Tree Borrows

NEVEN VILLANI[*], Univ. Grenoble Alpes, CNRS, Grenoble INP[†], France
JOHANNES HOSTERT[*], ETH Zurich, Switzerland
DEREK DREYER, MPI-SWS, Germany
RALF JUNG, ETH Zurich, Switzerland

The Rust programming language is well known for its ownership-based type system, which offers strong guarantees like memory safety and data race freedom. However, Rust also provides *unsafe* escape hatches, for which safety is not guaranteed automatically and must instead be manually upheld by the programmer. This creates a tension. On the one hand, compilers would like to exploit the strong guarantees of the type system—particularly those pertaining to aliasing of pointers—in order to unlock powerful intraprocedural optimizations. On the other hand, those optimizations are easily invalidated by "badly behaved" unsafe code. To ensure correctness of such optimizations, it thus becomes necessary to clearly define what unsafe code is "badly behaved". In prior work, *Stacked Borrows* defined a set of rules achieving this goal. However, Stacked Borrows rules out several patterns that turn out to be common in real-world unsafe Rust code, and it does not account for advanced features of the Rust borrow checker that were introduced more recently.

To resolve these issues, we present *Tree Borrows*. As the name suggests, Tree Borrows is defined by replacing the stack at the heart of Stacked Borrows with a tree. This overcomes the aforementioned limitations: our evaluation on the 30 000 most widely used Rust crates shows that Tree Borrows rejects 54% fewer test cases than Stacked Borrows does. Additionally, we prove (in Coq) that it retains most of the Stacked Borrows optimizations and also enables important new ones, notably read-read reorderings.

1 Introduction

Type systems have been enormously effective at ruling out entire classes of bugs in an efficient and compositional way. A recent success story is *Rust*, which uses affine ("ownership-based") types and lifetimes (a variant of region types) to statically ensure type safety, memory safety, and data race freedom for lower-level systems programming applications without requiring a garbage collector.

However, the benefits of type systems go beyond "just" ensuring that programs *don't go wrong*: type systems can also help to make programs *go fast*. Indeed, from very early on, the Rust compiler has used its type system not only to ensure type safety, but also to inform optimizations. Specifically, Rust's types encode powerful *aliasing information*: all references are either aliased or mutable, but never both at the same time. Alias analysis is a central pillar of modern optimizers, and so unsurprisingly Rust compiler developers want to feed the omnipresent type information into alias analysis. Let us consider a concrete example:

Example 1

```
fn write_both(x: &mut i32, y: &mut i32) -> i32 {
    *x = 13;
    *y = 20;
    *x // Transformation: return 13
}
```

This function takes two *mutable references* as arguments, so by the rules of the Rust type system, these references cannot alias. It follows that the function must always return 13, so we should be able to remove the read from `*x` on the last line and hard-code the return value to 13 without

[*] The two authors should be considered as joint first authors
[†] Institute of Engineering Univ. Grenoble Alpes

Authors' Contact Information: Neven Villani, neven.villani@univ-grenoble-alpes.fr, Univ. Grenoble Alpes, CNRS, Grenoble INP, 38000 Grenoble, France; Johannes Hostert, johannes.hostert@inf.ethz.ch, ETH Zurich, Zurich, Switzerland; Derek Dreyer, dreyer@mpi-sws.org, MPI-SWS, Saarbrücken, Germany; Ralf Jung, ralf.jung@inf.ethz.ch, ETH Zurich, Zurich, Switzerland.

affecting the program's behavior. Although this is a contrived example, it is representative of the kind of program transformation that unlocks powerful *intraprocedural* optimizations, despite the relevant memory locations having been allocated elsewhere in the program.

1.1 The Challenge of Allowing Aliasing Optimizations in the Presence of Unsafe Code

Unfortunately, the reality of Rust is more complicated. Although the Rust type system provides strong aliasing guarantees, they can be *too* strong, preventing programmers from implementing certain abstractions (*e.g.*, those that rely on mutation of shared state) or from achieving the desired performance. As a result, programmers sometimes circumvent the restrictions of the type system through the use of *unsafe escape hatches*—notably *raw pointers*, whose aliasing is untracked. To ensure that reasoning about type safety remains compositional, Rust programmers wrap their unsafe code in safe abstractions. For instance, the `Vec` type from the standard library, which provides a resizable array, is internally implemented with unsafe code—but the public API provided to the user is safe. This means that millions of lines of code can use `Vec` without worrying about type safety, and only the few thousand lines of code implementing `Vec` require extra scrutiny.

The problem with unsafe code is that it can do things like this:

Example 2

```
fn main() {
    let mut x = 42;
    let ptr = &mut x as *mut i32;
    let val = unsafe { write_both(&mut *ptr, &mut *ptr) };
    println!("{val}");
}
```

This code creates a raw pointer `ptr` to a mutable local variable `x`. It then uses the `unsafe` code block to convert the same raw pointer to a mutable reference *twice* (via `&mut *ptr`), thus ending up with two aliasing mutable references. The program will thus print `20` when compiled without optimizations, but would print `13` if `write_both` were optimized as described above. That's bad! Optimizations are not supposed to change program behavior.

Given that aliasing optimizations are something that the Rust developers clearly want to support, we need some way of "ruling out" counterexamples like the one above from consideration. Rust programmers are already used to the fact that using unsafe features does not mean they can do whatever they want: unsafe code authors must pay careful attention to special rules. For instance, when using the `get_unchecked` function, which accesses an array without a bounds check, unsafe code authors must ensure that the index is in-bounds of the array. Unsafe code which does not hold up those requirements is said to be inducing *Undefined Behavior* (UB), because the resulting program could misbehave in arbitrary ways. Consequently, unsafe code authors have an implied obligation to ensure that their code is free of UB; the compiler will in turn trust the programmer and just assume that the code it compiles does not have UB.

To save the reference-based optimization in Example 1, we thus need to find a way to make the program in Example 2 have UB. However, this is harder than it may seem. For instance, the following entirely safe Rust snippet may seem like it causes aliasing of mutable references as well:

Example 3

```
let mut x = 42;
let ref1 = &mut x;
let ref2 = &mut *ref1;
*ref2 = 12;
println!("{ref1}"); // Will print 12.
```

Both `ref1` and `ref2` are pointing to `x`. How can this be allowed, when Rust does not permit aliasing of mutable references? The answer is that this is a case of *reborrowing*: `ref2` is *derived from* `ref1`, and so the compiler allows the two to coexist. However, for the duration that ref2 is live, ref1 may not be used, and when ref1 is used again, that is the end of the lifetime of ref2. If we added a println!("{ref2}"); at the end of Example 3, it would get rejected for violating this discipline.

The challenging task ahead therefore lies in defining UB in such a way that the bad aliasing example above gets rejected, but all safe Rust code gets accepted. In fact, we have to go even further and ensure that enough "reasonable" unsafe Rust code is accepted: for unsafe Rust to be useful, it must be possible to write correct unsafe code with modest effort.

1.2 Prior Work: Stacked Borrows

We are not the first to tackle this challenge: Stacked Borrows [12] had exactly the same goal. The key observation behind Stacked Borrows was that in cases like Example 3, all currently valid pointers can be organized in a *stack*, with x at the bottom of the stack, ref1 on top, and finally ref2. The stack models the fact that these references may only be used in a *well-nested manner*: the moment ref1 is used, ref2 has been invalidated. Unfortunately, this simple stack model has three major problems.

The first problem is that Stacked Borrows *prohibits* some basic optimizations, such as reordering adjacent reads. The reason for this is simple: in Example 3, if we read from ref2 and then from ref1, this is perfectly fine. ref2 is "nested inside" ref1, so all accesses are permitted. However, after reordering the two operations, we would be first reading from ref1 and then reading from ref2. This is UB under Stacked Borrows, since it violates the well-nested requirement. Reordering reads is a crucial optimization, so this is a significant flaw.

The second problem is that Stacked Borrows restricts references to a *static range*: when a reference is created, the type of the reference determines the range of memory it may access, and no memory outside that range can ever be accessed with this reference or any raw pointer derived from it. In practice, this is often not what programmers expect. For example, programmers coming from C will routinely create a reference to the first element of an array, turn it into a raw pointer, and then use that to access the entire array. Stacked Borrows rejects this, since a raw pointer derived from a reference to the first element of an array may only access that one element. This mismatch between programmers' expectations and the model leads to frequent UB in real-world code, so it is worth exploring alternative, more permissive options.

