Safe, Flexible Aliasing with Deferred Borrows

Chris Fallin
Mozilla[1], Mountain View, CA, USA
cfallin@c1f.net

Abstract

In recent years, programming-language support for *static memory safety* has developed significantly. In particular, *borrowing and ownership* systems, such as the one pioneered by the Rust language, require the programmer to abide by certain aliasing restrictions but in return guarantee that *no unsafe aliasing can ever occur*. This allows parallel code to be written, or existing code to be parallelized, safely and easily, and the aliasing restrictions also statically prevent a whole class of bugs such as iterator invalidation. Borrowing is easy to reason about because it matches the intuitive ownership-passing conventions often used in systems languages.

Unfortunately, a borrowing-based system can sometimes be too restrictive. Because borrows enforce aliasing rules for their entire lifetimes, they cannot be used to implement some common patterns that pointers would allow. Programs often use pseudo-pointers, such as indices into an array of nodes or objects, instead, which can be error-prone: the program is still memory-safe by construction, but it is not *logically memory-safe*, because an object access may reach the wrong object.

In this work, we propose *deferred borrows*, which provide the type-safety benefits of borrows without the constraints on usage patterns that they otherwise impose. Deferred borrows work by encapsulating enough state at creation time to perform the *actual* borrow later, while statically guaranteeing that the eventual borrow will reach the same object it would have otherwise. The static guarantee is made with a *path-dependent type* tying the deferred borrow to the container (struct, vector, etc.) of the borrowed object. This combines the type-safety of borrowing with the flexibility of traditional pointers, while retaining logical memory-safety.

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases Rust, type systems, ownership types, borrowing

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.30

1 Introduction

Managing memory ownership properly is central to safe, correct programming in any programming language with a mutable heap. Parallel programs with shared memory can easily experience non-deterministic, undefined behavior if two concurrently-executing threads write to the same memory. Even sequential programs can experience subtle correctness issues related to memory ownership: for example, pointer invalidation occurs when a data structure traversal simultaneously mutates that data structure, leading to dangling or incorrect references.

Modern programming languages have developed support for managing ownership correctly by encoding various invariants statically in the type system. Many works propose to augment pointers with ownership information or capabilities (indicating temporary exclusive access) [5, 4, 3, 1, 6, 12, 15, 22, 20]. Another related approach categorizes heap objects into disjoint heap subregions, and annotates pointers to refer only to particular regions [8, 11, 3, 9]. Among languages in widespread use today, the Rust programming language [18] provides an *ownership and borrowing* system that adapts ideas from lexical regions [8] to annotate

[1] This work is independent of author's employer and author does not speak for Mozilla.

lifetimes on pointers. The language's type system tracks "borrows" of an owned object – pointers taken to the object, or copies of those pointers – and ensures that only *one* mutable view of the object can be used at a time by blocking access to the original object for the borrow's duration. All of these systems work to enforce memory safety in some way by either disallowing some pointers to *exist*, or else disallowing some pointers to be *accessed*, in order to prevent aliasing, concurrent accesses that would cause correctness issues.

Although these systems can improve correctness for many classes of programs, there is still resistance to their adoption because they are often not *flexible* enough. For example, since its release in 2015, the Rust language has become well-known for the introductory experience of "fighting the borrow checker"[2]. The patterns that its ownership-passing and temporary-borrowing system allows and encourages are unfamiliar to programmers accustomed to unrestricted pointers in C or Java. As a result, Rust programmers have developed creative workarounds in their data-structure designs. For example, programs that manipulate graphs of objects, or otherwise have unpredictable heap graphs, sometimes use indices into a vector of objects as pseudo-pointers, in place of true pointers (borrows). Unfortunately, this merely sidesteps the problem, because these pseudo-pointers can be error-prone (as we will show later), and in any case are more cumbersome to use.

In this paper, we observe that an ownership and borrowing memory-management discipline is sometimes inflexible for *artificial and unnecessary reasons*, and that by splitting the borrow operation into two parts – the lookup, producing a "deferred borrow" handle, and a later conversion into a real borrow – much of the ease-of-use of an unrestricted language is recovered. The key idea in this work is to encapsulate a reference to an object that is not (yet) a borrow, so it does not trigger the restrictive mutual-exclusion rules that mandate only one usable mutable reference exist. Our language extension uses *path-dependent types* to statically bind this reference to the container (such as a vector, key-value map, struct or tuple) that owns the object. The deferred-borrow reference can later, at the point of actual use, be recombined with a reference to the container to get a true reference (borrow) of the pointed-to object. This true borrow lasts only as long as needed (during the use itself), and thus does not prevent the program from holding many deferred-borrow references to the same object, several of them mutable at once. Flexibility is achieved without giving up Rust's memory safety or suitability for concurrency.

The structure of the remainder of this paper is as follows. In §2, we first provide a tour of the essential aspects of Rust's borrowing and ownership system. Then, in §3, we show how Rust's borrows do not provide as much flexibility as pointers do, and demonstrate the alternative approaches that programs often use. We show that while these approaches attain Rust's memory safety properties in a *literal* sense, they lack what we call *logical memory safety*: the pseudo-pointers (e.g., indices into vectors), if used incorrectly, can cause the program to silently access the *wrong* data. We describe what would be necessary to avoid these logical memory safety violations. In §4, we introduce an extension to the Rust type system, *static path-dependent types*, that provide a minimal building-block for object references that retain logical memory safety. We prove that static path-dependent types provide an essential *object-binding* property. In §5, we finally introduce the concept of a *deferred borrow*, which is a general pattern that can be implemented by several types of *containers*, such as vectors, key-value maps, structs, and tuples. In §6, we describe how we have emulated static path-dependent types in the existing Rust type system for evaluation

[2] In the 2018 Rust community survey [19], the second and third most difficult-rated topics to learn were ownership/borrowing and borrow lifetimes, after only the macro system.

purposes. In §7, we analyze several common data-structure design patterns and qualitatively describe how the deferred borrows approach compares to other techniques. In §8 we consider related work.

2 Background: Ownership and Borrowing

In order to understand the need for deferred borrows, we first describe *borrowing-and-ownership-based memory management* as it is practiced in the Rust programming language [18]. We will describe how the language uses *ownership* to establish unique, unaliased access to heap objects, which permits statically-verifiable safe parallelism (§2.1). Then, because a pure ownership tree is cumbersome and difficult to use, we describe how Rust allows *borrowing* of owned objects by a form of lifetime-restricted pointer (§2.2). Finally, we look at how Rust container types typically use borrowing to encode safety invariants and extend the memory safety of core Rust data structures to the library level (§2.3).

2.1 Object Ownership and Linear Move Semantics

The Rust language, as many other memory safe languages or type systems before it [5, 4, 3], begins with establishing *unique ownership* for every object. An instance of an object (in Rust, a `struct`) can either exist on the stack or on the heap. In the former case, the local variable binding owns the object; in the latter, the heap memory is ultimately managed by an object on the stack, even if this is just a `Box<T>` (a pointer type).

The language uses *affine types* to ensure that a given object instance is not duplicated. When a non-copyable type is assigned (only primitive data types are trivially copyable), the object is *moved out of* the original storage slot (e.g., local variable binding), and the storage slot becomes inaccessible.

Fig. 1 shows a simple demonstration of moving ownership. We can see several examples of both stack-allocated and heap-allocated objects in this program. The `s` local variable is assigned an object initializer for a struct of type `S`; the storage for `s` is allocated in the stack frame. Several of its fields own heap storage, however: for example, the `t` field owns an instance of `T` that is stored on the heap, and the `children` field owns a vector (array-backed list) of `S`, also stored on the heap.

There are two important aspects to Rust's ownership system on display here. First, without borrowing (which we introduce below), storage is organized into a strict *tree of access paths*. There is data at `s`, and `s.t`, and `s.children[0]`, and `s.children[0].t`. Each object has exactly *one* access path, however: there is no aliasing of heap references.

Second, Rust enforces *move semantics*. When the program assigns local variable `t` the value of `s.t`, the assignment *moves* the object, and `s.t` is no longer accessible. The program's attempt to access `s.t.a` later is thus a static compilation error. Likewise, `s.children` becomes inaccessible after a move. These assignment semantics preserve the above property: by never duplicating non-trivial (non-copyable) data, we never create aliasing pointers. Of course, this restriction will be relaxed later, but only in a controlled way.

