

A hybrid approach to semi-automated Rust verification

SACHA-ÉLIE AYOUN, Imperial College London, UK

XAVIER DENIS, Université Paris-Saclay, CNRS, ENS Paris-Saclay, INRIA, Laboratoire Méthodes Formelles, France

PETAR MAKSIMOVIĆ, Imperial College London, UK and Runtime Verification Inc., USA

PHILIPPA GARDNER, Imperial College London, UK

While recent years have been witness to a large body of work on efficient and automated verification of *safe* Rust code, enabled by the rich guarantees of the Rust type system, much less progress has been made on reasoning about *unsafe* code due to its unique complexities. We propose a hybrid approach to end-to-end Rust verification in which powerful automated verification of safe Rust is combined with targeted semi-automated verification of unsafe Rust. To this end, we present Gillian-Rust, a proof-of-concept semi-automated verification tool that is able to reason about type safety and functional correctness of unsafe code. Built on top of the Gillian parametric compositional verification platform, Gillian-Rust automates a rich separation logic for real-world Rust, embedding the lifetime logic of RustBelt and the parametric prophecies of RustHornBelt. Using the unique extensibility of Gillian, our novel encoding of these features is fine-tuned to maximise automation and exposes a user-friendly API, allowing for low-effort verification of unsafe code. 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 may use but not verify, demonstrating the feasibility of our hybrid approach.

1 INTRODUCTION

The Rust programming language [24, 29] has seen rapid adoption in recent years, particularly in the field of *systems programming*, to the point where it has become only the second language to be adopted by the Linux kernel [4]. The success of Rust is due to its rejection of false dichotomies between safety and performance: its *ownership type system* and *borrow checker* preserve memory safety while avoiding the need for a garbage collector.

As Rust finds its way into more critical systems, the need for stronger formal guarantees about the *behaviour* of Rust programs grows. In response to this need, the past several years have seen the emergence of a number of tools aimed at the verification of Rust programs, such as Aeneas [11], Creusot [7] 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.

Although most Rust code is written in the safe subset of the language, it is common for it to rely on *unsafe* code to interface with the underlying operating system or to provide low-level abstractions. In unsafe code, the programmer gains access to several ‘superpowers’, such as the ability to dereference raw pointers, cast between types, and even access (potentially) uninitialised memory. Unsafe code is an essential part of Rust’s design, allowing 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 programmer is now responsible for ensuring that unsafe 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, unsafe code is often the most complex and error-prone part of a Rust program, making it the most important 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 needed to be overcome by the implementation of Gillian-Rust requires a background in the foundational theory underpinning Rust. In 2018, Jung et al. published RustBelt [15], a theoretical framework that allows for semantic interpretation of Rust’s

ownership types using the higher-order separation logic Iris [16], thereby enabling reasoning about type safety. In 2022, the work on RustHornBelt [25] extended RustBelt with the ability to reason about functional correctness of unsafe Rust code, allowing for safe functions implemented with unsafe code to be given first-order logic specifications and providing the meta-theory that now underpins Creusot. However, both RustBelt and RustHornBelt operate on λ_{Rust} , a model of Rust that makes many simplifying assumptions relevant to a foundational formalisation and therefore cannot capture the intricacies of real Rust. Moreover, RustHornBelt proofs are manually performed in Coq [27], on code ported by hand from Rust to λ_{Rust} , and little automation is provided to alleviate the boilerplate of verification. As a result, while blazing the trail for end-to-end functional correctness verification of Rust programs, RustHornBelt cannot bring its approach to real-world Rust programs.

Challenge 1: 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 remains 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. In that setting, we believe that it would be extremely 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 the higher-level features used in safe Rust.

On the other hand, tools such as Creusot [7] have demonstrated that safe Rust verification can be performed with impressive automation and simplicity, permitting reasoning about some of the most high-level features of Rust [6], but sacrificing the ability to handle unsafe code. Ideally, one would like to reuse such a tool for safe code, and use another, more adapted tool, for analysing unsafe Rust, splitting the proof effort appropriately and thereby having the proverbial cake and eating it too. 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.

Challenge 2: Real Rust is really hard. Rust is primarily a systems programming language, and therefore comes with every associated complication, some previously known from the large research effort in C verification—e.g., low-level date representation, byte-level value manipulation, and memory allocator manipulation—and some new ones—e.g., exotically-sized types, such as zero-sized types, compiler-chosen layouts (whereas C has a standardised layout), and polymorphism.

While these aspects of Rust are invisible to safe Rust code, they become a proper concern when working with unsafe code. For example, it is crucial for a verifier to reason generically over the possible memory layouts of programs so that it could detect any potentially disallowed memory operations. This makes the reuse existing memory models from C verification tools difficult and requires development of new techniques to reason automatically and efficiently about real Rust and the way it represents objects in memory.

Challenge 3: Type safety, borrows, and raw pointers. The notion of type safety in Rust is much stricter than that found in languages like C. Specifically, the responsibility of a safe function, even an internally unsafe one, is not limited to its own body: it must ensure that no fully-safe program calling it may trigger undefined behaviours. This dramatically increases the complexity of integrating unsafe code into a Rust program.

The main tool employed by Rust to guarantee safety is a strict and static ownership discipline, wherein each value must always have a unique, exclusive owner. While this alone would be too restrictive, Rust also provides mutable references ($\&_{\text{mut}}^{\kappa} T$) and shared references ($\&^{\kappa} T$) which may *borrow* ownership for a statically-defined amount of time κ called a *lifetime*. 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. [Figure 1](#), bottom left), breaking the exclusive ownership discipline. In such cases, developers must resort to unsafe code in order to manipulate raw pointers (`*mut T`) which, unlike references, allow for unrestricted aliasing and do not provide any safety guarantees.

This mixed use of raw pointers and safe references causes the task of verifying type safety of unsafe code to (yet again) increase in complexity, as it requires reasoning about lifetime-dependent safety invariants. While this key challenge of unsafe Rust verification is formalised and addressed for λ_{Rust} by RustBelt, it is yet to be addressed for real Rust.