The third problem is quite technical. It relates to a feature of the Rust borrow checker that was developed concurrently with Stacked Borrows, called *two-phase borrows*. This feature weakens the requirements for some mutable references so that they do not have to be immediately unique; instead, they are created in a *reservation* phase that permits read-only aliasing. Uniqueness is only enforced upon *activation* of the mutable reference, which must occur before the first write.

The original Stacked Borrows model did not support two-phase borrows at all. The implementation of Stacked Borrows that Rust programmers can use to check their code for UB has rudimentary support: to check today's Rust code that routinely uses two-phase borrows, the implementation treats two-phase borrows basically equivalent to raw pointers. However, this rudimentary support has some very counter-intuitive consequences that regularly confuse Rust programmers, as is demonstrated by the example below:

Example 4

```
fn write(x: &mut i32) { *x = 10; }
let x = &mut 0;
let y = x as *mut i32; // derive raw pointer from x
write(x); // write to x
unsafe { *y = 15 }; // use raw pointer
```

This program is accepted by Stacked Borrows. However, if we replace the call to write by its body, then it gets rejected by Stacked Borrows.[1] The reason for this difference is that `write(x)` is implicitly expanded to `write(&twophase x)`, where `&twophase` is pseudo-syntax marking an *implicit* mutable reference that is subject to the rules of two-phase borrows, and hence treated like a raw pointer by the Stacked Borrows implementation. Not only does this difference confuse programmers, the fact that implicitly created mutable references are treated like raw pointers also limits the potential for optimizations.

1.3 Our Contribution: Tree Borrows

In this paper, we show how to overcome all these limitations with a new aliasing model for Rust, which we call **Tree Borrows**:

- To support two-phase borrows, we use a *tree* instead of a stack to track the relationship of references. A tree is a natural choice since each reference has a unique parent: the reference/local variable it is created from. A stack imposes the additional constraint that at most one child of each reference can matter at any given time, but with two-phase borrows this does not suffice: there may be a reserved mutable reference and several shared references peacefully co-existing in the reservation phase of the mutable reference. They have to be tracked as independent children of the same parent reference, so that when the mutable reference gets activated, its siblings can no longer be used.
- To allow reordering of reads, each node in the tree tracks a *state machine* representing the current permissions of the reference that is represented by this node. These permissions change as the memory the reference points to is read and written by this or other references. This is in contrast to Stacked Borrows, in which the permissions of a reference are fixed upon creation and never changed until it is invalidated.
- Finally, we replace Stacked Borrows' static reference ranges by *dynamic reference ranges*, which place no fixed bound on the region of memory for which a reference claims uniqueness (or immutability). Instead, that region is determined based on how the reference is used. This crucially relies on the tree structure that lets us track multiple siblings without deciding upfront which one will be unique for which part of memory.

To empirically evaluate the effectiveness of Tree Borrows, we have implemented it in Miri, the same tool that was also used to evaluate Stacked Borrows. We have carried out an experiment on the 30 000 most downloaded Rust libraries, and confirmed that Tree Borrows rejects 54% fewer test cases than Stacked Borrows does. We have also formally modeled Tree Borrows in Coq, and used the Simuliris [9] framework to build a relational program logic that can prove correctness of Tree Borrows optimizations. Almost all optimizations shown for Stacked Borrows still hold for Tree Borrows; the only exceptions are optimizations that insert new writes before the first write in the original program—this is part of the trade-off of adapting Tree Borrows to be more compatible with real-world code. In particular, the aliasing optimizations the Rust compiler already performs (like Example 1 above) are all, to our knowledge, supported by Tree Borrows. Moreover, unlike Stacked Borrows, Tree Borrows enables us to prove that reordering adjacent reads is sound.

The rest of the paper is structured as follows: We begin with an example-driven introduction to the core of Tree Borrows (§2). Next, we extend Tree Borrows with *protectors*, which significantly strengthens the optimizations we can perform (§3). Afterwards, we perform a detailed comparison of our new model with its predecessor, Stacked Borrows, including the empirical evaluation mentioned

[1] This may seem like Stacked Borrows does not permit inlining, but it is more subtle than that. The difference in behavior arises from a difference in how surface Rust is elaborated into the internal IR of the compiler; inlining on the level of that IR is sound with Stacked Borrows.

above (§4). Finally, we show some examples of optimizations that we have proven correct for Tree Borrows (§5), and conclude with a discussion of related and future work (§6).

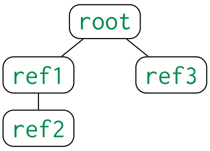
2 The Basics of Tree Borrows

In this section, we explain the basic mechanisms behind Tree Borrows' tracking of references and detection of UB. We assume no prior knowledge of Rust or Stacked Borrows.

2.1 Keeping Track of Aliases

The key idea of Tree Borrows is given away in its name: references are organized in a tree. For instance, consider this example:

```
let mut root = 42;
let ref1 = &mut root;
let ref2 = &mut *ref1;
let ref3 = &mut root;
```



Example 5

Here we have a local variable, root, and several references derived directly or indirectly from that variable. The tree on the right indicates how Tree Borrows views the relationship between these references and the local variable. Every newly created reference is associated with a new node in the tree, inserted as a child of the node that was used to create this reference. So, a reference is not just a location into memory; instead, it is defined by a pair of a location and an identifier that determines its corresponding node in the tree. We call this identifier the *tag* of the reference.

Effects of an access. Every time a memory access occurs, Tree Borrows "informs" all references about this access. Each reference, *i.e.,* each node in the tree, tracks a state machine, which defines how the reference "reacts" to that access: the access can either be permitted (possibly switching to a different state), or rejected, in which case the program has UB.

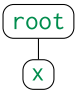
The transitions of the state machine are indexed by whether the access was a read or a write, and by the relationship of the currently considered reference with the reference that was used for the access. Specifically, we distinguish *local* accesses from *foreign* accesses. When computing the next state machine transition for a node with tag t_{sm} after an access to tag t_{acc}, the access is a local access to t_{sm} if t_{acc} is "derived from" t_{sm}, *i.e.,* if it is t_{sm} itself or one of its children; it is a foreign access if t_{acc} is a parent or cousin of t_{sm}. For instance, in the example above, if we consider t_{sm} = ref1, then an access to ref1 or ref2 would be considered local, and an access to ref3 or root would be considered foreign. Overall, the alphabet of the state machine is defined by {local read (↓R), local write (↓W), foreign read (↑R), foreign write (↑W)}.

2.2 Lifetime of a Mutable Reference

We will now slowly define the state machine, the full version of which is shown in Figure 1. Most of the state machine can be understood by discussing mutable references. We thus begin by explaining a simplified model of mutable references, and then extend it until it is able to accept all safe code, and allows our desired optimizations.

A naive model of mutable references. For inspiration on how to handle mutable references, we first turn to the component of the Rust compiler that ensures correct usage of references—the *borrow checker*. We start with a very simple example:

```
let mut root = 42;
let x = &mut root;
*x += 1;
root = 0
```



Example 6

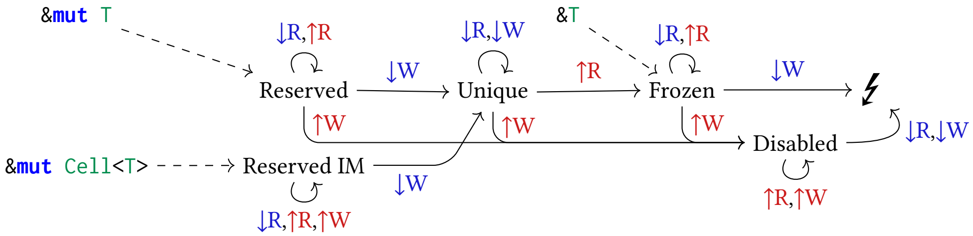


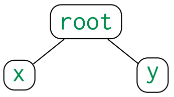
Fig. 1. Default state machine of permissions, featuring the entry point for mutable references marked by `&mut T`. Reaching ↯ means that the program has UB. Transitions are labeled by the events that cause them: (R)ead or (W)rite, each either ↑(foreign) or ↓(local).

Here, the mutable reference `x` is created by the expression `&mut root`, making it a child of `root`. The borrow checker will determine that the lifetime of `x` begins at `&mut root`, and cannot continue further than `root = 0`, where the parent reference reclaims ownership which must kill `x`. Any further attempt to use `x` will raise a compilation error that `x`'s lifetime has expired. In the interval between those operations, while `x` is live, we are free to read from and write to `x`. This desired behavior can easily be expressed in our framework of permissions and local/foreign accesses: we use two permissions, Unique and Disabled, to represent live and dead mutable references, respectively. Initially, the mutable reference is given the permission Unique, and as long as it is Unique it tolerates arbitrary local accesses. When a foreign access occurs, this means that a parent reference has reclaimed ownership of the pointee. We hence transition the permission to Disabled, and from that point onwards any local accesses to `x` trigger UB.