By providing exactly one path to any particular object, the ownership system can provide both (i) precise memory management – when a path goes out of scope, its subtree of memory resources is freed – and (ii) safe, deterministic (race-free) parallelism, by passing ownership of entire subtrees to separate parallel tasks. Despite these advantages, it is cumbersome: many common programming idioms rely on holding multiple references to particular heap objects. We now describe how pointers can be re-introduced in a limited but safe way.

```
struct S {
  value: u64,
  t: Box<T>,
  children: Vec<S>,
}
struct T {
  a: u64,
  b: u64,
}

fn main() {
  // `s` lives on the stack, and is the unique owner of:
  //   `t`, a `T` instance that is also on the stack
  //   `children`, a `Vec` whose storage (and S instances)
  //               are on the heap
  let mut s: S = S { value: 1,
                     t: Box::new(T { a: 1, b: 2 }),
                     children: vec![ /* ... */ ] };
  s.value += 1;
  s.t.a = 100;
  s.children.push(S { ... });

  let t = s.t;               // moves `t` out of `s`
  let children = s.children; // moves `children` out of `s`
  s.t.a = 1;                 // ERROR! (s.t moved out of already)
}
```

Figure 1 Rust ownership system: examples.

2.2 Safe Pointers: Lifetime-constrained Borrows

In order to allow references to a particular object, Rust permits *borrows*. A borrow is a kind of pointer that can be created from an owned access path, and that has an explicit *lifetime* that is statically restricted to maintain the safety properties that we introduced with unique ownership above. In particular:

- A borrow *temporarily restricts access* to an access path, just as a move out of that path permanently restricts access.
- A borrow can only exist as long as the original borrowed object exists: for example, it cannot be returned from a function if it borrows an object on that function's stack frame.

Both of these restrictions are implemented using *lifetimes*. A lifetime is, conceptually, a static description of a period of time during execution: either a local scope or a contiguous range of program points. Semantically, every borrow has a lifetime, and every local variable does, as well. In Rust's syntax, a lifetime is written as `'a`, and a borrow of an object of type `T` is written `&'a T`. Though every borrow semantically has a lifetime, lifetimes are usually inferred and so they can be omitted.

When a borrow is created, the borrow's lifetime is *constrained* such that the borrowed object must *outlive* the borrow, and the lifetime of the local variable that holds the borrow (as a value) must be outlived by the borrow lifetime. This is illustrated in Fig. 2.

Note that borrows can be included in data structures: when a an aggregate datatype (such as a `struct`) has a member whose type is a borrow, the lifetime of that borrow must be defined as a *generic parameter* of the type. This need arises because such a type is (usually) defined outside of any function scope, and hence no lifetimes are otherwise in scope. This implicitly creates an outlives-constraint: when one object contains a pointer to (i.e., a borrow of) the other, the latter must outlive the former. This is analogous to the local-variable case above.

```
struct S { ... }

fn main() {
  let mut s: S = S { ... };

  s.a = 1;

  let mut s_borrow: &mut S = &mut s;            // \
                                                // |
  s.a = 2;  // ERROR: s is currently borrowed   // | borrow lifetime
                                                // |
  s_borrow.a = 2;  // OK.                       // /

  // s_borrow is now dead -- `s` can be accessed again.

  s.a = 3;
}
```

Figure 2 Rust borrowing examples.

2.2.1 Access Path-based Disjointness

The borrow system supports two types of borrows: *mutable* (as `&mut T`) and *immutable* (as `&T`). Multiple immutable borrows may have overlapping lifetimes, and read-only accesses to the original, borrowed, object may occur while such borrows exist. In contrast, if a mutable borrow is created, then no other mutable borrow of that object may have an overlapping lifetime, and the original object is inaccessible during its lifetime, as illustrated in Fig. 2.

In order to maintain this single-access-path invariant that enables safe parallelism, the Rust borrow system tracks *disjointness* of borrows. In particular, for any given object stored within a local variable or field of a struct (identified by an access path starting from a local variable), the compiler tracks which borrows are active for which contiguous spans of static program points, and ensures that no two incompatible borrows or direct accesses overlap.

2.2.2 Two Guarantees: Safety and Unique Mutability

The borrow system in Rust provides pointers that have been statically verified to retain two important properties: *memory safety* and *unique mutability*. Memory safety arises from outlives-constraints, and means that a pointer cannot be dangling; every pointer dereference thus accesses valid memory, and the program can never crash from an invalid pointer access. Unique mutability arises from disjointness constraints, and means that if a pointer is mutable (can be used for writes), then it is the only available access path to its pointee. This is what allows for safe parallelism: because of unique mutability, a program cannot have data races. Both sorts of constraints, and both resulting guarantees, also allow for the creation of safe *container types* that extend the access path-based system to dynamically-sized, heap-resident containers.

2.3 Borrows and Container Types

The power of Rust's ownership and borrow system arises from the way in which its basic primitives – borrows with lifetimes, and constraints between lifetimes – can be used by *libraries* to build type-safe containers.

Consider, for example, the simple vector (array-backed ordered list with $O(1)$ element access) API in Fig. 3. This API has two functions: `get_index_mut`, which returns a pointer to a storage slot within the vector, and `append`, which appends a new element.

```
struct Vector<T> { ... }
impl<T> Vector<T> {
  // The 'a lifetime (usually implied, but written explicitly here) ties
  // the returned borrow to an implicit borrow of `self` at the callsite.
  // The callee cannot use the `Vector` in any other way while the
  // borrow to the element is live.
  fn get_index_mut<'a>(&'a mut self, index: usize) -> &'a mut T { ... }
  fn append(&mut self, t: T) { ... }
}

fn main() {
  let mut v = Vector<u32>::new(...);

  let elem0 = v.get_index_mut(0);  // borrows `v`, returns borrow to elem
  *elem0 = 1;
  // `elem0` borrow now dead (not used below). Borrow lifetime on `v` ends.

  let elem1 = v.get_index_mut(1);
  let elem2 = v.get_index_mut(2);  // ERROR: `v` already borrowed mutably.
  *elem1 = 2;
  *elem2 = 3;

  let elem3 = v.get_index_mut(3);
  // The mutable borrow also serves to "freeze" the underlying storage
  // location in place (no other method on the `Vector` can be invoked,
  // because we cannot borrow its `self` again). This is needed
  // because the borrow is just a pointer: the container must not
  // e.g. reallocate its storage to grow an array, rehash a table or
  // rotate a tree, etc.
  v.append(4);  // ERROR: `v` already borrowed mutably.
  *elem3 = 4;
}
```

Figure 3 Rust lifetime constraints used in container APIs.

In order to return a borrow to internal storage – an element contained within, and whose memory is managed by, the vector – the `get_index_mut` function must take a borrow of the vector itself, Given the borrow of the *whole* vector, it can safely return a borrow of a piece of that vector, as long as the returned borrow's lifetime is contained within the original borrow's lifetime (trivially true here as the lifetime is the same `'a`).

This lifetime-outlives constraint, relating a borrow of the original container to that of its element, serves two important purposes. First, the borrow on the container serves as a proxy for the borrow of the element itself. Because the Rust borrow-checker does not have a precise understanding of vector indices or hashmap keys (for example), it cannot directly track the access-paths that name these storage locations. Hence, although it might be perfectly valid (assuming a well-behaved data structure implementation) to borrow both `v[i]` and `v[j]` mutably if $i \neq j$, because the element storage slots are disjoint, the borrow checker cannot actually verify this. The container API leverages the borrow checker in a sound but conservative way, requiring a borrow of the *entire container* when an element is borrowed. Because it is always sound to "over-borrow," this maintains the no-dangling-pointer and no-aliasing-mutable-pointer safety properties of Rust's type system.

Second, and just as importantly, this lifetime constraint and container-borrow mechanism *freezes the container layout in place* so that the raw pointer (which is how a borrow is implemented) remains valid. Here, any access with `get_index_mut` borrows the container mutably, which prevents any other access to the original object. Idiomatic container APIs also allow immutable element borrows, which borrow the container immutably: this allows other

read-only access to the container, but still prevents any mutation. This works because any other function that mutates the container – e.g., `append`, which might cause the underlying storage to be reallocated if more space is needed – takes a `&mut self` parameter, requiring a mutable borrow, which is incompatible with the outstanding (immutable or mutable) borrow.

3 Inflexible Borrows and Workarounds

Now that we have seen how Rust enables safe aliasing through *borrows* with carefully checked disjoint-lifetime and disjoint-mutability properties, let us consider how these limitations impact program design.

3.1 Borrowing-Incompatible Data Structures

Two basic patterns of pointer-based data structure design are problematic when borrows are used in place of pointers, corresponding to each of the constraints that the borrowing system imposes.

