust leverages its features presented in the previous sections to prove the Pearlite specification of push_front method of LinkedList. In Figure 8, we give the full implementation of push_front, together with the auxiliary push_front_node method. We provide a specification only for the former, as Gillian-Rust can simply symbolically execute the latter.

When execution starts, the state contains: **a)** the ownership predicate for a mutable reference to a LinkedList at reference self, with representation self_repr; **b)** the ownership predicate for the element elt of type T; **c)** an observation that the length of the representation of the linked list is less than usize::MAX; and **d)** a lifetime token corresponding to the lifetime of the mutable reference self.

First, in line 3, the function allocates a new owned pointer, Box, which contains a new node constructed from the element elt, with previous and next pointers set to None. This pointer is immediately passed to the auxiliary function push_front_node.

In line 8, the access to self.head requires ownership of the corresponding location in memory, which is currently hidden in the full borrow contained in the resource **a)**. Thanks to the encoding of full borrows presented in §4.2, Gillian-Rust automatically *opens the borrow* by applying the UNFOLD-GUARDED rule, losing ownership of the lifetime token (resource **d)**), but obtaining ownership of the value contained at address self as well as the entire linked list, together with the prophecy controller corresponding to its representation.

The following three lines perform in-place heap updates, all handled automatically by Gillian-Rust, as per §3. Note that the matching of the value of self.head in line 10 and its dereferencing to access its prev field requires unfolding the dllSeg predicate once, also done automatically.

Next, in line 12, the len field of the list is updated, potentially resulting in an overflow. The current path condition is not sufficient to prove its absence and execution branches into a correct path where the overflow does not happen and an incorrect path that implicitly calls a panic. Before panicking, Gillian-Rust always checks that the current path condition (here, the overflow condition) does not contradict the observation, using the PROPH-SAT rule (which entails that $\langle$False$\rangle \Rightarrow$ False). Here, the observation, our resource **c)**, contradicts the overflow, and the incorrect path is discarded.

Next, `push_front_node` returns, and the `mutref_auto_resolve!` annotation on line 4 tells Gillian-Rust to apply the Mut-Update and MutRef-Resolve rules in sequence. The former requires the invariant of the linked list to have been restored, with a new representation. At this point, Gillian-Rust automatically folds the `dllSeg` predicate twice, once to revert the unfolding previously performed, and once to push the newly-added node and its ownership predicate (resource **b**)) to its front. Then, Gillian-Rust folds the ownership predicate of the linked-list, checking that the first and final pointer are `None`, and that its length field corresponds to the length of its new representation, which is `self_repr` with `elt_repr` prepended to it. Mut-Update is then successfully applied, updating the prophecy controller and observer to match the new representation.

When applying MutRef-Resolve, Gillian-Rust understands that the borrow needs to be closed. Since the invariant of the linked list has been correctly restored, the full borrow is automatically closed, and the lifetime token is recovered. MutRef-Resolve then discards the resource corresponding to the mutable reference (including the full borrow), and produces the observation required to prove the postcondition of the function.

Finally, the obtained state is matched against the postcondition, which requires: **1)** ownership of the return value, which is vacuously owned as the return type is unit; **2)** the lifetime token that was recovered when closing the borrow; and **3)** the observation obtained by applying Mut-Update. As the postcondition is satisfied, the specification is verified, and can be soundly used in Creusot.

7 Evaluation

We used our hybrid verification pipeline to perform a number of case studies, all making use of internally unsafe modules (IUMs); the results are shown in the table to the right. For each analysed IUM, we give: the number of executable lines of code (eLoc) and lines of annotations (specifications/predicate definitions/lemmas/proof tactics, aLoc); the type of properties verified (VP), with functional correctness (FC) subsuming type safety (TS); and the verification time. We note that verifying only TS allows for the use of a simpler encoding, which eschews prophecies to track value information. To our

	VP	eLoC	aLoC	Time
EvenInt	TS/FC	47	13	0.04s
LP	TS	32	40	0.03s
LP	FC	43	56	0.04s
LinkedList	TS	130	176	0.24s
LinkedList	FC	130	227	0.45s
MiniVec	FC	140	59	1.35s
Vec	TS	294	44	1.08s
Vec	FC	294	107	2.57s

knowledge, this is the the first verification of TS and FC of unsafe code from the Rust standard library—a subset of the `LinkedList` and `Vec` modules (with caveats for the latter)—with no or minor modifications to the original source code. All experiments were performed single-threaded, on a MacBook Pro 2019, with 16GB Memory and a 2.3GHz 9-Core Intel Core i9 processor.