Despite its simplicity, this model already captures most of the desired semantics of mutable references. For example, it rejects sequences of events that contain a foreign write followed by a local write, thereby properly enforcing unique ownership. To illustrate this, recall Example 2 in which unsafe code creates two mutable references to the same location, and passes both as arguments to a safe function. We found that this is in violation of uniqueness of mutable references, and therefore, the code in question should be declared Undefined Behavior. Below is the same piece of code (with Example 1 inlined for brevity) on which we will now execute the current model to show that it does indeed detect UB.

Example 7

```
let mut root = 42;
let ptr = &mut root as *mut i32;
let (x, y) = unsafe { (&mut *ptr, &mut *ptr) };
*x = 13;
*y = 20;
let val = *x;
```



We track the state of `x`: it is created by `&mut *ptr`, which means it is a mutable reference and it is derived indirectly from `root`. The reference `y` is a cousin of `x`, so accesses through `y` appear as foreign accesses from the point of view of `x`. At `*x = 13` (which is a local write), `x` is Unique. Then `*y = 20` is a foreign write, so `x` becomes Disabled. Finally, `let val = *x` is an attempted local read through a Disabled. This is UB, as expected.

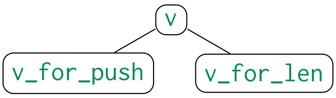
Despite these promising results for such a simple model, having only Unique and Disabled proves to be insufficient. We now show two ways in which the previous model should be refined.

Two-phase borrows and the Reserved state. In Example 4, we showed *two-phase borrows*, the handling of which is one of the main motivations for introducing Tree Borrows. Our simplified Unique/Disabled model currently lacks support for these.

The standard example [32] for a two-phase borrow is `v.push(v.len())`, where `v` is of type `Vec<usize>`. After desugaring the notations for method calls, this becomes roughly:

Example 8

```
let v_for_push = &mut v;
let v_for_len = &v;
let len = Vec::len(v_for_len);
Vec::push(v_for_push, len);
```



The first line computes the first argument for `push`, the next two lines compute the second argument (where the call to `len` has also been desugared to its more explicit form). If we take the perspective of `v_for_push`, then after the reference is created, it is subjected to a foreign read when `Vec::len` reads the length of the vector. According to the state machine as we have described it so far, that would transition its state to Disabled, thus causing Undefined Behavior later in `Vec::push`! The naive model thus fails to account for the fact that this safe code should be permitted.

Example 8, in fact, used to be rejected by the borrow checker, basically for the same reason that our naive model rejects it. To overcome this, the Rust developers introduced the concept of *two-phase borrows*, where some[2] mutable references begin their life with a *reservation phase*. During the reservation phase, arbitrary read accesses to the memory the reference points to are allowed, even via other references. After *activation*, which occurs the first time the mutable reference is written to, it becomes a full mutable reference and no longer tolerates aliases. We can incorporate this behavior in Tree Borrows by creating a new permission to model the behavior of mutable references during their reservation phase. We fittingly call this permission Reserved. From now on, mutable references start out as Reserved, not as Unique. Reserved allows local and foreign reads, thereby allowing all reads during the reservation phase. When the reference is *activated*, *i.e.*, when it first experiences a local write, its permission changes to a full-fledged Unique; and as before, when Unique experiences a foreign access, it becomes Disabled.

Enabling read-read reorderings with the Frozen state. Read-only accesses are usually considered to be free of side effects, so compilers are usually allowed to reorder adjacent read-only operations: the two programs `let vx = *x; let vy = *y;` and `let vy = *y; let vx = *x;` should be equivalent in any context.[3] However, in our aliasing model, read accesses are not side effect-free! Indeed, the simple act of reading data has an effect on the state machines maintained in each node. For example, a foreign read could very well change a Unique into Disabled.

Unfortunately, in the current draft of the state machine, adjacent reads cannot in general be reordered. For example, in a context where `x` is Unique and a child of `root` (this configuration is easily generated for example by `let mut root = 0; let x = &mut root; *x = 42;`) we find that reading from `x` then `root` is fine, but reading from `root` then `x` results in UB (the read through `root` is foreign for `x`, making it Disabled, and thus further reads are UB). We can pinpoint this issue on the specific transition from Unique to Disabled wherein a foreign read (through `root`) causes a readable reference (`x` currently Unique) to lose read permissions (becomes Disabled).

This suggests that upon a foreign read, a Unique reference should not immediately lose the ability to read, only to write. To fix this shortcoming, we introduce a new permission, which we call Frozen. This new permission is added as an intermediate step between Unique and Disabled,

[2] Two-phase borrows only apply to implicit reborrows, *i.e.*, they do not occur when the source code contains "`&mut`" syntactically. Tree Borrows ignores this distinction and gives a reservation phase to all mutable references.

[3] These are non-atomic accesses, so reordering reads is possible even in a concurrent program. If there were any race conditions, that would be UB.

reached when a Unique experiences a foreign read and exited when Frozen experiences a foreign write. A Frozen tag allows arbitrary local and foreign read accesses. Now both orderings of reading `root` then `x` or `x` then `root` result in the same state, with `x` being Frozen and still tolerating read accesses. By introducing Frozen, we have made read-read reorderings possible again, making it a rare case where having *less* UB leads to *more* optimizations!

Dealing with ranges of memory. So far, we pretended that each reference only has a single state. This is correct as long as we only consider a single location in memory. To support multiple locations, we use a CompCert-like [21, 19] memory model where memory is organized in contiguous *allocations*, each of which consists of a certain number of bytes. A memory location is a tuple (a, o), where a identifies the allocation, and o the offset into that block. Thus, a reference consists of three components, $((a, o), t)$, also featuring the tag t. Tree Borrows tracks a tree of tags for each allocation. Furthermore, there is an instance of the aforementioned state machine for each *location*, *i.e.,* the same tag can be in a different state at different offsets in the allocation. When an access affects more than one location (such as a 4-byte load that reads from 4 locations), we independently update the state machine for all tags in this allocation's tree for each of these locations. If any of these state machines transitions to UB, the overall access is UB.

2.3 Beyond Mutable References

So far we have considered only mutable references, but Rust has more pointer-like types: *shared references* are the dual to mutable references in the sense that they permit aliasing, but no mutation. And *raw pointers* are Rust's unsafe pointer construct that cannot be used in safe Rust, but permits arbitrary aliasing. In this section we show that our current model easily accomodates both of these, without requiring any new permissions to be introduced.

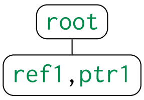
Shared references. Shared references are permitted to alias arbitrarily with other shared references (and also with mutable references in their reservation phase) and allow read accesses, but no write accesses, as long as they are live. In terms of our state machine, the permission that represents shared references must allow local and foreign reads, and trigger UB if a local write is attempted. On foreign writes, the permission should become Disabled. But look: this exactly describes the Frozen state! In addition to being a transitory permission for mutable references near the end of their lifetime, Frozen is thus also the initial permission for shared references. This gives a new interpretation to the transition from Unique to Frozen for mutable references: when a mutable reference can no longer uphold its uniqueness guarantee, it gets demoted to a shared reference.

Raw pointers. Raw pointers are Rust's most permissive pointer type, to be used only when doing low level manipulations that references cannot support. Multiple raw pointers to the same location can exist simultaneously, all aliasing each other, and all of them may have write permissions. Short of outliving their parent reference, there is very little that raw pointers are not allowed to do.

In Tree Borrows, the creation of a raw pointer is a no-op: raw pointers do not get a new tag and thus inherit the tag of the reference that they are created from. This means that if multiple raw pointers are created from the same reference, accesses through them can be arbitrarily interleaved—the aliasing model considers them all as coming from the same "source", namely the reference they are derived from. Once that reference comes to the end of its lifetime (*i.e.,* when its permission becomes Disabled), all the raw pointers are also invalidated.

Example 9

```
let mut root = 42;
let ref1 = &mut root;
let ptr1 = ref1 as *mut i32;
*ref1 = 43;
*ptr1 = 44;
*ref1 = 45;
```

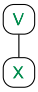


In this code, the entire line `let ptr1 = ref1 as *mut i32;` is a no-op from the point of view of Tree Borrows, and `ptr1` is thereafter considered identical to `ref1`. Although the final three assignments assign to syntactically different variables, Tree Borrows at runtime sees only the tags, which are all equivalent. So Tree Borrows accepts this code, as intended, since raw pointers are supposed to allow aliasing.