First, the borrowing system requires the borrowed object to *outlive* the borrow itself, to preserve the language's memory-safety property. When one object points to another, the lifetime of the former must be strictly shorter than the latter. This immediately rules out *cycles* of borrows. Common data structures that are cyclic, such as graphs, doubly-linked lists, and trees with parent pointers, thus cannot be implemented in safe Rust.

Second, the borrowing system requires borrows to be *safe* relative to each other, and in particular, allows no more than one mutable borrow to exist at a time. There are many programming idioms that require holding pointers to inner elements of a data structure: for example, a "secondary index" might refer to elements in a vector or map indexed by an alternate key, or an algorithm may keep a stack of pointers to nodes as it traverses a graph or tree. Some of these pointers may later be used to update the data structure. Unfortunately, Rust cannot allow these pointers (borrows) to be mutable.

Both of these problems can be seen in Fig. 4. Program 1 shows a simple graph-manipulation program in C++, demonstrating the ease by which the graph node type can be defined. In contrast, in Program 2, we run into trouble as soon as we try to define the node types, caused by both reference cycles and aliasing mutable borrows. We clearly cannot carry over our habits of freely handling pointers as we had done in C++.

One approach, sometimes seen in core data-structure libraries, is to use "unsafe" raw pointers that circumvent the type system. While the Rust language allows this C-like flexibility via an escape hatch, the memory safety then relies completely upon the programmer's care. We thus do not consider this approach further.

3.2 A Solution: Pseudo-Pointers

A common approach to allow arbitrary object references in safe Rust is to use names for objects that are not actually borrows (pointers), such as indices in a vector. Program 3 in Fig. 4 demonstrates this approach. Node references are by indices into a vector, and this vector is the true owner of the nodes. Rust's tree-ownership model that described in §2 is retained, and the program is completely memory-safe. There is no issue when `visitEdge` needs to take references to two different nodes to mutate, because these references (indices) are not actually potentially-aliasing borrows, only integers. We call these integers *pseudo-pointers*.

Program 1: C++, using unrestricted native pointers

```
// Define a graph as a list of pointers to nodes; define a node's edges
// simply as pointers to other nodes.
typedef vector<Node*> Graph;
struct Node { vector<Node*> outEdges; };
void visitEdge(Node* n, Node *neigh) { ... }

void updateGraph(Graph& g) {
  for (Node* n : g)
    for (Node* neighbor : n->outEdges)
      visitEdge(n, neighbor);
}
```

Program 2: Rust, using references (borrows)

```
// Problem 1: we will not be able to construct a graph instance of Node<'a>
// because each node needs to outlive its pointed-to nodes; the cyclic
// dependency is impossible to resolve.
//
// Problem 2: we cannot hold mutable borrows of neighboring nodes in
// `outEdges`, because more than one borrow might exist (the type
// system will not allow us to create these borrows).
type Graph<'a> = Vec<Node<'a>>;
struct Node<'a> { outEdges: Vec<&'a mut Node> }
```

Program 3: Rust, using node indices

```
// Define a graph as an owned vector of nodes; define out-edges as
// indices of other nodes in this vector.
type Graph = Vec<Node>;
type NodeIndex = usize;
struct Node { outEdges: Vec<NodeIndex> }

// NOTE: this is cumbersome: every access to a node `n' is really
// `g.nodes[n]'.
fn visitEdge(g: &mut Graph, n: NodeIndex, neighbor: NodeIndex) { ... }

fn updateGraph<'a>(g: &mut Graph<'a>) {
  for n_idx in 0..g.len() {
    // Note: we need to copy the outEdges list here because `visitEdge'
    // below takes temporary mutable ownership of the entire graph `g'.
    let neighs = g[n_idx].outEdges.clone();
    for neigh_idx in &neighs {
      visitEdge(g, n_idx, neigh_idx);
    }
  }
}
```

Figure 4 A graph-processing program, in C++ (first program) and Rust (second and third programs), that demonstrates the difficulties imposed by borrowing (second program) and the type-unsafety of the usual workaround (third program).

However, this approach has several downsides. First, and most directly, it is *cumbersome*. To make it work, we need to (i) pass a borrow to the true owner (here, the `Graph` object) everywhere along with the pseudo-pointers, and (ii) explicitly dereference the node by indexing the vector at each point of use. However, beyond the mere ergonomics issues, a potential *correctness* issue looms: the vector access `g[n_idx]` may not refer to the same `Node` at access time that it did when the index was taken! If, for example, the program removes a node and compacts the node-vector, all node indices become invalid, but the type system does not prevent their use. Even worse, if the program contains *multiple* vectors of the appropriate type (say, the program maintains several graphs simultaneously), an index intended for one may be used to access another. We define a new term to encapsulate these issues: *logical memory safety*.

3.3 Logical Memory Safety

Rust provides *memory safety* in the sense that any memory accessed by a Rust program must be valid memory and must be a valid, still-live instance of the object implied by the type of the pointer (borrow). Nothing in the vector-based approach invalidates this property, nor could it, because the guarantee is true for any safe Rust program. Unfortunately, nothing in the Rust type system ensures that the *correct* memory will be accessed, because correctness is a program-specific property.

We define *logical memory safety* in the context of a program that uses pseudo-pointers to mean that every access to an object via a reference (such as a vector index) accesses the same object that the reference was created to refer to. This property provides essentially the same guarantee that a borrow does: a borrow also ensures that the pointed-to object remains accessible, and remains *the same object*, during the lifetime of the borrow.

3.4 Maintaining Safety: Deferred Borrows with Irrevocable Binding

We can now concisely state the goal of this work: we wish to provide logical memory safety for object references without the limitations of borrows, i.e., in a way that retains the flexibility of pseudo-pointers.

We build on pseudo-pointers, because they resolve the conflicting-borrows problem right away. This is because they *defer* the actual borrow of the container until the referred-to object is used. In other words, `nodes[node_idx]` borrows `nodes` mutably, but only for as long as the particular operation on this node. This property is the origin of our term *deferred borrow*, which we expand further in §5.

To make pseudo-pointers logically memory-safe, let us consider what would be needed: in concrete terms, for the vector-based approach, we must ensure that an object's index in the vector is constant once added (by only appending to the vector). In addition, we must ensure that an index created for some particular vector is only ever used to access that vector, and not another, even if their static types match.

We can provide the append-only property at the library API level by encoding the invariant into the types: for example, provide a `.to_append_only()` method on `Vec<T>` that consumes the `Vec` (as we can do with linear types!) and returns an `AppendOnlyVec<T>`.

Ensuring that indices are only used with a *particular* vector, however, is more challenging. The existing Rust type system cannot encode this restriction. We must somehow *irrevocably bind* an index with a vector object, and require this binding when the access `node[node_idx]` occurs. In the following section, we now show how this can be done with *static path-dependent types*.

4 Static Path-Dependent Types in Rust

Our key contribution to the core Rust language that enables deferred borrows is the *static path-dependent type*. Path-dependent types have been proposed previously in a dynamic context [7, 14, 2], e.g. in Scala: in that context, a type is an element of some class, and each class instance has a *different* type (i.e., the dynamic object identity is part of the type). In contrast, our path-dependent types are static. This is an extension to an ordinary type that ties a value to *another particular value in-scope*, by its access path (local variable plus struct field(s)).

We define a type `T/x` to be a subtype of `T` that has the *path* `x`. Intuitively, a path can refer to any *storage place* that the borrow-checker tracks: e.g., a local variable binding or a subfield of one. The path can then be used to constrain function arguments so that, e.g., a

value of type `T/x` can only be combined with exactly the value in local `x`. This provides the necessary conditions for logical memory safety of deferred-borrow smart pointer objects, as each can be paired with the container from which it must eventually borrow.

4.1 Types with Static Dependent Paths

We augment the type system of Rust so that any type τ can be annotated with a path p to form type τ/p. The path describes a *storage slot*, or *place* in the terminology of Weiss et al. [21]'s formulation. Concretely, this is a *root binding* optionally extended with a path of struct fields. A root binding is (i) a local immutable variable binding in scope, within a function body; (ii) a formal parameter index, within a function type; or (iii) the *self* root, within a struct. In each of these contexts, the path corresponds to a fixed location that will hold the same value until it goes out of scope or is moved out of.

To communicate our intended semantics, we sketch a set of definitions and inference rules in Fig. 5. This scheme is built on top of that of Weiss et al. [21] for the *Oxide* language, which describes a small core language that captures Rust's ownership and borrowing semantics. We first extend the type unification and subtyping judgments with rules to allow the path annotations on types to flow through the program. We extend the expression typing judgment to weaken path-dependent types when the corresponding path is dropped or moved out of, or when a mutable binding is modified: formally, when the place π is removed from the outgoing typing context Γ in T-Move', we weaken any type τ/π in the typing context to simply τ. Finally, we modify the typing rules for function application and struct-field projection to translate the *roots* between the three domains (locals, function parameters, or a struct's *self*). This allows struct field types and function parameter types to naturally refer to "neighboring" values, as we will see below.