EvenInt. We start from a case study provided as part of the RefinedRust [10] evaluation. `EvenInt` is a structure that only contains a single value of type `i32`, and its ownership invariant requires the value to be even. We copy all applicable functions from this case study, eliding those that make use of shared references (cf. §8), and verify them in Gillian-Rust by giving Creusot specifications that correspond to the RefinedRust specifications provided. These functions include 2 unsafe functions (one constructor and one mutator) and 3 safe functions (two constructors and one mutator). We verify Creusot specifications for the three safe functions, and purposefully do not write specifications for the unsafe functions as they are not required by Gillian-Rust, reducing the annotation overhead. Full details about the verified functions can be found in [4].

The total verification time of Gillian-Rust for the `EvenInt` study is **0.04s**, several orders of magnitude faster than the **4m36s** of RefinedRust. To hint at the level of automation, verifying the safe mutator requires a single line of annotation with Gillian-Rust to resolve the prophecy (cf. line 4 of

Figure 8). In contrast, RefinedRust requires of the user to manually write a Rocq proof that if *i* is an even integer, then $i + 1 + 1$ is still a even integer.

LinkedPair. Next, we verified TS and FC of a "linked-pair" data-structure (LP) that we developed as a tutorial example for Gillian-Rust, the details of which we omit given space constraints.

LinkedList. Next, we verified TS and FC of a subset of the `LinkedList` API from the Rust standard library, extracted from commit ad2b34d0 (04/12/23) of the official Rust repository. The only modifications made were to add annotations required for verification as well as to manually inline calls to `Option::map`, whose parameter is a closure, which are not yet supported by the Gillian-Rust compiler. Once these are added, there will be no need for additional annotations, as Gillian-Rust will be able to symbolically execute them like any other function, without requiring a specification.

Using the ownership predicate given in §2.2 and the `dllSeg` predicate given in [4], we prove FC of six functions: `new`, `push_front`, `pop_front`, `push_back`, `pop_back`, and `front_mut`. The total verification time is **0.72s**, including verification of auxiliary proofs generated by the `extract_lemma` macro, as well as two additional lemmas required for proving `push_back` and `pop_back`. These lemmas, in particular: change the traversal direction of `dllSeg` (from head-to-tail to tail-to-head, and vice-versa); are not Rust-specific, but rather essential primitives for any doubly-linked-list formalisation in SL; and are written in Rust and proven within Gillian-Rust without requiring the use of external tools.

MiniVec. Next, we verified a subset of the API for `MiniVec`, a simple implementation of the `Vec` module used by RefinedRust as a case study. Using specifications that provide similar guarantees to those proven by RefinedRust, we verify FC of the `new`, `with_capacity`, `push`, `pop`, `get_mut`, and `get_unchecked_mut` functions, as well as a simple associated client function. Our hybrid pipeline performs verification in **1.35s**, 1.28s for Gillian-Rust and 0.07s for Creusot, in contrast with the **30m40s** of RefinedRust.

Vec. Next, we verify the `Vec` implementation from the Rust standard library (same commit as for `LinkedList`), targeting the same functions as for `MiniVec`. In addition, we also verify `index_mut`, which performs a similar operation to `get_unchecked_mut`, but adds a safety check and performs access in memory through slice indexing instead of raw pointer arithmetics. Verifying both of these functions ensures that we correctly support these two different ways of accessing memory in Rust.

The source code of `Vec` module is substantially more complex than that of `MiniVec`, explaining why the verification of this module takes longer than the verification of `MiniVec`, yielding (in our opinion, a still reasonable) **1.08s** for TS and **2.57s** for FC.

