GhostCell: Separating Permissions from Data in Rust

Alex Buzanis, Anton Outkine, Bayard Walsh

May 2023

1 High-Level Summary

This report concerns the theory, implementation, and correctness proof of GhostCell, a novel Rust-language API that expands the scope of compile-time memory safety verification. This is accomplished by providing more coarse-grained control over mutable data structures that incorporate internal sharing [11]. A prerequisite for understanding GhostCell is the Rust concept of lifetimes [4], which enforce strict regulations over the scope of reference validity. Lifetimes form the basis for the implementation of the GhostCell API, as they allow specifying which permissions apply to which pieces of data. By separating permissions from data, GhostCell obviates the need for locks and runtime reference counting. This allows catching bugs earlier in the development process, while also offering greater performance than the alternatives. Explicit examples of GhostCell API usage are detailed below, following the relevant background information.

2 Technical Summary

2.1 Background

Rust is a modern systems programming language designed to guarantee memory safety and data race freedom without relying on a garbage collector. Rust's unique approach to memory safety eliminates entire classes of bugs that plague programs in traditional systems languages like C, all without sacrificing performance. This is enabled in large part by its strict ownership and borrowing system.

In Rust, every value, upon creation, is assigned a unique owner, typically a variable. When a value is moved to another variable or passed as an argument to a function, ownership is transferred, and the value can no longer be accessed through its previous owner. Once a value's final owner goes out of scope, the value is considered "dropped" and the memory it previously occupied is automatically freed. By ensuring that no piece of memory can be freed more than once, Rust eliminates the "double-free" bugs that are possible in C.

To facilitate shared access to data, Rust allows references to be derived from owned data. Each reference is tagged with a "lifetime", or scope for which the reference remains valid. Since the lifetime of a reference is limited by the scope of ownership, this prevents dangling references and the resulting "use-after-free" bugs that are possible in C.

Additionally, Rust makes a clear distinction between immutable and mutable references:

- Immutable references, denoted `&T`, where `T` is some data type, permit read-only access to the referred value. An arbitrary number of immutable references to a value may concurrently exist, provided that no mutable references to that value exist. This enables multiple parts of a program to safely read from the same value with the guarantee that the value won't be tampered with.
- Mutable references, denoted `&mut T`, on the other hand, provide exclusive read/write access to the referred value. When a mutable reference to a value exists, no other reference to that value (whether mutable or immutable) may concurrently exist. This prevents data races in multithreaded contexts, since changes that are being made through a mutable reference cannot be observed nor interfered with in another thread.

This principle that a reference may either be aliased or mutable, but not both at the same time, is called the aliasing-XOR-mutability (AXM) principle [11], and it is fundamental to Rust's assurance of memory safety. The enforcement of this principle not only eliminates data races in multithreaded contexts, but it also makes single-threaded code easier to reason about. This is true for the programmer, but also for the compiler, which in some cases is able to make more aggressive optimizations under the assumption that AXM is not violated.

To ensure that these rules are upheld, the Rust compiler has a component called the "borrow checker", which statically analyzes programs and declines to compile those that break the rules. Since violating these rules results in undefined behavior, the borrow checker is forced to err on the side of caution, rejecting more programs than strictly necessary. This means that there are safe programs that are disallowed by the borrow checker due to its inability to understand more complex patterns of memory use. In such cases, Rust provides an escape hatch in the form of raw pointers and `unsafe {}` blocks.

Raw pointers, denoted `*const T` and `*mut T`, are akin to traditional pointers in C. Unlike references, raw pointers are not bound by Rust's borrowing rules, but rather allow for unrestricted memory access. To limit possible places where code can go wrong and to facilitate building safe abstractions around unsafe code, Rust only allows raw pointers to be used within explicit `unsafe {}` blocks.

