

Tree Borrows

Author: **Neven Villani** · ENS Paris-Saclay

Advisor: **Derek Dreyer** · MPI for Software Systems

Advisor: **Ralf Jung** · ETH Zürich

Abstract.

1 Introduction

While the purpose of type systems and typing information in programs is usually presented as primarily a matter of security (strict type systems can rule out at compilation-time a number of bugs), they also enable compilers to generate more efficient code in both space and time. Languages with strict compile-time type systems can avoid the need for typing metadata at runtime, have fewer bounds checks for memory accesses, or even in the case of Rust eliminate the need for a runtime garbage collector entirely.

In Rust the type system includes aliasing information (mutability and uniqueness), which is to be used not only for safety guarantees, but also to improve performance and enable optimizations that are only valid under certain aliasing guarantees. We aim to define formally what these aliasing guarantees are for Rust, as well as show how to check them and use them in proving the validity of compiler optimizations.

1.1 Motivating example

As a first concrete example, consider the following function:

```
fn example1(x: &mut u64, y: &mut u64) -> u64 {  
    let xval = *x; // First read of *x  
    *y = xval + 1;  
    let xval = *x; // Second (redundant ?) read of *x  
    return xval;  
}
```

Because mutable references in Rust are required to be unique, `x` and `y` must point to disjoint regions of memory. In particular the instruction `*y = xval + 1` constitutes a write to `y`, but it cannot affect the memory covered by `x`: the value `*x` is unaffected and thus the second read of `*x` is redundant. This function can be optimized to perform one fewer operation by deleting the second line on which `*x` is read, without modifying behavior.

Unfortunately there is a huge gap in the proof above: we have not specified by whom, in what contexts or in which manner mutable references are “required” to be unique. This is the purpose of Tree Borrows.

Indeed there exists in Rust the `unsafe` keywords which allows the programmer to bypass certain compiler checks by extending their available instruction set: among other things the `unsafe` keyword allows dereferencing raw pointers in the following manner:

```
fn context1() {
    let mut data = 42u64;
    let data_ptr = &mut data as *mut u64;
    let x: &mut u64 = unsafe { &mut *data_ptr };
    let y: &mut u64 = unsafe { &mut *data_ptr };
    let result = example1(x, y);
    assert!(result == 43);
}
```

Here we use `unsafe` instructions to obtain two mutable references to the same location, and we pass both of them to `example1`. While in this context the original version of `example1` will return 43, the optimized version will instead return 42.

The optimization shown here is thus not unconditionally valid: violating the unicity requirement of mutable references has enabled us to create a program in which the optimization does not preserve behavior.

However we want this optimization to be valid, on the grounds of `context1` being a “bad” program that violates some assumptions that we want to be able to make. This issue is solved by adjusting the operational semantics of Rust in a way that declares `context1` “Undefined Behavior” (UB): a program exhibits UB when it performs some forbidden instruction. Compilers can assume that programs do not exhibit UB, and optimizations are not required to preserve the observable behavior of such programs.

There is a tradeoff in what programs can be declared UB, indeed for compiler writers the more programs are declared UB the more powerful optimizations can be made and the more freedom there is in what is considered a correct compiler, while for language users the more programs are declared UB the less the execution closely matches the source code. Consider for example the two extreme cases:

- if all programs are UB then it is valid to compile all programs as an empty sequence of instructions, optimizations are too powerful and the language is so underspecified as to become useless;
- on the other hand if no program is UB, then few to no optimizations are possible.

Both sides however have an interest in the rules governing UB being clear and well-defined: if the rules are vague then compiler writers are unsure what assumptions they can make, and language users may accidentally write a program that exhibits UB.