It is important to note that `Vec` performs *untyped* allocations by explicitly providing the size of the allocation in bytes, yielding a raw pointer to an uninitialised array of bytes. The pointer is then cast to a pointer to the vector element type and is used to store the typed values. In this process, the corresponding Gillian-Rust heap object has some nodes indexed using the `u8` type and other nodes indexed using the vector element type (cf. §3.2), showcasing its resilience to low-level operations.

Vec-related Caveats. Our verification of `Vec` and `MiniVec` comes with three caveats. First, we disallow zero-sized types (ZSTs) as types of vector elements. Both `Vec` and `MiniVec` are special-cased for ZSTs, in which case there is no allocation and the vector is simply a counter for the length. However, Gillian-Rust is not able to express the ownership invariant for ZSTs, as Gillian is untyped and cannot exhibit *the only representative of the ZST type*. We will overcome this limitation by allowing the Gillian-Rust state model to produce these representatives. This does mean, however, that RefinedRust considers several more execution paths for MiniVec than Gillian-Rust.

Next, the borrow extraction lemma for FC of `get_index_mut` and `index_mut` requires the proof of a magic wand that Gillian-Rust cannot yet automate, and is left unproven for now. We will add support for manually specifying extract-lemmas proofs when the tool is unable to automate them.

Finally, we slightly modified the source code of the standard library `Vec` module, in the following ways. First, the vector type is parametric on an allocator, which we remove and perform all allocation

by calling the Gillian-Rust allocator. We also inline calls to functions such as `Result::map`, which receive a closure as parameter, due to the above-mentioned lack of support for closures. Finally, the real implementation of `index_mut` when using `usize` as an index is hidden behind a few layers of trait indirection; we manually inline layers so that `index_mut` is a single function.

Hybrid Verification. We argue that a *hybrid* approach combining Gillian-Rust with Creusot enables higher performance and flexibility in verification. To validate this, it is essential to answer two questions: (1) "Can Gillian-Rust effectively verify Creusot-style specifications?"; and (2) "Can those specifications then be efficiently used from Creusot's perspective?".

In §6, we presented our `hybrid` macros, which act as a bridge between the two tools, interpreting the specifications appropriately as either Pearlite or Gilsonite. We used these macros to specify and verify the the examples presented above, conclusively answering (1). The code generated by the hybrid macros is often identical to the raw Gilsonite specification we would write by hand. We have noticed no impact on verification times caused by use of the hybrid macros.

Our answer to (2) comes in two parts. Firstly, we note that Creusot provides the `creusot_contracts` crate, which provides standard, trusted specifications for common Rust types through its `extern_spec!` macro. These specifications are either identical or semantically equivalent to the ones proved by Gillian-Rust and at most a safe wrapper would be required by Creusot to prove the entailment.

Secondly, we implemented and verified several safe programs using specifications obtained by Gillian-Rust and observed favourable verification times: **Merge Sort** (55 lines of specification, 56 lines of generic lemmas about permutations missing from Creusot's standard library, and 68 executable lines of code), taking 6.3s (wall) 28.7s (user) to verify; **Gnome Sort** (6 lines of specification and 17 executable lines of code), taking 2.6s (wall), 4.6s (user) to verify; and **Right Pad** (11 lines of specification and 12 executable lines of code), taking 0.6s (wall) 0.4s (user) to verify.

8 Limitations and Future Work

Gillian-Rust is still a proof-of-concept, demonstrating, together with Creusot, the *viability* of hybrid Rust verification. We outline our current limitations below, noting that most of the improvements that we believe are required for removing the 'proof-of-concept' label are purely engineering challenges, but all together requiring substantial time and manpower.

Unimplemented Features. The Gillian-Rust compiler does not support all constructs of the MIR AST, as it was implemented by need. One such construct, for example, are closures, for which the extension should be straightforward, as Gillian-Rust already supports dynamic function calls.

