GhostCell: Separating Permissions from Data in Rust

JOSHUA YANOVSKI, MPI-SWS, Germany
HOANG-HAI DANG, MPI-SWS, Germany
RALF JUNG, MPI-SWS, Germany
DEREK DREYER, MPI-SWS, Germany

The Rust language offers a promising approach to safe systems programming based on the principle of *aliasing XOR mutability*: a value may be *either* aliased *or* mutable, but not both at the same time. However, to implement pointer-based data structures with internal sharing, such as graphs or doubly-linked lists, we need to be able to mutate aliased state. To support such data structures, Rust provides a number of APIs that offer so-called *interior mutability*: the ability to mutate data via method calls on a shared reference. Unfortunately, the existing APIs sacrifice flexibility, concurrent access, and/or performance, in exchange for safety.

In this paper, we propose a new Rust API called `GhostCell` which avoids such sacrifices by *separating permissions from data*: it enables the user to safely synchronize access to a *collection* of data via a single permission. `GhostCell` repurposes an old trick from typed functional programming: *branded types* (as exemplified by Haskell's `ST` monad), which combine phantom types and rank-2 polymorphism to simulate a lightweight form of state-dependent types. We have formally proven the soundness of `GhostCell` by adapting and extending RustBelt, a semantic soundness proof for a representative subset of Rust, mechanized in Coq.

CCS Concepts: • **Theory of computation** → **Type structures**; **Separation logic**.

Additional Key Words and Phrases: Rust, type systems, separation logics

ACM Reference Format:
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

1 INTRODUCTION

Most modern programming languages make a choice between *safety* and *control*: either they provide safe high-level abstractions or they provide low-level control over system resources, but not both. The Rust programming language [Matsakis and Klock II 2014] offers an exciting alternative: it provides fine-grained control over resource management and data layout *à la* C/C++, but with a strong ownership type system in place to ensure type safety, memory safety, and data race freedom.

The central tenet of Rust is that the most insidious source of safety vulnerabilities in systems programming is the unrestricted combination of mutation and aliasing—when one part of a program mutates some state in such a way that it corrupts the view of other parts of the program that have aliases to (*i.e.*, references to) that state. Consequently, Rust's type system enforces the discipline of *aliasing XOR mutability* (AXM, for short): a value of type `T` may *either* have multiple aliases (called *shared references*), of type `&T`, *or* it may be mutated via a unique, *mutable reference*, of type `&mut T`, but it may not be both aliased and mutable at the same time.

Authors' addresses: Joshua Yanovski, MPI-SWS, Saarland Informatics Campus, pythonsq@mpi-sws.org; Hoang-Hai Dang, MPI-SWS, Saarland Informatics Campus, haidang@mpi-sws.org; Ralf Jung, MPI-SWS, Saarland Informatics Campus, jung@mpi-sws.org; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, dreyer@mpi-sws.org.

2475-1421/2021/8-ART92
https://doi.org/10.1145/3473597

A defining feature of Rust's AXM discipline is that it *ties permissions to data*—that is, the permission to read or write an object through a reference is reflected in the type of the reference itself. In so doing, Rust is able to effectively piggyback ownership and permission tracking on top of automatic type checking, which is convenient and ergonomic for many systems programming scenarios. However, it is not always what programmers want: there are also common scenarios in which it would be much more natural to *separate permissions from data*—that is, to track permissions separately from the data they govern.

In this paper, we will (1) motivate the need for separating permissions from data, (2) propose a novel Rust API called `GhostCell` that addresses it, and (3) establish formally that `GhostCell` is a safe extension to Rust. As we shall see in §1.2, our approach relies crucially on some old tricks from typed functional programming, given new life in the context of Rust.

1.1 Motivation: Safely Implementing Data Structures with Internal Sharing

In Rust, the AXM discipline guides the design of not only the core type system but also data structure APIs, in particular *container* data structures that manage user-controlled data. Typically, containers will provide full mutable access to their content given a mutable reference of type `&mut Container`, while providing thread-safe, read-only access given a shared reference of type `&Container` (*i.e.*, the data structure can be read from multiple threads concurrently). Thanks to AXM, containers permit the creation of *interior pointers* that point directly into the data structure, *e.g.*, `Vec` hands out pointers pointing directly into the backing buffer that stores all elements of the vector [Jung et al. 2021]. Interior pointers are an example of what Rust means by "providing control over data layout", and they are crucial to avoid unnecessary copying or pointer indirections. As a result, performance of these data structures is on par with the equivalent data structures in C++, mostly because the Rust code is doing essentially the exact same work as the C++ code—the extra checks required for safety are almost entirely carried out at compile time.

When it comes to efficiently *implementing* these data structure APIs in Rust, the AXM discipline works great for tree-like data structures—*i.e.*, data structures that are acyclic and maintain at most one pointer to any internal node. For tree-like data structures, a (unique) mutable reference to the root of the data structure can be safely used to traverse and mutate the entire data structure, and shared references to the data structure can be safely used by multiple threads to read it concurrently. However, the AXM discipline is not such a good fit for implementing data structures with *internal sharing*—*i.e.*, data structures like graphs or doubly-linked lists, which may have cycles and/or aliased nodes (nodes with in-degree greater than 1). For implementing such data structures, Rust's type system is overly restrictive because it does not allow aliased nodes to be mutated.

In order to still support mutation of data structures with internal sharing, Rust currently provides two ways to work around the AXM discipline. The first is to simply "give up" and circumvent the restrictions of Rust's type system by employing unsafe features of the language (*e.g.*, using raw pointers, whose aliasing is untracked). For example, the `LinkedList` type in the standard library is realized this way. Obviously, this approach is viewed as a last resort.

The second, *safe* alternative is to wrap aliased nodes in a "cell" type that provides so-called *interior mutability*: the ability to mutate the underlying data *indirectly* via method calls on a shared reference to the cell. One may wonder: this sounds like a flagrant violation of the AXM discipline—how can it be safe in general? The answer is indeed, it is *not* safe in general, but it *can* be safe if certain restrictions are placed on the type of data being aliased and/or if certain dynamic checks are performed to ensure proper synchronization of read and write accesses.

For example, suppose we would like to implement a doubly-linked list in safe Rust, providing a Rust-style API that includes the ability to take interior pointers into the list and perform concurrent unsynchronized reads. In this case, since the nodes of the list are aliased by both their `previous`

and `next` neighbors, we must wrap them in *some* "cell" type if we want to be able to mutate them. But the only thread-safe "cell" types that provide interior pointers are `Mutex` and `RwLock` (which of these we choose depends on whether or not we want to allow nodes to be read concurrently). Both of these types will have the effect of protecting each node in the list with its own lock (a mutex or a reader-writer lock, respectively) to ensure all accesses to the node are properly synchronized. The resulting implementation of the linked-list `Node` type is as follows:[1]

```
struct Node<T> {
  data: T,
  prev: Option<NodeRef<T>>, // None for null pointers
  next: Option<NodeRef<T>>,
}
type NodeRef<T> = &RwLock<Node<T>>;
```

This implementation achieves the goal of providing a thread-safe AXM-style API with support for interior pointers, but performance when compared with C++ is abysmal! To enable a safe implementation of the desired API, we had to resort to *per-node locking*, so even a simple immutable iteration has to constantly acquire and release locks while traversing the list. As we will see in §5.1, this (unsurprisingly) causes a massive slowdown.

As a result, the Rust programmer is caught between a rock and a hard place: they can either use unsafe code and give up on guaranteed safety, or they can use interior mutability and give up on good performance. This is unsatisfying because the whole point of Rust is to "have our cake and eat it, too": we do *not* want to make a compromise between safety and performance. There must be a better way!

1.2 GhostCell: A Thread-Safe Zero-Cost Abstraction for Interior Mutability in Rust

In this paper, we show how to extend Rust to support the safe and efficient implementation of data structures with internal sharing, using a new API we call `GhostCell`. In particular, unlike all existing thread-safe interior-mutable APIs presently available in Rust, `GhostCell` is literally a *zero-cost abstraction*: its methods consist simply of type casts, whose soundness we justify in this paper but which are erased completely by the Rust compiler.

The design of `GhostCell` is rooted in the observation that Rust's existing approaches to interior mutability (*e.g.*, wrapping each node of a linked list with a `RwLock`) incur unnecessary overhead for data structures with internal sharing because they *tie permissions to data*. That is, they track "permission state"—*i.e.*, whether a single party has write permission or multiple parties have read permission—at the too-fine granularity of individual aliased objects (*e.g.*, the constituent nodes of a linked list). In contrast, `GhostCell` *separates permissions from data*: it enables one to associate a single permission state with a *collection* of objects (*e.g.*, the collection of nodes in a linked list). Whoever has read/write permission to the *collection* can read/write any object in it without additional synchronization or dynamic checks, but the collection is merely a logical (or "ghost") mechanism, not a piece of runtime data.

Concretely, `GhostCell` introduces two types, `GhostCell<'id, T>` and `GhostToken<'id>`, with the former representing some shared data of type `T`, and the latter representing the permission to access that shared data. The key to the API is that we separate the cell and the permission to access it into two different types, albeit with a common *brand* parameter `'id` serving to connect them. This separation is what enables a single `GhostToken<'id>` to act as the permission governing a

[1]The actual definitions are more verbose: the types are also annotated with a lifetime `'arena`, *e.g.*, `Node<'arena, T>` and `type NodeRef<'arena, T> = &'arena RwLock<Node<'arena, T>>`. This is due to our use of *region-based memory management*; see §3.2. Here, we elide these lifetimes to simplify the presentation.

whole collection of `GhostCell<'id, _>`'s. Note that the brand `'id` here takes the concrete form of a Rust "lifetime". Lifetimes in Rust are usually used to track the scopes during which mutable and shared references are valid, but in the case of `GhostCell` and `GhostToken`, the brand lifetime `'id` is not actually used *as a lifetime*—rather, it merely serves as a static representative of the *collection* to which all the nodes of type `GhostCell<'id, _>` belong.

Returning to our motivating example of a doubly-linked list, using the `GhostCell` API we would parameterize the type of nodes in a list by a brand `'id` representing the particular linked list that the nodes belong to (*i.e.*, the node type becomes `Node<'id, T>`), and we would share references to nodes by wrapping them in a `GhostCell`, *i.e.*, as `&GhostCell<'id, Node<'id, T>>`. However, given just a `&GhostCell<'id, Node<'id, T>>`, one cannot do anything: to actually access the node, we also need to show ownership of the `GhostToken<'id>`, which can be viewed as a coarse-grained "proxy" permission to every node in the list with brand `'id`. In other words, ownership of `GhostToken<'id>` plays the role that in Rust is usually played by ownership of the `Container`: given a mutable reference *to the token*, we can mutate the doubly-linked list; given a shared reference, we can traverse it immutably in multiple threads at the same time.

Since `GhostToken<'id>` is a regular Rust type, we can compose it with existing Rust libraries for ownership management. For instance, normally in Rust, to provide coarse-grained sharing of a `Container` type, we could protect it with a *single* reader-writer lock (*i.e.*, as `RwLock<Container>`). To achieve the same for our doubly-linked list, we would use `RwLock<GhostToken<'id>>`. Note that we don't *need* to use a `RwLock` here. We could compose `GhostToken<'id>` with any synchronization mechanism we want—be it message-passing (*e.g.*, `sync::mpsc`), fork-join (*e.g.*, `rayon::join` [Stone and Matsakis 2017]), or something else. The `GhostCell` API is agnostic to which mechanism is used because it decouples permission transfer from the data being shared.

There is a tradeoff here, of course: with fine-grained permission tracking (à la per-node `RwLock`) it is possible for multiple threads to both read and write *different* nodes within a collection concurrently, whereas with `GhostCell`'s coarse-grained permission tracking, it is not. Furthermore, the `GhostCell` API does not allow the user to mutate one `GhostCell`-wrapped node in a collection while simultaneously holding *interior* pointers into other nodes from the same collection. Still, for the common case where these restrictions are acceptable—*e.g.*, the linked-list and graph operations we present in §3—`GhostCell` eliminates the significant space and time overhead of fine-grained permission tracking by avoiding the need to record and maintain extra state alongside each node.

The basic technique of *branded types*—using a static brand like `'id` as a representative of some dynamic, stateful data structure—is an old one, dating back at least to the work of Launchbury and Peyton Jones [1995] on the `ST` monad in Haskell. It involves using a combination of *phantom types* [Fluet and Pucella 2006] and *rank-2 polymorphism* [Kfoury and Wells 1994] in order to simulate a lightweight form of stateful, dependently-typed API, and has been explored more recently by Kiselyov and Shan [2007] in the context of OCaml and Haskell, and by Beingessner [2015] in the context of Rust. However, proving that the aforementioned APIs are sound is actually far from straightforward because their implementations typically make use of potentially *unsafe* operations (*e.g.*, unchecked array accesses or type casts) to avoid unnecessary dynamic checks. In prior work, it was claimed (and sometimes argued formally) that the unsafe operations used were nevertheless *safely encapsulated* by their strongly-typed APIs—*i.e.*, that the APIs were *observably* safe. However, we know of no prior work that formalizes the soundness of this approach for abstractions that rely crucially on the ownership-based (substructural) nature of the type system, as `GhostCell` does.

Our work makes two key contributions compared to the prior work on branded types:

(1) With `GhostCell`, we demonstrate a novel application of branded types to the problem of building safe and efficient data structures with internal sharing in Rust.