4.2 Static Path-Dependent Types in Rust: Syntax and Examples

We now show how static path-dependent types might appear in several Rust snippets to give a flavor of their integration into the language. First, consider that we have a struct definition:

```
struct S<'a> { c: &'a Container, r: Option<ContainerRef/self.c> }
```

This struct holds a borrow (over whose lifetime it is parameterized) to some `Container` type that we have presumably defined elsewhere. It also holds a value of type `ContainerRef`. In this struct definition, however, we have augmented this type with a path `self.c`. In a struct field type context, any path on a path-dependent type must start with the `self` path prefix, and this prefix indicates that the following path refers to a neighboring field in the same struct instance. Here, whatever `ContainerRef` that is stored in `r` of a given instance will be irrevocably tied via its path to the container in `c`.

To use this value, we might write a function like the following:

```
fn foo() {
  let c = Container::new_with_contents();
  let mut s = S { c, r: None };
  let r = s.c.deferred_index(/* index = */ 42);
  s.r = Some(r);

  let elem = s.c.defborrow(s.r.unwrap());
  // ...
}
```

Static Path-Dependent Types

Note: This formalization extends that of Oxide [21], which captures the Rust ownership and borrowing system in a small core language. We omit a repetition of most of Oxide's rules and definitions, and show only those relevant to tagged types below.

e	*Expression*	x	*Variable/Identifier*	p	*Path*
τ	*Type*	π	*Storage Place*	i, n	*Naturals*

Language Extensions

τ	::=	$\ldots \mid \tau/p$	*Path-dependent type*
p	::=	π	*Storage-place path (in expression context)*
		$\mid \text{i}.x_1.\ldots.x_n$	*Formal parameter path (in function type)*
		$\mid \text{self}.x_1.\ldots.x_n$	*Struct path (in struct field type)*

$\boxed{T; \Sigma; \Delta; \Gamma \vdash \pi : \tau \Rightarrow \Gamma'}$ *(Typing judgment)*

$$\text{T-Move'}\ \frac{\Gamma \vdash_{\text{uniq}} \pi : \tau^s \qquad \tau_s \text{ noncopyable}}{\Sigma; \Delta; \Gamma \vdash \pi : \tau^s \Rightarrow \text{filter}(\Gamma - \pi, \pi)}$$

Similar adaptation to T-Let, omitted for brevity: filter path x from Γ in the out-context of let $x \ldots$.

$$\text{T-App'}\ \frac{\begin{array}{l}\Sigma; \Delta; \Gamma \vdash e_f : (\tau_1^s, \ldots, \tau_n^s) \to \tau_f^s \Rightarrow \Gamma' \\ \Sigma; \Delta; \Gamma_{i-1} \vdash e_i : \text{funcpath}(\tau_i^s, (e_1, \ldots, e_n)) \Rightarrow \Gamma_i,\ 1 \le i \le n\end{array}}{\Sigma; \Delta; \Gamma \vdash e_f(e_1, \ldots, e_n) : \tau_f^s \Rightarrow \Gamma_n}$$

places-typ meta-function modified so that struct-field types are filtered through structpath; if $x : \{x_1 : \tau/self.\pi, \ldots\} \in \Gamma$, then $x.x_1 : x.\pi \in Gamma$.

$$\begin{array}{lcl}\text{filter}(\Gamma, \pi) & = & \Gamma[\tau/\pi.\ldots \mapsto \cdot] \\ \text{funcpath}(\tau, (e_1, \ldots)) & = & \tau[\tau'/i.\pi \mapsto x.\pi \text{ if } e_i = x] \\ \text{structpath}(\tau, \pi) & = & \tau[\tau'/self.\pi' \mapsto \pi.\pi']\end{array}$$

$\boxed{\tau_1 \sim \tau_2 \Rightarrow \tau}$ *(Type unification)*

$$\text{U-Path}\ \frac{\tau_1 \sim \tau_2 \Rightarrow \tau}{\tau_1/p \sim \tau_2/p \Rightarrow \tau/p}$$

$\boxed{\Sigma \vdash \tau_1 <: \tau_2 \rightsquigarrow \delta}$ *(Subtyping)*

$$\text{S-Path}\ \frac{\Sigma \vdash \tau_1 <: \tau_2 \rightsquigarrow \delta}{\Sigma \vdash \tau_1/p <: \tau_2 \rightsquigarrow \delta}$$

Figure 5 Static path-dependent types: definitions and rules as an extension of the Oxide formalization [21].

In this function, we create an instance of `S`, and initially fill its field `c` with a new `Container`. Let us say that the method `Container::deferred_index` returns a value of type `ContainerRef/s.c` (we will see how such a function can be declared below). Then `r` has this type as well. This type unifies with the type of field `s.r` because `T/self.c` in struct-field type context maps to `T/s.c` for the struct at the particular place `s`.

How do we pass these path-qualified values between functions? Analogously to our approach for struct-field types, we allow *parameter* roots in parameter (and return value) types. Thus the type of one parameter can be bound to the value of another parameter. For example, we might define `deferred_borrow` above as follows:

```
impl Container {
  fn deferred_index(&self, index: usize) -> ContainerRef/0 {
    ContainerRef { /* ... */ }
  }
}
```

The return type `ContainerRef/0` refers to the 0-th parameter, in this case `self`: thus, the returned `ContainerRef` is bound to the passed-in `Container`.

Paths can always be stripped from types, but cannot be added in ordinary value dataflow, due to the subtyping rule S-Path which specifies that $\tau/p <: \tau$. A value acquires a path at construction time: e.g., above, the return value is constructed with the struct-literal form for `ContainerRef`, and implicitly has type `ContainerRef/0`.

Finally, the function `defborrow` can place a path on an inbound parameter type, requiring that parameter to have the path in order for the function call to typecheck. In this running example, `defborrow()` requires the passed-in `ContainerRef` to be associated with the `self Container` in order for the call to succeed:

```
impl Container {
  fn defborrow(&self, r: ContainerRef/0) -> &Elem {
    // ...
  }
}
```

In summary, we see that static path-dependent types qualify arbitrary types with *storage paths*. The paths available depend on the context. For an expression in a function body, these are precisely the local storage paths, rooted at local variables. For a struct definition's fields, these are paths rooted at `self` and a field name in the same struct. Finally, for function parameter and return types, these are the numbered roots `0` to `n-1` referring to the n parameters of the function.

4.3 Correctness: Value-Correspondence Lemma

Building on the above intuition for "binding" one value to another, we can now describe the condition that path-dependent types ensure:

▶ **Lemma 1.** *If a storage place π is in scope with any type and with value v at a program point defined by typing environment Γ, and another storage place π' is in scope with type τ/π and with value v' at the same program point, then for any dynamic execution, the value v' is stored in a place of type $\tau' <: \tau/\pi$ only as long as the value v remains in storage place π.*

Proof sketch. Follows from the dynamic semantics. As long as the binding π it remains in scope, the value remains v (because bindings in Oxide are immutable). The rules T-Let and T-Move in the Oxide formalization [21] on which our formalization is built ensure that π is dropped from the typing context Γ when it is moved out of or when it falls out of scope. Our modifications to these rules apply our meta-function filter() to the typing context, weakening any type τ/π to simply τ. ◀

4.4 Generics and Path Parameters

For brevity, we have not included a formalization of *generic path parameters*, though we describe them informally here. In order to allow data structures to contain path-dependent types that refer to paths *outside* of the `struct` in question (i.e., outside of the `self` path), we allow generic parameters to provide paths. Field types are known at instantiation time according to the actual path provided as a parameter, as for type and lifetime generics.

4.5 Alias Analysis and Type-Tag Propagation

We note briefly that this type system can be seen as the combination of path-dependent types (in the dynamic sense of earlier work [7, 14, 2]) with a static must-alias analysis, so that all dynamic checks are replaced with static reasoning at type-check time. The rules for unifying dependent paths attached to types essentially form a very simple path-based intraprocedural must-alias analysis. When seen this way, one can also imagine several precision enhancements by adopting more advanced static must-alias techniques, as in e.g. Kastrinis et al. [10].

5 Deferred Borrows

In §3, we saw how the existing Rust borrow system can be inflexible in the face of some common program design patterns. We described the foundation of a solution in §3.4, describing how one might combine an index-based scheme (what we call *pseudo-pointers*) with some sort of strong *binding* between these indices and the particular container instances to which they refer in order to attain *logical memory safety*, a stronger property than Rust's language-level memory safety. Now that we have introduced static path-dependent types, we can show how to use these to achieve exactly this goal.