Contributions and paper outline. In this paper, we introduce Gillian-Rust, a proof-of-concept semi-automated verification tool for Rust, built on top of the Gillian compositional symbolic analysis platform and capable of reasoning efficiently about type safety and functional correctness of unsafe Rust code. 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, thereby enabling Gillian-Rust to reason about type safety of mutable references. In [§5](#), we present a way of embedding within Gillian-Rust the ability to reason about parametric prophecies as proposed by RustHornBelt. Further in [§5](#), we give a systematic encoding of Creusot specifications into the specification language of Gillian-Rust, demonstrating successful integration of safe and unsafe Rust verification. In [§6](#), we evaluate Gillian-Rust by verifying type safety and functional correctness for a subset of the functions of the `LinkedList` module of the Rust standard library that make use of mutable references, effectively demonstrating that our hybrid approach is viable and that the resulting verification process is fast. Finally, in [§7](#) and [§8](#), we discuss the current limitations of Gillian-Rust in detail, provide a pathway for short- and long-term explorations required to overcome these limitations, and place Gillian-Rust in the context of overall related work.

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.

2.1 A hybrid approach: Creusot + Gillian-Rust

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

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*¹ 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 `[T]` (cf. [Figure 1](#) (left)).

¹Ownership predicates for Rust types were introduced in RustBelt, but did not connect types to a pure representation

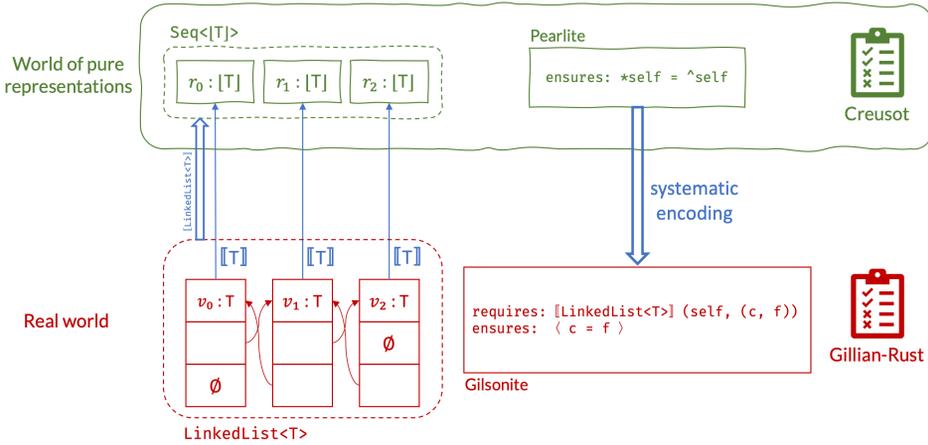


Fig. 1. A high-level illustration of the differences and connections between the world of pure representations, observed by Creusot, and the world of real representations, observed by RustHornBelt and Gillian-Rust.

```

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

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

```

Fig. 2. `LinkedList` structure definition and its ownership predicate. The `dllSeg` predicate is given in §3.3.

To verify all Rust code, including that which contains unsafe blocks, we propose a hybrid approach where Creusot verifies all of the proof obligations within its reach and delegates the verification of unsafe code to another tool. Such a tool, however, does not yet exist, and we therefore propose a proof-of-concept called Gillian-Rust, introduced in §2.3. Gillian-Rust has the ability to perform separation-logic 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 Figure 1 (right), and detailed in §5.4.

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* (cf. Figure 2, left), 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.

```

#[show_safety]
// Expands to:
// #[unsafe_spec(
//   requires { self.own(_) }
//   ensures { result.own(_) }
// )]
fn pop_front(&mut self) -> Option<T> {
    // Implementation
}

#[ensures(match result {
    None => (^self)@ == Seq::EMPTY,
    Some(x) => self@ == Seq::cons(x, (^self)@
)}}]
fn pop_front(&mut self) -> Option<T> {
    // Implementation
    mutref_auto_resolve!(self)
}

```

Fig. 3. The type safety (left) and Pearlite (right) specifications for `pop_front`

We show the process of using Gillian-Rust to prove a Pearlite specification for the `pop_front` function, extracted from the Rust standard library, which modifies a `LinkedList` in place by removing its first element if it is not empty.

Implementing Ownable. The first step is to connect the real Rust structure to its pure representation used by Creusot. To do so, users must implement the `Ownable trait`,² and define the type of its representation `ReprTy` (denoted by $[\cdot]$ in mathematics), and the ownership predicate, which takes two parameters, the structure itself (`self`) and the representation.

In the case of `LinkedList<T>`, its representation type is a sequence, of which each element has type $T :: \text{ReprTy}$. Note that, in order for this type to be properly defined, \top itself must implement `Ownable`, a constraint specified using a trait bound (the `: Ownable` part in `<T : Ownable>`).

Type safety. Once the ownership predicate is defined, we can already verify type safety of a function by simply adding the `#[show_safety]` attribute on top. This attribute, also exposed as part of the Gilsonite API, expands into a Gilsonite specification which requires all input parameters to be owned when entering the function, and ensures that the resulting value be owned when the function returns. This type safety specification (cf. Figure 3, left) corresponds to that proposed in RustBelt, which also requires a lifetime token in the pre- and post-condition. This token is added automatically by the Gillian-Rust compiler (cf. Figure 6), and 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 (cf. Figure 3, right), by describing the update performed on the sequence which represents the `LinkedList`: *when the mutable reference expires*, if the return value is `None`, the representation will be the empty sequence, otherwise, it will be the tail of the input sequence and the return value will be the first element of the input sequence.

Pearlite, inspired by RustHorn [26], uses prophecy variables and the final value operator \wedge 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 §5.4, together with further explanations. Finally, after adding a single line which triggers a semi-automatic tactic during verification, Gillian-Rust is able to perform the proof for this specification.

2.3 Building Gillian-Rust on top of Gillian

Gillian-Rust is built on top of Gillian [9, 23], a compositional symbolic execution platform that is parametric on a target language being analysed. To instantiate Gillian, one needs to implement a symbolic state model in OCaml, which exposes:

²A trait is, akin to a Haskell typeclass, a form of interface describing a list of items that can be implemented for a type.

- a representation of the symbolic state, in the form of an OCaml type;
- *actions*, which are primitive operations for manipulating the state; and
- *core predicates*, which are the building blocks of a separation-logic assertion language that allows for specifying states.

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 a list of arguments \vec{v} yields a state σ' , value v_o , and path condition π' . As symbolic execution may branch, i.e. executing an action may produce several outcomes.

For each core predicate ρ , the tool developers using Gillian have to implement two special actions called the *consumer* and the *producer* of ρ , denoted by cons_ρ and prod_ρ . Intuitively, consumers and producers, respectively, remove from and add to the symbolic state the resource corresponding to a given core predicate. On top, Gillian then extends consumption and production to entire assertions. This ability to provide custom consumers and producers is what makes Gillian uniquely extensible in the design space of semi-automated compositional verification tools, as it allows one to automate the basic rules of their custom separation logic.