While `unsafe` may sound alarming, it does not mean the code inside the block is definitely unsafe. Rather, it merely instructs the compiler to trust the programmer. It remains crucial to ensure that usages of raw pointers within the unsafe block still uphold Rust's safety rules, because violating the rules is always undefined behavior. As such, `unsafe` blocks are best used sparingly,

ideally only in the implementations of safe abstractions in libraries whose code is seen by many eyes.

The Rust standard library is one such library, as it provides certain "interior mutability" primitives, which are abstractions that internally use unsafe code to enable shared mutable state. These wrapper types, such as `RefCell` and `RwLock`, may seem to contradict the AXM principle, but they maintain reference counts and use locks at run-time to ensure that the borrowing rules are upheld before allowing their wrapped values to be mutated.

The standard library's interior mutability primitives are useful in that they allow for more complex sharing and mutation scenarios, such as in the case of cyclic data structures with internal sharing (e.g., doubly linked lists), which would otherwise be rejected by the borrow checker. However, their use incurs a runtime performance cost, and in the case that they are used incorrectly, they can cause deadlocks or crashes. This is why verification of memory safety at compile-time is preferred whenever possible.

2.2 GhostCell and GhostToken

In "GhostCell: Separating Permissions from Data in Rust", the authors introduce a novel interior mutability data type, GhostCell, which leverages Rust's borrow checker in a clever way to expand the set of situations where memory safety can be verified at compile-time. This brings increased runtime performance compared to the standard library's interior mutability primitives, and it enables bugs to be caught earlier in the development process (at compile-time rather than at run-time).

The idea behind GhostCell is to decouple data from the permissions to access it. This is accomplished through the introduction of two new data types: `GhostCell<'id, T>` and `GhostToken<'id>`. The first type, `GhostCell<'id, T>`, acts as a wrapper around a piece of data of type T, while the second type, `GhostToken<'id>`, represents permission to access any GhostCell marked with that same branding lifetime, `'id`.

An immutable reference to a `GhostCell<'id, T>` alone cannot be used to access the data. However, when paired with an immutable reference to the corresponding `GhostToken<'id>`, the GhostCell provides an immutable reference to its data. Alternatively, when paired with a mutable reference to the corresponding `GhostToken<'id>`, the GhostCell provides a mutable reference to its data.

`GhostToken` thus allows for a more coarse-grained form of permission checking than the standard library's data types. Since the permissions are separate from the data, multiple GhostCells with the same brand can be shared freely without concerns of unauthorized access, eliminating the need to count references or lock each individual piece of data. This is all made possible by the fact that access to the shared data is determined by access to the corresponding `GhostToken<'id>`.

2.2.1 Invariant Lifetime as Unique Brand

In Rust, lifetimes are normally used to track how long references remain valid. In the case of `GhostCell<'id, T>` and `GhostToken<'id>`, however, the lifetime parameter `'id` is used differently, as a unique brand linking a GhostCell with a specific GhostToken. The problem with implementing this naively is that lifetimes in Rust are covariant by default: if `'a:'b` (meaning lifetime `'a` outlives lifetime `'b`), then `'b` can be substituted wherever `'a` is expected [5]. This is acceptable in most cases, because the substituted lifetime will be valid for as long as the original. However, in the case of `GhostCell` and `GhostToken`, this usual covariant behavior could allow one GhostToken to be substituted for another. This, in turn, would allow for the creation of multiple mutable references to the contents of the same GhostCell, which would violate the AXM principle.

To solve this problem, the authors designed `GhostCell` and `GhostToken` to be invariant over their branding lifetime parameter `'id`. This means that the only lifetime that can be used in place of `'id` is `'id` itself. Any attempt to substitute a different lifetime will result in a compile-time error. This strict control over the branding lifetime is what allows `GhostCell` and `GhostToken` to uphold AXM, even when multiple GhostCells share the same brand. We delve into the implementation details of the branding lifetime in section 4.