Our *key insight* in this work is that we can alleviate the inflexibility of the borrow system, caused by the conflict of multiple outstanding borrows, by avoiding a borrow until the point of use while retaining safety in other ways. This is the origin of the name *deferred borrow*.

A deferred borrow is an *API concept* that uses path-dependent types to provide a more flexible interface to a container. As we described in §2.3, borrows of container elements perform proxy borrows (borrows with the same lifetime) of the *entire* container; this approximation is necessary to retain memory safety because the borrow checker cannot reason about abstract index spaces or storage locations such as vector indices. The idea of a deferred borrow is to:

1. Freeze the *existence* of the reference element (e.g., disallow element deletion); and
2. Return some state that can allow a lookup and true borrow of the element later, even if the internal storage of the container has been rearranged in the meantime. This later lookup performs a borrow of the entire container, as an ordinary element reference does.
3. Tie this state to the container with a path-dependent type.

This strategy provides all the same guarantees as a true Rust borrow. First, while the true borrow is outstanding at the point of use, we have memory safety simply by reduction to the usual Rust container access idiom: the entire container is borrowed for the duration of the element access. However, for the *entire existence of the deferred-borrow element reference*, we also have a substantially similar guarantee: (i) the container is put into a state so that the element cannot disappear; (ii) the element reference is tied to this particular container; so (iii) when the deferred borrow is converted into a real borrow, the borrow will refer to exactly the desired element.

It is important to note that all of these properties were provided by a *true* borrow simply because a true borrow performs a borrow of the entire container for the duration of the element access. In contrast, a deferred borrow *synthesizes* this same guarantee from pieces. By doing so, without holding an outstanding borrow on the container, many of the unnecessary restrictions are avoided. In particular, while a deferred borrow exists, any of the container elements can be accessed or mutated, even if the outstanding deferred borrow also allows mutable access; and, if the container is implemented properly, the program can also append new elements to the container. In other words, we separate a "bookmark" phase, in which an element is identified, from a "use" phase, in which it is exposed for access.

5.1 Definition and Correctness Conditions

A deferred borrow idiom properly implemented by a container grants *logical memory safety*, as long as the container upholds the contract: a deferred borrow object returned by a lookup operation *must* convert into the same borrow at any future point if dereferenced with the same container object. Combining the value-correspondence lemma of §4.3 with this contract, we have the full guarantee to the user of the container API.

In slightly more precise terms, we can define a deferred-borrow implementation as an API pattern with the following conditions. Given a mutable container datatype that contains values of type `V` indexed by keys of type `K` and provides the following operations:

- insert(c, k, v), which inserts a new *storage slot* dynamically into container c at abstract address, or key, k with initial value v,
- remove(c, k), which removes a key k from c,
- immutable_borrow(c, k) which returns an immutable borrow (pointer) to the storage slot for k, and has lifetime constraints such that c is immutably borrowed as long as the returned borrow is in scope,
- mutable_borrow(c, k) which likewise returns a mutable borrow tied to a mutable borrow of the container,

and the usual key-value map semantics (the value seen under the pointer returned by immutable_borrow or mutable_borrow is that value last stored to a pointer fetched by mutable_borrow for that key), a deferred-borrow pattern is implemented with the operations:

- deferred_get(c, k), which returns some abstract reference type r, tied to the container with a path-dependent type,
- deferred_borrow(c, r), which given any r returned by deferred_get(c, k) at *any* point in the past with no interceding remove(c, k) operation, returns an immutable borrow to the storage slot *currently* backing k, with lifetime tied to a full-container borrow as above, and
- deferred_get_mut(c, k), likewise but with mutable borrows.

The most important aspect of a deferred-borrow implementation is the property that a reference *remains valid* even if the container is later mutated. This implies that the abstract reference type must somehow keep a *logical* notion of storage-slot address, rather than a true pointer, unless the implementation can guarantee that the storage layout will never change. It is exactly this property that allows us to retain logical memory safety without freezing the container completely with an ordinary borrow.

The correctness of the deferred-borrow concept arises largely by definition from the above, in concert with the value-correspondence lemma of §4.3. By using path-dependent types on the interface above, a deferred-borrow implementation can statically ensure that a reference r produced from a particular container c is only ever used with that container. The application in concrete type terms is simple, and will be shown in the next section.

Table 1 Examples of containers and associated element-reference (deferred borrow) types, with implementation strategies. A library that provides deferred-borrow types may preserve memory safety in several ways, trading off allowed mutation with the amount of state that is kept in the reference and the amount of (deferred) work to convert it into a true borrow.

		Deferred-Borrow Element Reference		
Base Container	*Derived Type*	*Reference State*	*Deferred Work*	*Borrow Result*
`Vec<T>`	`Vec`	index	bounds check; base plus index	`Option<&T>`
	`AppendOnlyVec`	index	base plus index	`&T`
	`FrozenVec`	true pointer	none	`&T`
`HashMap<K, V>`	`HashMap`	key, lookup hint	hash-table lookup	`Option<&T>`
	`AppendOnlyHashMap`	key, lookup hint	hash-table lookup	`&T`
	`FrozenHashMap`	true pointer	none	`&T`

5.2 Containers and Deferred-Borrow APIs

Let us now see how deferred borrows can be implemented concretely. First, we define a Rust trait that generalizes over any state that, in association with some container, can be converted into a borrow of an element in that container:

```
trait DefBorrow<T, Container> {
  fn def_borrow<'a>(&self/1, cont: &'a Container) -> &'a T;
}

trait DefBorrowMut<T, Container> {
  fn def_borrow_mut<'a>(&self/1, cont: &'a mut Container) -> &'a mut T;
}
```

These definitions simply mean that a particular "deferred borrow" object is associated with a *particular* container, and if a method on the deferred-borrow state is invoked with that container (invoking it with any other container instance is a type error), it will return a reference to (borrow of) the element. This borrow creates a true borrow of the container, but only as long as this particular access needs it; e.g., a program may hold many mutable deferred borrows, some of them aliasing, and dereference each in turn to perform a single mutation before dropping the true borrow.

How might this interface be implemented? Let us consider several real container types: the `Vec` (vector) and `HashMap` (key-value map built with a hashtable). Table 1 summarizes several options for each.

Recall that we need to *freeze the existence of the referred-to element* when the deferred borrow is created. A container whose element index space can dynamically shrink and grow (true for both `Vec` and `HashMap`) might do so in one of several ways. it could freeze its structure entirely, not allowing addition or removal, or it could still allow addition, simply prohibiting element removal.

In the first case, if any insertion or removal is prohibited, any actual pointer that refers to the internal storage for a particular element should remain valid, because the container will not need to reallocate to grow, and so the deferred-borrow object can carry an actual pointer. The abstraction is thus erased at runtime, and serves only to translate accesses to one of a large collection of true pointers into borrows of the container to ensure mutual exclusion.

```
impl Vec<T> {
  // ...
  pub fn to_append_only(self) -> AppendOnlyVec<T> {
    AppendOnlyVec { vec: self }
  }
}

pub struct AppendOnlyVec<T> {
  vec: Vec<T>,
}

impl AppendOnlyVec<T> {
  pub fn deferred(&self, index: usize) -> AppendOnlyVecRef<T>/0 {
    AppendOnlyVecRef { index }
  }

  pub fn push(&mut self, t: T) {
    self.vec.push(t);
  }
}

pub struct AppendOnlyVecRef<T> {
  index: usize,
  _phantom: PhantomData<T>,  // keep the Rust type-checker happy by using T.
}

impl DefBorrow<T, AppendOnlyVec<T>> for AppendOnlyVecRef<T> {
  fn def_borrow<'a>(&self/1, cont: &'a AppendOnlyVec<T>) -> &'a T {
    &cont.vec[self.index]
  }
}
```

Figure 6 Excerpt of the implementation for `AppendOnlyVec`, one variant of a vector that implements deferred borrows. This variant ensures the existence of elements that have outstanding deferred borrows simply by disallowing element removals. Deferred borrows are just vector indices internally; type erasure makes this approach equivalent to "pseudo-pointers" at runtime, but it is more type-safe.

In the second case, however, the container is still allowed to grow by insertion, and so it must compute the element location only when the deferred borrow is converted into a true borrow. In this case, the deferred borrow will contain the *logical* element address – e.g., a vector index. This essentially emulates the pseudo-pointer idiom, but with more type-level safety.