(2) We formally establish the soundness of our `GhostCell` API (as well as a variant of Beingessner [2015]'s "unchecked indexing" API) by extending RustBelt [Jung et al. 2018a], a machine-checked soundness proof for a core subset of Rust that is formalized in the Coq proof assistant. Adapting RustBelt to handle these branded-types APIs turned out to be quite subtle, seeing as they (ab)use Rust's lifetime mechanism for a purpose for which it was not intended. In particular, as we explain in §4, it required us to make a non-trivial change to the way that RustBelt models lifetime inclusion.

The rest of the paper is structured as follows:

- §2: As a warmup, we explain some basic concepts of Rust by example, and we illustrate the idea of branded types by reviewing Beingessner [2015]'s unchecked-indexing API.
- §3: We present our new `GhostCell` API, and show how to use it to make the safe implementation of a doubly-linked list data structure significantly more efficient. We also demonstrate the flexibility of the API with a graph traversal implementation.
- §4: We briefly review RustBelt [Jung et al. 2018a] and then explain at a high level the salient aspects of how we extended it to prove soundness of both the unchecked-indexing API and `GhostCell`. Our proof of soundness is fully formalized in Coq [Yanovski et al. 2021].
- §5: We empirically evaluate the performance and expressiveness of `GhostCell` compared to Rust's existing interior-mutable types.
- §6: We conclude by comparing with related work.

2 BRANDED TYPES IN RUST

In this section, we introduce the idea of *branded types* using a simple example drawn from prior work. In particular, we present the "unchecked indexing" API of Beingessner [2015], which shows how we can eliminate dynamic bounds checks when accessing a vector with an index, and instead achieve the in-bounds guarantee statically using Rust's type system. However, as we do not wish to assume prior knowledge of Rust on the part of the reader, we begin this section with a review of Rust's basic concepts of *ownership*, *borrowing*, and *lifetimes*.

2.1 Basic Rust with Vectors

First, let us see how to create a vector in Rust, along with some pointers (references) into it.

```
1  let mut vec: Vec<u8> = vec![0,1,2];
2  vec.push(3);
3  println!("{:?}", vec);          // Prints [0, 1, 2, 3]
4  let v0: &u8 = &vec[0];          // an immutable reference into `vec`
5  println!("{:?}", v0);           // Prints 0
6  let v1: &mut u8 = &mut vec[1];  // a mutable reference into `vec`
7  *v1 += 1;
8  println!("{:?}", vec);          // Prints [0, 2, 2, 3]
```

Here, we create an 8-bit unsigned integer vector (`Vec<u8>`) with 3 elements (line 1) and push a new element onto it (line 2). In line 4, we can create an interior pointer into the vector: in this case, we specifically create a *shared (immutable) reference* `v0` of type `&u8` to the first element `vec[0]` of `vec`. The shared reference `v0` only gives us *read permission* to `vec[0]`, which suffices for printing it (line 5). We can also create a *mutable reference* `v1`, now of type `&mut u8`, to the second element `vec[1]`, which allows us to mutate that element (line 7)—the effect is made visible in line 8.

Although this is a very simple example, we can already see Rust's ownership principle at work here. In particular, in line 1, after the creation, we *own* the vector `vec`. Then, in line 2, to mutate `vec`, `push` must *mutably borrow* the ownership of `vec` for the duration of the function call. This is

expressed concretely in the type of `push: fn(&mut Vec<u8>, u8) -> ()`.[2] Then, in line 4, the shared reference `v0` is created by *immutably borrowing* the ownership of `vec`. Finally, once (after line 5) the borrow by `v0` has ended, the mutable reference `v1` can be created by mutably borrowing `vec` again in line 6.

The durations of borrows are determined by *lifetimes*, which Rust's type system tries to infer automatically. The Rust "borrow checker" makes sure that conflicting borrows do not overlap, so as to uphold the AXM principle. To see this in action, let us look at another example, in this case one that is *rejected* by the type system.

```
1 let v0: &/* 'a */ u8 = &vec[0];
2 // REJECTED: cannot borrow `vec` as mutable because it is also borrowed as immutable
3 vec.push(4);                                                    Lifetime 'b
4 // REJECTED: cannot borrow `vec` as mutable because it is also borrowed as immutable
5 let v1: &/* 'c */ mut u8 = &mut vec[1];
6 *v1 += 1;                                                       Lifetime 'c
7 println!("{:?}", v0);                                          Lifetime 'a
```

Here, we annotate the implicitly inferred lifetimes for references to explain how Rust maintains AXM with lifetimes. First of all, Rust's type system infers that the immutable borrow of `vec` by `v0` has lifetime `'a`, spanning from line 1 to 7. Thus the type `&'a u8` of `v0` is implicitly tagged with the lifetime `'a`. Meanwhile, `push` needs to mutably borrow `vec` (line 3) for lifetime `'b`, which is the duration of the function call. The type system then rejects the call to `push` because `'a` and `'b` are overlapping and conflicting: `vec` cannot be borrowed immutably *and* mutably at the same time in line 3. Indeed this prevents a critical bug: if the code were allowed, `push` could have reallocated its internal array, leaving `v0` as a dangling pointer, in which case the type system would have missed the memory error caused by the access to `v0` in line 7. Similarly, the mutable borrow of `vec` by `v1` with lifetime `'c` also conflicts with the immutable borrow by `v0` with lifetime `'a`, and thus is also rejected. In this case, allowing the code would not actually lead to an AXM violation; it is conservatively rejected because the Rust compiler makes no attempt to reason about vector indices.

2.2 Phantom Lifetimes and Unchecked Indexing

To ensure memory safety, Rust performs dynamic bounds checks on vector accesses. For example, when the `i`th element of `vec` is accessed through `vec[i]`, a runtime check is performed to see if `i` is in bounds. If it is not, then the program will *panic* (*i.e.*, clean up the program state and abort the current thread). However, in many situations such runtime checks are unnecessary because we know that the indices are always in bounds. We would like to avoid these checks but still be guaranteed *statically* that such accesses are safe. In this section, we review a solution to this problem in Rust, based on Beingessner [2015]'s "unchecked indexing" API, which in turn is a close descendant of the idea of *branded types* [Kiselyov and Shan 2007].

2.2.1 An API for Branded Vectors. Let us enhance the vector type `Vec<T>` with an additional lifetime parameter `'id` to form a "branded" version `BrandedVec<'id, T>`. The key idea is that `'id` is not used as a normal lifetime which defines the duration of a borrow, but instead is used as a *brand* that *uniquely* identifies a runtime vector value. In other words, for every concrete brand `'id`, the type `BrandedVec<'id, T>` is a singleton type, inhabited only by *the* vector (call it `bvec`) of type `BrandedVec<'id, T>`. We then also have a type `BrandedIndex<'id>` describing integers that are known to be *valid* indices for `bvec`, with which we can access `bvec` without runtime checks. Since the lifetime `'id` serves here as a brand rather than a normal lifetime, we call it a *phantom* lifetime.

[2]Note that `vec.push(3)` here is syntactic sugar for `Vec::push(&mut vec, 3)`.

```
1  // A vector branded with a phantom lifetime `'id`, holding an underlying vector `Vec<T>`.
2  struct BrandedVec<'id, T> { ... }
3  // A branded index into a vector with the same brand `'id`.
4  struct BrandedIndex<'id> { ... }
5  impl<'id> BrandedVec<'id, T> {
6    // Turns a regular `inner: Vec<T>` into a branded vector of type `BrandedVec<'id, T>`,
7    // then passes it to a closure `f`.
8    pub fn new<R>(inner: Vec<T>, f: impl for<'a> FnOnce(BrandedVec<'a, T>) -> R) -> R { ... }
9    // Pushes to the vector and returns the index of the pushed element.
10   pub fn push(&mut self, val: T) -> BrandedIndex<'id> { ... }
11   // Bounds-checks an index; inbounds indices are returned as `BrandedIndex`.
12   pub fn get_index(&self, index: usize) -> Option<BrandedIndex<'id>> { ... }
13   // Given an index with the same brand, returns a reference to that element in
14   // the vector without performing any bounds checks.
15   pub fn get(&self, index: BrandedIndex<'id>) -> &T { ... }
16   pub fn get_mut(&mut self, index: BrandedIndex<'id>) -> &mut T { ... }
17 }
```

The API of branded vectors let us create a branded vector from a standard vector with `new` (line 8); then we can `push` a new element to the vector (line 10, where **`self`** refers to the vector itself). With `get_index` we can perform bounds-checking on an index, and in case of success we can now use that index with `get` and `get_mut` (line 15 and 16) without any further checks. We now go over these operations in more detail.

Pushing to a branded vector. When pushing to a vector `bvec: BrandedVec<'id, T>` with brand `'id`, we will receive in return a new index `i: BrandedIndex<'id>` to that pushed element with exactly the same brand `'id`, which allows us to make references to the element later. With `get_index` we can also turn a regular integer **`usize`** into a checked index—the method returns `None` when the desired index is out-of-bound. Crucially, the vector *does not* have a `pop` operation, because it can only be grown but not shrunk: monotonicity of the vector length is required to ensure that an index, once checked, will always remain valid.

Unchecked indexing. Once we have an index `i: BrandedIndex<'id>`, we can create references to the `i`th element of the branded vector `bvec` with the same brand `'id`. Whether we can create shared or mutable references depends on whether we can borrow `bvec` immutably or mutably, respectively. Most importantly, these functions do not have to perform any runtime checks, and they will always succeed (unlike with regular vectors, there is no `Option` in the return type). Furthermore, accessing a branded vector with indices created for another vector will be rejected at compile time by the type system, because it cannot unify the two different lifetimes (see below).

Creating a branded vector. The creation of a branded vector is a bit unusual. The function `new` consumes the ownership of some standard vector `inner: Vec<T>` to create a branded vector, but it also requires a closure `f` whose type universally quantifies over any lifetime `'a` (with the syntax **`for`**`<'a>`). This is an instance of *rank-2 polymorphism*: `new` is allowed to pick a fresh lifetime `'id` to brand the vector `inner`, and then it passes the branded vector to the client `f`. As `f` must work with any lifetime it receives from `new`, it knows nothing about `'id` except the fact that there *exists* such a lifetime, and thus must treat the brand `'id` opaquely.

To make this more concrete, let us look at how the API can be used in the following example.

```
1  let vec1: Vec<u8> = vec![10, 11];
2  let vec2: Vec<u8> = vec![20, 21];
3  BrandedVec::new(vec1, move |mut bvec1: BrandedVec<u8>| { // bvec1: BrandedVec<'id1, u8>
```

```
4      bvec1.push(12); let i1 = bvec1.push(13); // i1: BrandedIndex<'id1> is an index into bvec1
5      BrandedVec::new(vec2, move |mut bvec2: BrandedVec<u8>| { // bvec2: BrandedVec<'id2, u8>
6          let i2 = bvec2.push(22);            // i2: BrandedIndex<'id2> is an index into bvec2
7          *bvec2.get_mut(i2) -= 1;            // No bounds check! Updates to 21
8          println!("{:?}", bvec2.get(i2));    // No bounds check! Prints 21
9          println!("{:?}", bvec1.get(i1));    // No bounds check! Prints 13
10         // println!("{:?}", bvec2.get(i1)); // Rejected: i1 is not an index of bvec2
11     }); // end of `bvec2`'s closure
12 }); // end of `bvec1`'s closure
```

Here, we have two standard vectors `vec1` and `vec2` which we turn into two differently-branded vectors `bvec1` (with brand `'id1`) and `bvec2` (with brand `'id2`), by calling `new` in line 3 and 5, respectively. The client of `bvec1` is the closure from line 3 to 12, while that of `bvec2` is the closure from line 5 to 11. In line 6, we create `i2` as an `'id2`-branded index of `bvec2` by `push`-ing to it. This allows us to access `bvec2` through `i2` in line 7 and 8 without any bounds checks. Similarly, the index `i1` created in line 4 can also be used to access `bvec1` in line 9. Crucially, however, one cannot use the indices of one branded vector to access another: line 10 (if uncommented) would be rejected by the type system, because it cannot unify the brand `'id1` of `i1` (that is, the brand of `bvec1`) and the brand `'id2` of `bvec2`. This is needed for safety, for as we can see, `i1` is out of `bvec2`'s bounds.

2.2.2 Implementation of Branded Vectors. The implementation of the branded vector API needs to satisfy two conditions: (1) branding must be truly static, so that it does not incur runtime cost, and (2) brands must be unique, so that the type will reject the mixing of brands like in line 10 above. This leads us to the following implementation:

```
// Phantom lifetime type that can only be subtyped by exactly the same lifetime `'id`.
struct InvariantLifetime<'id>(PhantomData<*mut &'id ()>);
struct BrandedVec<'id, T> { inner: Vec<T>, _marker: InvariantLifetime<'id> }
struct BrandedIndex<'id> { idx: usize, _marker: InvariantLifetime<'id> }
```

The types `BrandedVec<'id, T>` and `BrandedIndex<'id>` are simply newtype wrappers around their underlying types (`Vec<T>` and `usize`, respectively), pairing them with the "phantom" type `InvariantLifetime<'id>`. The exact implementation of `InvariantLifetime<'id>` is not so important to our discussion; what is relevant is its interaction with *variance*. In general, the Rust compiler will automatically infer the variance of lifetime and type parameters such as `'id` and `T`, and here the type `InvariantLifetime<'id>` is carefully defined so that the compiler will infer the lifetime `'id` to be *invariant*. This ensures that we cannot change the brand of a `BrandedVec<'id, T>` or `BrandedIndex<'id>` via subtyping. In particular, it is guaranteed that indices of brand `'id` can only be used to access the vector with the *exact same* brand `'id`, as desired. In addition, through the use of the `PhantomData` constructor in its definition, `InvariantLifetime<'id>` will have size *zero*, and thus be compiled away and incur no runtime cost.