An explicit design decision of Rust is that programs that do not use the `unsafe` keyword cannot be declared UB. For `unsafe` to be useful, it should also hold that it is not too difficult to write unsafe Rust that is not UB. Tree Borrows should thus:

- **Declare enough programs UB that some useful optimizations are valid.** We evaluate Tree Borrows on this aspect by providing proofs of validity of such optimizations using the aliasing assumptions allowed by the Tree Borrows semantics.
- **Declare as little UB as possible in programs that have already been written.** Each program retroactively declared to exhibit UB is a violation of backwards compatibility, we must ensure that these are few and justifiable. To check this aspect we implement the Tree Borrows semantics in the Miri interpreter and run the OS-independent part of the Rust standard library test suite using this semantics. We found very few rejected instances, and the suggested changes were accepted and merged back into the standard library.
- **Have rules that are simple, consistent and intuitive.** While it is difficult to measure this objectively, we argue that compared to its predecessor Stacked Borrows, Tree Borrows is more consistent in its handling of pointers. In particular a number of bug reports on

the Github page of Miri show that users misunderstand some details of the behavior of Stacked Borrows, and we have ensured that such misunderstandings are not present in Tree Borrows.

- **Be efficient.** In addition to being humanly understandable, Tree Borrows should also be verifiable in practice without too much overhead in Miri. We compare it to benchmarks of Stacked Borrows and observe a noticeable slowdown, but not to the point that it would be an obstacle to the usability of Miri.

2 Aliasing in Rust

In this section we explain the features of Rust that we are interacting with – mostly the different kinds of pointers available – and provide an intuition for the aliasing constraints they should satisfy.

We call “safe Rust” the subset of the complete Rust language (more explicitly called “unsafe Rust”) that does not use the `unsafe` keyword (and thus does not use any `unsafe` instructions).

2.1 References and raw pointers

Rust offers a kind of pointer with more guarantees than its counterpart in most languages: references. Written `&(let x: &T = &t;)` or `&mut (let x: &mut T = &mut t;)` for immutable and mutable references respectively.

These references should satisfy “aliasing XOR mutability”: for a given memory location, if a mutable reference exists there cannot also be other mutable or immutable references. We can have access to either unique write permissions or shared read permissions, not both.

As a means of interfacing with C code and expressing complex pointer manipulations, Rust also offers another type of pointers, called raw pointers, written `*const T` for immutable raw pointers and `*mut T` for mutable raw pointers. These pointers can bypass some requirements of references (there can exist multiple `*mut T` to the same location), but their use is more dangerous and dereferencing raw pointers can be done only in blocks marked by the keyword `unsafe { ... }` thus their use is usually restricted to situations where they are absolutely necessary. Several `*mut T` can exist simultaneously.

The following snippet shows some conversions between these kinds of pointers:

```
let mut data = 42u64;
let some_mut_ref: &mut u64 = &mut data;
let const_from_mut: &u64 = &*some_mut_ref;
let reborrow_mut: &mut u64 = &mut *some_mut_ref;
let mut_ptr: *mut u64 = &mut *some_mut_ref as *mut u64;
let ref_from_ptr: &mut u64 = unsafe { &mut *mut_ptr };
```

3 Limitations of existing tools

3.1 The Borrow Checker

The Borrow Checker is a compile-time verification of some aliasing rules, and in the presence of safe Rust it is able to guarantee that mutable references have exclusive access and that shared references have access to data that will not be mutated. However it is in several aspects not fine-grained enough compared to the model we want to develop.

We show here that there are both places where strictly adhering to the behavior of the Borrow Checker would lead to too little UB and others where there would be too much UB.

However in places where Tree Borrows does not follow the same rules as the Borrow Checker the following should always hold: code that does not use `unsafe` and is accepted by the Borrow Checker should never be UB.