Finally, if we extend the notion of a deferred borrow to one that can return an *optional* borrow (i.e., an element borrow or `None`), we can allow even a standard container with insertions and removals to produce deferred borrows. In this case, we also must retain a logical address only in the deferred-borrow object, and perform the lookup late.

Referring again to Table 1, these container variants can be seen as a form of typestate encoding the restrictions on container mutations that are allowed. The library user can convert containers only to more constrained variants. Each base type has `.to_append_only()` and `.to_frozen()` methods that consume the original (i.e., have non-borrowed `self` arguments) and return the appropriate constrained type, and the append-only type has `.to_frozen()` as well.

To provide a complete example, we show a simple implementation of `AppendOnlyVec`, the variant of the vector that allows insertions but not removals, in Fig. 6. The main highlights are that deferred-borrow objects reduce simply to vector indices at runtime (there need not

be any dynamic checks that the "correct" vector is accessed, because the path-dependent types ensure that statically), and that the vector indexing operation at true-borrow time can be assured of success because the underlying vector is not allowed to shrink after any deferred-borrow objects are produced.

Note that these types are not exhaustive by any means: one can imagine several other variants that make still different tradeoffs. For example, a container might allow individual element deletions yet still provide a deferred-borrow type that returns a `&T` rather than `Option<&T>` by dynamically tracking which elements have outstanding element references. Internal data structure design may also facilitate the creation of more efficient element references with less deferred lookup work: for example, a container might allocate stable memory storage (via, e.g., `Box<T>`) for each element in order to provide element references that just store pointers even while the container is allowed to grow, or might lazily move elements for which references are created to such indirected storage. This section's proposed types are merely the simplest design points in a large space enabled by a flexible language mechanism.

5.3 Auto-Dereferencing for Syntactic Sugar

As one final ergonomic improvement, we note that by including the access path to the associated container, a deferred-borrow value contains all the information necessary to convert it to a true borrow automatically. The Rust language today contains a feature known as "auto-dereferencing" wherein the compiler inserts calls to the `deref()` or `deref_mut()` methods on smart pointer types when necessary. (This is similar to e.g. the use of operator overrides in C++ to implement smart pointers.) This allows transparent implementation of borrow/pointer-like values by library authors. We propose a modification to this desugaring step that, for a value `t` of type `T/a` that implements the `DefBorrow` or `DefBorrowMut` trait, invokes the deferred borrow `t.def_borrow(&a)` or `t.def_borrow_mut(&mut a)` as appropriate. This will make deferred-borrow references as ergonomic as true borrows in most circumstances, without additional user intervention.

5.4 Chained Deferred Borrows

We note that there is nothing preventing a deferred borrow's path from referring to *another* deferred borrow object. In particular, consider the case where one vector container contains vectors as elements. A deferred borrow reference `p1` to an element of the outer vector `v1` might have type `Ref/v1`, and could in turn be used to produce a deferred borrow reference `p2`, which might have type `Ref/p1`. Auto-dereferencing could then chain *two* true borrows at the time of use, so that a write to `p2.a` becomes:

```
let v1: FrozenVec<FrozenVec<u32>> = ...;
let p1 = v1.deferred(0);  // type FrozenVecRef/v1
let p2 = p1.deferred(0);  // type FrozenVecRef/p1

// `p2.a = x` becomes:
let tmp1 = p1.def_borrow_mut(&mut v1);  // type &mut FrozenVec<u32>, lifetime <: `v1`
let tmp2 = tmp1.def_borrow_mut(tmp1);   // type &mut u32, lifetime <: `tmp1`
*tmp2 = x;
// tmp2 and tmp1 now out of scope; mutable borrow on `v1` ends.
```

6 Deferred-Borrow Prototype: Emulating Paths in Stable Rust

In order to evaluate the utility of deferred borrows, we implemented a prototype library of container types that provide deferred-borrow element references. Ideally, such a library would make use of true path-dependent types, as we described in §4. For expediency of implementation and experimentation, we instead chose to emulate path-dependent types with type-tagging, which is a strategy that works in stable Rust today (§6.1). We then illustrate several examples of our container library using this strategy (§6.2).

6.1 Emulating Path-Dependent Types with Type Tagging

Because a path-dependent type is a sort of dependent type – that is, because the type depends on a value in the program – we cannot implement our proposed system as written in today's stable Rust language. Instead, we can *emulate* many of the type-safety benefits of path-dependent types, albeit without the convenience of auto-dereferences and with slightly more syntactic noise, by (i) *tagging* the container and its references with an extra type parameter, such that the "tag" type must match for a deference to work, and then (ii) using a *unique type for every container allocation site*. This strategy is less powerful than a true path-dependent type because it will let different objects from the same allocation site intermingle references, but is sufficient to understand the annotation burden and verify that the general approach can work.[3]

To understand this approach, consider first the following snippet using the "true" library design with path-dependent types:

```
fn main() {
  let v: AppendOnlyVec<T> = ...;
  let ref1: AppendOnlyVecRef<T>/v = v.deferred(i);
  let w: AppendOnlyVec<T> = ...;
  let ref2: AppendOnlyVecRef<T>/w = w.deferred(j);
  *ref1 = ...;   // auto-derefs to: *ref1.def_borrow_mut(&mut v) = ...;

  // This is a type error (we used `w`, not `v`, but ref1 is of type `.../v`)
  // *ref1.def_borrow_mut(&mut w) = ...;
}
```

Instead, we define our container type with an extra type parameter `Tag`: hence, the container type becomes `AppendOnlyVec<T, Tag>`, and its deferred-borrow references are of type `AppendOnlyVecRef<T, Tag>`. Thus the example becomes:

```
fn main() {
  struct Tag1 {}  // We can wrap this in a macro! (see below)
  let v: AppendOnlyVec<T, Tag1> = ...;
  let ref1: AppendOnlyVec<T, Tag1> = v.deferred(i);

  struct Tag2 {}
  let w: AppendOnlyVec<T, Tag2> = ...;
  let ref2: AppendOnlyVec<T, Tag2> = w.deferred(j);
  *ref1.def_borrow_mut(&mut v) = ...;
}
```

[3] Note that this is subtly different than ownership-type approaches that encode ownership as a generic parameter, such as Potanin et al. [17]: while that work's system enforces particular object instances as owners in a principled way, our prototype approach simply ties the path-dependent type to a *static allocation site*, which may produce many object instances. The only advantage of our scheme is that it can be written in Rust's existing type system.

Note that this tag-type approach is *not* as strict as a true static path-dependent type, even as our example illustrates that the tag types can distinguish references from ‘v‘ and ‘w‘. Consider the case where a container is allocated within a loop: each instance, on each iteration, must have the same type, but logical memory safety requires disallowing a deferred-borrow from one to be used in a dereference with another.

However, this prototype has some value: it lets us see an approximation of the annotation burden, in that deferred-borrow types `Ref<T>/v` become `Ref<T, V>`. This is enough to evaluate the feasibility of large-program refactorings.

6.2 Macros for Tag Types

In order to make this strategy feasible and ergonomic enough for reasonable prototype use, we define several macros alongside our library of container types so that (i) container definition (with tag type) and (ii) deferred-reference access, which would become an invisible auto-deref with true path-dependent types, are both relatively simple.

First, we define a macro that defines a new empty tag type and parameterizes a container constructor:

```
fn main() {
  let v = vec![1,2,3,4];
  let mut v = freeze!(FrozenVec, v);
```

This expands to a struct-type definition inside a new scope (hence invisible to the rest of the program) and a constructor invocation parameterized on this tag type. The multiple instances that are produced by this expression are not distinguished by the type system as they would be with path-dependent types, but they are distinguished from other containers in the program that happen to coincide in the element type. (This is thus a form of allocation-site newtype idiom.)

Then, we define a macro that provides a short form for the deferred borrow itself, allowing the above dereferences to become simply:

```
fn ref<Tag>(v: &mut FrozenVec<u32, Tag>, elem: FrozenVecRef<u32, Tag>) {
  let value = *d!(v, elem);
  *dmut!(v, elem) += 1;
}
```

7 Qualitative Evaluation: Newly Possible Programming Patterns

We briefly describe several programming patterns that are possible with deferred borrows but not while restricted to true borrows, and discuss how API design considerations change when more flexible object references are possible.