As mentioned in the introduction, implementing this API requires unsafe operations (code marked with an **`unsafe`** block) in the body of the `get` and `get_mut` functions, since they perform a vector access without bounds checks. This implies that we, as the developer of the library, have the obligation to prove that such accesses are actually safe under all possible interactions with well-typed clients. We will present this proof in §4.

3 GHOSTCELL: ZERO-OVERHEAD, THREAD-SAFE INTERIOR MUTABILITY

In the previous section, we have seen how branding (encoded via phantom lifetimes) can be used in Rust to achieve static guarantees and eliminate runtime checks for array indexing. This trick, of course, is already well-known in the functional programming world. However, branding can be

generalized further in combination with Rust's rich substructural type system to support *separating permissions from data*.

In this section, we introduce `GhostCell`, a library that statically enforces Rust's AXM discipline on shared data via a separately tracked "ghost token" (§3.1). As described in the introduction, `GhostCell` can be used to build data structures with internal sharing entirely in safe code without making compromises that might impact performance. We demonstrate this with the implementations of a doubly-linked list (§3.2) and a graph (§3.3).

3.1 The API of GhostCell

The key idea of `GhostCell` is to separate the *permission* to access a data structure from the *data* itself. As such, `GhostCell` introduces two types: `GhostToken<'id>` and `GhostCell<'id, T>`.

- `GhostCell<'id, T>` describes *data* of type `T` that is marked with the brand `'id`. When this "cell" type is shared, the data it contains can only be accessed by whoever holds the corresponding `GhostToken<'id>`.
- `GhostToken<'id>` represents the *permission* to access all data in shared `GhostCell`s marked with the brand `'id`.

The core API of `GhostCell` is as follows.

```
/// A single "branded" permission to access the data structure.
/// Implemented with a phantom-lifetime marker type.
struct GhostToken<'id> { _marker: InvariantLifetime<'id> }
/// Branded wrapper for the data structure's nodes, whose type is T.
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.
    fn new<R>(f: impl for<'new_id> FnOnce(GhostToken<'new_id>) -> R) -> R { ... }
}
impl<'id, T> GhostCell<'id, T> {
    /// Wraps some data T into a GhostCell with brand `'id`.
    fn new(value: T) -> Self { ... }
    /// Turns an owned GhostCell back into owned data.
    fn into_inner(self) -> T { ... }
    /// Turns a mutably borrowed GhostCell to and from mutably borrowed data.
    fn get_mut(&mut self) -> &mut T { ... }
    fn from_mut(t: &mut T) -> &mut Self { ... }

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

`GhostToken<'id>` only has one method: a constructor `GhostToken<'id>::new`, using the same pattern as we already saw with branded vectors. That is, `new` requires a client closure `f` that must be able to work with a `GhostToken` with an arbitrary brand `'new_id`. Thus `new` picks a fresh brand `'new_id`, creates the `GhostToken<'new_id>`, and then passes it on to `f`.

Moving on to `GhostCell`, the first four methods here are rather simple. They allow one to convert owned values 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 `GhostCell<'id, T>` has an identical representation to `T`, and that a *uniquely owned* `GhostCell<'id, T>` carries the same ownership as `T` and can be directly accessed without a `GhostToken`. This is common for interior-mutable types in Rust, *i.e.*, types that permit mutation of shared state: when the state is not actually shared, exclusive mutable access works as usual (*e.g.*, `Cell` also has these methods).

The more interesting part of the `GhostCell` API comprises the methods `borrow` and `borrow_mut`, which allow access to the content `T` of a `GhostCell<'id, T>`. Both of these methods only require a shared reference `&GhostCell<'id, T>`, which means this data can be subject to arbitrary aliasing. To ensure that AXM and hence safety are nevertheless maintained, a `GhostToken` with the same brand `'id` is required. The `GhostToken` determines which kind of access is granted to the content `T`: to return a `&T`, we require a `&GhostToken<'id>` (*i.e.*, the token can be shared with others); to return a `&mut T`, we require a `&mut GhostToken<'id>` (*i.e.*, the token must be exclusively borrowed). Thus, we delegate the management of borrowing the data `T` to the management of borrowing `GhostToken<'id>`. Effectively, this separates the *knowledge* about the data structure from the *permission* to access it: we track the knowledge via `&GhostCell<'id, T>` (which is a shared reference and hence freely duplicable) and the permission via `GhostToken<'id>` (which is not).

Thread safety. Thanks to Rust's borrow checker, `GhostCell` is naturally thread-safe. The borrow checker ensures that `GhostToken<'id>` is subject to the AXM discipline, and because of the types that we chose for `borrow` and `borrow_mut`, this implies that the contents of the `GhostCell` are also complying with AXM. Hence, each `GhostCell` is *either* mutated by one thread *or* accessed immutably by many threads at the same time.

As one would expect from a system where permissions are separate from data, multiple threads can coordinate accessing `GhostCell`s via ownership transfer of the corresponding `GhostToken`. For example (see §3.2.3), the `GhostToken` can be put into a lock; in that case the thread holding the lock has full access to all associated `GhostCell`s. Other ownership transfer idioms such as message passing are also supported.

Zero-cost abstraction. `GhostCell<'id, T>` is a *zero-cost abstraction*, meaning that it is merely a newtype wrapper (around `T`), whose dynamic representation is the same as `T`'s. Correspondingly, the methods `borrow` and `borrow_mut` do not actually do anything—they are just type coercions, which can be erased during compilation. (In fact, this is true for all `GhostCell` methods.) As for `GhostToken<'id>`, it is in fact a *zero-space abstraction*—*i.e.*, it is implemented as a zero-sized type with no runtime representation at all.

Of course, in case synchronization is actually required, `GhostCell` does not magically make that cost disappear. As with other Rust data structures, the client has to pick an appropriate synchronization mechanism (see §3.2.3). The point is that each client can make its own cost-expressiveness tradeoff; `GhostCell` itself does not add any unavoidable costs here.

Coarse-grained sharing. Designed as it is, `GhostCell` does not support fine-grained sharing: if we have multiple same-branded `GhostCell<'id,T>`'s for different nodes, we can only *mutably* borrow one `GhostCell` at a time, because we can only mutably borrow `GhostToken<'id>` once at a time. As we will see in the next section, we can still work with multiple nodes at the same time and even mutate all of them; the only restriction is that we cannot hold an *interior* pointer to one node while mutating another (since the type system has no way of ensuring that the "other" node is in fact not an alias to the node we are holding a pointer to).

3.2 Implementing a Doubly-Linked List with GhostCell

In this section we demonstrate how `GhostCell` can be used to build efficient doubly-linked lists in Rust. The goal is to build our doubly-linked list using only safe code or safely-encapsulated libraries, so that we can benefit from the safety guarantees of the type system. At the same time, we want to provide the kind of API that Rust programmers expect—*i.e.*, we aim to support interior pointers directly into the data structure as well as thread-safe read-only access.

The reason this is challenging is that doubly-linked lists inherently have internal sharing caused by a node being referenced from both its predecessor and its successor. The AXM discipline usually mandates that such shared nodes cannot be mutated. In the introduction, we have seen that existing interior-mutable types such as `RwLock` can be used to work around that limitation, but this comes at a steep performance cost due to per-node locking. With `GhostCell`, we can implement such a doubly-linked list without any unnecessary runtime checks.

3.2.1 Node Structure. A doubly-linked list data structure is rather simple: each `node` has some `data` and two pointers `prev` and `next` which point to the previous and the next nodes, respectively. For the case where there is nothing to point to (*i.e.*, at the ends of the list), the pointers can be *null*, which in Rust is represented with the `Option` type.

```
1  struct Node<'arena, 'id, T> {
2    data: T,
3    prev: Option<NodeRef<'arena, 'id, T>>,
4    next: Option<NodeRef<'arena, 'id, T>>,
5  }
6  type NodeRef<'arena, 'id, T> = &'arena GhostCell<'id, Node<'arena, 'id, T>>;
```

This should mostly match the usual expectations for what the definition of a doubly-linked list looks like—all the subtleties are concentrated in the definition of `NodeRef`. As the name suggests, this type describes references (pointers) to nodes, but this is also where we encode how the memory storing the node and the associated permissions are handled: we use an arena for memory management (which we will discuss immediately), so the reference has the lifetime `'arena` of the arena; and we use `GhostCell` to manage the permissions. `GhostCell` enables us to establish and exploit the AXM discipline for `Node` *even though* the nodes themselves are subject to unrestricted sharing: whoever holds the permission `GhostToken<'id>` has control over what happens to the list. The brand parameter `'id` and the arena lifetime have to be propagated everywhere, which is why they show up as parameters of both `Node` and `NodeRef`.

Region-based memory management with arenas. `TypedArena` [Fitzgerald and Sapin 2020] is a Rust implementation of *region-based memory management* [Grossman et al. 2002; Tofte et al. 2004]. Instead of individually managing the memory used for each `Node`, using an arena we can efficiently allocate nodes in a growing region of memory managed centrally by the arena. All these objects are deallocated together when the arena is destroyed. Hence all objects in the arena can use the lifetime `'arena` of the arena itself. This common lifetime makes arenas particularly suited for managing cyclic data structures such as our doubly-linked list.

The downside of arenas is that individual nodes cannot be deallocated; if that is required, reference-counting might be a more suitable form of memory management (but of course, that will incur some runtime overhead). We have chosen arenas here since they lead to simpler code, but `GhostCell` is agnostic about the underlying memory management mechanism and in our supplementary material we also provide a doubly-linked list based on `Arc`, a reference-counting implementation in the Rust standard library.

Note that in our `Node` implementation, the two lifetimes `'id` and `'arena` have distinct roles: `'id` is just a phantom brand which uniquely identifies a related set of nodes, while `'arena` is an "actual" lifetime of the arena which restricts the lifetimes of all objects—some of which can be differently-branded—allocated by the arena.

3.2.2 The API of Doubly-Linked Lists. Our simple lists with `GhostCell` have five **pub**lic methods:

(1) `new` wraps a value of type `T` to create an isolated node that is not yet linked to a list;
(2) `iterate` walks through the list starting from `node` and calls the function `f` with immutable interior pointers to all the `data` fields;
(3) `iter_mut` is the mutable counterpart of `iterate`;
(4) `insert_next` inserts `node2` immediately after `node1` in the list;
(5) `remove` disconnects `node` from its adjacent nodes, which are then directly connected.