In Gillian-Rust, a symbolic state $\sigma = (h, \xi, \gamma, \phi, \chi)$ is a quintuple 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).

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.

Until now, Rust verification tools have been consistently working around these issues. For instance, Prusti encodes structures using the object-oriented memory model of Viper, allowing efficient field access but preventing reasoning about pointer arithmetic. Kani, on the other hand, compiles Rust code to a C-like representation by choosing one specific layout for each structure, thereby sacrificing the guarantee that a verified program would be correct had the compiler made different layout choices authorised by the language [10].

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 [21] 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

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

represents either: an offset of e times the size of the type T , where e is a symbolic integer, denoted by $+^T e$; or the relative³ offset of the i -th field of a structure τ , denoted by $\cdot^T i$; or the relative offset of the i -th field of the j -th variant of an enum, denoted by $\cdot^T j i$.

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. This means, in particular, that the position of an element within a projection does not affect its interpretation: for example, the projection $[\cdot^T i, \cdot^T j]$ is equal to the projection $[\cdot^T j, \cdot^T i]$.

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 always 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;
- a tree representing an enum with a concrete discriminant⁴, consisting of: an internal node containing that discriminant; and children nodes representing the fields of the corresponding enum variant.

The types annotating the nodes must always be sized (i.e must have a size known at compile-time, chosen by the compiler⁵), thereby providing an interpretation for each node. The primitive

```
struct S { x: u32, y: u64 };
```

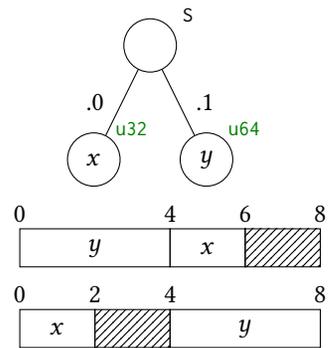


Fig. 4. A structure S , its representation as a structural node, and two of its potential interpretations

³The offsets are relative with respect to the beginning of the structure.

⁴A symbolic enum (i.e., an enum with a symbolic discriminant) would be represented as a single node with a symbolic value.

⁵In contrast to unsized types, such as the slice type $[T]$, for which the size is only known at run-time.

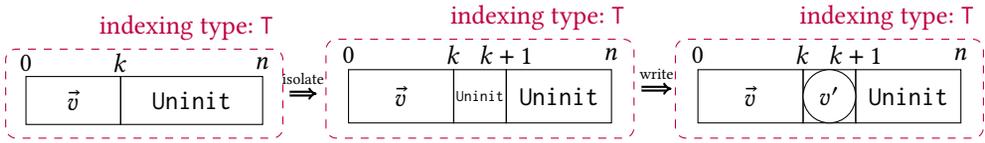


Fig. 5. Update of a laid-out node corresponding to $n * \text{size_of}::\langle T \rangle()$ bytes.

operations *load* and *store* are provided in the interface of the symbolic heap and are in charge of ensuring that the validity invariants [13] of values written in memory are maintained (e.g., Boolean values are represented only by the bit-patterns `0b0` and `0b1`). They are also responsible for enforcing other important aspects of the Rust semantics: for example, loading a value from memory in the context of a *move* will deinitialise that memory.

In Figure 4, 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 $(x, \text{u32})$ and $(y, \text{u64})$. 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 field ordering chosen by the compiler: 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 `.S0` or `.S1`.

Laid-out nodes. While structural nodes facilitate efficient resolution for a large majority of memory accesses, they are not a novel concept: they are similar to, for example, the representation used by Viper for its objects. The novelty of our approach lies in the combination of structural nodes with laid-out nodes, inspired by Gillian-C [23], 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 $n * \text{size_of}::\langle T \rangle()$ with respect to 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, Figure 5 (left) shows a laid-out node with indexing type τ 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, Figure 5 (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 (Figure 5, 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 (Figure 5, 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 τ .

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, $a \mapsto_{\tau} v$, 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 a theme, 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 a `LinkedList`:

$$\begin{aligned} \llbracket \text{LinkedList} \langle T \rangle \rrbracket(l, r) &\triangleq l = \text{dllSeg} \langle T \rangle(l.\text{head}, \text{None}, l.\text{tail}, \text{None}, r) * l.\text{len} = |r| \\ \text{dllSeg} \langle T \rangle(h, n, t, p, r) &\triangleq (h = n * t = p * r = []) \vee \\ &\quad (\exists h', v, z, r'. h = \text{Some}(h') * h' \mapsto_{\text{Node} \langle T \rangle} \{v, z, p\} * \llbracket T \rrbracket(v, r_v) * \\ &\quad \text{dllSeg} \langle T \rangle(z, n, t, h, r') * r = r_v :: r') \end{aligned}$$

The doubly-linked-list-segment predicate, a staple of separation logic literature, can be reused in the Rust context with one adaptation, which requires that the value of each node value must be owned by the list (captured by the $\llbracket T \rrbracket(v, r_v)$ ownership predicate), effectively making the predicate parametric on the type of the values that the list holds.

4 AUTOMATING REASONING ABOUT MUTABLE BORROWS

The handling of 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 [15], thankfully, provides a theoretical framework for reasoning about mutable borrows within Iris and proves its correctness in Coq, this comes at the cost of the reasoning itself being 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 $0 < q \leq 1$, 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., **LFTL-NOT-OWN-END** states that a lifetime cannot be alive and expired at the same time; **LFTL-END-PERSIST** states that an expired lifetime token is persistent (i.e. it can be duplicated); while **LFTL-TOK-FRACT** states that alive lifetime tokens may be split into fractions (for $0 < q, q'$).

$$\begin{array}{lll} \text{LFTL-NOT-OWN-END} & \text{LFTL-END-PERSIST} & \text{LFTL-TOK-FRACT} \\ [\kappa]_q * [\dagger\kappa] \Rightarrow \text{False} & \text{persistent}([\dagger\kappa]) & [\kappa]_{q+q'} \Leftrightarrow [\kappa]_q * [\kappa]_{q'} \end{array}$$

Lifetimes form a partial ordering, in that $\kappa \sqsubseteq \kappa'$ if κ' is alive for at least as long as κ is. For this reason, we encode lifetimes in Gillian as opaque sets of integers, which gives us inclusion reasoning at the SMT level for free. A lifetime context ξ is then simply a partial finite map from lifetimes to either a symbolic value ranging over real numbers in the interval $(0, 1]$, corresponding to the currently owned fraction of the lifetime token, or \dagger , capturing that the lifetime has expired.