Tree Borrows even allows raw pointers to access memory outside the range determined by the type of their parent reference, as in the example below:

Example 10

```
let mut v = [0, 0];
let x = &mut v[0];
unsafe { *(x as *mut i32).add(1) = 1; }
```



In this example, `x as *mut i32` is a raw pointer to the first element of `v`. Using pointer arithmetic, we use that pointer to access the second element of `v`. The way this works is that `&mut v[0]` creates a permission and associated state machine for the new reference *everywhere* in the current allocation, not just in the memory range that this reference points to. Through the use of raw pointers, the memory range actually accessible by a reference can be dynamically extended to be larger than what its type indicates. This is crucial to support some low-level programming patterns where the reference only points to a prefix of the relevant data, and based on information stored in that prefix, a raw pointer is used to access further data stored after the prefix [3].

2.4 Interior Mutability

Rust references prevent mutation of aliased state, but having access to shared mutable state is sometimes necessary. Rust supports this via the mechanism of *interior mutability*, which permits exposing carefully controlled forms of shared mutable state using a library type that is internally implemented with unsafe code. Types with interior mutability do not uphold the usual aliasing guarantees of Rust, and are therefore required to use the special type `UnsafeCell<T>` that acts as a marker to indicate that the inner data of type `T` can be mutated through a shared reference.

In terms of aliasing, shared references to interior mutable types are basically equivalent to raw pointers, so Tree Borrows models them the same: interior mutable shared references inherit the tag of their parent reference. This immediately trivializes the handling of these references; just like raw pointers, they can freely alias with other raw pointers derived from the same parent.

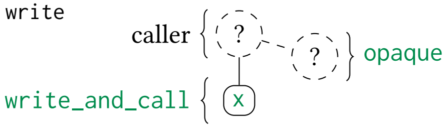
Mutable references to interior mutable types are indistinguishable from regular mutable references as soon as they become Unique. However, during their reservation phase, shared references can exist, and since these references permit mutation, a mutable reference must tolerate not only foreign reads but also foreign writes. To model this, we introduce a new permission ReservedIM which is identical to Reserved except for the fact that it allows foreign writes. ReservedIM, is the new entry point into the state machine for mutable references to interior mutable types.

3 Advanced Tree Borrows: Ensuring That References Stay Alive

The goal of this section is to permit optimizing code like this:

Example 11

```
fn write_and_call(x: &mut i32, opaque: impl Fn()) {
    *x = 13; // Transformation: eliminate this write
    opaque();
    *x = 20;
}
```

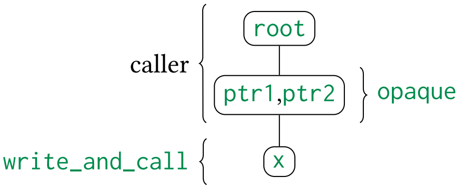


Here, opaque is a closure that takes no arguments, and does not return anything. Our goal is to eliminate the first write. Note that this is a quite powerful optimization: x points to memory that was allocated in an unknown way, and opaque is arbitrary unknown code—it is generally not possible to do any aliasing optimizations in a situation like this!

Intuitively, Tree Borrows should make this possible because opaque cannot possibly access the memory that x points to. After all, opaque would have to use some alias that is foreign to x (as indicated to the right of the code) to access the same memory. That should be UB since it violates uniqueness of x: such an access would transition x from Unique to Disabled, so `*x = 20` would be UB. However, the problem is that opaque might never return, in which case the `*x = 20` would never be executed and cannot cause UB. This is demonstrated by the following context:

Example 12

```
let mut root = 42;
let ptr1 = &mut root as *mut i32;
let opaque = move || {
    let ptr2 = ptr1;
    println!("{}", unsafe { *ptr2 });
    std::process::exit(0); // quit program
};
write_and_call(unsafe { &mut *ptr1 }, opaque);
```



Here, opaque is purposefully implemented so that it performs an access with a tag that is foreign to x, and then never returns. Applying the desired optimization will make the code print 42 instead of 13, showing that this code is in conflict with the desired optimization. At the same time, this context does not exhibit UB according to the model of Figure 1: x transitions from Unique to Frozen due to the foreign read `*ptr2`, and is never accessed again due to the infinite loop.

To fix this, we introduce a new mechanism called *protectors* that changes what happens at the `*ptr2` inside opaque: currently, this just transitions x to Frozen; if x is *protected*, this operation will instead be Undefined Behavior, even if x is never used again.

3.1 Putting Tags Under Protection

Concretely, we attach a boolean flag to each tag that indicates that this tag must currently not become Disabled. Such a tag is called *protected*. We also introduce some virtual operations at the beginning of each function call that equip all references in function arguments with a fresh tag (as if they had been reborrowed), and mark that fresh tag as protected. The protector flag is removed when the function returns. Intuitively, this ensures that for the duration that the function runs, references that are passed as arguments to the function cannot be invalidated. In the example above, the tag of x is protected for the entirety of the execution of the function `write_and_call`.

Note that protectors special-case function boundaries. This is justified because in Rust, the borrow checker ensures that a reference passed to a function always outlives that function. Protectors are the operational semantics equivalent of this outlives requirement. Furthermore, the Rust compiler already special-cases references in function argument position when passing code to its LLVM backend, to let LLVM perform optimizations based on the non-aliasing of references. Tree Borrows protectors are ensuring that the requirements LLVM imposes are upheld by UB-free Rust code.

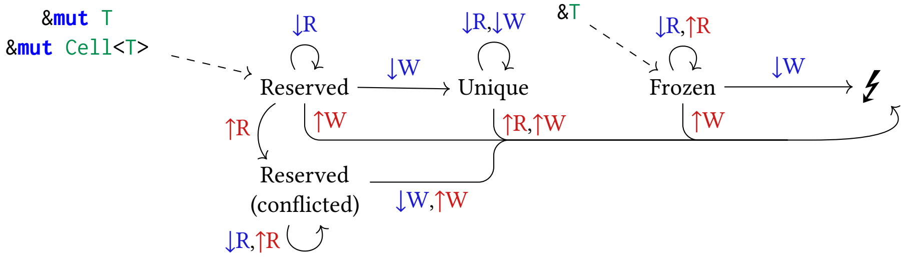


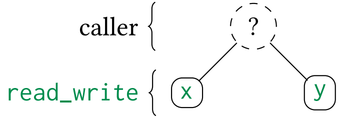
Fig. 2. Modified state machine of permissions for protected references. UB triggers upon reaching ϟ. Transitions are labeled by the events that cause them: (R)ead or (W)rite, each either ↑(foreign) or ↓(local).

When a tag is protected, instead of transitioning to Disabled, the corresponding operations immediately trigger UB. This is the core function of protectors: they ensure that a reference stays alive while it is protected. Furthermore, a protected Unique reference triggers immediate UB on a foreign read (instead of transitioning to Frozen); this is required to ensure that Example 12 is UB.

This is enough to justify the transformation in Example 11. However, it turns out it is not enough to justify everything the Rust compiler already assumes about mutable references. To see why, consider the example below, which represents an optimization that the Rust compiler performs:

Example 13

```
fn read_write(x: *const i32, y: &mut i32) -> i32 {
    let val = unsafe { *x };
    *y = 20; // Transformation: move write up
    val
}
```



Given all we said so far, this transformation would introduce UB in a program where x and y alias. The write to x changes its state to Unique, and therefore the subsequent foreign read through y is not permitted. We must thus ensure that if a mutable reference is written to at any point of the function call, then there was no foreign read at any point during the call. Protectors as explained above already rule out foreign reads *after* the write (since a foreign read on a Unique reference is UB), but extra care needs to be taken to ensure there hasn't been a foreign read *before* the write. To do this, we introduce a boolean flag in the Reserved state which records that a foreign read took place. This flag is called conflicted. A conflicted Reserved reference cannot become Unique; a local write to such a reference will instead trigger UB.

To ensure that this optimization can also be performed when y is a mutable reference to an interior mutable type, we change the initial state for protected mutable references to be *always* Reserved, no matter whether they point to interior mutable data or not.

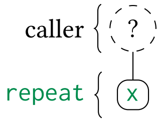
The end result is that for any mutable reference that is passed as argument to a function, it is UB for this reference to encounter both a foreign read and a local write—in any order.