```
impl<'arena, 'id, T> Node<'arena, 'id, T> {
  /// Create a new isolated node from T. Requires an arena.
  pub fn new(
    data: T, arena: &'arena TypedArena<Node<'arena, 'id, T>>
  ) -> NodeRef<'arena, 'id, T> {
    GhostCell::from_mut(arena.alloc(Self { data, prev: None, next: None }))
  }
  /// Traverse immutably.
  pub fn iterate(node: NodeRef<'arena, 'id, T>, token: &GhostToken<'id>, f: impl Fn(&T)) {
    let mut cur: Option<NodeRef<_>> = Some(node);
    while let Some(node) = cur {
      let node: &Node<_> = node.borrow(token); // immutably borrow `node` with `token`
      f(&node.data);
      cur = node.next;
    }
  }
  /// Traverse mutably.
  pub fn iter_mut(
    node: NodeRef<'arena, 'id, T>, token: &mut GhostToken<'id>, mut f: impl FnMut(&mut T)
  ) { ... }
  /// Insert `node2` right after `node1` in the list.
  pub fn insert_next(
    node1: NodeRef<'arena, 'id, T>, node2: NodeRef<'arena, 'id, T>, token: &mut GhostToken<'id>
  ) {
    // Step 1: remove node2 from its adjacent nodes.
    Self::remove(node2, token);
    // Step 2: let node1 and node1_old_next point to node2.
    let node1_old_next : Option<NodeRef<_>> = node1.borrow(token).next;
    if let Some(node1_old_next) = node1_old_next {
        node1_old_next.borrow_mut(token).prev = Some(node2);
    }
    node1.borrow_mut(token).next = Some(node2);
    // Step 3: link node2 to node1 and node1_old_next.
    let node2: &mut Node<_> = node2.borrow_mut(token);
    node2.prev = Some(node1); node2.next = node1_old_next;
  }
  /// Remove the links of this node to and from its adjacent nodes and connect those nodes.
  pub fn remove(node: NodeRef<'arena, 'id, T>, token: &mut GhostToken<'id>) { ... }
}
```

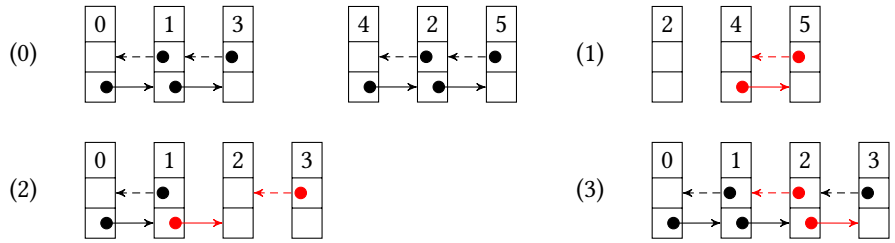

Fig. 1. A example run of `insert_next`.

Note how in `new`, we call `alloc` on the arena reference passed in by the client to perform allocation. This returns a `&'arena mut Node`, which we can convert to `&'arena mut GhostCell<Node>` using `from_mut`; that mutable reference is implicitly coerced to `&'arena GhostCell<Node>` which matches the return type.

Let us have a closer look at `iterate` and `insert_next`.

`iterate`. Since the function `f` only needs read-only access to the list, `iterate` only requires a shared reference to the `token`, which represents the permission to read the whole list. As such, this method can be called concurrently from multiple threads. `iterate` simply walks through every node in the list following the `next` field (line 14). For each node, it uses `token` to immutably borrow the actual node: `borrow` turns a `&GhostCell<Node>` into a `&Node` (line 12). With a shared reference to the node, we can get a shared reference to its `data` field, which is sufficient to call `f` (line 13).

`insert_next`. Since `insert_next` needs to modify the structure of the list, it requires a mutable reference to the `token`. We explain how the method works with an example run in Figure 1, where node 2 is inserted into the list (0,1,3), immediately after node 1. (The nodes are presented in the figure as vertically-stacked boxes, with names on top.) In this example run, `node1` is the node 1, while `node2` is the node 2.

- At the beginning (Fig. 1(0)), `node1` is linked to nodes 0 and 3, while `node2` is linked to nodes 4 and 5. In Fig. 1, we draw `next` and `prev` pointers as solid and dashed arrows, respectively.
- In step 1 (line 26, Fig. 1(1)), `node2` is removed from its adjacent nodes 4 and 5. In Fig. 1, we highlight updated pointers as red arrows.
- In step 2 (line 28-32, Fig. 1(2)), `node1`'s `next` field and node 3's `prev` field are updated to point to `node2`. In this step, we first take out the current value (3) of `node1.next` into `node1_old_next` to be used in step 3.
- In step 3 (line 34-35, Fig. 1(3)), `node2`'s fields are updated to point to `node1` and node 3.

In `insert_next`, we need to update several nodes in the list, but we only need to update one node at a time. For each node, we call `borrow_mut` using our `token` of type `&mut GhostToken<'id>` (lines 30, 32, and 34, and also the call of `remove` in line 26). This relinquishes access to that token until we stop using the reference returned by `borrow_mut`. Typically, we just mutate a single field, so `token` is immediately available again to mutate the next node. Remember that `borrow_mut` is just a type-changing identity function, so there is no performance cost to calling it many times.

Limitations. When compared with `LinkedList` from the standard library (implemented using **`unsafe`** code), the API surface of our list is a lot smaller. Most of the missing functionality could be added, but there is one key limitation that is not easily lifted: the API for mutable iteration over a `LinkedList` cannot be safely implemented even with `GhostCell`. This API provides mutable references to all nodes of the linked list *at the same time—i.e.*, with the same lifetime—but it is also

only sound for *acyclic* lists, since cycles would lead mutable iteration to generate multiple mutable references to the same node (which would violate AXM). In contrast, our list API permits the creation of cyclic lists through `insert_next`, but our iteration only provides mutable references to the elements of a list *one node at a time*. Building on `GhostCell`, this is the best we can do—indeed, it is unclear how to capture the non-local invariant of acyclicity (on which a safe implementation of mutable iteration depends) in the Rust type system. (For *read-only* iteration, we did implement an iterator providing references to all nodes at the same time. This works because, with shared references, aliasing due to cycles is not a problem.)

3.2.3 A Client of the Linked List. To show the kind of usage patterns that are enabled by separating permissions from data, we give a small program that performs concurrent accesses to the same list in two ways: unsynchronized read-only access, and mutable access synchronized with a lock.

```
1  GhostToken::new(|mut token| { // We first need a token. Note that the lifetime 'id is implicit.
2    // Allocate a list of size 50.
3    let arena = TypedArena::with_capacity(50); // pre-allocate some space (for efficiency).
4    let list = init_list(&arena, &mut token, 50);
5    // Share the token with two threads immutably, without synchronization.
6    rayon::join(
7      || Node::iterate(&list, &token, |n| print!("{:?}, ", n)),
8      || Node::iterate(&list, &token, |n| print!("{:?}, ", n)),
9    );
10   // Put the token into an RwLock to share it (mutably) across threads.
11   let lock_token : RwLock<GhostToken> = RwLock::new(token);
12   rayon::join( // fork two child threads
13     || { let token : &GhostToken = &lock_token.read().unwrap(); // acquire read lock
14          Node::iterate(&list, token, |n| print!("{:?}, ", n)); }, // print (old or new) content
15     || { let token : &mut GhostToken = &mut lock_token.write().unwrap(); // acquire write lock
16          Node::iter_mut(&list, token, |n| *n += 100); }, // add 100 to all nodes' data
17   );
18 });
```

After creating a list (`init_list` can be implemented with `insert_next`), we use `rayon::join` to spawn two threads and wait for their completion (lines 6 – 9). Both of these threads can iterate the list and print its content without any synchronization, since we can just pass `&GhostToken<'id>` to both of them without violating AXM. This is the kind of read-only thread-safety that (almost) all Rust data structures provide.

To share a data structure across threads *and mutate it*, synchronization is of course required to ensure absence of data races. If coarse-grained synchronization is sufficient, the usual approach is to wrap the entire data structure with some kind of lock type. For example, to share a `Vec<T>` across threads, we would use `RwLock<Vec<T>>`. Read-locks (resp. write-locks) could then be acquired to obtain a shared (resp. mutable) reference to the vector.

For doubly-linked lists, however, which type should we put into the `RwLock<_>`? One might think that we would end up with `RwLock<NodeRef>`, but `NodeRef` only represents a single node. Instead, the type that represents "the list as a whole" is `GhostToken`, since it captures the permission that is required to access and mutate the list. Thus, by putting the `GhostToken` into an `RwLock` (line 11), we can provide exclusive mutable or shared read-only access to the whole list to any number of threads, analogously to what `RwLock<Vec<T>>` would provide for a vector (line 12-16).

Considering that the `GhostToken` is just a phantom type holding no data, it may seem strange to put "nothing" into the `RwLock`. However, what really happens here is that `RwLock` wraps the *permission* instead of (as usual) the data. The nodes of the list may not seem like they are inside

the lock, but conceptually, they are. In that sense, the lifetime `'id` in `RwLock<GhostToken<'id>>` and `GhostCell<'id, T>` replaces the usual comments saying "this data is protected by that lock". Separating permissions from data lets us implement flexible locking disciplines such as protecting different fields of a type by different locks (using multiple brand lifetimes)—and the compiler can still statically enforce that we are using the locking discipline correctly!

Of course, using locks is just one way to perform synchronization. An alternative would be to use message-passing to transfer ownership of the token from one thread to another; we could use any Rust implementations of channels for this purpose. Again, no actual data is being transmitted, but the mere signal that a message of type `GhostToken<'id>` has been sent suffices to ensure that the next thread can pick up the token and access all the data it guards without further synchronization.

3.3 Implementing Graph Traversals with GhostCell

In this section, we briefly give a safe and efficient `GhostCell`-based implementation of the depth-first search (DFS) algorithm on a possibly cyclic graph data structure that supports interior pointers.

3.3.1 Node Structure. A graph node is a slight generalization of a linked list node: instead of having just one incoming and one outgoing edge, a graph node can have multiple incoming and outgoing edges. For simplicity, we store only the outgoing edges of each node, using an *adjacency list*.

```
struct Node<'arena, 'id, T> { data : T, uid : u32, edges : Vec<NodeRef<'arena, 'id, T>>, }
type NodeRef<'arena, 'id, T> = &'arena GhostCell<'id, Node<'arena, 'id, T>>;
```

Here, we also use arenas for memory management. The adjacency list is implemented as a vector of references to the outgoing nodes. The field `uid` is a fixed unique identifier for a node, which is used to efficiently implement DFS (see below). We also implement a `Graph` data structure whose main jobs are to create nodes with unique identifiers and to create the DFS data structure.

```
impl<'arena, 'id, T> Graph<'arena, 'id, T> {
  /// Create and assign a unique id to a node. Requires an arena.
  pub fn add_node(&mut self, data: T, arena: ...) -> NodeRef<'arena, 'id, T> {...}
  /// Create a DFS visitor data structure starting from the given root.
  pub fn dfs_visitor(&self, root: NodeRef<'arena, 'id, T>) -> DFSVisitor<'arena, 'id, T> { ... }
}
```

3.3.2 DFS Traversal. The DFS data structure manages the data for a traversal, which contains a stack to maintain the depth-first order, and a visit map to track already-visited nodes.

```
struct DFSVisitor<'arena, 'id, T> { stack: Vec<NodeRef<'arena, 'id, T>>, mark: FixedBitSet }
```

We implement the stack as a vector of references to nodes, and the visit map as a bitset [fixedbitset 2021]—which relies on the unique identifiers of nodes. We could have implemented the visit map as a hash-based set of references, but that is not as efficient as a bitset.

A DFS iteration that mutates the nodes' data can be implemented as follows:

```
impl<'arena, 'id, T> DFSVisitor<'arena, 'id, T> {
  /// Mark the node identified by uid as visited.
  fn visit(&mut self, uid: u32) { ... }
  /// Check if the node identified by uid is visited.
  fn is_visited(&self, uid: u32) -> bool { ... }
  /// A DFS visit that can mutate the data of each node (of type T).
  pub fn iter_mut(&mut self, token: &mut GhostToken<'id>, mut f: impl FnMut(&mut T)) {
    while let Some(node) = self.stack.pop() {
      let node_mut: &mut Node<_> = node.borrow_mut(token); let uid = node_mut.uid;
      if !self.is_visited(uid) {
```