3 Digest of Related Work

When looking through GhostCell and grasping the concept of separating data from permissions, we found the reference to implicit dynamic frames to be useful. The earlier paper, titled "Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic" [9] was cited in the GhostCell paper as a foundational inspiration for the GhostCell API, specifically the notion of Separation Logic. This paper formalizes Separation Logic, explaining how it enables precise reasoning about memory sharing and ownership. While the paper was not designed with Rust in mind (and therefore the essential Rust lifetimes used in GhostCell), the approach of reasoning about dynamic memory allocation and deallocation by automatically inferring frame predicates based on Separation Logic annotations is very pertinent to the GhostCell approach. Therefore, observing the concept of separating data from permission in an application gave us a deeper appreciation for how GhostCell creates this separation through a type system via a user-defined library in Rust.

One of the obstacles to our understanding we encountered in our analysis of GhostCell was the usage of Coq in the paper's formalized proof. While we had experience working with more basic type soundness proofs throughout the class, the automated process of creating proofs was a challenge to wrap our heads around. Therefore, we consulted the "lambda-rust" [3] repository linked by the paper to better understand formal proof management systems. Referencing this repository enabled us to consider a code-focused implementation of the proofs offered in the paper, which in the long run made the unfamiliar symbols in the

proof given by the paper far less abstract and help us understand the formal safety of GhostCell.

While not explicitly referenced by the paper, in our search into the current status of GhostCell, we encountered a paper that built on GhostCell, titled, "Linear types for large-scale systems verification". This material enabled us to see how the system was criticized, as the paper mentioned: "the doubly linked list example in [Yanovski et al. 2021] has to expose the GhostToken lifetime variables to the clients of the doubly linked list, since Rust currently lacks existential lifetime variable quantification, whereas our Linear Dafny doubly linked list can hide the use of regions from clients" [6]. When we used GhostCell ourselves, we too found ourselves frustrated by the abundance of lifetime annotations, as well as the necessity for using GhostToken within a closure. The lack of support for existential lifetimes in Rust certainly complicates the GhostCell API and puts a greater burden on the user. We discuss existential lifetimes as a solution to this complication in section 4.

4 Constructive Engagement with Work

To begin our engagement with this paper, we implemented our own version of the GhostCell API without consulting the paper's reference implementation. Afterwards, we checked that our implementation matched with the official one, and indeed it did. Most of the methods in the API consist of simple type-casts, which speaks to the the fact that GhostCell is truly a zero-cost abstraction.

Source Code 1: GhostCell API Implementation

```
use std::{cell::UnsafeCell, marker::PhantomData};

// Phantom lifetime type that can only be subtyped by exactly the same lifetime `'id`.
struct InvariantLifetime<'id>(PhantomData<*mut &'id ()>);

/// A single "branded" permission to access the data structure.
/// Implemented with a phantom-lifetime marker type.
pub struct GhostToken<'id> {
    _marker: InvariantLifetime<'id>,
}

/// Branded wrapper for the data structure's nodes, whose type is T.
pub struct GhostCell<'id, T: ?Sized> {
    _marker: InvariantLifetime<'id>,
    value: UnsafeCell<T>,
}

impl<'id> GhostToken<'id> {
    /// Creates a fresh token that GhostCells can be tied to later.
```

```
    pub fn new<R>(f: impl for<'new_id> FnOnce(GhostToken<'new_id>) -> R) -> R {
        f(Self {
            _marker: InvariantLifetime(PhantomData),
        })
    }
}

impl<'id, T> GhostCell<'id, T> {
    /// Wraps some data T into a GhostCell with brand `'id`.
    pub fn new(value: T) -> Self {
        Self {
            _marker: InvariantLifetime(PhantomData),
            value: UnsafeCell::new(value),
        }
    }

    /// Turns an owned GhostCell back into owned data.
    pub fn into_inner(self) -> T {
        self.value.into_inner()
    }

    /// Turns a mutably borrowed GhostCell to and from mutably borrowed data.
    pub fn get_mut(&mut self) -> &mut T {
        self.value.get_mut()
    }
    pub fn from_mut(t: &mut T) -> &mut Self {
        unsafe { &mut *(t as *mut T as *mut Self) }
    }

    // Immutably borrows the GhostCell with the same-branded token.
    pub fn borrow<'a>(&'a self, token: &'a GhostToken<'id>) -> &'a T {
        unsafe { &*self.value.get() }
    }

    /// Mutably borrows the GhostCell with the same-branded token.
    pub fn borrow_mut<'a>(&'a self, token: &'a mut GhostToken<'id>) -> &'a mut T {
        unsafe { &mut *self.value.get() }
    }
}
```