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 Figure 6.⁶ While simple, these rules are illustrative of the relationship

⁶In 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.

$$\begin{array}{c}
\text{LFT-PRODUCE-ALIVE-ADD} \\
\frac{\xi(\kappa') = q' \quad \pi \vdash (\kappa = \kappa' \wedge 0 < q \wedge q + q' \leq 1) \quad \xi' = \xi [\kappa \leftarrow q + q']}{(\xi, \pi).prod_{[\cdot]}(\kappa, q) \rightsquigarrow (\xi', \pi)} \\
\\
\text{LFT-PRODUCE-OWN-END} \\
\frac{\xi(\kappa') = \dagger \quad \pi \vdash (\kappa = \kappa')}{(\xi, \pi).prod_{[\cdot]}(\kappa, q) \text{ vanishes}} \\
\\
\text{LFT-CONSUME-EXP} \\
\frac{\xi(\kappa') = \dagger \quad \pi \vdash (\kappa = \kappa')}{(\xi, \pi).cons_{[\dagger]}(\kappa) \rightsquigarrow (\xi, \pi)} \\
\\
\text{LFT-PRODUCE-EXPR-DUP} \\
\frac{\xi(\kappa') = \dagger \quad \pi \vdash (\kappa = \kappa')}{(\xi, \pi).prod_{[\dagger]}(\kappa) \rightsquigarrow (\xi, \pi)}
\end{array}$$

Fig. 6. Consumer and producer rules for lifetime tokens (simplified, excerpt)

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 **LFTL-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 **LFTL-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-EXPR-DUP** automate **LFTL-END-PERSIST**.

4.2 Modelling full borrows: guarded predicates

In Rust, a mutable reference of a value of type T during lifetime κ , denoted by $\&_{\text{mut}}^{\kappa} T$, 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 rule for manipulating mutable borrows are as follows:

$$\begin{array}{c}
\llbracket \&_{\text{mut}}^{\kappa} T \rrbracket(p) \triangleq \&^{\kappa} (\exists v. p \mapsto v * \llbracket T \rrbracket(a)) \\
\text{LFTL-BORROW-ACC} \\
\&^{\alpha} P * [\kappa] \rightleftharpoons \triangleright P * (\triangleright P \rightleftharpoons \&^{\alpha} P * [\kappa])
\end{array}$$

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, \triangleright , is omitted; in §7.2, 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 away higher-orderness. 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 $\llbracket \&_{\text{mut}}^{\kappa} \text{LinkedList} \langle T \rangle \rrbracket$. When using the Gilsonite API, a user may instantiate the full borrow assertion using the `#[borrow]` attribute, as follows:

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

obtaining an ownership predicate for mutable references of type τ^7 . 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 τ , 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 $(\delta, \vec{v}) \in (\text{Str} \times \text{List}(\text{Val}))$, 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 $\gamma : [\text{Str} \times \text{Lft} \times [\text{Val}]]$ 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 $(P \Rightarrow^* \&^K P * [\kappa]_q)$.

UNFOLD-GUARDED

$$\frac{\begin{array}{l} p.\text{predDefs}[\delta(\kappa, \vec{x})] = P \\ (\sigma, \pi).\text{cons}[\cdot, (\alpha, q)] \rightsquigarrow (\sigma', \pi') \\ \sigma' = (\mu', \gamma') \quad \delta(\alpha, \vec{v}) \in \gamma' \\ \gamma'' = \gamma' \setminus \delta(\alpha, \vec{v}) \quad \sigma'' = (\mu', \gamma'') \\ P' = P * C_\delta(\kappa, q, \vec{v}) \end{array}}{p \vdash (\sigma, \pi).\text{gunfold}(\delta(\alpha, \vec{v})) \rightsquigarrow (\sigma'', \pi'')} (\sigma'', \pi').\text{prod}(P'[\kappa/\alpha][\vec{x}/\vec{v}]) \rightsquigarrow (\sigma''', \pi'')$$

The **UNFOLD-GUARDED** rule describes the successful execution of `gunfold`. For conciseness, we decompose symbolic states into a pair (μ, γ) , where μ represents the remaining components of the state. 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 $p.\text{predDefs}$ maps predicates to their definitions.

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_node`,

⁷For now, instantiation of the mutable reference ownership predicate is performed as a primitive operation by the compiler because we do not support cross-crate compilation. However, the construct can be used by users within their crate.

```
#[extract_lemma]
#[unsafe_spec { <forall: head, tail, len, p>
  requires: (head != None) // F * list_ref_mut_frozen(list, head, tail, len) // &^κP
  ensures: Ownable::own(&mut (*p.as_ptr()).element) // &^κQ
}]
fn extract_head<T: Ownable>(list: &mut LinkedList<T>); // Implicitly parametric on κ
```

Fig. 8. Example instantiation of a borrow extraction lemma

`LinkedList::push_front_node` becomes completely automatic once the safety invariants of `LinkedList` has been properly specified as in §3.3.

4.3 Proving safety of borrow extraction

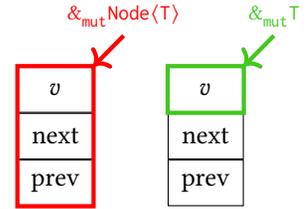
Unfortunately, opening and closing are not the only operations that one needs when working with full borrows. We identify several recurring patterns in unsafe Rust programs and provide ways of instantiating lemmas that allow us to analyse code that uses these patterns.

In particular, borrow extraction—the process of cutting a borrow up into a smaller borrow—is a common pattern in unsafe Rust programming. In particular, 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`), as 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, let us explain what would happen when extracting an incorrect mutable reference from the `LinkedList` structure.

Consider the case where the library implementer would create 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. 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`) is perfectly fine, the intuition being that one can *remove* the resource associated with the element and obtain a *remainder*. To that remainder we can then add any other element that satisfies the invariant of `T`, in order to recover a structure satisfying the `LinkedList` invariant. This principle is embodied by the **BORROW-EXTRACT** rule—which we have proven in Iris using RustBelt (it is a trivial corollary of the already existing rules)—where P is the invariant of the `LinkedList`, Q is the invariant of `T`, and $Q * P$ 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).