```
11          self.visit(uid); f(&mut node_mut.data); // mark the node as visited and apply f
12          let node: &Node<_> = node.borrow(token);
13          for child in node.edges.iter() {        // push unvisited child nodes into the stack
14            if !self.is_visited(child.borrow(token).uid) { self.stack.push(child) }
15          }
16    } } }
17  }
```

The `iter_mut` function takes a mutable reference to a `DFSVisitor` (`&mut self`), a mutable reference to the `GhostToken`, and a function `f` that will be called on each node (in DFS order) and can mutate the node's data. The DFS implementation is straightforward: the function pops the stack for a `node` (line 8), marks it as visited, applies the function `f` to the node's data (line 11), then pushes all un-visited children of `node` to the stack (line 14), and repeats.

The most important detail is how the `GhostToken` is borrowed twice to perform the task. First, in order to mutate the node's data, we need to get a mutable reference to the node (`&mut Node`) by calling `borrow_mut` with `token` (line 9). Then, only after that mutable borrow has ended do we get back the `token` so that we can *immutably* borrow the outgoing `child` nodes *while holding a reference to the current* `node` (line 12 and line 14).[3] We need immutable (shared) borrows here because we are holding references to two nodes (the current node and a child node) simultaneously and, in case there is a self-loop, one of these child nodes could very well be the current node itself.

4 PROVING SOUNDNESS OF BRANDED-TYPES APIS IN RUST

In this section, we explain how to prove safety of branded types in general, and `GhostCell` specifically. Our soundness proof is done within RustBelt [Jung et al. 2018a], a machine-checked proof of safety for a significant subset of Rust called λ_{Rust}. RustBelt formally proves the soundness of λ_{Rust}'s type system, and provides a framework to verify libraries that use **`unsafe`** features, such as `BrandedVec` and `GhostCell`. In §4.1, we review the general structure of RustBelt, and in particular the basic requirements for the verification of **`unsafe`** libraries. Then, in §4.2, we present a new technique for associating lifetimes with logical (ghost) state, which plays a crucial role in the soundness proofs of branded-types APIs, and we explain how it required us to make some changes to the modeling of lifetime inclusion in RustBelt. In §4.3, as a warmup, we sketch the soundness proof for a simplified version of `BrandedVec`, before proceeding in §4.4 with the one for `GhostCell`.

4.1 The Semantic Approach of RustBelt

Rust follows an extensible approach to safety: it enforces a sound ownership-based type system with the AXM principle, but when that type system becomes too restrictive, developers can opt out and use **`unsafe`** operations such as unchecked indexing and type casts in the implementation of their libraries. It is then the developers' obligation to show that, despite their use of **`unsafe`** features, their libraries are actually *observably safe*, in the sense that they never exhibit any unsafe/undefined behaviors (such as use-after-free or data races) when used by safe code.

The RustBelt work [Jung et al. 2018a] formalizes this approach with a *semantic model*. This model defines a *safety contract* that each function needs to uphold: if the function is called on well-formed arguments, it must be well-behaved (*i.e.*, not cause any unsafety) and return a well-formed result. The notion of "well-formedness" is given by the *semantic interpretation* of types, which (roughly speaking) defines the representation invariant of the type. For safe functions, the safety contract is upheld by construction, as established by the type safety proof of RustBelt. However, libraries that internally use **`unsafe`** features need to have their safety contract proven manually. This is done

[3]Note that in order to end the mutable borrow here within a conditional branch, we rely on Rust's "non-lexical lifetimes" [Matsakis 2016].

in two steps: (1) one first picks a semantic interpretation for the library's types; and then (2) one proves that the implementation satisfies the library-specific safety contract generated by the API.

We will later consider this contract in more detail for `BrandedVec` and `GhostCell`, but first we briefly review RustBelt's semantic interpretation of types, and we explain how that interpretation needs to change to support branding.

Iris and the lifetime logic. The first key design decision of a semantic model is to pick the logic that is used to express the safety contract. This logic determines the basic vocabulary of abstractions that are available in the semantic model. The usual approach is to interpret a type as the set of values that inhabit the type, which works great for simple languages but falls short for languages like Rust: Rust types denote *ownership* of resources such as memory, so it is beneficial to pick a logic that comes with built-in support for reasoning about ownership, such as a *separation logic* [Reynolds 2002]. Toward this end, RustBelt depends on Iris [Jung et al. 2015, 2018b], a framework for concurrent separation logic with strong support for interactive proofs in Coq [Krebbers et al. 2017, 2018].

While Iris makes it easy to model the ownership that is implicit in Rust's types, a semantic model also has to account for the concepts of *borrowing* and *lifetimes*. RustBelt overcomes this problem by introducing the *lifetime logic*. Here, we can only give an extremely high-level summary of the lifetime logic; we refer the reader to the RustBelt paper [Jung et al. 2018a] and Jung's PhD thesis [Jung 2020] for further details.

The key idea of the lifetime logic is to extend separation logic with a proposition $&^{\kappa}_{\mathbf{full}} P$, called a *full borrow* of P, which expresses *temporary ownership* of P for lifetime κ. This is most clearly reflected in the following (slightly simplified) central proof rule to create new borrows:

$$P \quad \Rrightarrow \quad \underbrace{&^{\kappa}_{\mathbf{full}} P}_{\text{ownership } \textit{during } \kappa} \quad * \underbrace{\left([\dagger\kappa] \Rrightarrow P\right)}_{\text{ownership } \textit{after } \kappa}$$

This rule says that whenever we own some proposition P, we can split P into two pieces: a full borrow $&^{\kappa}_{\mathbf{full}} P$, which grants access to P while the lifetime κ is ongoing; and an *inheritance* $[\dagger\kappa] \Rrightarrow P$, which grants access to P after κ has ended.[4] (The $\Rrightarrow$ connective is basically a fancy version of the magic wand, *i.e.*, a form of implication suitable for separation logic.) Even though both of these pieces are about P (*i.e.*, they overlap *in space*), we can treat them as separate propositions because they are disjoint *in time*: at any given point in time, only one of the two propositions can be used to access P. This justifies the use of a separating conjunction between the two pieces.

The other ingredient of the lifetime logic is the mechanism of *lifetime tokens*: $[\kappa]_q$ expresses ownership of some fraction q of lifetime κ. Ownership of the lifetime token reflects that the lifetime κ is currently still ongoing. This is required to get the P out of $&^{\kappa}_{\mathbf{full}} P$. *Ending* a lifetime requires ownership of the full token $[\kappa]_1$. The lifetime token then turns into $[\dagger\kappa]$, which (as we already saw above) serves as a witness that the lifetime has indeed ended.

RustBelt's semantic interpretation of types. As mentioned before, the semantic interpretation of a type in RustBelt is an Iris predicate. However, it is not simply a predicate on values: (1) the predicate is slightly more involved, and (2) we need two predicates, not just one. Regarding (1), instead of just considering individual values, Rust types describe data as it is laid out in memory, potentially spanning multiple locations. The predicate thus considers a *list* of values.[5] Regarding (2), *sharing* in Rust is extremely flexible: while most types treat shared data as read-only (following

[4] To simplify the presentation, we omit the "later" modality $\triangleright$ and the masks $\mathcal{E}$, which are related to handling step-indexing and to avoid reentrancy issues with Iris's invariants, respectively. For further details, see [Jung et al. 2018b; Jung 2020].

[5] The full model also requires a *thread ID* to reflect Rust's ability to reason about whether a type is safe to send to another thread, or safe to share across thread boundaries. To simplify the presentation here, we omit the thread IDs.

AXM), types that involve interior mutability actually use arbitrarily complex sharing protocols to ensure soundness—from `Cell`, which ties the data to a particular thread, to `RwLock`, which implements run-time checks in a fine-grained concurrent protocol. In order to support all these types, RustBelt considers *two* predicates for each type: the *ownership predicate* governs values that are uniquely owned, and the *sharing predicate* governs values that are shared.

$$[\![\tau]\!].\mathrm{own} \in \mathit{List}(\mathit{Val}) \to \mathit{iProp} \qquad [\![\tau]\!].\mathrm{shr} \in \mathit{Lft} \times \mathit{Loc} \to \mathit{iProp}$$

Here, *iProp* is the type of Iris propositions. The ownership predicate, $[\![\tau]\!].\mathrm{own}(\overline{v})$, defines whether the list of values $\overline{v}$ is a valid *owned* inhabitant of τ. It is also used to define the semantic interpretation of mutable references (`&mut T`). The sharing predicate, $[\![\tau]\!].\mathrm{shr}(\kappa, \ell)$, defines whether the location ℓ is a valid *shared reference* to an inhabitant of τ (`&T`), borrowed for the lifetime κ.

4.2 Key Idea: Associating Lifetimes with State

The key idea underlying the soundness proof of both `BrandedVec` and `GhostCell` is to *associate the brand phantom lifetime with some state*, such as the current length of the vector or the current state of the `GhostToken` (*i.e.*, whether it is currently being used to access some `GhostCell`). In this subsection we will discuss how this idea is reflected in our proofs, and how RustBelt had to be adjusted so that the idea could be used in the soundness proof of a Rust library.

Before we can explain how a lifetime can be associated with state, we first need to explain how state is handled in Iris. Iris has a very general built-in notion of *ghost state*, which is a purely logical concept that is not tied to any actual program variables. As such, ghost state is governed by a set of proof rules; we will see some examples of that shortly. Iris then uses *invariants* to tie this ghost state to observably physical state such as the length of a vector; in RustBelt proofs, the semantic interpretation of a type typically plays the role of that invariant. The details of how that works do not matter yet; the key point is that given established Iris techniques, the one new ingredient we need for proofs involving branding is to associate lifetimes with *ghost state*. As an example, we consider the ghost state that will be required for the verification of `BrandedVec`.

Monotone counters. At the heart of the `BrandedVec` proof lies the fact that the length of the vector is monotonically increasing, and hence any `BrandedIndex` is a lower bound on whatever the current length is, even if that length has changed since the index has been bounds-checked. A monotone counter can be constructed as ghost state in Iris with the following properties:

$$\textsc{MonoInit}\quad \mathsf{True} \Rrightarrow \exists \gamma.\, \mathsf{MonoVal}(\gamma, n) \qquad \textsc{MonoIsLb}\quad \mathsf{MonoVal}(\gamma, n) * \mathsf{MonoLb}(\gamma, m) \mathrel{-\!\!*} m \le n$$

$$\textsc{MonoUpdate}\quad \frac{n \le m}{\mathsf{MonoVal}(\gamma, n) \Rrightarrow \mathsf{MonoVal}(\gamma, m)} \qquad \textsc{MonoMakeLb}\quad \frac{m \le n}{\mathsf{MonoVal}(\gamma, n) \mathrel{-\!\!*} \mathsf{MonoLb}(\gamma, m)}$$

Here, $\mathsf{MonoVal}(\gamma, n)$ expresses *ownership* of the counter named γ, which implies exact knowledge of its current value n and the ability to increase the counter via MonoUpdate.[6] A counter can be allocated with MonoInit, which gives a $\mathsf{MonoVal}(\gamma, n)$ with a fresh name γ. $\mathsf{MonoLb}(\gamma, n)$ expresses *knowledge* that the current value of the counter is *at least* n, as expressed by MonoIsLb.[7] Since MonoLb is pure knowledge, it can be freely duplicated (we omitted this proof rule for space reasons).