4.1 API Type Definitions

The GhostCell API defines two user-facing types: `GhostCell<'id, T>` and `GhostToken<'id>`. Both types have a lifetime parameter, `'id`, which serves as a brand to establish an association between them. Lifetimes are usually used to enforce the scope for which a reference remains valid, ensuring a reference

is not used after its referent is dropped. In this case, however, `'id` is not used like a normal lifetime, but rather as a statically checked tag that links each GhostCell with a particular GhostToken.

`GhostCell<'id, T>` represents shared data of type `T`, marked with a brand `'id`. It serves as a wrapper for the data, ensuring safety without adding any runtime overhead. As a zero-cost abstraction, `GhostCell<'id, T>` has the same memory representation as the underlying type `T`.

`GhostToken<'id>` represents the permission to access the shared data in `GhostCell<'id, T>` branded with the same `'id`. This type is used to enforce the safety rules at compile time, with its borrowing patterns guiding the data access in the associated `GhostCell<'id, T>`.

4.1.1 Invariant Lifetime Implementation

In Rust, the variance of a struct is determined by the variance of the types of its fields [5]. To enforce the invariance of GhostCell and GhostToken over their branding lifetimes, the authors create a struct, `InvariantLifetime<'id>`, which includes a piece of phantom data that acts like it contains a mutable pointer, `*mut &'id ()`. Since a mutable pointer `*mut T` can act as both a source an a sink, the Rust compiler treats it (and thus any struct that contains it) as invariant over `T`. This means that `*mut T` cannot be substituted for `*mut U`, even if `T` and `U` are related (for instance, if `U` is a subtype of `T`). The authors exploit this property of `*mut T` in the definition of `InvariantLifetime`: By including a `PhantomData<*mut &'id ()>` in `InvariantLifetime<'id>`, the authors render `InvariantLifetime<'id>` invariant over `'id`. Then, by including an `InvariantLifetime<'id>` within the `GhostCell` and `GhostToken` types, the authors enforce the desired invariance over the branding lifetime.

This consequence of this is that a `GhostToken<'a, T>` cannot be used to access a `GhostCell<'b, T>` just because `'a:'b`. Rather, `GhostToken<'a, T>` is only compatible with `GhostCell<'a, T>`. This invariant lifetime is what ensures that each "brand" of `GhostCell` is only accesible by via the correct `GhostToken`, thus preventing AXM violations. It's important to note that `PhantomData` and `InvariantLifetime<'t>` are zero-sized types, so they do not bring any run-time overhead. They are merely marker types that enforce the compile-time link between `GhostCell` and `GhostToken`.

4.2 API Methods

The core GhostCell API consists of the following methods:

- **GhostToken::new:** This is a constructor method that requires a closure which can work with a GhostToken with an arbitrary brand `'new_id`. The method picks a fresh brand `'new_id`, creates a new `GhostToken<'new_id>`, and then passes it to the closure.
- **GhostCell::new, GhostCell::into_inner, GhostCell::from_mut, and GhostCell::get_mut:** These methods allow converting between types

`T` and `GhostCell<'id, T>`, as well as between mutable references `&mut T` and `&mut GhostCell<'id, T>`. This reflects the fact that a uniquely owned `GhostCell<'id, T>` carries the same ownership as `T` and can be directly accessed without a GhostToken.