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 `&^κP` and `&^κQ` as well as the persistent assertion F , as illustratively

Fig. 7. Example invalid and valid mutable references extracted from a `LinkedList`.

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

done in Figure 8.⁸ Gillian itself is unable to prove that **BORROW-EXTRACT** holds, or to manipulate borrows using such a rule. Instead, the Gillian-Rust compiler produces two lemmas: one corresponding to the conclusion of the rule, which is marked as trusted and left unproven, and one corresponding to the hypotheses of the rule, which needs to be proven. As we have proven that the rule itself holds in Iris, the meta-theory of Gillian-Rust therefore ensures that if we prove the second lemma, then 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 [5], to Gillian’s parametric separation logic; the details of this extension are out of scope of this presentation.

5 FUNCTIONAL CORRECTNESS REASONING AND CONNECTION WITH CREUSOT

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 [26] introduced prophecy variables, later used in Creusot. However, only the latter has been given a foundational formalisation in RustHornBelt [25], an extension of RustBelt which describes how prophetic specifications interact with full borrows.

In this section, we briefly remind the reader of the workings of RustHornBelt, and show how its concepts are encoded in Gillian-Rust. To conclude our technical presentation, we show how Pearlite specifications can be compiled to Gilsonite, hence 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 \mathbb{T} , the type of its representation is denoted by $\lfloor \mathbb{T} \rfloor$. 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. $\lfloor \text{LinkedList}\langle \mathbb{T} \rangle \rfloor = \text{Seq}\langle \lfloor \mathbb{T} \rfloor \rangle$.

Mutable references, on the other hand, are represented as a pair of representations of the inner type (i.e., $\lfloor \&\text{mut } \mathbb{T} \rfloor = \lfloor \mathbb{T} \rfloor \times \lfloor \mathbb{T} \rfloor$), 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 $\uparrow x$.

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

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

MUT-AGREE

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

MUT-UPDATE

$$\text{VO}_x(a) * \text{PC}_x(a) \rightleftharpoons \text{VO}_x(a') * \text{PC}_x(a')$$

⁸In Figure 8, `list_ref_mut_frozen` denotes a borrow predicate obtained from the ownership predicate of `&mut LinkedList` (automatically derived by the compiler) by freezing the existential variables corresponding to the `head`, `tail` and `len` fields of the structure. Freezing existential variables is a common strategy for extracting borrows, fully supported by the Gilsonite API; we do not go into full detail due to space limitations.

$$\begin{array}{c}
\text{OBS-MERGE} \\
\langle \psi \rangle * \langle \psi' \rangle \vdash \langle \psi \wedge \psi' \rangle
\end{array}
\qquad
\begin{array}{c}
\text{PROPH-SAT} \\
\langle \psi \rangle \Rightarrow \exists \varepsilon. \varepsilon(\psi)
\end{array}
\qquad
\begin{array}{c}
\text{PROPH-TRUE} \\
\frac{\forall \varepsilon. \varepsilon(\psi)}{\langle \psi \rangle}
\end{array}$$

Fig. 9. Rules for observations from RustHornBelt (excerpt)

$$\begin{array}{c}
\text{OBSERVATION-PRODUCE} \\
\frac{\pi \wedge \phi \wedge \phi' \text{ SAT}}{(\phi, \pi). \text{prod}_{(\cdot)}(\phi') \rightsquigarrow (\phi \wedge \phi', \pi)}
\end{array}
\qquad
\begin{array}{c}
\text{OBSERVATION-CONSUME} \\
\frac{(\pi \wedge \phi \Rightarrow \phi') \text{ VALID}}{(\phi, \pi). \text{cons}_{(\cdot)}(\phi') \rightsquigarrow (\phi, \pi)}
\end{array}$$

Fig. 10. Consumer/producer rules for observations (excerpt)

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** 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.

Finally, RustHornBelt introduces *observations*, denoted by $\langle \psi \rangle$, where ψ is a pure assertion, which contains information known about prophecy values. Observations act as a second layer of truth preventing information about the future to leak 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 $PcyVar \rightarrow Value$).

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 $SVar \rightarrow Value$, 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. [17]. 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. Follows the core idea behind encoding prophecy variables: 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 may depend on both prophecy variables and symbolic variables. Figure 9 presents some of the rules that apply to observations in RustHornBelt, while Figure 10 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 appropriate, 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

$$\begin{array}{c}
\text{VOBS-PRODUCE-WITHOUT-CONTROLLER} \\
\frac{x \notin \text{dom}(\chi) \quad \chi' = o[x \leftarrow (a, \text{true}, \text{false})]}{(\chi, \pi). \text{prod}_{\text{VO}}(x, a) \rightsquigarrow (\chi', \pi)} \\
\hline
\text{VOBS-PRODUCE-WITH-CONTROLLER} \\
\frac{\chi(x) = (a', \text{false}, \text{true}) \quad \chi' = \chi[x \leftarrow (a', \text{true}, \text{true})] \quad \pi' = \pi \wedge (a = a')}{(\chi, \pi). \text{prod}_{\text{VO}}(x, a) \rightsquigarrow (\chi', \pi')}
\end{array}$$

Fig. 11. Excerpt rules for consumer/producer of the prophecy context

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* $\chi = \text{PcyVar} \rightarrow \text{Expr} \times \mathbb{B} \times \mathbb{B}$ 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.

Figure 11 provides rules for successfully producing a value observer into the state. 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 $(a, \text{true}, \text{false})$, 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 $(a', \text{false}, \text{true})$ 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 having modified the contents of a mutable reference $p: \&\text{mut } \tau$, one still needs to apply this rule before being able to close the mutable borrow. The current implementation of Gillian does not allow us to fully automate this process, but we are able to provide the **MUT-AUTO-UPDATE** lemma which can be used by the verification engineer by simply writing `p.prophecy_auto_update()`. This lemma updates the current value of the prophecy by automatically choosing the appropriate value that will allow the borrow to be closed again.

Finally, Gillian-Rust also provides a manual way of *resolving* mutable references, as described by **MUTREF-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.

5.4 A systematic encoding of Creusot specifications

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 Pearlite⁹ specifications of their APIs are assumed as axioms. Operations

⁹Pearlite is a first-order logic, including the usual connectives for conjunction, disjunction, implication, and quantification, and also support for functions and predicate definitions.

on mutable borrows are specified using the *final* operator \wedge which accesses the prophecy of a mutable reference. For example, the Creusot specification of `pop_front` is as follows:

```
#[ensures(match result {
  None => (^self).shallow_model() == Seq::EMPTY,
  Some(x) => self.shallow_model() == Seq::cons(x, (^self).shallow_model())
}]]
fn pop_front(&mut self) -> Option<T> { .. }
```

noting that Creusot allows users to use a post-fix `@` operator to invoke `shallow_model` (cf. Figure 3), but we have desugared the syntax here for clarity.