3.2 Implicit Accesses

Given that protectors guarantee that their tags are live for the entirety of the function call, this should enable optimizations that introduce *spurious reads*: speculatively reading from a location that is not guaranteed to be read from in all executions. For example, consider the function below, in which a shared reference x is accessed n times:

```
fn repeat(x: &i32, n: usize, opaque: impl Fn(i32)) {
//  let val = *x;      // Transformation: insert load
    for _ in 0..n {
        opaque(*x);    // Transformation: replace *x by val
    }
}
```

Example 14



The value of `*x` is constant for the entirety of the execution of `repeat`; this is guaranteed by the protector on `x` which prevents foreign writes. Thus, we would like to optimize this function to read the value only once instead of `n` times. However, if `n` is 0, then we are inserting a read where none occurred, and this might trigger UB.

To ensure that inserting a read cannot trigger UB, we define our semantics such that even if the original function did not explicitly read `x`, we *implicitly* apply all the effects of a read access. Specifically, every time a fresh tag is generated, we immediately update the entire tree as-if the program performed a read via this fresh tag. The size of the read access is determined by the type of the reference. We refer to operations that create a fresh tag as *retags* (they occur every time a reference is created, and at the beginning of each function call), so we can summarize this section as saying that retags implicitly execute a read.

It is important to note that while the implicit read of a fresh reference, say of type `&mut i32`, is only as large as the type of the reference indcates (in this case, 4 bytes), it is still possible to use raw pointers derived from this reference outside that memory range. The protector is enforced for all locations that the reference has actually been *used* for. For instance, earlier in Example 10 where we reborrowed an array only partially, `let x = &mut v[0]` initializes `x` with the permission Reserved on all 8 bytes of the allocation, but only the first 4 bytes receive an implicit read. These 4 bytes are always considered *used*; any usage beyond that is determined dynamically. These are the *dynamic reference ranges* we mentioned in the introduction.

Protector end semantics. For reasons that are more subtle (see §5), we also need implicit accesses when a tag's protector is being removed: all used Reserved or Frozen locations are subject to an implicit read, and all used Unique locations are subject to an implicit *write*. In both cases, the access is only applied to foreign tags. Intuitively, by performing implicit accesses both at the beginning and end of a reference's lifetime, we give the model maximal freedom to reorder accesses and insert new spurious accesses without introducing UB.

4 Experimental Comparison with Stacked Borrows Using Miri

In the introduction, we motivated Tree Borrows as overcoming several limitations of Stacked Borrows. We now return to this point with a more in-depth explanation of how Tree Borrows differs from Stacked Borrows, along with an experimental evaluation of whether we succeeded in better aligning the model with programmers' expectations.

4.1 Differences between Tree Borrows and Stacked Borrows

Overall, Tree Borrows accepts many patterns not accepted by Stacked Borrows, but there are some cases in which Stacked Borrows accepts more code than Tree Borrows. In the following, we look at the differences between the models in a bit more detail: we explain how the general structure of Stacked Borrows differs from that of Tree Borrows, how Stacked Borrows handles references (which is structurally fairly similar to how Tree Borrows handles references, just with a different state machine), and how Stacked Borrows handles raw pointers (which is quite different).

Like Tree Borrows, every reference and raw pointer in Stacked Borrows carries a *tag* to distinguish it from other references and raw pointers with the same address. However, as the name indicates, Stacked Borrows works with a stack, not a tree. We can cast Stacked Borrows in the same mental framework as Tree Borrows by considering the stack to be equivalent to a tree with a branching factor of 1, where the bottom of the stack is the root of the tree.[4]

References. Stacked Borrows references can be fairly easily described in the framework of Tree Borrows. There is no Reserved state; mutable references start out Unique and they become Disabled on any foreign access—just like the first naive model of mutable references at the beginning of §2.2. Shared references are treated basically the same in Stacked Borrows and Tree Borrows, but since Stacked Borrows cannot represent branching, all shared references derived from the same mutable reference are stacked on top of each other. Since shared references are unaffected by foreign reads, and do not allow any writes, it does not make a difference whether they are tracked as unordered siblings, or whether they are put into some total order.

As already mentioned, Stacked Borrows also has no proper support for two-phase mutable references, and the implementation treats them like raw pointers. This is an oversimplification of how two-phase borrows work, and accepts way too much code—it accepts arbitrary aliasing of the two-phase mutable reference with a raw pointer. Tree Borrows rejects such aliasing if it violates the guarantees of two-phase borrows, *i.e.,* the two-phase mutable reference must be truly unique after its activating write. Furthermore, Tree Borrows makes no difference between mutable references with or without two-phase treatment by the borrow checker: they all start in the Reserved state. This avoids surprising changes in behavior when an implicit reference is made explicit.

Stacked Borrows has protectors, and they work like a simpler version of Tree Borrows protectors: there is no implicit access when a protector is removed, and no tracking of which locations are conflicted or *used.* Tree Borrows needs that tracking to support two-phase borrows and dynamic reference ranges, neither of which are supported by Stacked Borrows.

Raw pointers. Raw pointers work quite differently in Stacked Borrows: in the original model as described in the paper, raw pointers are *untagged.* When a reference is cast to a raw pointer, the stack records that this reference has an untagged "child"; accesses through an untagged pointer are only allowed when such a suitable untagged record exists on the stack. However, it turned out that this has some rather problematic consequences [24, 4], and therefore the Stacked Borrows implementation was eventually updated to also track a tag for raw pointers [2]. However, this tag is still distinct from the tag of the reference that the raw pointer is created from.

The initial permission of a raw pointer in Stacked Borrows is SharedReadWrite, which does not correspond to any Tree Borrows permission. It is unaffected by most accesses, except for foreign writes performed via a Unique pointer—those disable even a SharedReadWrite pointer. In other words, to cast Stacked Borrows in terms of Tree Borrows, the state machine needs to know not only the kind of access and the relative position in the tree of the tag that was used for the access and the tag under consideration, but also the permission of the tag used for the access.[5]

Furthermore, reads via a pointer with SharedReadWrite have a very special interaction with Unique children. This is best demonstrated by the following example:

Example 15

```
let mut root = 0;
let ptr = &mut root as *mut i32;
*ptr = 1;
let _val = root;
*ptr = 2;
```

[4] So unlike normal trees, this one grows upwards.

[5] Stacked Borrows can be described more elegantly than that in its own terms. Here, we are deliberately casting it in the terms of Tree Borrows to be able to properly compare both systems.

This code is accepted by Stacked Borrows, but rejected by Tree Borrows. Tree Borrows rejects this code because `ptr` is a child of `root`, and after the assignment storing 1 in `ptr`, it has Unique permission. This means that the read from `root` in the next line transitions `ptr` to Frozen, and the subsequent store of 2 is Undefined Behavior. However, Stacked Borrows accepts Example 15 because it considers `ptr` to be a *grandchild* of `root`. There is an unnamed mutable reference (the one created via `&mut root`) in between `root` and `ptr`. As a special "quirk" (sic),[6] a read from `root` invalidates that unnamed reference, but does *not* invalidate `ptr`. Therefore, further writes to `ptr` are permitted. This behavior cannot be explained while upholding a tree structure, and in fact even the stack structure of Stacked Borrows is arguably violated here.

The reason Stacked Borrows behaves in this quirky way is to accept a fairly common pattern of unsafe code. Specifically, the code that is given in the Stacked Borrows paper as the motivation for this "quirk" is similar to Example 15 but does not involve any writes to `ptr` in between the creation of `ptr` and the read from `root`. Such code is still accepted by Tree Borrows thanks to Tree Borrows' treatment of *all* mutable references as two-phase borrows. In contrast, code like Example 15, in which writes to `ptr` are interleaved both before and after the read from `root`, is not discussed in the Stacked Borrows paper and is perhaps only accepted under Stacked Borrows by accident.

On the other hand, there are many cases where Tree Borrows accepts more code than Stacked Borrows. In particular, Stacked Borrows statically bounds the memory ranges that can be accessed by a reference, thus rejecting all code that uses raw pointers outside the range of the reference they were derived from. Another factor for accepting more code in Tree Borrows is the fact that a raw pointer inherits the identity of its parent reference: when a raw pointer is created like `&raw mut root`, Tree Borrows allows mixing accesses to the pointer and to `root`, while Stacked Borrows does not. Both of these points (but especially the static bounds) regularly confuse programmers, leading to bug reports and online discussions asking why code examples get rejected [1, 3, 5, 6, 8].

4.2 Experimental Setup

As we have seen, Stacked Borrows and Tree Borrows are incomparable in terms of which model accepts more code. However, we claim that Tree Borrows accepts a lot more code *in practice*. To support this claim, we conducted an experiment around the following questions:

- *Q*1: How much existing Rust code is *supported* in Tree Borrows, but not in Stacked Borrows, due to UB that Tree Borrows has removed?
- *Q*2: How much existing Rust code is *broken* in Tree Borrows, but not in Stacked Borrows, due to UB that Tree Borrows has introduced?