[6] The $\Rrightarrow$ in these two rules indicates that applying these lemmas has the side-effect of *updating* the ghost state—but for the purpose of our high-level tour of the soundness proof, the difference between $\Rrightarrow$ and $\mathrel{-\!\!*}$ does not matter much.

[7] It may seem like this rule *consumes* ownership of MonoVal, but in fact that is not the case—in Iris, when the right-hand side of a magic wand consists of statements like $m \le n$ and $\mathsf{MonoLb}(\gamma, n)$ that are freely duplicable (in Iris lingo, *persistent*), it can be applied without consuming the left-hand side [Jung et al. 2018b].

The rule MonoMakeLb takes a "snapshot" of the current counter value and produces a corresponding MonoLb; this snapshot may be weakened to any smaller value.

Notice how in these proof rules, the *name* γ of the counter plays an important role: rules like MonoIsLb only work when a MonoVal and a MonoLb *for the same counter* are brought together. When verifying branded types, what we need is a way for the phantom lifetime `'id` to take the role of that γ—then we can have a MonoVal in `BrandedVec<'id>` reflecting the length, and a MonoLb in `BrandedIndex<'id>`, and we would know that whenever the brand matches, both are actually talking about the same underlying piece of ghost state. In other words, we need a way to *map a phantom lifetime to a ghost name.*

Associating lifetimes with ghost names. To achieve this, we have extended the lifetime logic with a new proposition $\mathsf{GhostLft}(\kappa, \gamma)$, a (freely duplicable) witness that the lifetime κ is associated with some *unique* ghost name γ, where uniqueness is expressed by this key rule:

$$\mathsf{GhostLft}(\kappa, \gamma_1) * \mathsf{GhostLft}(\kappa, \gamma_2) \twoheadrightarrow \gamma_1 = \gamma_2 \qquad \text{(GhostLftLookup)}$$

One way to think about this is to imagine a kind of "immutable heap" where lifetimes serve as locations, and at each location we can store the corresponding ghost name. GhostLft then serves as the equivalent of the usual points-to assertion of separation logic for this particular heap.

With GhostLft, we can now define syntactic sugar for a "monotone counter identified by a lifetime" via $\mathsf{MonoVal}(\kappa_{\mathsf{id}}, m) := \exists \gamma.\, \mathsf{GhostLft}(\kappa_{\mathsf{id}}, \gamma) * \mathsf{MonoVal}(\gamma, m)$, and similarly for MonoLb. Thanks to GhostLftLookup, most of the proof rules above still hold when using κ_{id} as an identifier instead of a ghost name γ. Only the initialization rule MonoInit changes; it has to be tied to the allocation of the brand lifetime κ_{id} itself.

To summarize, GhostLft and GhostLftLookup reflect the core power of branding: if we can make sure that two types with the same brand lifetime `'id` work on shared ghost state, we can use Iris to establish arbitrary kinds of coordination between these two types.

However, before we sketch in §4.3 and §4.4 how this general approach is applied to the concrete cases of `BrandedVec` and `GhostCell`, let us first discuss the technical challenges we had to overcome in order to add support for GhostLft in RustBelt:

(1) Previously, in RustBelt, the lifetime logic made no guarantee that a newly allocated lifetime is *fresh*—a property needed to perform the association with a ghost name in `BrandedVec::new` and `GhostToken::new`. To minimize changes to the lifetime logic, we develop a "reservation" system which separates the table that maps lifetimes κ_{id} to their associated γ from the rest of the lifetime logic, while still making sure that the newly allocated lifetime is fresh in that table. However, for space reasons, we elide further discussion of this reservation system here and focus instead on the second, more conceptually interesting challenge.
(2) In the original RustBelt, lifetimes were treated *extensionally* (see below). However, the assertion $\mathsf{GhostLft}(\kappa_{\mathsf{id}}, \gamma)$ is inherently tied to the *syntactic* representation of the lifetime κ_{id}, so we had to find a way to make RustBelt less extensional without unduly affecting the type system or soundness proofs of other libraries. We will now explain this point in detail.

The extensional treatment of lifetimes. To explain the extensional treatment of lifetimes in RustBelt, we consider the *lifetime inclusion* relation, $\kappa_1 \sqsubseteq \kappa_2$, which says that κ_1 will end before κ_2 does.[8] Lifetime inclusion is crucial for subtyping; for example, reference types permit replacing longer lifetimes by shorter ones:

$$\kappa_1 \sqsubseteq \kappa_2 \vdash \&^{\kappa_2}_{\mathbf{mut}}\, \tau \sqsubseteq \&^{\kappa_1}_{\mathbf{mut}}\, \tau$$

[8]This is RustBelt terminology. In Rust lingo, one typically states this in reverse: "$\kappa_1 \sqsubseteq \kappa_2$" means "$\kappa_2$ *outlives* κ_1".

The symmetric closure of lifetime inclusion induces an equivalence relation, defined as follows: $\kappa_1 \equiv \kappa_2 := \kappa_1 \sqsubseteq \kappa_2 \land \kappa_2 \sqsubseteq \kappa_1$. RustBelt requires the subtyping of *all* types to respect this lifetime equivalence relation $\equiv$. For example, for `BrandedIndex`, we have to prove the subtyping property:[9]

$$\kappa_1 \equiv \kappa_2 \vdash \mathbf{BrandedIndex}(\kappa_1) \sqsubseteq \mathbf{BrandedIndex}(\kappa_2) \qquad \text{(BrandedIndexSub)}$$

However, **BrandedIndex** cannot satisfy BrandedIndexSub! This is caused by the use of GhostLft: $\kappa_1 \equiv \kappa_2$ does not imply $\mathrm{GhostLft}(\kappa_1, \gamma) \Rightarrow \mathrm{GhostLft}(\kappa_2, \gamma)$, because lifetime inclusion is not antisymmetric, *i.e.*, $\kappa_1 \equiv \kappa_2 \not\Rightarrow \kappa_1 = \kappa_2$.

This lack of antisymmetry arises because lifetime inclusion ($\sqsubseteq$) is defined by RustBelt *semantically*: κ_1 is included in κ_2 if one can always "trade" a token for κ_1 (witnessing that this lifetime is still alive) for a token for κ_2. This "trade" may use Iris invariants and arbitrary ghost state protocols, so it is "dynamic" in the sense that as the proof state evolves, the relation can grow to relate more lifetimes with one another. The dynamic nature of semantic inclusion plays a crucial role in the soundness proof of types such as `RwLock`.

Syntactic lifetime inclusion. Since GhostLft cannot be compatible with $\equiv$, we have to change RustBelt's notion of lifetime inclusion so that BrandedIndexSub holds. To this end, we extend RustBelt to provide a second, *syntactic* form of lifetime inclusion, $\kappa \sqsubseteq_{syn} \kappa'$. Syntactic lifetime inclusion is defined in terms of the lifetime intersection operation in RustBelt as follows:

$$\kappa \sqsubseteq_{syn} \kappa' := \exists \kappa''.\, \kappa'' \sqcap \kappa' = \kappa$$

In other words, κ is syntactically included in κ' if it can be written as the intersection of κ' with some other lifetime κ''. Unlike semantic inclusion, syntactic inclusion is *static*: its structure cannot be changed even as the proof state evolves.[10] And crucially, it enjoys antisymmetry. BrandedIndexSub may thus assume proper equality $\kappa_1 = \kappa_2$, making the proof trivial.

To use syntactic lifetime inclusion for subtyping, we made the following changes to RustBelt:

- We adjusted the semantic interpretation of *external lifetime contexts*, a component of the typing judgments that expresses assumptions about how the currently available lifetimes are included in each other. In RustBelt, this was modeled with semantic inclusion; we had to change this to use syntactic inclusion instead.
- This changed interpretation broke the proof of one typing rule called "lifetime equalization". We discuss the adaptation of that rule below.
- Furthermore, we had to redo the proof of a key lemma involved in calling functions and ensuring that their assumptions about lifetime parameters do indeed hold. The old proof exploited *semantic* lifetime inclusion in external lifetime contexts in a crucial step. The proof was fixed by adjusting the semantic interpretation of the *local lifetime context*, which tracks the lifetimes that were started inside the local function and can be ended using the appropriate typing rule. This change affected the semantic interpretation of all judgments that have a local lifetime context, but not in any fundamental way.

Except for this "lifetime equalization" rule, the type system is unaffected by these changes. The remaining proofs (for the type system and the previously verified unsafe libraries) required barely any updates. In particular, the changes to the semantic model did not affect soundness proofs that internally relied on semantic lifetime inclusion, such as the one of `RwLock`: those lifetime inclusions never end up in the syntactic lifetime contexts of the type system, so the soundness proofs can proceed as before.

[9] We use **BrandedIndex** as notation for the mathematical model of the Rust type `BrandedIndex` in RustBelt.
[10] Technically, semantic inclusion is an Iris proposition *iProp*, while syntactic inclusion is a Coq proposition *Prop*.

Lifetime equalization. The one typing rule that no longer holds under the new semantic model is the "lifetime equalization" rule [Jung 2020, §9.4] (not presented in the RustBelt paper). This rule makes two lifetimes *equal* by applying semantic lifetime inclusion to give up the "right to end" one of the involved lifetimes. This is fundamentally incompatible with making lifetime inclusion syntactic, and indeed a slight variant of this rule can be used to typecheck a program that breaks branding (*i.e.*, under this adjusted type system, branding would be unsound). The lifetime equalization rule was originally added to typecheck Rust programs that the official Rust compiler does not actually accept but that the Rust developers hope to accept in the future, with the next-generation borrow checker *Polonius* [Matsakis 2018]. We replaced this rule by a weaker variant that is compatible with syntactic lifetime inclusion and still sufficiently strong to typecheck the example motivating the original rule (see the `README.md` in our supplementary material for a reference to the new rule).

4.3 Soundness of a Simplified Variant of BrandedVec

We are now ready to show soundness of (a simplified version of) `BrandedVec` by modeling that type in RustBelt and verifying that it satisfies the safety contract determined by its public API.

To focus attention on the interesting part of the verification, we reduce the `BrandedVec` API to its core functionality: ensuring that all values of type `BrandedIndex` are in-bounds of their associated vector. Other aspects of `BrandedVec`, in particular actually storing the elements of the vector somewhere, are completely orthogonal to the use of branding to avoid a dynamic bounds-check, and have thus been omitted from our formalization. (For example, the type parameter `T` for the element type disappears from our model.) Thus, in our core model, a `BrandedVec` is simply an integer representing the current length of the vector, whose value is incremented by every `push`. Correspondingly, the indexing methods `get` and `get_mut` simply get stuck if the given index is out-of-bounds. Our safety proof then establishes that these stuck paths are unreachable, by showing that a `BrandedIndex<'id>` is always in-bounds of the corresponding `BrandedVec<'id>`.

The interpretations of the types. The semantic interpretation of *owning* a `BrandedVec<'id>` and a `BrandedIndex<'id>` crucially relies on $\mathrm{MonoVal}(\kappa_{\mathrm{id}}, n)$ and $\mathrm{MonoLb}(\kappa_{\mathrm{id}}, n)$, defined in §4.2:

$$[\![\mathbf{BrandedVec}(\kappa_{\mathrm{id}})]\!].\mathrm{own}(\overline{v}) := \exists n.\, \overline{v} = [n] * \mathrm{MonoVal}(\kappa_{\mathrm{id}}, n)$$
$$[\![\mathbf{BrandedIndex}(\kappa_{\mathrm{id}})]\!].\mathrm{own}(\overline{v}) := \exists m.\, \overline{v} = [m] * \mathrm{MonoLb}(\kappa_{\mathrm{id}}, m+1)$$

In other words, owning a `BrandedVec` corresponds to ownership of a counter that reflects its length,[11] whereas owning a `BrandedIndex` implies owning a witness that the length of the corresponding `BrandedVec` is *at least* $m+1$—*i.e.*, m is an in-bounds index.

Soundness of `get_mut`. We are now ready to verify correctness of `get_mut`. As we have simplified the vector implementation to just storing its length, the type signature looks as follows:

```
fn get_mut<'id>(vec: &mut BrandedVec<'id>, idx: BrandedIndex<'id>)
```

In our simplified model, `get_mut` causes a stuck state (by dereferencing a bad pointer) if `idx` is not in-bounds for `vec`.

To prove that `get_mut` is safe—*i.e.*, it does not get stuck—the contract of `get_mut` gives us ownership of the (borrowed) vector $[\![\&^{\kappa}_{\mathbf{mut}}\, \mathbf{BrandedVec}(\kappa_{\mathrm{id}})]\!].\mathrm{own}(\overline{v}_{\mathrm{vec}})$ and ownership of the index $[\![\mathbf{BrandedIndex}(\kappa_{\mathrm{id}})]\!].\mathrm{own}(\overline{v}_{\mathrm{idx}})$. Note that the vector argument is wrapped in a mutable reference. The semantic model of mutable references in RustBelt is defined as follows:

$$[\![\&^{\kappa}_{\mathbf{mut}}\, \tau]\!].\mathrm{own}(\overline{v}) := \exists \ell.\, \overline{v} = [\ell] * \&^{\kappa}_{\mathbf{full}}\big(\exists \overline{v}'.\, \ell \mapsto \overline{v}' * [\![\tau]\!].\mathrm{own}(\overline{v}')\big)$$

[11] To model the original `BrandedVec`, we would of course also need to account for ownership of the underlying data buffer.

That is, a mutable reference is a location ℓ, and the associated ownership is described using the $\&^{\kappa}_{\mathbf{full}}$ connective of the lifetime logic: with the mutable reference comes *temporary* ownership of the data $\overline{v}'$ that ℓ points to. Ownership is temporary in the sense that it has been borrowed for lifetime κ. However, we know that the associated lifetime κ is alive (RustBelt provides an appropriate lifetime token), so we can access the borrowed resources for the duration of our proof. In our case, this means we obtain ownership of the `BrandedVec`, *i.e.*, we obtain $[\![\mathbf{BrandedVec}(\kappa_{\mathsf{id}})]\!].\mathrm{own}(\overline{v}'_{\mathsf{vec}})$.

The remaining reasoning is summarized by the following lemma:

$$[\![\mathbf{BrandedVec}(\kappa_{\mathsf{id}})]\!].\mathrm{own}(\overline{v}'_{\mathsf{vec}}) * [\![\mathbf{BrandedIndex}(\kappa_{\mathsf{id}})]\!].\mathrm{own}(\overline{v}_{\mathsf{idx}}) \mathrel{-\!\!*} \exists n, m.\, \overline{v}'_{\mathsf{vec}} = [n] * \overline{v}_{\mathsf{idx}} = [m] * m < n$$

The proof of this lemma follows directly from GhostLftLookup and MonoIsLb, and finally exploiting that $m + 1 \le n$ is equivalent to $m < n$. From this lemma, it follows that `get_mut` indeed cannot get stuck, which completes the proof.

Shared references and shared borrows. As a second example, we want to sketch the soundness proof of `get_index`. This function takes as argument a shared reference `&BrandedVec<'id>`, so before we can talk about its proof we need to discuss the *sharing predicate* of `BrandedVec`. As already explained, the sharing predicate is how RustBelt accounts for the fact that Rust types have a lot of freedom in how to justify correctness of the shared interactions offered by the type. However, a few common patterns are enough to account for the sharing predicate of most Rust types, so RustBelt provides some reusable reasoning principles for this purpose. For `BrandedVec`, the two relevant kinds of sharing are captured by *fractured borrows* and *atomic borrows*.

The proposition $\&^{\kappa}_{\mathbf{frac}}\, \lambda q.\, \Phi(q)$, called a *fractured borrow*, captures the idea that $\Phi(1)$ is borrowed for some lifetime κ, but there are many parties that have access to this borrow, so each party can only get some *fraction* $\Phi(q)$ of the overall resources. For this to work, we require that summing fractions corresponds to separation conjunction, *i.e.*, $\Phi(q_1 + q_2) \Leftrightarrow \Phi(q_1) * \Phi(q_2)$.

Like fractured borrows, an *atomic borrow* $\&^{\kappa}_{\mathbf{at}}\, P$ is shared between many parties; unlike fractured borrows, each party can get full access to the entire P—but only for a single step of computation; nobody is allowed to "hold on" to P for longer periods of time.

Sharing a `BrandedVec`. The sharing predicate of `BrandedVec` is now defined as follows:

$$[\![\mathbf{BrandedVec}(\kappa_{\mathsf{id}})]\!].\mathrm{shr}(\kappa, \ell) := \exists n.\, \left(\&^{\kappa}_{\mathbf{frac}}\, \lambda q.\, \ell \overset{q}{\mapsto} n\right) * \&^{\kappa}_{\mathbf{at}}\, \mathsf{MonoVal}(\kappa_{\mathsf{id}}, n)$$

Here, we use a fractured borrow to manage ownership of the underlying memory where the length of the vector is stored. The borrow gives access to some fraction of the ownership of that memory location, which is good enough to perform reads, reflecting that the vector is indeed immutable for as long as the sharing lasts. Meanwhile, the atomic borrow is used to provide everyone with instantaneous access to the counter via $\mathsf{MonoVal}(\kappa_{\mathsf{id}}, n)$. The counter value n cannot be changed (notice how it is existentially quantified *outside* the atomic borrow and tied to the value stored at ℓ), but MonoMakeLb will let us take a snapshot of the current value, which is sufficient to create a new `BrandedIndex`, as we will see next.

Soundness of `get_index`. The core of this proof is reflected by the following key lemma:

$$[\kappa]_q * [\![\mathbf{BrandedVec}(\kappa_{\mathsf{id}})]\!].\mathrm{shr}(\kappa, \ell_{\mathsf{vec}}) \Rrightarrow \quad \text{(GetIndexCore)}$$
$$\exists n, q'.\, \ell_{\mathsf{vec}} \overset{q'}{\mapsto} n * \left(\forall m.\, m < n \Rrightarrow [\![\mathbf{BrandedIndex}(\kappa_{\mathsf{id}})]\!].\mathrm{own}([m])\right) *$$
$$\left(\ell_{\mathsf{vec}} \overset{q'}{\mapsto} n \Rrightarrow [\kappa]_q * [\![\mathbf{BrandedVec}(\kappa_{\mathsf{id}})]\!].\mathrm{shr}(\kappa, \ell_{\mathsf{vec}})\right)$$

This lemma is a lot to take in, so let us go over it slowly. First, GetIndexCore takes the assumptions that RustBelt's interpretation of the type of `get_index` gives us: a shared instance of `BrandedVec`, and some fraction of the corresponding lifetime token κ (witnessing that κ is alive).

Then GetIndexCore gives us ownership of *some fraction* q' of the memory ℓ_{vec} where the counter's current length n is stored, which allows us to read and compare the length with the input `idx`. This part of GetIndexCore is proven by unfolding the sharing predicate and opening the fractured borrow.

GetIndexCore also gives us $\forall m.\, m < n \Rrightarrow [\![\mathbf{BrandedIndex}(\kappa_{\text{id}})]\!].\text{own}([m])$, which says that for any in-bounds m (strictly less than n), we can get a well-formed `BrandedIndex<'id>` for m. So if the input `idx` is indeed in-bounds, we can use this to construct the corresponding `BrandedIndex` as the return value. This part of GetIndexCore is proven by accessing the atomic borrow, using MonoMakeLb to take a snapshot of the length, and immediately closing the borrow again (satisfying the requirement that atomic borrows can only be opened for a single step of computation).

Finally, GetIndexCore provides a way to close the fractured borrow again, which consumes ownership of ℓ_{vec} in order to be able to give back the lifetime token.

The full soundness proof of our simplified variant of `BrandedVec` can be found in the Coq artifact that is part of our supplementary material [Yanovski et al. 2021].

4.4 Soundness of GhostCell

Finally, we come to the soundness proof of `GhostCell` itself. As with `BrandedVec`, we need to define the semantic interpretation of both `GhostCell<'id, T>` and `GhostToken<'id>`, and we need to show that all public functions of the `GhostCell` API are safe under this interpretation. For space reasons, we cannot discuss the interpretations and associated proofs in detail. Instead, we try to give a high-level intuition for the structure of the proof, and for the key challenge: tying the state of a `GhostCell` to the state of the associated `GhostToken`.

***The interpretation of* `GhostToken<'id>`.** A ghost token is always in one of two possible states: either it is currently *owned* (state StOwn), and can be used in a call to `GhostToken::borrow_mut` to grant mutable access to some `GhostCell`, or it is currently *shared* (state $\text{StShr}(\kappa)$, with κ denoting for how long the sharing lasts), and can be used in any number of calls to `GhostToken::borrow` to grant shared access to several different `GhostCell`s. As we did for monotone counters, we use Iris ghost state to manage this state of `GhostToken`. Its proof rules are as follows:

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

TokUpdate
$$\text{TokState}_1(\gamma, s) \Rrightarrow \text{TokState}_1(\gamma, s')$$

TokSplit
$$\text{TokState}_{q_1+q_2}(\gamma, s) \Leftrightarrow \text{TokState}_{q_1}(\gamma, s) * \text{TokState}_{q_2}(\gamma, s)$$

TokCombine
$$\text{TokState}_{q_1}(\gamma, s) * \text{TokState}_{q_2}(\gamma, s') \mathrel{-\!\!*} (s = s') * (q_1 + q_2 \leq 1)$$

Here, $\text{TokState}_q(\gamma, s)$ says that the current state of the ghost token named γ is s (which can be either StOwn or $\text{StShr}(\kappa)$), and that we own fraction q of that state. Owning the entire state (fraction 1) lets us use TokUpdate to switch to a different state, and when multiple fractions of the state are brought together, we learn that they must agree and cannot sum up to more than 1 (TokCombine).

As with `BrandedVec`, we will use GhostLft to tie the brand lifetime κ_{id} to the name γ of the ghost state. We will write $\text{TokState}_q(\kappa_{\text{id}}, s)$ as shorthand for $\exists \gamma.\, \text{GhostLft}(\kappa_{\text{id}}, \gamma) * \text{TokState}_q(\gamma, s)$, hiding the indirection from lifetimes to ghost names. This is justified by GhostLftLookup.

Based on this, the ownership and sharing predicate of `GhostToken` are defined (essentially) as:

$$[\![\mathbf{GhostToken}(\kappa_{\text{id}})]\!].\text{own}(\overline{v}) := \overline{v} = [\,] * \text{TokState}_1(\kappa_{\text{id}}, \text{StOwn})$$

$$[\![\mathbf{GhostToken}(\kappa_{\text{id}})]\!].\text{shr}(\kappa, \ell) := \exists \kappa'.\, \kappa \sqsubseteq \kappa' * \&^{\kappa'}_{\mathbf{frac}}\, \lambda q.\, \text{TokState}_q(\kappa_{\text{id}}, \text{StShr}(\kappa'))$$

The ownership predicate says that a token stores no data: the list $\overline{v}$ has to be empty. Moreover, it appropriately reflects the fact that the `GhostToken` is fully owned with the ghost state StOwn.

Similarly, the sharing predicate says that when the token is shared for the lifetime κ, that fact is reflected in the corresponding ghost state $\mathsf{StShr}(\kappa')$ for some lifetime κ' that is *at least* κ ($\kappa \sqsubseteq \kappa'$). The use of $\sqsubseteq$ allows subtyping with respect to lifetimes. As before, the use of a fractured borrow implies that for the duration of κ', anyone can borrow *some fraction* of this ghost state.

***The interpretation of* `GhostCell<'id, T>`.** For `GhostCell<'id, T>`, the ownership predicate is trivial—when the cell is not shared, it behaves exactly like `T`. This matches what RustBelt does for `Cell<T>` and also makes it easy to verify the safety contract of all methods that work with owned or mutably borrowed `GhostCell`s (*e.g.*, `GhostCell::new`, `GhostCell::get_mut`).

$$[\![\mathbf{GhostCell}(\kappa_{\mathsf{id}}, \tau)]\!].\mathrm{own}(\overline{v}) :\equiv [\![\tau]\!].\mathrm{own}(\overline{v})$$

For the sharing predicate, we focus on the parts that interact with the ghost token state:

$$[\![\mathbf{GhostCell}(\kappa_{\mathsf{id}}, \tau)]\!].\mathrm{shr}(\kappa', \ell) :\equiv \&^{\kappa'}_{\mathbf{at}}\Big(\big(\&^{\kappa'}_{\mathbf{full}}(\exists \overline{v}.\, \ell \mapsto \overline{v} * [\![\tau]\!].\mathrm{own}(\overline{v}))\big) \vee \tag{1}$$

$$\big(\exists \kappa.\, (\&^{\kappa}_{\mathbf{full}}\, \mathsf{TokState}_1(\kappa_{\mathsf{id}}, \mathsf{StOwn})) * \ldots\big) \vee \tag{2}$$

$$\big(\exists \kappa.\, (\&^{\kappa}_{\mathbf{frac}}\, \lambda q.\, \mathsf{TokState}_q(\kappa_{\mathsf{id}}, \mathsf{StShr}(\kappa))) * [\![\tau]\!].\mathrm{shr}(\kappa \sqcap \kappa', \ell) * \ldots\big)\Big) \tag{3}$$

The sharing predicate of `GhostCell` is a big atomic borrow, corresponding to an invariant that is maintained for the duration of the sharing (*i.e.*, lifetime κ'). This invariant consists of three possible states, somewhat similar to the three possible states of an `RwLock`—but unlike the state of a reader-writer lock, the state of a `GhostCell` is purely logical! The three possible states are reflected in a three-way disjunction: it is currently not accessed at all (1, corresponding to the lock being unlocked), accessed mutably (2, "write-locked"), or accessed in a shared way (3, "read-locked").

In case (1), the sharing predicate fully owns the (borrowed) content of the `GhostCell` at type τ. Just like with a lock that is not held, ownership fully resides in the `GhostCell`, and there is no interaction with the `GhostToken`. In case (2), that ownership of τ has been taken out of the sharing predicate by a call to `GhostCell::borrow_mut`; in exchange, (borrowed) ownership of the `GhostToken` that was passed to `borrow_mut` is held inside the sharing predicate, ensuring that the token cannot be used by anyone else until the borrow ends. Similarly, in case (3), the `GhostCell` sharing predicate holds ownership of a *shared* `GhostToken`—*i.e.*, it borrows some fraction of the token's ghost state which has to be $\mathsf{StShr}(\kappa)$. Moreover, it holds a proof that the `GhostCell` content is currently satisfying the sharing predicate of `T` ($[\![\tau]\!].\mathrm{shr}(\ldots)$), for a lifetime that corresponds to the *intersection* of the lifetime κ' for which the `GhostCell` is shared, and the lifetime κ for which the corresponding `GhostToken` is shared. This is crucial because the sharing of the `GhostToken` could end earlier than that of the `GhostCell`, in which case the `GhostCell` content needs to *stop* being shared to be ready for another `borrow_mut`.

To see how the sharing predicate works in action, we consider what happens in the proofs of `borrow_mut` and `borrow`, respectively.

`borrow_mut`. Here, we are given a shared `GhostCell` and a mutable reference to a `GhostToken`. The function returns a mutable reference `&mut T` to the content of the `GhostCell`; the proof needs to justify this by showing that the corresponding ownership can be obtained. We perform case distinction on the current state of the `GhostCell`. If it is currently in state (1), *i.e.*, "unlocked", then we "write-lock" it by transitioning to state (2). We can take out a full borrow of $\exists \overline{v}.\, \ell \mapsto \overline{v} * [\![\tau]\!].\mathrm{own}(\overline{v})$, which exactly corresponds to a mutable reference to `T` (the return type of `borrow_mut`), and we put the mutable reference of the `GhostToken` into the invariant. If the state is currently (2) or (3),

i.e., "locked", we can show a contradiction: the mutable reference to the `GhostToken` that we got as an input lets us borrow ownership of $\mathsf{TokState}_1(\kappa_{\mathsf{id}}, \mathsf{StOwn})$, so nobody else can own any fraction of this ghost state. And yet, in both (2) and (3), the `GhostCell` has at least some fraction q of this ghost state that can be borrowed. Combined with TokCombine, this leads to $1 + q \leq 1$ for a positive fraction q: a contradiction.[12]

`borrow`. Again, we perform a similar case distinction. Remember that we are given a shared `GhostCell` and a shared `GhostToken`, and we have to prove that we can return `&T`, a shared reference to the content of the `GhostCell`. If the `GhostCell` is currently in state (1), *i.e.*, "unlocked", we transition to state (3), "read-locking" it. Again we can take out borrowed ownership of the content at type `T` and initiate sharing of that `T`. If the `GhostCell` is in state (2) "write-locked", there is a contradiction, with arguments similar to the case of `borrow_mut` above. Finally, if the `GhostCell` is in state (3), then it is already read-locked. The sharing predicate for `T` is stored in the one for `GhostCell`, so we can successfully return a shared reference.[13]

This completes the proof sketch for `GhostCell`. In our supplementary material [Yanovski et al. 2021], we provide the full formal proof in Coq.

5 COMPARISON WITH OTHER INTERIOR-MUTABLE TYPES

In this section, we briefly compare `GhostCell` with Rust's existing types for interior mutability.

5.1 Micro-benchmarks

Conceptually, it is clear that `GhostCell` should perform better than the interior-mutable types of Rust that do dynamic, fine-grained permission tracking. To validate such a claim, we set up two micro-benchmarks for our doubly-linked list and graph examples.

The environment. We ran our benchmarks on a 3.3 GHz Intel Core i5 with 4 cores, running Debian 9, using Rust 1.52.1. Measurement and data gathering were done by *criterion* [Heisler and Aparicio 2020], a statistics-driven benchmarking library for Rust.

Doubly-linked lists. We compare our `GhostCell`-based linked list implementation against those of the only other thread-safe interior mutable types that permit interior pointers: `RwLock` and `Mutex`. We use a simple client similar to that in §3.2.3: we initialize linked lists of 100,000 integer nodes with `insert_next`, then perform parallel immutable iteration on 4 threads using `iterate`.

In Figure 2, we report the median execution time as well as the median absolute deviation (MAD), both in milliseconds (*ms*). The `GhostCell` version performs at least 10x better than the other two, confirming the expectation that both `Mutex` and `RwLock` have prohibitive performance costs. It is somewhat surprising that `Mutex` is faster than `RwLock` despite only `RwLock` permitting concurrent reads; likely, this is caused by the fact that the critical section is tiny, so the overhead is dominated by the lock implementation itself—and `Mutex` is simpler than `RwLock`.