To compile these specifications to Gilsonite, we first need to interpret Creusot’s types in Gillian-Rust. We interpret Rust types using their *representations*, so that `LinkedList<T>` is interpreted as `Seq<T::ReprTy>`. However, we must also interpret the *logical* types of Creusot, which we do by defining an instance of `Ownable` for these types as well. This means that Creusot’s `creusot::Seq<T>` is interpreted as Gillian-Rust’s `gillian_rust::Seq<T::ReprTy>`.

Specification interpretation is done by *elaboration*, whose general schema is as follows:

$$\begin{array}{ccc} \{P\} & & \{(\otimes_{i=1}^n \llbracket T_i \rrbracket(x_i, m_i)) * \langle P[x_i/m_i] \rangle\} \\ \text{fn } f(x_1 : T_1, \dots, x_n : T_n) \rightarrow T_{\text{ret}} & \implies & \text{fn } f(x_1 : T_1, \dots, x_n : T_n) \rightarrow T_{\text{ret}} \\ \{Q\} & & \{\exists m_{\text{ret}}. \llbracket T_{\text{ret}} \rrbracket(\text{ret}, m_{\text{ret}}) * \langle Q[x_i/m_i][\text{ret}/m_{\text{ret}}] \rangle\} \end{array}$$

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, substituting occurrences of Rust variables with their corresponding representation values. Following this process, our specification for `pop_front` translated to Gilsonite is as follows:

```
#[unsafe_spec( <forall: self_val, result_val>
  requires { self.own(self_val) }
  ensures { result.own(result_val) * $ match result_val {
    None => (^self_val).shallow_model() == Seq::EMPTY,
    Some(x) => self_val.shallow_model() == Seq::cons(x, (^self_val).shallow_model())
  } $
}]]
fn pop_front(&mut self) -> Option<T> { .. }
```

6 EVALUATION

While Gillian-Rust is still only a proof-of-concept, we were able to use it to verify properties of a subset of the APIs provided by the `LinkedList` module from the Rust standard library¹⁰: to our knowledge, this is the second time that type safety and the first time that functional correctness of Rust unsafe code has been verified.¹¹ The source code was copied from the library, specifically from commit `ad2b34d0` dated April 12, 2023. The sole modification made involved manually inlining calls to `Option::map`, as its parameter, a closure, is not yet supported by the Gillian-Rust compiler. It is important to note that Gillian can symbolically execute functions without annotations, so when support for closures is added, verification will not necessitate additional annotations.

We present results and measurements for two experiments conducted on the `LinkedList` library: verifying type-safety and verifying functional correctness. All measurements are conducted on a

¹⁰The currently present limitations that do not allow us to verify the remaining parts of the API are listed in detail in §7.

¹¹Verifast verified type-safety of a simplified, monomorphised `Cell<i32>::set` function, addressing challenges such as the manipulation of non-atomic borrows, but with little automation and no functional correctness guarantees.

MacBook Pro 2019, with 16GB Memory and a 2.3GHz 9-Core Intel Core i9 processor. All proofs are executed sequentially, without any parallelisation.

Verifying type safety for `LinkedList`. We have verified type safety of four functions from the `LinkedList` API: `new`, `push_front`, `pop_front`, and `front_mut`, which takes a total of **0.16s**. In addition to the definition of the safety invariant, which is a cost that we believe cannot be cut for any valid approach to verifying type safety, no function other than `front_mut` requires additional annotations.

The function `front_mut` is a case of borrow extraction, as presented in §4.3. As such, it requires the instantiation and manual application of 2 additional lemmas: an extraction lemma, and an existential freezing lemma. While these lemmas need to be manually declared, their proofs are entirely automatic. We believe that, in the future, the declaration and usage of these lemmas can also be automated in trivial cases.

Functional correctness for `LinkedList`. When it comes to functional correctness of `LinkedList`, we verified specifications of `new`, `push_front_node` and `pop_front_node`, taking a total of **0.18s**. The functional properties proven are those expected and specified by Creusot and are the strongest possible specifications one can give in our framework.

We are not yet able to verify the functional correctness specification for `front_mut`, as the `BORROW-EXTRACT` rule needs to be enhanced in order to enable reasoning about borrow extraction in the presence of prophecies. This rule has been designed and awaits implementation.

7 LIMITATIONS AND FUTURE WORK

In its present form, our infrastructure is a preliminary proof of concept, demonstrating the feasibility of our hybrid methodology in end-to-end Rust verification. As such, it comes with a number of limitations, both in the implementation and meta-theory. We identify and discuss these limitations, and outline how they will be addressed.

7.1 Unimplemented features

Importantly, the five features presented here only pose engineering challenges and mostly require time rather than further insight.

Compiler coverage. The Gillian-Rust compiler is missing support for some language constructs, such as closures, but we do not foresee any complications arising from these extensions. In particular, Gillian supports dynamic calls, extensively tested with JavaScript, and therefore would not have problems dealing with closures and other function pointers.

Multi-crate support. The Gillian-Rust compiler does not yet support multi-crate compilation, a difficult engineering challenge for Rust verifiers. In particular, one cannot import logic definitions, such as specifications or predicates from another crate. Because of this, predicate definitions provided as part of the Gilsonite crate, such as the ownership predicate for mutable references, are hardcoded in the compiler. Creusot was able to overcome this difficulty with heavy engineering work, and we plan to follow their approach in the future.

Functional correctness of borrow extraction. Reasoning about borrow extraction in the presence of prophecies requires an enhanced `BORROW-EXTRACT`, mirroring the need to enhance ownership

predicates of mutable references in the presence of prophecies. This lemma is yet to be implemented in Gillian-Rust, preventing us from verifying a functional specification for `LinkedList::first_mut`.

Automated encoding of Pearlite specifications. While §5.4 provides a systematic way of encoding Pearlite specifications into Gilsonite, this encoding is currently performed manually. Automating this translation is part of our immediate future work.