To answer these questions, we tested large amounts of existing Rust for their conformance to Tree Borrows' (and Stacked Borrows') rules, and compared the differences. We now discuss how we did this, and then present and interpret the results.

Our experiments used *crates.io* and *Miri*. The first, crates.io, is Rust's central package registry, similar to `npm` for JavaScript. Almost all widely used Rust libraries are published there. The second component is Miri. Miri is an implementation of Rust's abstract machine that evaluates Rust (strictly) according to the operational semantics. It is intended to be used in testing, letting unsafe code authors detect UB bugs in their code.

For our experiment, we extended Miri with support for Tree Borrows. Miri already had support for Stacked Borrows (as discussed in [12]), but introducing support for Tree Borrows and making the specific aliasing model configurable required a moderate rearchitecturing of Miri's internals. Besides this rearchitecturing, the Tree Borrows implementation in Miri is mostly a direct translation of the formal semantics. We also added some optimizations, *e.g.*, one which can compact the tree

[6] See the discussion of READ-2 on page 21 of the Stacked Borrows paper [12].

Cause	Number	Percent
Timeout	71 270	10.56%
Unsupported	108 210	16.04%
UB	10 311	1.53%
Failure	24 775	3.67%
Pass	460 182	68.20%
Σ	674 748	100.00%

(a) Filtering Run Results

SB TB	Pass	Borrow UB	Other UB	Timeout	Unsupported	Failure	
Pass	452 790	3564	2	51	34	36	456 477
Borrow UB	31	2992	0	0	0	0	3023
Other UB	4	4	15	0	0	0	23
Timeout	431	7	0	12	0	1	451
Unsupported	33	0	0	0	39	0	72
Failure	44	1	0	0	0	91	136
Σ	453 333	6568	17	63	73	128	460 182

(b) Results of Comparing Tree (TB) and Stacked Borrows (SB)

Fig. 3. Results of Running 674 748 Tests From 29 990 Crates

if intermediate nodes are determined to be unreachable, which avoids severe slowdowns and out-of-memory errors. With a working Miri that can test code for both Tree Borrows and Stacked Borrows violations, as well as all other kinds of UB, we then ran the test suite of the 30 000 most often downloaded crates on crates.io.

We would like to know the "amount of code" that is affected by Tree Borrows. As a first approximation, we could count the number of crates that are affected by Tree Borrows or Stacked Borrows. But this would skew the numbers, since smaller crates with smaller test suites would be over-represented. Instead, we analyze each *test* individually. Since larger crates tend to have more features, which are tested by more tests, this also allows detecting if *e.g.*, a crate is mostly compatible with Tree Borrows, except for one feature, while being mostly incompatible with Stacked Borrows. This allows our analysis to be more finely-grained.

For each of the 30 000 crates, the experiment then proceeds as follows: first, the test suite (each test individually) is run under Miri, but with both Tree Borrows and Stacked Borrows disabled. We call this run the "filtering run." This run serves to eliminate tests that, for one reason or another, cannot be tested. These include tests that use features not supported by Miri (*e.g.*, more advanced interaction with the host OS, inline assembly, platform-dependent features, calls to functions written in C, *etc.*), tests that simply fail due to bugs in the program they are testing,[7] or those that exceed the 60 second timeout we imposed on each execution. Also filtered are tests that cause UB already without Tree Borrows or Stacked Borrows, due to *e.g.*, out-of-bound accesses or use-after-frees.

Once only passing tests remain, we then run these again with Tree Borrows checking turned on, and then a third time with Stacked Borrows. These two runs are the "comparison runs." This then detects code that is incompatible with Tree Borrows or Stacked Borrows or both, but would work if there was no aliasing UB in Rust. For these runs we increased the timeout to 150 seconds, since the added Tree Borrows and Stacked Borrows checking takes extra time.

4.3 Experimental Results

The raw results of the experiment are shown in Figure 3. As can be seen, there were 674 748 tests in total. A bit more than half of all crates actually had a testsuite. 10 crates had to be excluded from the analysis because their test suite did not run to completion.[8]

Of the 674 748 total tests, about 68% passed the filtering stage. The others were filtered, the precise distribution of which can be seen in Figure 3a. Tests listed under *Timeout* ran into the

[7] Miri is specifically designed to trigger usually under-used codepaths around weak memory, which might explain how some published libraries have failing tests.

[8] We aborted crates that took more than a week to finish.

60s timeout. Those under *Unsupported* used a feature that Miri does not support. Those under *UB* triggered Undefined Behavior unrelated to Tree Borrows or Stacked Borrows (*e.g.,* a use-after-free). Finally, those under *Failure* failed due to an assertion failure or panic in the test.

Of those that passed the filtering stage, the vast majority did not trigger any issues under either Tree Borrows or Stacked Borrows. The fact that most tests work is not too surprising, since most libraries on crates.io are written entirely in safe code.

Answering the research questions. We now analyze the results of the two comparison runs, which are shown in Figure 3b. There, we can see that 6568 tests failed with aliasing model violations ("Borrow UB") under Stacked Borrows, while 3023 tests failed under Tree Borrows. This is a reduction by 53.97%, which is quite substantial. Looking at it in more detail, we can see that almost all of the tests still violating Tree Borrows rules also violate Stacked Borrows. Only 31 tests that previously passed Stacked Borrows were in violation of Tree Borrows rules. Thus, we have answers to our research questions: More than half of the code that was previously unsupported by Stacked Borrows is now accepted by Tree Borrows ($Q1$), while the amount of code newly broken by Tree Borrows is extremely small ($Q2$).

Note that these numbers are biased against Tree Borrows, in the sense that parts of the Rust ecosystem have already been using Miri for a few years and crate authors have been guided towards writing Stacked Borrows-compliant code. Despite this bias, Tree Borrows still fares significantly better than Stacked Borrows.

These findings are also confirmed by a recent preprint studying UB bugs in Rust FFI code [25].[9]

Timeouts and flaky tests. We also briefly discuss what the other cells in Figure 3b tell us. The first, somewhat worrying trend, is that there was an order of magnitude more timeouts under Tree Borrows than under Stacked Borrows. This is worrying because each such timeout could have potentially been a test that had UB under Tree Borrows, but aborting the test early prevented that UB from being observed. We suspect that this is not the case, since such tests usually consist of a rather large loop, and experience shows that aliasing violations usually appear quickly, within the first few iterations of the loop. Nonetheless, even if we assume that a disproportionately large number of these timeouting tests should actually count as UB under Tree Borrows (say 15% instead of the normal 1%), then the number of tests newly broken would still remain double-digit and thus extremely small. Those numbers do however tell us that the performance of Miri under Tree Borrows should be improved even more, beyond the speedups we already developed here.

Unrelated to timeouts, we observe that there were several dozen tests that passed the filtering run, but then used unsupported features, exhibited UB unrelated to Tree Borrows or Stacked Borrows, or failed in one of the two comparison runs. This is a bit surprising, since enabling Tree Borrows or Stacked Borrows should not affect program behavior, aside from adding more UB. After looking at these tests in more detail, we found out that they are so-called *flaky tests*, *i.e.,* tests that rely on non-determinism. Miri emulates non-determinism with randomness, and running such flaky tests several times can lead to some runs succeeding, and other runs failing, depending on the randomness. This is somewhat expected, especially since Miri contains other features that make usually under-utilized parts of tests more common. It is also consistent with the test results, since there seems to be no correlation between whether such an unexpected failure occurred in the Tree Borrows or Stacked Borrows run.

A closer look at newly broken code. Finally, we closely inspected some of the tests that were newly broken by Tree Borrows. These 31 tests are from 12 crates in total. All crates except five

[9] Since the Tree Borrows implementation has been integrated into Miri, it has already been available to developers and other researchers for some time.

also had Stacked Borrows violations in other parts. For those five that took care to comply with Stacked Borrows, we further analyzed the reasons for the Tree Borrows failure. For four of them, the reason for incompatibility was reliance on "the quirk" (see discussion in §4.1). This is usually due to superfluous references being created for ergonomics reasons;[10] these can then be fixed by making the code slightly more verbose. For the last case, the reason was that a struct should have been marked as having interior mutability, but Stacked Borrows did not complain due to a tricky interaction of both the quirk and its overtly simplistic handling of two-phased borrows. In all cases, the required fixes were small, crate authors acknowledged our bug reports as legitimate, and most have already merged our fixes.