- **GhostCell::borrow and GhostCell::borrow_mut:** These methods allow access to the content `T` of a `GhostCell<'id, T>` while maintaining safety by requiring a `GhostToken` with the same brand `'id`. The `borrow` method requires a shared reference to the GhostToken to return a shared reference to the content, whereas the `borrow_mut` method requires a mutable reference to the GhostToken to return a mutable reference to the content. These methods leverage the Rust borrow checker to enforce the AXM principle.

After implementing the GhostCell API ourselves, we decided to put it to use by constructing a doubly-linked list. Before doing this, however, we attempted an implementation without GhostCell.

Source Code 2: Doubly-Linked List Without GhostCell

```
1  struct Node<'a, T> {
2      data: T,
3      prev: Option<&'a Node<'a, T>>,
4      next: Option<&'a Node<'a, T>>
5  }
6
7  fn construct_list<'a>() -> Node<'a, u32> {
8      let mut node3 = Node { data: 3, prev: None, next: None };
9      let mut node2 = Node { data: 2, prev: None, next: None };
10     let mut node1 = Node { data: 1, prev: None, next: None };
11
12     node3.prev = Some(&node2);
13     node2.next = Some(&node3);
14     node2.prev = Some(&node1);
15     node1.next = Some(&node2);
16     node1
17 }
18
19 fn main() {
20     let head = construct_list();
21 }
```

The core datatype that defines the doubly-linked list is the `Node` struct, which contains a piece of data, a reference to the next node, and a reference to the previous node. Lines 8-10 initialize unconnected nodes, and lines 12-15 set their `next` and `prev` properties.

However, we discovered that there is no way for this code to compile, since it breaks the AXM principle. Specifically, the problem occurs on lines 13 and 15. First, on line 12, we set the third node's `prev` property to the second node, thereby creating an immutable reference to the second node. Then, on line 13, we mutate the second node by setting its `next` property to a reference to third node. Although we are able to create this immutable reference to the third node, since we are no longer mutating it, we cannot create the mutable reference the second node that would require updating its `next` property, since it is still borrowed as immutable. A similar problem occurs on line 15, and this is what causes the compilation error.

One possible way to get around this is to leverage the "interior mutability" primitives that are discussed earlier in the paper. The benefit of this approach is that it makes it possible to mutate data from multiple different places, since Rust does not restrict the number of immutable references that point to a piece of data.

The following is an attempt to fix the doubly-linked list example by using `RwLock`, which is one such primitive.

Source Code 3: Doubly-Linked List With RwLock

```
1  use std::sync::{Arc, RwLock};
2  struct Node<T> {
3      data: T,
4      prev: Option<NodeRef<T>>,
5      next: Option<NodeRef<T>>,
6  }
7
8  type NodeRef<T> = Arc<RwLock<Node<T>>>;
9
10 fn construct_list<'a>() -> NodeRef<u32> {
11     let node1 = Node { data: 1, prev: None, next: None };
12     let node2 = Node { data: 2, prev: None, next: None };
13     let node3 = Node { data: 3, prev: None, next: None };
14
15     let node1 = Arc::new(RwLock::new(node1));
16     let node2 = Arc::new(RwLock::new(node2));
17     let node3 = Arc::new(RwLock::new(node3));
18
19     RwLock::write(&node3).unwrap().prev = Some(Arc::clone(&node2));
20     RwLock::write(&node2).unwrap().next = Some(Arc::clone(&node3));
21     RwLock::write(&node2).unwrap().prev = Some(Arc::clone(&node1));
22     RwLock::write(&node1).unwrap().next = Some(Arc::clone(&node2));
23
24     node1
25 }
26
```

```
fn main() {
    let head = construct_list();
}
```

Unlike Source Code 2, this program compiles. Thanks to the use of an interior mutability primitive, there is no longer a problem with breaking AXM while connecting the nodes of the doubly linked list. The `write` method on `RwLock` checks that the node is not currently locked by another call to write, and then returns a temporary mutable reference to it. Once that reference is dropped (at the end of the line of code), the node is again unlocked.