Additionally, Gillian-Rust can reason about at most one lifetime *in specifications* (multiple lifetimes are already supported in function bodies). This does not allow us to verify, for instance, iterators such as `IterMut<'_>`. The solution lies in extracting annotated lifetimes from the Rust compiler, and may involve creating a custom borrow-checker pass or modifying the Rust compiler itself.

Meta-Theory Simplifications. The meta-theory of Gillian-Rust builds on that of RustBelt and RustHornBelt, with two simplifications that are beyond the scope of this project and that do not limit what Gillian-Rust can do, but rather leave small gaps in the justification of its soundness.

First, we remove 'later' modalities due to Gillian's non-step-indexed separation logic, believing that if Gillian were to be formalised in Iris, our ghost commands would 'take a step', justifying our approach. Second, RustHornBelt type definitions ensure the absence of causal loops in prophecy variables, by requiring an additional proof obligation that is not required by Gillian-Rust. However, Gillian enforces an ins-to-outs data flow of predicate parameters [24] which, we believe, naturally enforces this constraint. However, formalising these properties within RustHornBelt would require a deep embedding of the assertion language, which would enable the formalisation of the dataflow analysis performed by Gillian.

Unexplored Topics. We have not yet addressed shared references and their ownership predicates, which would require defining a `Shareable` trait and adding support for fractured borrows and non-atomic borrows to Gillian-Rust. To support fractured borrows, the Gillian-Rust heap would need to be extended with fractional permissions, which was done in the past for other state models without apparent roadblocks. We would also need to implement the behaviour of opening shared borrows, which is doable by following the blueprint we give for full borrows. Non-atomic should be simpler to implement, as they are more similar to full borrows, with an additional token in the guard.

In addition, while our specifications apply in concurrent contexts, we do not address concurrency-specific constructs or thread-safe types (e.g., `Send`/`Sync` proof obligations).

Finally, we do not model StackedBorrows[16] or TreeBorrows[34], noting that no current theoretical framework integrates these models with the semantic typing of RustBelt.

9 Related Work

As our focus is on reasoning about unsafe Rust, we provide an overview of, to our knowledge, the only other four tools capable of performing such reasoning, none of which explores the idea of hybrid verification. Given this focus, we do not address in detail the many tools other than Creusot for verifying safe Rust (e.g., Prusti [2], Aeneas [12], or Flux [21]).

RefinedRust. In line with RefinedC [30], RefinedRust [10] allows users to annotate functions with refinement types and interactively verify functional correctness (FC) of Rust programs with unsafe code. It compiles real-world Rust into an intermediate representation shallowly embedded in Rocq. Its trusted computing base is smaller than that of Gillian-Rust and it reuses RustBelt's lifetime logic to perform *foundational* proofs, extending it with new techniques for automating and simplifying reasoning about unsafe code, in some instances automating reasoning that requires annotations in Gillian-Rust. Some of these automations may be worth incorporating into our work.

However, while RefinedRust can verify FC of unsafe code, it does not explore hybrid verification, the key feature of our approach, meaning that its verification of safe code will be substantially slower and less automatic. Moreover, our preliminary evaluation suggests that Gillian-Rust is several orders of magnitude faster than RefinedRust, even for unsafe verification.

VeriFast for Rust. Rahimi Foroushaani and Jacobs [29] describe a Rust front-end for VeriFast [13] which provides a way of verifying semantic type safety for unsafe Rust. This frontend covers an extensive set of unsafe Rust features, and it has been used to verify type safety for a large corpus of real unsafe Rust code (in comparison to other unsafe Rust verification tools, including Gillian-Rust). For instance, it has been used to prove numerous proof obligations for the linked list module of the standard library. However, this frontend focuses solely on verifying type safety of unsafe code and does not explore hybrid verification or new automations (e.g., for opening and closing borrows).

Verus. In contrast to Gillian-Rust, VeriFast, and RefinedRust, Verus [20] does not use separation logic (SL) but rather linear ghost types to encode ownership properties. This approach allows it to leverage the borrow checker of the Rust compiler to drastically improve the encoding into SMT. It also means that writing proofs *feels like* writing Rust code, providing a familiar user experience. In short, Verus is great for verifying code that is written as target for verification (sometimes called proof-oriented programming).