7.1 Use Case #1: Graph Library

Consider the use-case of a general graph library: a top-level `Graph` object owns many `Node` instances, and each node contains out-edges to other nodes, with some `Edge` data attached to each out-edge. A straightforward Rust implementation, without deferred borrows, would simply hold `Node`s in an array, and use indices to refer to them from other nodes:

```
pub struct Graph {
  nodes: Vec<Node>,
}

pub type NodeIndex = usize;
pub struct Node {
  out_edges: Vec<(Edge, NodeIndex)>,
}

pub type EdgeIndex = usize;
```

We can then provide an API that allows for insertion of nodes and edges, and allows for accessing or mutating the data at each node or edge:

```
impl Graph {
  pub fn add_node(&mut self, node: Node) -> NodeIndex { ... }
  pub fn add_edge(&mut self, from: NodeIndex, to: NodeIndex, data: Edge) { ... }

  pub fn node<'a>(&'a self, node: NodeIndex) -> &'a Node { ... }
  pub fn node_mut<'a>(&'a mut self, node: NodeIndex) -> &'a mut Node { ... }
  pub fn edge<'a>(&'a self, node: NodeIndex, edge: EdgeIndex) -> &'a Edge { ... }
  pub fn edge_mut<'a>(&'a mut self, node: NodeIndex, edge: EdgeIndex)
    -> &'a mut Edge { ... }

  // ...
}
```

The two primary issues with such an API, as we have described already in motivating our approach, are (i) verbosity in use, due to the need to handle indices differently than native pointers/borrows, and (ii) logical unsafety because indices may be forged by the user, or erroneously taken from other contexts (e.g., another graph).

Thus, the following code is valid, but produces an unexpected result, because the programmer mixes indices from different domains (two different graphs) and erroneously uses an index in the wrong domain (here, a node index for graph `g1` used to index into `g2`'s nodes):

```
fn graphs_are_isomorphic(g1: &Graph, g2: &Graph) -> Mapping {
  'l: for mapping in generate_all_mappings() {  // brute-force
    for n1 in g1.node_ids() {
      let n2 = mapping.map_node(n1);
      //            TYPO / logical error --v
      if !nodes_are_isomorphic(g1, g2, n1, n1, &mapping) {
        continue 'l;
      }
    }
    return mapping;
  }
}
```

In addition, accessing node data is cumbersome: each access must be written as `g.nodes[i]` rather than simply `p` (where `p` is a borrow). Note that the API user cannot simply take and save multiple borrows: if any borrow is live (mutable or immutable), no other mutable borrow can be created or used. (This may not be an issue for code that simply queries a data structure, though the lifetime annotations can still be difficult to manage. However it is surely an issue for any code that updates a data structure while holding multiple pointers into its inner structure.) As a result of this limitation, typical function bodies might look as follows:

```
fn nodes_are_isomorphic(g1: &Graph, g2: &Graph, n1: NodeIndex, n2: NodeIndex,
                        mapping: &Mapping) {
  if !data_is_equal(&g1.nodes[n1].data, &g2.nodes[n2].data) {
    return false;
  }
  // ...
}

fn mutate_nodes<F: Fn(&mut Node)>(g: &mut Graph, root: NodeIndex, mutate: F) {
  mutate(&mut g.nodes[root]);
  for i in 0..g.nodes[root].neighbor_count() {
    let neighbor = g.nodes[root].get_neighbor(i);
    if /* ... !visited(neighbor) ... */ {
      mutate_nodes(g, neighbor, mutate);
    }
  }
}
```

In contrast, a graph library that makes use of the deferred borrow pattern and the path-dependent types extension to Rust could define an API as follows:

```
impl Graph {
  pub fn add_node(&mut self, node: Node) -> NodeRef/0 { ... }
  // ...
}

impl DefBorrow<Node, Graph> for NodeRef {
  fn def_borrow<'a>(&self/a, g: &'a Graph) -> &'a Node { ... }
}
impl DefBorrowMut<Node, Graph> for NodeRef {
  fn def_borrow_mut<'a>(&mut self/a, g: &'a Graph) -> &'a mut Node { ... }
}
```

Given this API, the above functions could be rewritten as below, using generic path parameters (§4.4) to the `Mapping` object (definition omitted here) so that it can accept and produce indices associated with each graph object:

```
fn graphs_are_isomorphic(g1: &Graph, g2: &Graph) -> Mapping</0, /1> {
  'l: for mapping in generate_all_mappings() {  // brute-force
    for n1 in g1.nodes() {           // n1 is of type: NodeRef/g1
      let n2 = mapping.map_node(n1); // n2 is of type: NodeRef/g2
      // typecheck error caught statically ---------v
      if !nodes_are_isomorphic(g1.node(n1), g2.node(n1), &mapping) {
        continue 'l;
      }
    }
    return mapping;
  }
}
```

Furthermore, if auto-dereference behavior is implemented for the deferred-borrow traits, then node accesses become much more convenient:

```
fn nodes_are_isomorphic(g1: &Graph, g2: &Graph, n1: NodeRef/0, n2: NodeRef/1,
                        mapping: &Mapping</0, /1>) {
  if !data_is_equal(&n1.data, &n2.data) {
    return false;
  }
  // ...
}
```

```
fn mutate_nodes<F: Fn(&mut Node)>(g: &mut Graph, root: NodeRef/0, mutate: F) {
  mutate(&mut *root);  // auto-deref.
  for i in 0..root.neighbor_count() {
    let neighbor = root.get_neighbor(i);
    if /* ... !visited(neighbor) ... */ {
      mutate_nodes(g, neighbor, mutate);
    }
  }
}
```

Note that the *efficiency* of these node accesses can be adjusted in a tradeoff with graph-mutation flexibility. If the user is willing to accept that the graph is frozen at a certain node count (many graph algorithms have this property: they do not mutate the graph topology, only data at each node/edge), then nodes can be stored in a `FrozenVec`, and a `NodeRef` is exactly as efficient as a true pointer, because it compiles down to exactly that. In contrast, if the programmer desires to allow graph expansion, an `AppendOnlyVec` could be used. If deletions are also desired, a more complex reference type might be used that dynamically prevents deletions of nodes with outstanding references. In short, the deferred-borrow idiom allows code to be *generic to the particular reference/addressing scheme*.

Finally, we note that the lessons we have learned from this example also apply to more heterogeneous or general data structures with arbitrary object-to-object linkage. For example, a tree with parent pointers (thus creating cycles between parent and children) or cross-links, could store tree nodes in an array and use deferred-borrow references throughout.

7.2 Use Case #2: Entity-Component Systems

Many programs operate on a large number of objects that fall into a small number of categories, and must hold references to these objects throughout their data structures. A common pattern is the *entity-component system*: the program has some global context with a few arrays or dynamically-sized vectors, one per object type; and a reference to an object of type `T` is simply an index into the vector of all `T`s. This provides efficiency advantages by allowing for more compact references (e.g., 32-bit indices instead of 64-bit pointers), more compact heap layout, and more efficient access patterns (i.e., streaming through an array in order rather than pointer-chasing). As such, the pattern is often used in high-performance scenarios such as game programming.

If we use deferred-borrow references for every entity in the system, we again have the guarantee against incorrect use of indices: e.g., an index into the array of all `T` objects cannot be used to index into the array of `U` objects instead. The path-dependent types are particularly ergonomic in this use pattern, however, because it is already the case that nearly every function will be passed a top-level "context" that allows access to the entity arrays; the reference types simply have types with paths starting at that context, and are thus automatically usable anywhere in the program without further plumbing. For example:

```
fn f(ctx: &mut Ctx, t: TRef/0, u: URef/0) {
  t.do_stuff(ctx);
  u.mutate(ctx);
  t.operate_with(ctx, u);
}
```

As a real-world test of this hypothesis, we took a small program, a microarchitectural CPU simulator, consisting of around 6000 lines of Rust. The simulator is written approximately in the style described above: references to major components of the simulated system (CPU

cores, caches, memory banks, etc.) are all by IDs that are indices into simulator-wide arrays. We adapted the system so that, instead, it would use deferred borrows to refer to CPU cores. The diff for this change (using the tag-type-based prototype library described in §6) was approximately 200 lines, almost all of which were in function signatures or field types within `struct` definitions.

7.3 Use Case #3: Logical Safety in an Array-based Algorithm

We note that static path-dependent types have uses outside of the deferred-borrow pattern. In fact, they are applicable as a means of tying references or handles to a particular context wherever such handles occur.

To see one such example, let us consider the case of an algorithm that operates on arrays of data, and manipulates indices into those arrays. It is often the case that such code is error-prone to write: the programmer might confuse which index corresponds to which array.