Bypassing the Borrow Checker with unsafe code. The Borrow Checker does not track borrows for raw pointers, so the easiest — but usually incorrect — way to resolve compilation errors raised by the Borrow Checker is to insert round trips to cast references to and from raw pointers as follows

```
// Rejected by the Borrow Checker.
// Expected to be UB.
fn alternate_writes() {
    let x = &mut 0u64;
    let y = &mut *x;
    let z = &mut *x;
    *y += 1;
    *z += 1;
}

// Accepted by the Borrow Checker.
// Expected to be UB.
fn alternate_writes_raw() {
    let x = &mut 0u64;
    let y = unsafe { &mut *(x as *mut u64) };
    let z = unsafe { &mut *(x as *mut u64) };
    *y += 1;
    *z += 1;
}
```

Since the explicit purpose of Tree Borrows is to also verify and optimize `unsafe` code, it should be more robust than this. This is both so that `unsafe` code actually gets checked and so that the use of `unsafe` does not completely block all optimizations.

Borrow conflicts undecidable at compile-time. As we have shown in `example1` above, whether a function call triggers UB is in general dependent on what arguments it receives at runtime. Since the usual reason that `unsafe` code is used in the first place is usually that its safety relies on properties undecidable at compile-time, we define UB at runtime. If some piece of code is unreachable then it cannot produce UB.

This is however not true of the Borrow Checker, which has to operate at compile-time, as shown by the following example

```
// Rejected by the Borrow Checker.
// Expected to not be UB.
fn unreachable_borrow() {
    let x = &mut 0u64;
    let y = &mut *x;
    if false {
        let z = &mut *x;
        *z += 1;
    }
    *y += 1;
}
```

3.2 Stacked Borrows

Stacked Borrows already addresses the two issues we have raised above with why the Borrow Checker is insufficient: it executes at runtime and it handles `unsafe` code as well. There are however a few aspects in which Stacked Borrows behaves in ways that are too strict or too unpredictable.

Reads should not invalidate other reads. An optimization that should always be possible is to permute two adjacent and independent read accesses. Since a read access cannot alter the outcome of another, permuting two reads produces a program that is guaranteed to have the same outcome.

Unfortunately that is not an optimization that Stacked Borrows always allows because under some circumstances permuting two adjacent reads can introduce new UB that was not in the original program, and of course introducing UB in a program that did not contain any is not a valid optimization.

Thus of the two following functions, in which the only difference is a reordering of read-only accesses, only one is UB and replacing the other with it is not a valid program transformation.

```
// UB according to Stacked Borrows.
fn read_xy() {
    let x = &mut 0u8;
    let y = unsafe { &mut *(x as *mut u8) };
    let _val = *x;
    let _val = *y;
}

// Not UB according to Stacked Borrows.
fn read_yx() {
    let x = &mut 0u8;
    let y = unsafe { &mut *(x as *mut u8) };
    let _val = *y;
    let _val = *x;
}
```

Accesses, not creations, are the actual violations. As shown by the code below, Stacked Borrows considers creating a mutable reference to already be a violation of the requirement that there is no mutable access to the data under a shared reference, even if there is no write access involved. Since this is needlessly strict and also an instance of the previous concern on reads not invalidating reads, we do not wish for Tree Borrows to declare a simple creation without access of a mutable reference to be a write access.

```
// UB according to Stacked Borrows.
// Should not be UB according to Tree Borrows.
fn unused_borrow() {
    let x = &mut 0u64;
    let y = unsafe { &*(x as *const u64) };
    let _z = unsafe { &mut *(x as *mut u64) };
    let _val = *y;
}
```

Proper implementation of 2-phase borrows. As stated in [?], 2-phase borrows should act as shared references during their reserved phase. This is not how Stacked Borrows defines them:

in Stacked Borrows, 2-phase borrows are raw pointers before their activation.