However, Verus does not support traditional raw pointers, meaning that it is not able to verify 'traditional' unsafe code. For example, using Verus, one cannot verify the standard library implementation of `LinkedList` in the way that we propose. Instead, Verus developers have verified their own implementation of the `LinkedList` library, keeping track of linear ghost objects (denoted by `PointsTo<T>`, a Verus primitive) to implement links between nodes. In that sense, Verus could be considered a verifier shallowly embedded in an extension of Rust, rather than a Rust verifier.

In addition, Verus does not yet support reasoning about functions that return mutable references, which we support in Gillian-Rust thanks to our RustHornBelt-inspired SL foundations.

Kani. Kani [32] is an industrial-strength bounded model checker for Rust, which compiles an impressively large fragment of Rust to the intermediate representation ingested by CBMC [5] for its analysis. However, it does not propose solutions to the challenges solved by our work: Kani cannot verify type safety; it picks a specific layout for each structure; and it treats all safe and unsafe code in the same way, not leveraging the safe Rust guarantees to enhance analysis.

10 Conclusions

We have introduced a hybrid approach to end-to-end verification of real-world Rust programs, in which, through a separation of concerns, the safe and unsafe parts of the code are handled by two different tools, each specialised for their task at hand. We have demonstrated the feasibility of this approach by connecting Creusot, a state-of-the-art automatic verification tool for safe Rust, with Gillian-Rust, a novel proof-of-concept semi-automatic verification tool for unsafe Rust. As part of the design and implementation of Gillian-Rust, we have shown how the complex concepts underpinning reasoning about unsafe Rust, such as lifetime logic and prophetic reasoning, can be brought from the interactive world of RustBelt and RustHornBelt to the world of compositional symbolic execution. We have conducted case studies that have demonstrated that Gillian-Rust is able to verify functional correctness of real-world unsafe Rust, including (to our knowledge, for the first time) code extracted from the Rust standard library, with high automation and in times several orders of magnitude faster than existing work. We have also shown that specifications verified by Gillian-Rust can be re-used by Creusot, demonstrating the feasibility of our hybrid approach.

Data-Availability Statement

The supplementary material, including our implementation of Gillian-Rust and the various case studies described in §7, is available online [3].

Acknowledgments

References

[1] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers. 2020. How do Programmers use Unsafe Rust? *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 136:1–136:27.

[2] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 147:1–147:30.

[3] Sacha-Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. 2025. Artifact: A Hybrid Approach to Semi-Automated Rust Verification. Zenodo. https://doi.org/10.5281/zenodo.15183201

[4] S.-É. Ayoun, X. Denis, P. Maksimović, and P. Gardner. 2025. A Hybrid Approach to Semi-Automated Rust Verification (Extended Version). https://arxiv.org/abs/2403.15122

[5] E. Clarke, D. Kroening, and F. Lerda. 2004. A Tool for Checking ANSI-C Programs. In *TACAS'04*. Springer, 168–176.

[6] T. Dardinier, G. Parthasarathy, N. Weeks, P. Müller, and A. J. Summers. 2022. Sound Automation of Magic Wands. In *CAV'22*. Springer International Publishing, 130–151.

[7] X. Denis and J.-H. Jourdan. 2023. Specifying and Verifying Higher-order Rust Iterators. In *TACAS'23 (Lecture Notes in Computer Science)*. 93–110.

[8] X. Denis, J.-H. Jourdan, and C. Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *FMSE'22*. 90–105.

[9] J. Fragoso Santos, P. Maksimović, S.-É. Ayoun, and P. Gardner. 2020. Gillian, Part I: A Multi-Language Platform for Symbolic Execution. In *PLDI'20*. 927–942.

[10] L. Gäher, M. Sammler, R. Jung, R. Krebbers, and D. Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proceedings of the ACM on Programming Languages* 8, PLDI, Article 192 (2024), 25 pages.