Connection to the Borrow Checker. At the time Gillian-Rust was implemented, the Rust compiler provided no API to extract the lifetimes information computed by the borrow checker. As a consequence, Gillian-Rust is currently restricted to considering only programs with a single lifetime. Note that the Gillian-Rust back-end does support multiple lifetimes, as presented in this paper, making this limitation only a matter of front-end implementation.

7.2 Meta-theory simplifications

The meta-theory of Gillian-Rust presented in this paper heavily relies on both RustBelt and RustHornBelt, but makes two simplifications that need to be formally justified; we believe that this is possible, though it involves a substantial amount of additional work.

Later modalities. Since Iris is a step-indexed logic, original rules from RustBelt and RustHornBelt, such as `LFTL-BORROW-ACC`, make use of later modalities. We simplify later modalities away, as the meta-theory of Gillian cannot account for them, since it has been formalised using a separation logic that is not step-indexed. We believe that, should Gillian be formalised in Iris, the unfolding and folding ghost commands would be formalised as view shifts which “take a step”, as they are formalised similarly to primitive memory operations. This would be enough to justify the soundness of our approach. In addition, all described paradoxes that would arise in Iris without step-indexing make extensive use of the impredicativity of the logic. On the other hand, Gillian uses a predicative logic, and it is unclear that any paradoxes could arise even without step-indexing. Unfortunately, providing a more formal version of these arguments would either require formalising the meta-theory of Gillian in Iris, or proving RustBelt and RustHornBelt rules using the meta-theory of Gillian, both of which exceed the scope of the current project.

Prophecy dependencies and type well-formedness. Although not presented in the associated paper, the Coq development of RustHornBelt attaches an additional proof obligation to the definitions of Rust types, called `ty_own_proph`, which stipulates that the type ownership predicate must permit access to the value observers for all prophecy variables on which its representation depends. We believe that the current implementation of Gillian-Rust naturally enforces this constraint, thanks to a dataflow requirement imposed on assertion definitions. In Gillian, all predicate parameters must be declared with a mode, `In` or `Out`, such that out-parameters can be uniquely learned by the in-parameters. For example, in the core predicate $a \mapsto_{\top} v$, address a and type \top are in-parameters, as v can be learned uniquely by querying the heap.

In Gillian-Rust, the ownership predicate `fn own(self, repr: Self::RepresentationTy)` declares `self` to be an in-parameter and `repr` to be an out-parameter, and Gillian performs an analysis which ensures that the latter must indeed be entirely learned from the former. Because the ownership predicate of a mutable reference is the only way to obtain a representation which depends on prophecy variables, and this predicate does provide the associated value observer, it is impossible to construct an ownership predicate that that does not satisfy `ty_own_proph`. However, formally proving this property within RustHornBelt is extremely difficult, as it would require a deep embedding of the assertion language on which the dataflow analysis performed by Gillian could be formalised.

7.3 Unexplored topics

Finally, our work, while advancing the state of the art, leaves some topics unexplored.

Extracting knowledge from observations. As mentioned in §6, the specification of `push_front_node` obtained through our systematic encoding from Pearlite cannot be verified as-is by Gillian-Rust. The function increments the `len` field of the structure, and, in order to be executed without overflow, requires the additional Pearlite precondition that `self.shallow_model().len() < usize::MAX`. When encoding this pre-condition into Gilsonite, one would obtain an observation expressing that the length of the representation of the list is less than `usize::MAX`. However, since this information is hidden inside the observation, Gillian-Rust is unable to use it to prove the absence of overflows.

In RustHornBelt, there exists a rule which allows extracting information from an observation if it is independent from prophetic information. Automating this rule is highly challenging, but would enable Gillian-Rust to extract the necessary pre-conditions from Pearlite specifications without user intervention. This challenge is likely to be of the same difficulty for any semi-automated verification tool.

Shared references. For the moment, we do not explore shared references and their ownership predicates. The path forward is to introduce another trait to the Gilsonite API, called `Shareable`, which allows one to define the *sharing* predicate for a given type, as defined by RustBelt. Furthermore, we would need to implement a variant of the guarded predicate algebra which behaves according to the rules governing the behaviour of *fractured borrows*, which are the shared counterpart of full borrows. Gillian-Rust would then be able to derive the ownership predicate for shared references of type `&T`, where `T` is `Shareable`. These enhancements require substantial additional work, which we consider outside the scope of the current presentation, as it aims only at proving the feasibility of our approach. In addition, prophecies are separate from the representation of shared references, suggesting that the step between supporting them for type safety verification and functional correctness reasoning is minimal.

Concurrency. While all of our proven specifications are valid in a concurrent context, we do not explore constructs specific to concurrency. In particular, ownership predicates in RustBelt receive an additional argument corresponding to a thread identifier. Types that are thread-safe—said to be `Send` in Rust—may have an ownership predicate which depends on this identifier. Our approach would have to be extended with thread identifiers in order to prove properties about such types.

xyzBorrows. Finally, we do not model `StackedBorrows` [14] or `TreeBorrows` [18], which are operational semantic models of the aliasing model of Rust. While preliminary research has been conducted on creating a logic to reason about `StackedBorrows` [22], symbolic reasoning in the presence of these models has, to our knowledge, not been performed before. Moreover, there is still no foundational framework that marries any of these models with the theory of semantic typing proposed by RustBelt, and we can therefore not model them confidently within Gillian.

8 RELATED WORK

We provide an overview of the relevant literature on unsafe Rust verification. While there exist many tools other than Creusot for safe Rust verification (such as Prusti [2], Aeneas [11], or Flux [20]), we do not address them in detail as our goal is to reason about unsafe Rust. To our knowledge, currently there exist only three tools capable of performing this reasoning—VeriFast [8], Verus [19], and Kani [28]—none of which explore the idea of hybrid verification.

VeriFast for Rust. Rahimi Froushaani et al. [8] describe a Rust front-end for VeriFast [12] which provides a way of verifying semantic type safety for a fragment of unsafe Rust. By design, this work is similar to ours, as both VeriFast and Gillian rely on a similar theoretical framework of

compositional symbolic execution through consumers and producers. However, VeriFast exists at a different point of the design space, sacrificing some automation that Gillian can provide, in exchange for more speed, predictability, and a robust capacity to explore innovative semi-automatic proof techniques. In particular, VeriFast does not support encoding custom separation algebras, which therefore requires manual application of the rules which Gillian-Rust automates.