We note that our test run here tries to measure a moving target. In fact, the test run outlined here is actually the second such run, since the first run had to be aborted and restarted after we discovered a critical bug in Miri. Nonetheless, we already started analyzing the crates that were found in the earlier test run. As we reported the incompatibility with Tree Borrows, one crate author implemented our fix so quickly that the UB had already disappeared by the time we solved our own bug in Miri and restarted the run. That crate, too, had relied on "the quirk". If Tree Borrows grows to be more accepted by the Rust community in the future, we expect the number of crates "in the wild" which are only incompatible with Tree Borrows to decrease.

5 Proving Validity of Aliasing Optimizations

The goal of Tree Borrows is to enable optimizations around memory accesses, even in the presence of unknown code. In this section, we give the key ideas for how to prove some of the optimizations outlined in the earlier sections. The supplementary material for this paper contains a Coq [31] proof of those optimizations, based on a program logic that helps to justify a wide range of similar transformations. Unfortunately, we do not have the space to present the program logic here. Overall, the proof amounts to around 32 000 lines of code.

Deleting reads. In the first example below, we aim to optimize a read of a shared reference, but the reference we use is passed to a foreign function before being read the second time:

Example 16

```
let x : &i32 = &... // arbitrary code before
let val = *x;
f(x);             // arbitrary code in-between, which can also use x
let ... = *x;     // Transformation: use `val` instead
...               // arbitrary code afterwards
```

In order to justify this transformation, we need to delete a read (of `x`) in the transformed program (or *target program*), and prove that the read in the original program (or *source program*) returns the value `val`. The key proof idea is to attach an invariant to the tag in the tree corresponding to `x`. This invariant says that either the tag is Disabled, or it is Frozen and the value stored at `x` is `val`. We then prove that this invariant is maintained by all program steps, which means it must still hold after the call to `f(x)`. The reason it is maintained is that Frozen tags become Disabled on foreign writes, and disallow local writes, so it is impossible to change the value while keeping the tag Frozen. Once `f` returns, we check whether the tag has transitioned to Disabled. If yes, then the source program has UB due to reading from a Disabled reference, which is a contradiction—for the optimization, we get to assume that the source program does not have UB. Therefore, the tag must be Frozen, and thus we know that we read the expected value.

At this point, the proof seems done, but this is not yet the case: The read that we removed not only reads from memory, it can also modify the permissions tracked in the Tree Borrows state

[10] Rust code using raw pointers is usually more verbose than a seemingly equivalent version using references.

machines. Therefore, the tree might be in a different state in the target than it was in the source. Specifically, a protected Reserved cousin of the tag x might be conflicted in the source, while still being unconflicted in the target.

Luckily, the remaining executions of both programs will be identical despite this difference. In particular, if Reserved tags in the target are not conflicted, while their source counterparts are, this can never change the behavior of a UB-free program. We can therefore use the standard simulation relation technique to show that despite the states not being exactly the same, the two programs still produce the same result.

Optimizing with protectors. Next, we consider Example 17 shown below, where two writes are combined into one, across unknown code in the middle. The main difficulty here is that when g runs, the memory between source and target differs: The source has x storing 10, but in the target, it is 0. The only reason this works is because x is protected, and because g is not given access to a reference or raw pointer derived from x. This ensures that g cannot observe the value stored in x. Thus the values can diverge temporarily, without affecting g.

Example 17

```
fn write_opt(x: &mut i32, f: impl FnOnce(), g: impl FnOnce()) {
  // x is protected, since it is a function argument
  f();
  *x = 10;         // Transformation: write 0
  g();
  *x = 0;          // Transformation: delete write
}
```

In our proof, this means that we can again attach an invariant to x, expressing that it is Unique, protected, and the value stored there is 10 in the source and 0 in the target. Since foreign writes to protected Unique tags are immediate UB, we can prove that this invariant is maintained by arbitrary code that does not have access to x. Thus, the invariant still holds after the call to g.

Like before, we need to also prove that deleting the write in the target is admissible. Unlike before, the main difficulty is proving that afterwards, the states are still in simulation. This is where the *protector end semantics* described in §3.2 becomes relevant, *i.e.,* the implicit reads and writes that are executed when a reference loses its protector: the simulation relation for protected references is quite weak to account for the possible differences between source and target that can be introduced by transformations such as this. It would not be correct to use a similarly weak simulation relation for *unprotected* references. The implicit accesses on protector removal ensure that even if the states were quite different while the protector was active, they become (almost) the same when the protector is removed, thus letting the proof pass through.[11]

A program logic for Tree Borrows optimizations. Formally, we carry out these proofs using Simuliris [9], a relational separation logic framework built atop Iris [13, 17] for proving *fair, termination-preserving contextual refinements* of concurrent programs, which comes equipped with built-in reasoning rules for unknown code. Simuliris has previously been used to reason about Stacked Borrows [9, §7]; we use this as the basis for our own proof efforts. Using Simuliris, we built a Tree Borrows Optimizations Program Logic, which in turn is able to prove that the two examples above are correct transformations, even in the presence of concurrency.

On top of this, the program logic can be used to prove a wide range of further reorderings for *protected* references. Specifically, the logic permits arbitrary insertions, deletions, and reorderings of reads and writes, subject to three restrictions, which apply mostly to &mut references. First, such mutable references may not be used by the arbitrary code in between the accesses we are

[11] And indeed, without the implicit accesses, we could construct counterexamples to these optimizations.

transforming (*e.g.*, not be passed to g in the example above). Secondly, the activating write to a mutable reference, which transitions it from Reserved to Unique, cannot be moved, and new writes can only be inserted after this activating write. Lastly, once the protector ends, or once the mutable reference is passed to unknown code, the values stored in that mutable reference must again be in simulation, and no further insertions/deletions/reorderings are allowed. This logic is sufficiently strong to verify the optimizations sketched in Example 1 and Example 14.

Optimizing without protectors. Although Tree Borrows permits a great deal of optimizations involving protectors, the optimization potential for references without protectors is more limited: without a protector, the lifetime of a reference ends at its last use, so transformations that move the last use down are generally not permitted. Verifying these transformations for unprotected references is made even more challenging by concurrency: such transformations are only correct in Rust because data races are UB, which for simplicity we have not modeled in this paper (we use a simple sequentially consistent semantics that allows racy behaviors). For the proofs above, this was not a problem because protectors already cause enough UB without any special treatment of data races; therefore, we did not have to reason about data races.

As a consequence, we were not able to use Simuliris to verify correctness of transformations on unprotected references, with the exception of Example 16 above. However, some simple and common optimizations such as read-read reordering *can* be verified in a sequential model of Tree Borrows without reasoning about unknown code. Specifically, we have proven that starting in an arbitrary state, if we execute Example 18 in its original form and in its transformed form, we end up in the *exact same state* on both sides:

Example 18

```
let x : &mut i32 = &mut ...;
let y : &mut i32 = &mut ...;
let xv = *x;  // Transformation: move this read down just below `let yv`
let yv = *y;
...
```

As explained in §2.2, this crucially relies on the fact that a foreign read causes a mutable reference to transition to Frozen, not Disabled. In contrast, Stacked Borrows does *not* support this reordering.

Comparison with Stacked Borrows. Tree Borrows permits slightly fewer optimizations than Stacked Borrows. Specifically, Stacked Borrows allows inserting and reordering writes without any regard for the activating write. This is a direct consequence of Stacked Borrows' *not* having support for two-phase borrows, and thus considering all mutable references to be immediately Unique.

In terms of which optimizations have actually been *proven* correct, the original Stacked Borrows paper [12] proved that under certain conditions, a read can be hoisted up across unknown code even for an unprotected reference. Specifically, this requires that next to the place that the read is moved to, there already is an access or retag of the relevant reference. However, this was proven only for sequential programs. Simuliris [9] later claimed to extend this proof to the concurrent setting; however, the formal operational semantics they use has a fatal flaw that invalidates this result. (This was confirmed by the authors.) We are convinced that the optimization is correct even for concurrent code in both models; however, proving this would require combining data race reasoning with the aliasing model, which we leave to future work.

6 Related and Future Work

The most directly related piece of work is Stacked Borrows. We have extensively compared Tree Borrows with Stacked Borrows in §4.1. In summary, Tree Borrows loses some optimizations on mutable references in their reservation phase, but it allows read-read reordering and permits significantly more real-world code than Stacked Borrows.

Looking beyond Rust, we discuss prior work that studied the aliasing models of C and LLVM, as well as low-level pointer semantics in general.