Let us say, for example, that we wish to develop an edit-distance algorithm that takes two strings (arrays of characters):

```
fn edit_distance(s1: &[u8], s2: &[u8]) -> usize {
  for i in 0..s1.len() {
    for j in 0..s2.len() {
      do_stuff(s1[i], s2[j]);  // correct
      do_stuff(s1[i], s2[i]);  // logical error!
    }
  }
}
```

One could make use of static path-dependent types to make such a logical error impossible, by defining a type `SafeSlice<T>` that wraps a `&[T]` and provides deferred-borrow-like element references:

```
fn edit_distance(s1: SafeSlice<u8>, s2: SafeSlice<u8>) -> usize {
  for i in s1.indices() {
    for j in s2.indices() {
      do_stuff(s1, s2, i, j);
      do_stuff(s1, s2, i, i);  // caught at compile time!
    }
  }
}
```

In fact, path-dependent types are far more powerful than our examples have demonstrated so far: they serve, in brief, as a way to ensure *unforgeable values* that are produced and consumed by some opaque manager object and usable only with that object. We believe this type-system primitive would have many other use cases in a systems programming language such as Rust.

8 Related Work

Many prior works have explored the tradeoff space in type-system approaches to sound, usable memory-safety with useful aliasing guarantees that allow for parallelism. While we build on the particular programming language Rust in this work, Rust borrows from a long line of work on *ownership-based* and *region-based* memory-management. In addition, we adapt *path-dependent types*, which are a limited form of generalized dependent types, to provide type safety. To our knowledge, this is the first work to combine path-dependent types with a region-based ownership or borrowing system to provide static safety guarantees for a more flexible form of "borrowed" ownership.

Ownership tracking originated with Ownership Types [5] and spawned a long line of followup works [4, 3, 1, 6, 9, 15] to encode "contexts" or ownership domains, reason about split or fractional ownership and varying levels of access permission, allow "ombudsmen" that provide a safe external reference to internal state, and other approaches. In general, these past works begin with a highly restricted system – objects form a strict ownership tree, and access to an object is possible only by its owner – and then systematically relax that constraint to allow for common programming idioms to be expressed.

Rust's borrowing system adapts *region-based memory management* ideas from Cyclone [8] to provide safe borrows of subtrees of the ownership tree. The borrows have constrained lifetimes, based on lexical regions, and the lifetimes of borrows of the same owned object are mutually disjoint. Abstractly, borrowing can also be seen as a form of permission system; effects and permissions have been widely studied as a means to provide memory safety and allow for deterministic parallel processing [9, 12].

Path-dependent types are a subset of dependent types [23], and have been extensively studied in the context of object-oriented languages, starting with Ernst [7] and Odersky et al. [14]. These works use path-dependent types to tie together related specific object instances, exactly as we use them to tie element references to the appropriate containers. In the context of the Scala language, path-dependent types have been formalized and serve as a foundational abstraction in the language [2], although Scala's path-dependent types associate a new type with each value or class instance at runtime, and are hence a dynamic concept; the Scala implementation performs dynamic checks as a result. In contrast, our use of path-dependent types is purely static. Other sorts of dependent types, such as constraint types [13], can also encode restrictions on values which can in principle be used to build safe containers as we have. Such schemes are powerful and general; in contrast, our approach is specific to the problem of associating one object with another specific object.

There has been some work on how to build container data types in the context of ownership systems, or to fit a more regimented ownership system. Potanin et al. [16] modify standard Java container types to conform to an *owner-as-accessor* ownership system: this means that they must ensure that direct object accesses to internal state only go through the container itself. This is similar to how our deferred-borrow objects contain state that allows a container to return a direct reference at time of use.

9 Future Work and Conclusion

In this paper, we have examined the shortcomings of *ownership and borrow-based memory management* as practiced in the Rust programming language. Although borrows with static lifetimes allow the compiler to verify that (i) no dangling pointers exist, and (ii) no aliasing mutable pointers exist, the imprecision in borrow tracking that arises when *container data types* are used frequently creates friction for programmers. A standard idiom is to refer to container elements by index, manually indexing an element each time a short-lived borrow is required. However, this approach is cumbersome and is *logically* unsafe because an index is not tied to the container. We introduce *deferred borrows*, which encapsulate a memory-safe reference to an element of a container. A deferred borrow provides the *same static guarantees* as an ordinary borrow: it will never be dangling and it will never allow aliasing mutable pointers to exist. This is achieved by performing a true borrow of the associated container only when the deferred borrow is actually used, rather than when it is created. The associated container is tied to the deferred borrow with the use of *static path-dependent types*.

While we have presented a complete proposal, we believe there are several angles for further expansion of this language feature that are promising. First, while the deferred-borrow implementation is currently manual (the container type must return a value with a type that implements the `DefBorrow` trait), it could be auto-derived from an ordinary access method that returns a true borrow. The compiler should be able to analyze the method body to determine whether actions are safe to defer based on just a few assumptions (e.g., that the internal storage of a `Vec` will not be reallocated).

Second, we have not discussed *field* borrows so far, but conceptually a field accessor (`x.a`) is a deferred-borrow operator, or selector, whose state happens to be constant. The Rust borrow checker obviates the need for deferred field borrows in many common cases because it can directly track individual fields (it natively understands field access paths), but reformulating field accesses in terms of deferred borrows may provide an opportunity to simplify the borrow checker.

Finally, deferred borrows, when seen as operators (curried with specific state) that convert the root container borrow to an element borrow at the time of use, should be *composable* as well. This is especially useful when considering field accessors as deferred-borrow state. In essence, deferred borrows encapsulate an arbitrary access path, possibly constructed by concatenating access path components, allowing the program to refer to state without restricting access to portions of the state tree in the meantime.

We believe that the deferred-borrow approach will be a valuable addition to the repertoire of safe memory management techniques in Rust and related ownership-and-borrowing-based type systems in the future.

References

1 J Aldrich and C Chambers. Ownership domains: Separating aliasing policy from mechanism. *ECOOP*, 2004.
2 N Amin, T Rompf, and M Odersky. Foundations of path-dependent types. *OOPSLA*, 2014.
3 C Boyapati, A Sălcianu, W Beebee Jr., and M Rinard. Ownership types for safe region-based memory management in real-time Java. *PLDI*, 2003.
4 D Clarke and S Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. *OOPSLA*, 2002.
5 D G Clarke, J M Potter, and J Noble. Ownership types for flexible alias protection. *OOPSLA*, 1998.
6 W Dietl, S Drossopoulou, and P Müller. Generic universe types. *ECOOP*, 2007.
7 E Ernst. Family polymorphism. *ECOOP*, 2001.
8 D Grossman, G Morrisett, T Jim, M Hicks, Y Wang, and J Cheney. Region-based memory management in Cyclone. *PLDI*, 2002.
9 R L Bocchino Jr., V S Adve, D Dig, S V Adve, S Heumann, R Komuravelli, J Overbey, P Simmons, H Sung, and M Vakilian. A type and effect system for Deterministic Parallel Java. *OOPSLA*, 2009.
10 G. Kastrinis, G. Balatsouras, K. Ferles, N. Prokopaki-Kostopoulou, and Y. Smaragdakis. An efficient data structure for must-alias analysis. *CC*, 2018.
11 K R M Leino, A Poetzsch-Heffter, and Y Zhou. Using data groups to specify and check side effects. *PLDI*, 2002.
12 K Naden, R Bocchino, J Aldrich, and K Bierhoff. A type system for borrowing permissions. *POPL*, 2011.
13 N Nystrom, V Saraswat, J Palsberg, and C Grothoff. Constrained types for object-oriented languages. *OOPSLA*, 2008.
14 M Odersky, V Cremet, C Röckl, and M Zenger. A nominal theory of objects with dependent types. *ECOOP*, 2003.

15 J Östlund and T Wrigstad. Multiple aggregate entry points for ownership types. *ECOOP*, 2012.

16 A Potanin, M Damitio, and J Noble. Are your incoming aliases really necessary? counting the cost of object ownership. *ICSE*, 2013.

17 A Potanin, J Noble, D Clarke, and R Biddle. Generic ownership for generic java. *OOPSLA*, 2006.

18 Rust community. The Rust programming language. `https://www.rust-lang.org/`.

19 Rust community. Rust survey 2018 results. `https://blog.rust-lang.org/2018/11/27/Rust-survey-2018.html`.

20 S Stork, K Naden, J Sunshine, M Mohr, A Fonseca, P Marques, and J Aldrich. Aeminium: A permission-based concurrent-by-default programming language approach. *ACM Trans. Prog. Lang. Sys.*, 36, March 2014.

21 A Weiss, D Patterson, ND Matsakis, and A Ahmed. Oxide: the essence of Rust, 2019. `arXiv:1903.00982`.

22 E Westbrook, J Zhao, Z Budimlić, and V Sarkar. Practical permissions for race-free parallelism. *ECOOP*, 2012.

23 Hongwei Xi and Frank Pfenning. Dependent types in practical programming. *POPL*, 1999.