Unfortunately, there are two drawbacks to this approach. The first is conceptual: in most of the scenarios that involve mutating this doubly-linked list, we will need to modify multiple nodes. It therefore is more conceptually sound to have the ability to unlock the list as a whole instead of having to independently do so for every node that we care about.

The second drawback has to do with performance: this current implementation requires creating an `RwLock` primitive for every single node in our list, and thus also storing the associated `RwLock` state for every single node. This will cause performance issues for programs that create large lists and frequently mutate them.

The GhostCell implementation addresses both of these drawbacks.

Source Code 4: Doubly-Linked List With GhostCell

```
struct Node<'a, 'id, T> {
    data: T,
    prev: Option<&'a GhostCell<'id, Node<'a, 'id, T>>>,
    next: Option<&'a GhostCell<'id, Node<'a, 'id, T>>>,
}

fn main() {
    GhostToken::new(|mut token| {
        let node1 = Node { data: 1, prev: None, next: None };
        let node2 = Node { data: 2, prev: None, next: None };
        let node3 = Node { data: 3, prev: None, next: None };

        node1.borrow_mut(&mut token).prev = Some(&node2);
        node2.borrow_mut(&mut token).next = Some(&node3);
        node2.borrow_mut(&mut token).prev = Some(&node1);
        node1.borrow_mut(&mut token).next = Some(&node2);
    })
}
```

There are two notable differences with this implementation: first, `GhostCell` primitives are replaced by `GhostCell` and `RwLock::write` method invocations are replaced by `GhostCell::borrow_mut` invocations. Second, the list initialization code is moved into a closure (i.e., an anonymous function) that is passed

to the `GhostToken::new` method. By virtue of being in this closure, the list initialization code has access to the `GhostToken`, which is passed as an argument to the closure.

`GhostCell::borrow_mut` is called just as many times as `RwLock::write`, but the invocation of `borrow_mut` does not cause any impact on the performance of the program, since GhostCell is a zero-cost abstraction. Instead, the analogue to `RwLock` locking occurs in the invocation of `GhostToken::new`, which happens just once, before we begin work on the list, but even that gets compiled away.

4.3 Hypothetical Rust Feature: Existential Lifetimes

Although this restructuring that the `GhostCell` library imposes on programs addresses the two drawbacks discussed earlier, it also comes with its own drawback: any code that requires working with the data that is wrapped in `GhostCell` structs must occur within the closure passed to `GhostToken::new`, since this is the only part of the program where the branded lifetime is defined. This not an issue if the goal of invoking `GhostToken::new` is to obtain an end-value that is independent of the data that is wrapped in `GhostCell`. If, however, this data needs to be stored as part of global state, and accessed in other parts of the program, then `GhostCell` would not work.

As mentioned earlier, related work [6] notes how existential lifetimes could simplify the user-facing API of `GhostCell`. Existential lifetimes would allow for a kind of anonymous or hidden lifetime that can be bound to a scope without the need for explicit declaration or closure. Currently, `GhostToken::new` takes a closure that accepts a `GhostToken` with an arbitrary lifetime:

```
GhostToken::new(|token| {
    let cell = GhostCell::new(42);
    let value = cell.borrow(&token);
    // ... use value ...
});
```

If Rust supported existential lifetimes, `GhostToken::new` could instead bind an existential lifetime `'id` to the returned `GhostToken` as well any `GhostCell` that uses it. Whereas a normal lifetime must exist for a specific scope of the program, an existential lifetime would simply "exist" but have unspecified scope. Since GhostCell and GhostToken would still be invariant over their branding lifetime, this would provide the same link between GhostCell and GhostToken without the need for a closure. The simplified code could then look like this:

```
let token = GhostToken::new();
let cell = GhostCell::new(&token, 42);
let value = cell.borrow(&token);
// ... use value ...
```

5 Formal Proof of GhostCell Correctness

The implementation of GhostCell includes several `unsafe` blocks, which means that a manual correctness check is necessary to ensure that the library is safe to use. GhostCell performs this check by extending and adapting the work of the RustBelt project [2]. Before explaining the specific modifications that GhostCell proposes, it is instructive to go over the general approach of this project.