In particular the following piece of code shows the kind of pattern that this improperly allows. The fact that explicit reborrows of the form `&mut *x` are never 2-phase borrows adds to the confusion: this makes programs suddenly become UB if we remove some `&mut*` or if we inline some functions. We wish for Tree Borrows’s handling of 2-phase borrows to be more strict and more consistent.

```
// Not UB according to Stacked Borrows.
// Should be UB according to Tree Borrows.
fn write_during_2phase() {
    let x = &mut 0u8;
    let xraw = x as *mut u8;
    print(
        x,
        unsafe { *xraw += 1; },
    );
}

// UB according to Stacked Borrows.
// Should also be UB according to Tree Borrows.
fn write_during_reborrow() {
    let x = &mut 0u8;
    let xraw = x as *mut u8;
    print(
        &mut *x,
        unsafe { *xraw += 1; },
    );
}

fn print(x: &mut u8, -: ()) {
    println!("{}", x);
}
```

On handling pointee types of unknown size. Stacked Borrows needs to know at the moment of reborrow the range that a pointer covers. This in particular makes it unable to handle types of unknown size and accesses outside of the reborrowed range. As an example the following code is rejected by Stacked Borrows:

```
// UB according to Stacked Borrows.
// Should not be UB according to Tree Borrows.
fn offset_outside_reborrowed_range() {
    let mut data = [0u8, 1, 2];
    let x1 = (&mut data[1]) as *mut u8;
    unsafe { *(x1.add(1)) = 3 };
}
```

4 The Tree Borrows aliasing model

4.1 Tree Structure

The core principle of Tree Borrows is that in order to detect aliasing between pointers we associate a *permission* to each pointer. Conflicting aliases manifest in the form of attempted

accesses with insufficient permissions: if a tag does not allow write accesses it means that writing through this tag would violate the immutability assumptions of other pointers.

These permissions are dynamic during the execution of the program, and are updated based on the various pointer creations and accesses that occur. As long as all accesses are done through pointers with sufficient permissions there is no UB.

Pointers are identified by their unique *tag*. Over the lifetime of a pointer, various read or write accesses may cause its permissions to be updated. Tree Borrows has the property that the update of permissions is a process that only requires local information concerning which pointers were derived from which other pointers, and we store this information in a *borrow tree*.

4.1.1 Creation of a new Node

For each memory location, we maintain a borrow tree which contains the following data:

```
// Per-tag data consists of
// - permissions
// - local tree structure
struct Node {
    // The pointer (identified by its tag) that this node represents
    tag: Tag,
    // The parent pointer; the root has no parent
    parent: Option<Tag>,
    // The child pointers
    children: Vec<Tag>,
    // The current permissions associated with this tag
    permissions: ReadWritePerms,
    /* Some lazy initialization and debug information omitted */
}

// Per-location data consists of
// - aggregated per-tag data for all relevant tags
struct Tree {
    root_tag: Tag,
    nodes: Map<Tag, Node>,
}
```

All pointers tracked by Tree Borrows have an associated tag. Several pointers can share the same tag, making them indistinguishable from each other.

When a new pointer *y* is derived from an existing pointer *x*, depending on the kind of operation that caused the creation of *y* and the underlying pointee type,

- either *y* will receive the same tag as *x* meaning that from this point onward *x* and *y* are indistinguishable from the point of view of Tree Borrows,
- or a new fresh tag will be created, associated with *y*, and recorded in the tree structure as a child of the tag of *x*. This new tag will have its own associated permissions.

Moreover for the purposes of what we will discuss in ??, creation of a child tag from a parent tag *x* counts as a read access through *x*. Among other things, this action ensures that *x* actually has permission to access the locations it is being reborrowed on.

No read access will be performed and no new tag will be created in the following cases:

- mutable references `&mut` whose underlying type is `!Unpin`;
- shared references `&` whose underlying type is `!Freeze` (has interior mutability);

- raw pointers `*mut` and `*const`.