The current Rust front-end of VeriFast is somewhat limited and excludes features such as polymorphism. Moreover, it focuses solely on verifying semantic type safety and not functional correctness—essentially, it encapsulates a portion of RustBelt, but does not extend beyond it to RustHornBelt. It also does not provide an approach for reasoning about mutable borrow extraction, but only for borrow opening and closing. Furthermore, it uses an encoding of borrows which requires manual management on the part of the users, while the guarded predicate mechanism of Gillian-Rust facilitates much greater automation. Having said that, the insights of this work and VeriFast itself have contributed to overcoming some of the difficulties encountered during the design and implementation of Gillian-Rust.

Verus. As opposed to Gillian-Rust and VeriFast, Verus [19] does not use separation logic 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 an enjoyable user experience.

However, the linear ghost type approach of Verus renders it unable to verify “traditional” unsafe code. For example, using Verus, one would not be able to verify the standard library implementation of `LinkedList` the way we propose, as they would not be able to directly use raw pointers. Instead, they would require the target code to use different kinds of linear ghost types; in particular, they verify only their own implementation of the `LinkedList` library, using permissioned pointers (denoted $\text{PPtr}<T>$, 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, since Verus does not use separation logic, its specification of the heavily pointer-based invariants such as that of doubly-linked list is more verbose than their separation logic counterparts. Moreover, we have found that, for the verification of heavily pointer-based code, Verus requires many more annotations than Gillian-Rust, as it does not have the ability to use separation logic. Moreover, without separation logic, Verus’s specification of a doubly-linked-list also contains universal quantifiers that are encoded into SMT, while Gillian-Rust’s version of the same specification is entirely quantifier-free.

Finally, Verus cannot reason about functions that return mutable references, and does not present a pathway towards enabling support for that feature.

Kani. Kani [28] is a bounded model checker for Rust, which compiles Rust to the intermediate representation ingested by CBMC [3] for its analysis. While Kani is a well-engineered industrial-strength tool which covers an impressive fragment of the existing Rust language, it does not propose solutions to the challenges solved by our work.

First, as it uses CBMC as a backend, it performs bounded model checking rather than unbounded verification, which is our current task. Kani encodes Rust programs into a C-like representation, instantiating a specific layout for each structure, and performing model checking in that context. As a result, layout-sensitive Rust code is beyond the current ability of Kani. Finally, Kani treats all safe and unsafe code alike, and, as such, is unable to leverage any of the safe Rust guarantees to enhance analysis, unlike our hybrid approach.

9 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 demonstrated how the complex concepts underpinning reasoning about unsafe Rust, such as lifetime logic and prophetic reasoning, can be brought from the interactive theorem proving world of RustBelt and RustHornBelt to the world of semi-automatic verification.

REFERENCES

- [1] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 136:1–136:27. <https://doi.org/10.1145/3428204>
- [2] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 147:1–147:30. <https://doi.org/10.1145/3360573>
- [3] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, Berlin, Heidelberg, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15
- [4] Kees Cook. 2022. [GIT PULL] Rust introduction for v6.1-rc1. <https://lore.kernel.org/lkml/202210010816.1317F2C@keescook>. Accessed: Nov. 16th 2023.
- [5] Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. 2022. Sound Automation of Magic Wands. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Sharon Shoham and Yakir Vizel (Eds.). Springer International Publishing, Cham, 130–151. https://doi.org/10.1007/978-3-031-13188-2_7
- [6] Xavier Denis and Jacques-Henri Jourdan. 2023. Specifying and Verifying Higher-order Rust Iterators. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer Nature Switzerland, Cham, 93–110. https://doi.org/10.1007/978-3-031-30820-8_9
- [7] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: a Foundry for the Deductive Verification of Rust Programs. Springer Verlag. <https://hal.inria.fr/hal-03737878>
- [8] Nima Rahimi Foroushaani and Bart Jacobs. 2022. Modular Formal Verification of Rust Programs with Unsafe Blocks. <https://doi.org/10.48550/arXiv.2212.12976> arXiv:2212.12976 [cs].
- [9] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, part i: a multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 927–942. <https://doi.org/10.1145/3385412.3386014>
- [10] 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.
- [11] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 116:711–116:741. <https://doi.org/10.1145/3547647>
- [12] Bart Jacobs, Jan Smans, and Frank Piessens. 2010. A Quick Tour of the VeriFast Program Verifier. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Kazunori Ueda (Ed.). Springer, Berlin, Heidelberg, 304–311. https://doi.org/10.1007/978-3-642-17164-2_21
- [13] Ralf 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.
- [14] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked borrows: an aliasing model for Rust. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 41:1–41:32. <https://doi.org/10.1145/3371109>
- [15] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec. 2017), 66:1–66:34. <https://doi.org/10.1145/3158154>

- [16] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. <https://doi.org/10.1017/S0956796818000151>
- [17] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2019. The future is ours: prophecy variables in separation logic. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 45:1–45:32. <https://doi.org/10.1145/3371113>
- [18] Ralf Jung and Neven Villani. 2023. From Stacks to Trees: A new aliasing model for Rust. <https://www.ralfj.de/blog/2023/06/02/tree-borrows.html> Accessed: Nov. 16 2019.
- [19] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 85:286–85:315. <https://doi.org/10.1145/3586037>
- [20] Nico Lehmann, Adam Geller, Niki Vazou, and Ranjit Jhala. 2022. Flux: Liquid Types for Rust. <https://doi.org/10.48550/arXiv.2207.04034> arXiv:2207.04034 [cs].
- [21] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2*. report. INRIA. <https://hal.inria.fr/hal-00703441> Pages: 26.
- [22] Daniël Louwink. 2021. *A Separation Logic for Stacked Borrows*. Report. <https://eprints.illc.uva.nl/id/eprint/1790/>
- [23] Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 827–850. https://doi.org/10.1007/978-3-030-81688-9_38
- [24] Nicholas D. Matsakis and Felix S. Klock. 2014. The rust language. *ACM SIGAda Ada Letters* 34, 3 (Oct. 2014), 103–104. <https://doi.org/10.1145/2692956.2663188>
- [25] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 841–856. <https://doi.org/10.1145/3519939.3523704>
- [26] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *ACM Transactions on Programming Languages and Systems* 43, 4 (Oct. 2021), 15:1–15:54. <https://doi.org/10.1145/3462205>
- [27] The Coq Team. 2023. The Coq Proof Assistant. <https://coq.inria.fr/> Accessed: Nov. 16th 2023.
- [28] 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.
- [29] The Rust Team. 2023. Rust Programming Language. <https://www.rust-lang.org/> Accessed: Nov. 16th 2023.