RustBelt works by utilizing what is called a "semantic model," which relies on a sequence of "safety contracts," that each function in a program need to satisfy. Because of the `unsafe` blocks used by the GhostCell library, the authors of this paper had to prove these safety contracts manually.

Proofs of safety contracts begin with the choice of a logic system. The specific system that the RustBelt authors chose, and that the GhostCell authors work with, is called Iris.

Because the proof strategy that the authors use in their paper is complex, this report will focus on one specific part of it: how the proof of GhostCell ties each token to the branded lifetime that uniquely identifies it.

Although lifetimes are formalized in RustBelt, the notion of "token" is not. Iris does, however, have a notion of "ghost state," and the authors decided that this would be a good fit for representing tokens. What complicates their task is that "ghost state" is purely logical and connects to physical state only through invariants.

In order to represent tokens through ghost state, the authors create an invariant for ghost token names with a new proposition that extends RustBelt lifetime logic called GhostLft (κ, γ), where κ is a lifetime and γ is a unique ghost name. They enforce uniqueness of the token name that parameterizes this proposition with the following rule:

$$\text{GhostLft}\,(\kappa, \gamma_1) * \text{GhostLft}\,(\kappa, \gamma_2) \mathrel{-\!\!*} \gamma_1 = \gamma_2.$$

The "$*$" in this statement is the separating conjunction from separation logic, which is a logical system that allows for reasoning about divided data in computer systems, and the "$-\!\!*$" operator is a type of implication operator that is used in this system. In this case, the division addressed by the separating conjunction is between two different occurrences of ghost names. The rule states that if two pieces of ghost state are associated with the same lifetime, then the two ghost names that parameterizes those instances of state must be the same.

After formalizing this piece of ghost state, the authors then add additional rules for how it interplays with actual ghost tokens. In order to do this, they introduce another proposition that represents ghost tokens called TokState (γ, s), where γ is a ghost name and s is the current state of the ghost token, which can either be StOwn if there is currently full ownership of this token, or StShare(κ) if this token is shared and governed by lifetime κ. A subscript is used to denote the extent to which token is owned. For example TokState_1 corresponds to full ownership.

The authors introduce several rules that work with this proposition (which require the use of "$\Rrightarrow\!\!\!*$", an alternative implication operator).

$$1.\ \text{True} \Rrightarrow \exists\gamma.\text{TokState}_1(\gamma, \text{StOwn}).$$

This rule allows for initializing a new ghost token, and gives it a state of StOwn, since it will be fully owned after initialization.

$$2.\ \text{TokState}_1(\gamma, s) \Rrightarrow \text{TokState}_1\left(\gamma, s'\right).$$

This rule allows for changing the ownership of a fully owned token.

$$3.\ \text{TokState}_{q_1}(\gamma, s) * \text{TokState}_{q_2}\left(\gamma, s'\right) \Rrightarrow \left(s = s'\right) * \left(q_1 + q_2 \leq 1\right).$$

Lastly, this rule is similar to the uniqueness rule that the authors defined for GhostLft, and states that if two token names are equivalent, then their states must be the same (either owned or shared), and the degree of ownership of these tokens must not exceed 1. The addition of the $q_1 + q_2 \leq 1$ property is necessary because it is impossible to own more than a whole token.

After introducing these rules, the authors connect their ghost token predicate to their ghost state predicate by defining the following shorthand:

$$\text{TokState}_q\left(\kappa_{\text{id}}, s\right) := \exists\gamma.\text{GhostLft}\left(\kappa_{\text{id}}, \gamma\right) * \text{TokState}_q(\gamma, s).$$

This TokState shorthand is parameterized by a degree of ownership (q), a lifetime ($\kappa_i d$), and the current owernship state of the ghost token. Given this configuration, the shorthand asserts that existence of a ghost name γ, and then creates a conjunction between the ghost state that wraps this name (with $\text{GhostLft}(\kappa_i d, \gamma)$) and the token that instance that corresponds to the name (with $\text{TokState}_q(\gamma, s)$).