[11] Unsafe Code Guidelines Working Group. 2023. Structs and Tuples - Memory Layout - Unsafe Code Guidelines. https://github.com/rust-lang/unsafe-code-guidelines/blob/50f8ff4b6892f98740de3b375e4d4bda10b9da9f/reference/src/layout/structs-and-tuples.md Accessed: Nov. 16 2019.

[12] S. Ho and J. Protzenko. 2022. Aeneas: Rust Verification by Functional Translation. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 116:711–116:741.

[13] B. Jacobs, J. Smans, and F. Piessens. [n. d.]. A Quick Tour of the VeriFast Program Verifier. In *Programming Languages and Systems (Lecture Notes in Computer Science)*. Berlin, Heidelberg, 304–311.

[14] R. Jung. 2016. The Scope of Unsafe. https://www.ralfj.de/blog/2016/01/09/the-scope-of-unsafe.html Accessed: March 3rd 2025.

[15] R. Jung. 2018. Two Kinds of Invariants: Safety and Validity. https://www.ralfj.de/blog/2018/08/22/two-kinds-of-invariants.html Accessed: June 19th 2023.

[16] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 41:1–41:32.

[17] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 66:1–66:34.

[18] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. 2018. Iris from the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *Journal of Functional Programming* 28 (2018), e20.

[19] R. Jung, R. Lepigre, G. Parthasarathy, M. Rapoport, A. Timany, D. Dreyer, and B. Jacobs. 2019. The Future is Ours: Prophecy Variables in Separation Logic. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 45:1–45:32.

[20] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85:286–85:315.

[21] N. Lehmann, A. Geller, N. Vazou, and R. Jhala. 2022. Flux: Liquid Types for Rust. http://arxiv.org/abs/2207.04034

[22] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. 2012. *The CompCert Memory Model, Version 2*. Technical Report. Inria. https://hal.inria.fr/hal-00703441 Pages: 26.

[23] A. Lööw, D. Nantes-Sobrinho, S.-É. Ayoun, C. Cronjäger, P. Maksimović, and P. Gardner. 2024. Compositional Symbolic Execution for Correctness and Incorrectness Reasoning. In *ECOOP'24*. 25:1–25:28. https://doi.org/10.4230/LIPIcs.ECOOP.2024.25

[24] A. Lööw, D. Nantes-Sobrinho, S.-É. Ayoun, P. Maksimović, and P. Gardner. 2024. Matching Plans for Frame Inference in Compositional Reasoning. In *ECOOP'24*. 26:1–26:20. https://doi.org/10.4230/LIPIcs.ECOOP.2024.26

[25] P. Maksimović, S.-É. Ayoun, J. Fragoso Santos, and P. Gardner. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *CAV'21*. 827–850.

[26] N. D. Matsakis and F. S. Klock. 2014. The Rust Language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104.

[27] Y. Matsushita, X. Denis, J.-H. Jourdan, and D. Dreyer. 2022. RustHornBelt: A Semantic Foundation for Functional Verification of Rust Programs with Unsafe Code. In *PLDI'22*. 841–856.

[28] Y. Matsushita, T. Tsukada, and N. Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *ACM Transactions on Programming Languages and Systems* 43, 4 (2021), 15:1–15:54.

[29] N. Rahimi Foroushaani and B. Jacobs. 2022. Modular Formal Verification of Rust Programs with Unsafe Blocks. http://arxiv.org/abs/2212.12976

[30] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *PLDI'21*. 158–174.

[31] The Coq Team. 2023. The Coq Proof Assistant. https://coq.inria.fr/ Accessed: Nov. 16th 2023.

[32] The Kani Team. 2023. How Open Source Projects are Using Kani to Write Better Software in Rust | AWS Open Source Blog. https://aws.amazon.com/blogs/opensource/how-open-source-projects-are-using-kani-to-write-better-software-in-rust/ Accessed: Nov. 13th 2023.

[33] The Rust Team. 2023. Rust Programming Language. https://www.rust-lang.org/ Accessed: Nov. 16th 2023.

[34] N. Villani, J. Hostert, D. Dreyer, and R. Jung. 2025. Tree Borrows. *Proceedings of the ACM on Programming Languages* PLDI (2025). https://doi.org/10.1145/3735592