DFS traversals on graphs. We compare our `GhostCell`-based DFS implementation (§3.3) against naive versions based on `Mutex` and `RwLock`. We initialize a cyclic graph with 100,000 nodes with integer data and 400,000 edges, and then perform (1) singled-threaded immutable traversals, (2) parallel immutable traversals with 4 threads, and (3) single-threaded mutable traversals. The results are shown in Figure 3: again, `GhostCell` performs much better, for the same reason as before.

For a more realistic comparison, we also benchmark our implementation against an implementation using the `Graph` type of `petgraph` (version 0.5.1)—the currently most widely used Rust library

[12] There is another case of the proof here that we are omitting: if the lifetime κ that is quantified in the `GhostCell` sharing predicate has already ended, resources that we have omitted here can be used to transition to state (1) and proceed as before.
[13] Again, to simplify matters, we omitted what happens in cases (2) and (3) when the lifetime κ has already ended.

Benchmarks (**median time** ± the median absolute deviation)		
Data structure: Doubly-linked lists (100K nodes)	Initialization (*ms*) (insertions)	4-thread Parallel Iteration (*ms*) (immutable reads)
GhostCell	**0.26** ±.001	**0.14** ±.001
Mutex	**10.4** ±.01	**2.0** ±.03
RwLock	**14.1** ±.02	**3.3** ±.02

Fig. 2. Comparing doubly-linked list implementations: `GhostCell` vs. `Mutex` and `RwLock`.

Benchmarks (**median time** ± the median absolute deviation)				
Algorithm: DFS (100K nodes, 400K edges)	Initialization (*ms*)	1-thread Immutable Iteration (*ms*)	4-thread Immutable Iteration (*ms*)	1-thread Mutable Iteration (*ms*)
GhostCell Vec	**5.83** ±.02	**1.11** ±.01	**1.26** ±.004	**1.18** ±.02
GhostCell List	**1.33** ±.002	**1.36** ±.001	**1.58** ±.006	**1.52** ±.003
petgraph's Graph	**2.11** ±.002	**1.60** ±.003	**1.73** ±.004	**1.58** ±.004
Mutex	**16.97** ±.02	**5.09** ±.04	**5.54** ±.02	**5.07** ±.09
RwLock	**21.86** ±.04	**7.84** ±.06	**11.54** ±.14	**6.93** ±.07

Fig. 3. Comparing DFS implementations: `GhostCell` vs. `Mutex`, `RwLock`, and `petgraph::graph::Graph`.

for graphs [petgraph 2021]. `petgraph` uses a `Vec`-based representation, where nodes are stored in a vector and the adjacency lists use indices into the vector to refer to child nodes. This permits an efficient implementation in safe Rust, at the cost of an extra indirection through the node index.

The results in Figure 3 show that iteration on `GhostCell` is consistently slightly faster than on `petgraph`'s `Graph`. We hypothesize that the difference may be due to `petgraph`'s need to perform bounds checks when indexing into the nodes array. Initialization is slightly slower with "GhostCell Vec", which uses a `Vec` to store the list of edges (as in §3.3). Those arrays are stored outside the arena on the regular heap, making allocation more costly. The "GhostCell List" version improves initialization performance by implementing adjacency lists as a linked list stored inside an arena (which is more comparable with `petgraph`, where an index-based linked list data structure is used for the adjacency list).

Our results show that a `GhostCell`-based graph can achieve performance competitive with a state-of-the-art production-grade Rust graph library.

5.2 Rust's Interior-Mutable Types

We have focused our discussion in this paper on comparing `GhostCell` against `RwLock`, since (together with `Mutex`) it is the only interior-mutable type available in Rust that offers interior pointers and thread-safety. Here we briefly discuss why the remaining interior-mutable types of Rust do not provide the desired API for the doubly-linked list.

(1) **No interior pointers**: `Cell<T>` and `AtomicCell<T>`[14] provide a thin wrapper around a type `T`, but only support operations to get and set the current value of type `T` as a whole. One cannot make interior pointers into the underlying data `T`, so one cannot perform in-place mutation, and passing the data around would require a full copy.

[14] `AtomicCell` is not part of the Rust standard library but provided a widely-used user library [crossbeam 2021].

(2) **Not thread-safe**: `RefCell<T>` is more flexible than `Cell` in that it supports interior pointers and in-place mutation; however (like `Cell`) it is not thread-safe, so neither concurrent read-only access, nor transfer of ownership between threads, are possible. It is also not zero-cost: `RefCell` is a single-threaded reader-writer lock, and tracking that state incurs overhead.

To summarize, unlike `GhostCell`, none of the existing types for interior mutability in Rust is able to combine support for interior pointers with a zero-cost implementation.

6 RELATED WORK

In their development of the `ST` monad, Launchbury and Peyton Jones [1995] discovered that rank-2 polymorphic types [Kfoury and Wells 1994]—in conjunction with a style of data type that later came to be known as *phantom types* [Fluet and Pucella 2006]—could be used to simulate a lightweight form of state-dependent types. Under this approach, the phantom type parameters of a type—*i.e.*, the type parameters that merely played a role in restricting the *use* of the type but did not serve any actual syntactic function in its underlying implementation—could serve as *brands*: static representatives of dynamically generated state. For example, with the `ST` monad, the type `ST s t` refers to a monadic computation producing a value of type `t`, which may contain references into a piece of state represented abstractly by the brand `s`. The use of rank-2 polymorphism enables the API to enforce that clients work with an arbitrary, abstract brand. If a term inhabits the type `ST s t` for an arbitrary brand `s`, its monadic computation cannot depend on any external state, so the `runST` primitive allows one to safely escape the monad by executing the monadic computation and producing a pure term of type `t`.

Kiselyov and Shan [2007] took this style of API further, showing how branded types could be used (in the context of a standard functional language like ML or Haskell) to enforce protocols on the use of a data structure, which could guarantee that accesses to the data structure were safe without requiring run-time checks. They referred to this approach as "lightweight static capabilities". Later, Beingessner [2015] applied a similar idea to Rust in the development of her "unchecked indexing" API, which we discussed in §2. In the Rust setting, it became necessary to represent brands using phantom *lifetimes*, rather than phantom *types*, because Rust only supports rank-2 polymorphism over lifetimes, not types. Although `GhostCell` is quite different in detail and application from Beingessner's API, the inspiration for it came directly from her work.

Some of the aforementioned approaches to branded types have been proven sound, to varying degrees of formality. In their original work on the `ST` monad, Launchbury and Peyton Jones [1995] argued that the `runST` mechanism preserves referential transparency, using relational reasoning over a denotational semantics, but they did not connect this to a more realistic operational semantics of mutable state with a global heap and in-place update. Moggi and Sabry [2001] subsequently verified safety of `runST` against an operational semantics, and more recently, Timany et al. [2018] extended their results to also establish equational properties in the presence of `runST`. They did so formally in Coq using the Iris framework, which we have also used in this paper. Kiselyov and Shan [2007] described a formal methodology for establishing the safety of branded-types APIs, but they only applied this methodology to purely functional examples. To our knowledge, we are the first to prove the soundness of branded-types APIs that make essential use of ownership-based (substructural) typing. Moreover, our proof is fully formalized in Coq, as an extension to RustBelt.

`QCell` [Peters 2019] is an experimental Rust crate providing a library of alternatives to `Cell`. One of the types in the library is `LCell`, whose API (as the author acknowledges) is directly derived from an implementation of `GhostCell` that we posted online several years ago. However, `QCell` also includes several other interesting interior-mutable types that decouple shareable cells from their permissions. These cell types follow the style of `GhostCell` in distinguishing between a "cell"

and a "token" type (the latter is called "owner" in `QCell`). The difference between the types is in how they connect the cell and token types—*i.e.*, how they represent the *brand*. In `LCell`, as with `GhostCell`, the brand is represented as a phantom lifetime parameter. In `TCell` and `TLCell`, the brand is represented as a type parameter, and uniqueness of brands is enforced dynamically using a global type-indexed table of singleton token instances. This has the advantage of avoiding rank-2 polymorphism, but introduces new failure cases when the singleton is already in use elsewhere. The last alternative is the eponymous `QCell` type itself, which represents the brand using an extra integer field in both the cell and the token types. Accessing the contents of a cell using a token incurs a dynamic check to make sure the integers match. This makes it easier to support a finer granularity of permission tracking than is convenient with `GhostCell`, but it comes at the cost of introducing new dynamic failure paths, additional space consumption, and dynamic brand checks. The soundness of `QCell`, unlike `GhostCell`, has not been formally verified.

The core idea underlying `GhostCell`, namely separating permissions from data, also forms the foundation of *implicit dynamic frames* [Smans et al. 2009; Jacobs et al. 2011]. Here, the accessibility predicate **acc**(`x.f`) roughly corresponds to `GhostToken`. The technique of implicit dynamic frames has been successfully applied in tools for automated program verification, most notably Viper [Leino and Müller 2009; Müller et al. 2016]. The key novelty in our work is that we carry out this separation of permissions from data in the context of a *type system*, via a user-defined library.

The API of `GhostCell` bears some resemblance to a number of earlier ideas from the literature on substructural typing. In particular, *adoption and focus* [Fähndrich and DeLine 2002] is a type system mechanism introduced to provide temporary exclusive access to shared data. The concept of "nesting" [Boyland 2010] generalizes adoption and focus to managing arbitrary permissions. Nesting is implemented in Mezzo [Balabonski et al. 2016] as an axiomatized library of operations that are just type coercions, much like in `GhostCell` (but without a soundness proof). One minor advantage of `GhostCell` over these prior approaches is that it leaves the choice of how to perform memory management to the client; Mezzo relies on garbage collection for this, and the original adoption needs some run-time tracking. However, the main advantage of `GhostCell` is expressivity: adoption/nesting are irreversible operations, whereas `GhostCell` provides methods such as `into_inner` and `get_mut` that build on Rust-style borrowing to support regaining full control over ownership that was previously managed by some `GhostToken`. Thanks to Rust's concept of a shared reference with a lifetime, `GhostCell` also supports *temporarily* sharing the data for multiple concurrent read-only operations via `borrow`, but with the ability to regain full exclusive control via `borrow_mut` later on. Prior approaches do not offer that kind of flexibility.

Mezzo also supports a variant of adoption and focus that the authors dubbed "adoption and abandon". This mechanism has very different trade-offs than nesting or `GhostCell`: it supports exclusive access to multiple distinct "adoptees" at the same time, at the cost of a runtime check for each access to ensure that each individual adoptee is only accessed once (akin to a per-node `RwLock`, but with less overhead). That makes it unsuited for zero-overhead data structures like our doubly-linked list, but in other situations a runtime check might be acceptable or even truly needed. We leave it to future work to write and verify a Rust version of Mezzo's adoption and abandon—as with `GhostCell`, we expect that the integration with borrowing and lifetimes could give a significant boost to the expressivity of this mechanism.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive suggestions for improvement. This research was supported in part by European Research Council (ERC) Consolidator Grants for the projects "RustBelt" and "PERSIST", funded under the European Union's Horizon 2020 Framework Programme (grant agreements 683289 and 101003349, respectively).

REFERENCES

Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2016. The design and formalization of Mezzo, a permission-based programming language. *TOPLAS* 38, 4 (2016). https://doi.org/10.1145/2837022

Alexis Beingessner. 2015. *You can't spell trust without Rust.* Master's thesis. Carleton University, Ottawa, Ontario, Canada.

John Tang Boyland. 2010. Semantics of fractional permissions with nesting. *ACM Trans. Program. Lang. Syst.* 32, 6 (2010), 22:1–22:33. https://doi.org/10.1145/1749608.1749611

crossbeam. 2021. crossbeam. https://crates.io/crates/crossbeam.

Manuel Fähndrich and Robert DeLine. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. In *PLDI*. https://doi.org/10.1145/512529.512532

Nick Fitzgerald and Simon Sapin. 2020. The Typed-Arena library. https://crates.io/crates/typed-arena.

fixedbitset. 2021. fixedbitset. https://crates.io/crates/fixedbitset.

Matthew Fluet and Riccardo Pucella. 2006. Phantom Types and Subtyping. *J. Funct. Program.* 16, 6 (2006). https://doi.org/10.1017/S0956796806006046

Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *PLDI*. https://doi.org/10.1145/512529.512563

Brook Heisler and Jorge Aparicio. 2020. The Criterion library. https://crates.io/crates/criterion.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*. https://doi.org/10.1007/978-3-642-20398-5_4

Ralf Jung. 2020. *Understanding and Evolving the Rust Programming Language.* Ph.D. Dissertation. Universität des Saarlandes.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the Foundations of the Rust Programming Language. *PACMPL* 2, POPL, Article 66 (2018). https://doi.org/10.1145/3158154

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe Systems Programming in Rust. *Commun. ACM* (April 2021). https://doi.org/10.1145/3418295

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *Journal of Functional Programming* 28, e20 (Nov. 2018), 1–73. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*. https://doi.org/10.1145/2676726.2676980

A. J. Kfoury and J. B. Wells. 1994. A Direct Algorithm for Type Inference in the Rank-2 Fragment of the Second-Order λ-Calculus. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*. 196–207.

Oleg Kiselyov and Chung-chieh Shan. 2007. Lightweight Static Capabilities. *Electron. Notes Theor. Comput. Sci.* 174, 7 (2007), 79–104. https://doi.org/10.1016/j.entcs.2006.10.039

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *PACMPL* 2, ICFP, Article 77 (2018), 77:1–77:30 pages. https://doi.org/10.1145/3236772

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. https://doi.org/10.1145/3009837.3009855

John Launchbury and Simon L. Peyton Jones. 1995. State in Haskell. *LISP and Symbolic Computation* 8, 4 (Dec. 1995), 293–341. https://doi.org/10.1007/BF01018827

K. Rustan M. Leino and Peter Müller. 2009. A Basis for Verifying Multi-threaded Programs. In *ESOP*. https://doi.org/10.1007/978-3-642-00590-9_27

Nicholas D. Matsakis. 2016. Non-lexical lifetimes: Introduction. http://smallcultfollowing.com/babysteps/blog/2016/04/27/non-lexical-lifetimes-introduction/.

Nicholas D. Matsakis. 2018. An alias-based formulation of the borrow checker. https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/ Blog post.

Nicholas D. Matsakis and Felix S. Klock II. 2014. The Rust language. In *SIGAda Ada Letters*, Vol. 34. https://doi.org/10.1145/2663171.2663188

Eugenio Moggi and Amr Sabry. 2001. Monadic encapsulation of effects: A revised approach (extended version). *JFP* 11, 6 (Nov. 2001), 591–627.

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

Jim Peters. 2019. The QCell library. https://crates.io/crates/qcell.

petgraph. 2021. petgraph. https://crates.io/crates/petgraph.

John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *LICS*. https://doi.org/10.1109/LICS.2002.1029817

Jan Smans, Bart Jacobs, and Frank Piessens. 2009. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In *ECOOP*. https://doi.org/10.1007/978-3-642-03013-0_8

Josh Stone and Nicholas D. Matsakis. 2017. The Rayon library. https://crates.io/crates/rayon.

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A Logical Relation for Monadic Encapsulation of State: Proving Contextual Equivalences in the Presence of runST. *PACMPL* 2, POPL, Article 64 (Jan. 2018), 28 pages. https://doi.org/10.1145/3158152

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *High. Order Symb. Comput.* 17, 3 (2004), 245–265. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. Coq development and supplementary material accompanying this paper. https://plv.mpi-sws.org/rustbelt/ghostcell/.