Aliasing rules for C. The C memory model [11], loosely speaking, requires that memory that was written using a pointer of a given type is only read using pointers of the same type (with a special exception for character types). The point of this restriction is that compilers can now be sure that, for instance, writing to a `float*` pointer cannot change the result of a read through a `int*` pointer. This is often referred to as type-based alias analysis (TBAA) or strict aliasing.

However, many large-scale C programs such as the Linux kernel disable type-based alias analysis since it is incompatible with common C idioms and the only programmer-controlled way to opt out of this mechanism (by using pointers of character type) often leads to a loss of performance. In contrast, Tree Borrows is designed to be compatible with low-level Rust programming patterns, and by using raw pointers programmers can opt out of the aliasing rules without any loss of efficiency.

A very strict interpretation of the strict aliasing rules has been formalized as part of the first version of the CompCert memory model [22]; however, this model disallows the use of unions to convert data from one type to another, even though the C standard does permit this at least under some conditions. The conditions given for this in the standard are however quite ambiguous.[12] (Later versions of CompCert dropped the strict aliasing rules in favor of a simpler memory model [20].) Krebbers [15] closes that gap by giving rules for union accesses that are based on how the GCC developers interpret the standard. He shows correctness of a simple alias analysis based on this, but does not prove correctness of any transformation.

Aside from strict aliasing, C has another source of aliasing assumptions available to the compiler: the `restrict` keyword. This keyword can be used with a pointer type (as in `int* restrict`) to indicate that this pointer cannot alias with any other in-scope pointer. Specifically, `restrict` requires that accesses done through this pointer and pointers derived from it never overlap with accesses done through other pointers, except when both accesses are reads. This is quite similar to the behavior of both mutable and shared references (without interior mutability) in Rust, and we suspect that Tree Borrows could be used as the basis for a formal definition of `restrict`.

Hathhorn et al. [10] give a semantics for both strict aliasing and `restrict`, and their framework can be used to generate a tool that checks those semantics. However, they do not explain how their model navigates the ambiguities of the standard, do not prove any theorems about their model, and do not evaluate how well those semantics align with widely-used C compilers or real-world C code. It remains an open problem to resolve the tension between C as standardized, C as it is implemented in compilers, and C as it is written by programmers [27, 28].

Aliasing rules for LLVM IR. LLVM is a project providing a low-level compiler backend and many optimizations organized around a shared intermediate language, the LLVM IR. The IR supports a `noalias` attribute on function arguments which closely matches the C `restrict` qualifier (in particular, it allows aliasing of read-only accesses). In particular, LLVM is used as the main backend by the Rust compiler, and Rust annotates function arguments of reference type with `noalias`. Tree Borrows (like Stacked Borrows) was designed with this in mind, so that a Rust program that complies with the rules of Tree Borrows should translate into an LLVM IR program that satisfies all the assumptions implied by `noalias`. However, while the meaning of `noalias` is described informally in the LLVM Language Reference [30], it has not been formalized to our knowledge, so

[12] Specifically, the standard requires “that a declaration of the completed type of the union is visible”, without giving a precise definition of “visibility”.

we cannot formally relate Tree Borrows and `noalias`. Notably, while Alive2 [23] and VellVM [33, 7] formalize large parts of LLVM IR's semantics, they do not support `noalias`.

Low-level memory models. Modeling low-level pointer operations correctly has been the topic of a lot of recent work [7, 26, 18, 14, 16, 29]. However, these models do not impose fine-grained restrictions on how multiple pointers pointing to the same memory are allowed to alias each other.

Future work. In future work, it would be interesting to extend our framework for proving correctness of Tree Borrows optimizations. This framework currently has two major limitations: (a) the activating write of a mutable reference can not be reordered down (as in Example 11)[13] and (b) optimizations that additionally require reasoning about data races cannot be verified. Reasoning about the interaction of data races and the tree-based aliasing model is quite complicated, so this will likely require a complete re-design of our optimization framework.

Acknowledgments

We thank Ben Kimock (saethlin) for their tool `crater-at-home`, which we used for the experiment described §4.

References

[1] Unsafe Code Guidelines Issue #134. 2019. Stacked Borrows: raw pointer usable only for T too strict? URL redacted for double-blind review.

[2] Unsafe Code Guidelines Issue #248. 2020. Stacked Borrows: track raw pointers more precisely. https://github.com/rust-lang/unsafe-code-guidelines/issues/248.

[3] Unsafe Code Guidelines Issue #256. 2020. Storing an object as &Header, but reading the data past the end of the header. URL redacted for double-blind review.

[4] Unsafe Code Guidelines Issue #273. 2021. Stacked Borrows: using the topmost "untagged" item is not "monotonic". https://github.com/rust-lang/unsafe-code-guidelines/issues/273.

[5] Miri Issue #3082. 2023. Is a stacked borrows error in iced-x86 a problem? . URL redacted for double-blind review.

[6] Miri Issue #3657. 2024. Possible false positive of stacked borrow rules. URL redacted for double-blind review.

[7] Calvin Beck, Irene Yoon, Hanxi Chen, Yannick Zakowski, and Steve Zdancewic. 2024. A Two-Phase Infinite/Finite Low-Level Memory Model: Reconciling Integer–Pointer Casts, Finite Space, and undef at the LLVM IR Level of Abstraction. *Proc. ACM Program. Lang.* 8, ICFP, Article 263 (2024), 29 pages. https://doi.org/10.1145/3674652

[8] Anton Fetisov. 2024. Aliasing of raw pointers. URL redacted for double-blind review.

[9] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498689

[10] Chris Hathhorn, Chucky Ellison, and Grigore Rosu. 2015. Defining the undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015.* 336–345. https://doi.org/10.1145/2737924.2737979

[11] ISO. 2017. *C17 Standard.* ISO/IEC 9899:2018.

[12] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked borrows: an aliasing model for Rust. *Proc. ACM Program. Lang.* 4, POPL (2020), 41:1–41:32. https://doi.org/10.1145/3371109

[13] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[14] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A formal C memory model supporting integer-pointer casts. In *PLDI.* 326–335. https://doi.org/10.1145/2737924.2738005

[15] Robbert Krebbers. 2013. Aliasing Restrictions of C11 Formalized in Coq. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings.* 50–65. https://doi.org/10.1007/978-3-319-03545-1_4

[16] Robbert Krebbers. 2015. *The C standard formalized in Coq.* Ph. D. Dissertation. Radboud University Nijmegen. https://robbertkrebbers.nl/thesis.html

[13] Reordering the activating write up is usually impossible, but moving down could be supported.

[17] 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. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

[18] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling high-level optimizations and low-level code in LLVM. *PACMPL* 2, OOPSLA (2018), 125:1–125:28. https://doi.org/10.1145/3276495

[19] Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert memory model, version 2.* Technical Report RR-7987. Inria. https://hal.inria.fr/hal-00703441

[20] Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert memory model, version 2.* Technical Report RR-7987. Inria.

[21] Xavier Leroy and Sandrine Blazy. 2008. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *J. Autom. Reason.* 41, 1 (2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

[22] Xavier Leroy and Sandrine Blazy. 2008. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *JAR* 41, 1 (2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

[23] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 65–79.

[24] Daniël Louwrink. 2021. *A Separation Logic for Stacked Borrows.* Master's thesis. https://eprints.illc.uva.nl/id/eprint/1790/.

[25] Ian McCormack, Joshua Sunshine, and Jonathan Aldrich. 2024. A Study of Undefined Behavior Across Foreign Function Boundaries in Rust Libraries. *CoRR* abs/2404.11671 (2024). URL redacted for double-blind review (the preprint is not ours, but it cites an earlier draft document on Tree Borrows).

[26] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C semantics and pointer provenance. *PACMPL* 3, POPL (2019), 67:1–67:32. https://doi.org/10.1145/3290380

[27] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the depths of C: Elaborating the de facto standards. In *PLDI.* 1–15. https://doi.org/10.1145/2908080.2908081

[28] Kayvan Memarian and Peter Sewell. 2016. N2014: What is C in practice? (Cerberus survey v2): Analysis of Responses. ISO SC22 WG14 N2014, http://www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html.

[29] Michael Norrish. 1998. *C formalised in HOL.* Ph. D. Dissertation. University of Cambridge.

[30] LLVM Project. 2024. *LLVM Language Reference.* https://llvm.org/docs/LangRef.html.

[31] The Coq Team. 2023. The Coq proof assistant. https://coq.inria.fr/.

[32] The Rust Team. 2020. Rust Compiler Development Guide: Two-phase borrows. https://rustc-dev-guide.rust-lang.org/borrow_check/two_phase_borrows.html.

[33] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473572