In this way, the shorthand allows the authors to hide the connection between lifetimes and ghost names while proving safety contracts for the GhostCell library. This entire mechanism thus establishes a strong correspondence between the logical assertions of Iris and the concrete implementation of GhostCell, enabling verification of the safety properties of Ghostcell despite the presence of `unsafe` blocks.

6 Current Status of Work

Research for GhostCell was in part driven by the explosion in Rust's popularity over the past decade (starting with 6.6% of developer's preferred language in 2017 to 17.6% in 2022 [1]). In terms of the current status of GhostCell users, we found that the project was predominately self-contained in academia (13 citations on Google Scholar [10]), and there are a few papers that adopt and build off its findings. These papers were predominately in the theoretical systems safety research space. This is largely caused by the fact that GhostCell was created as a paper within the larger RustBelt project [2], meaning that it was one of several projects on proving the logical foundations for safe systems programming in Rust. However, it was the top paper featured on the RustBelt

website and seems to be one of the central achievements of the project. We also found a YouTube video made by the supervisor of the RustBelt project, Derek Dreyer, where he walks through and explains GhostCell in depth.

So far, we could not find any examples of wide-scale industry adoption. Due to its recent release date in 2021, it is possible that GhostCell could be implemented into commercial code projects in the future. However, its extra syntax could dissuade some developers, especially on projects where the performance benefit is not essential for functionality. In terms of activity for online communities, we found some online repositories for novel applications from developers inspired by the paper. For example, we came across a GitHub repository inspired by researchers with 367 stars [7] and a Crates.io page with 5,973 downloads all-time [8]. These online versions were easy to download and use, though we opted to design a custom implementation.

A notable aspect of GhostCell is how central its concept and implementation are tied to the features of Rust itself. Without lifetimes or some other form of branded type support in Rust, the API would not be possible. Furthermore, its emphasis on type safety and speed through controlling scope and permissions for variables exactly matches the spirit which initially inspired the surge in Rust's popularity.

References

[1] Pavan Belagatti, *Rust by the Numbers: The Rust Programming Language in 2021.* The New Stack. May 12th, 2021. https://thenewstack.io/rust-by-the-numbers-the-rust-programming-language-in-2021.

[2] Derek Dreyer and Rustbelt Team, *ERC Project "RustBelt"*, https://plv.mpi-sws.org/rustbelt/.

[3] Ralf Jung (2023), *lambda-rust*, GitLab repository, https://gitlab.mpi-sws.org/iris/lambda-rust

[4] Steve Klabnik and Carol Nichols, *References and Borrowing*, The Rust Programming Language, https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html.

[5] The Rust Programming Language. *Subtyping and Variance*, The Rust Reference, https://doc.rust-lang.org/reference/subtyping.html.

[6] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2022. *Linear Types for Large-Scale Systems Verification.* Proc. ACM Program. Lang. 6, OOPSLA1, Article 69 (April 2022), 28 pages. https://doi.org/10.1145/3527313

[7] matthieu-m (2022), *ghost-cell*, GitHub repository, https://github.com/matthieu-m/ghost-cell.

[8] matthieu-m (2022), *ghost-cell*, Rust crate, https://crates.io/crates/ghost-cell.

[9] Jan Smans, Bart Jacobs, and Frank Piessens. 2009. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In ECOOP. https://doi.org/10.1007/978-3-642-03013-08

[10] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. *Ghostcell: separating permissions from data in rust, Google Scholar* https://scholar.google.com/scholar?cites=13429561981895750 35assdt =400005sciodt=0,14hl=en.

[11] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. *GhostCell: Separating Permissions from Data in Rust.* Proc. ACM Program. Lang. 5, ICFP, Article 92 (August 2021), 30 pages. https://doi.org/10.1145/3473597