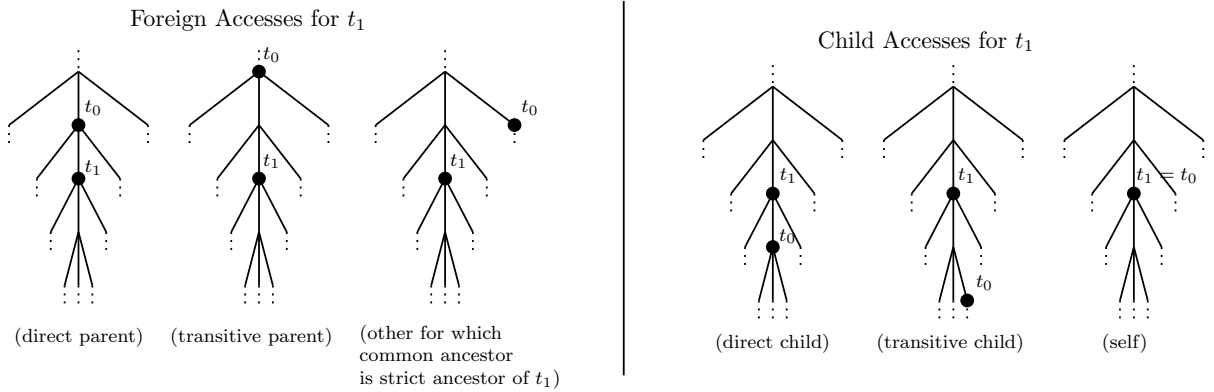
Those kinds of pointers share the property that they do not satisfy the rule of "aliasing XOR mutability", and must thus be handled by Tree Borrows differently from other more restricted pointers.

As an immediate optimization, one can notice that the tree structure will be identical for all locations of an *allocation*, meaning that although the permissions must be stored and updated on a per-location basis, the borrow tree itself can be shared at the allocation level.

4.1.2 Navigating the Tree

For the purposes of the permission update mechanism described in ??, we introduce here some terminology.

Consider a tag t_0 through which an access was performed, and a tag t_1 from the same allocation whose permissions will be updated. From the point of view of t_1 we call an access through t_0 a *child access* if t_0 is a transitive child of t_1 (including t_1 itself). In all other cases — which include strict transitive parents of t_1 as well as all pointers who do not share a branch with t_1 — we call an access through t_0 a *foreign access*.



4.2 Pointer permissions

4.2.1 Available permission combinations

Pointers can have the following permissions:

- read permissions: `Read` or `!Read`;
- write permissions: `Write` or `Future Write` or `!Write`.

We choose to introduce `Future Write` permissions as a way for Tree Borrows to natively handle 2-phase borrows, as is explained in ??. This does not directly allow write accesses through this pointer, but it reserves a right to obtain such `Write` permissions later in the execution.

Tree Borrows uses the following combinations of permissions:

```
enum ReadWritePerms {
    Reserved = Read + Future Write,
    Active = Read + Write,
    Frozen = Read + !Write,
    Disabled = !Read + !Write,
}
```

They are to be interpreted in the following way:

- a `Disabled` tag corresponds to a pointer whose lifetime has ended;

- a **Frozen** tag gives read-only permissions to a pointer, so it is suitable for shared references without interior mutability;
- an **Active** tag corresponds to a mutable reference that is currently being used mutably;
- a **Reserved** tag corresponds to a mutable reference that has not been used mutably yet.

4.2.2 The purpose of Future Write

The following code features a so-called “2-phased borrow”:

```
impl X {
    fn method(&mut self, ...) { ... }
}
```

```
x: &mut X
x.method(arg1, arg2, ...);
```

where the method call desugars to approximately

```
let x_bor: &mut *x;

let arg1 = ...;      |----- special zone within which x is still accessible,
let arg2 = ...;      |         but only immutably
```

```
X::method(x_bor, arg1, arg2, ...);
```

As a concrete example, consider

```
v: &mut Vec<usize>
v.push(v.len());
```

which the Borrow Checker accepts despite rejecting the following

```
v: &mut Vec<usize>
let w = &mut *v;
let l = v.len();
Vec::push(w, l);
```

Therefore the existence of 2-phase borrows suggests the possibility of a mutable borrow being “delayed”: as long as it has not yet been accessed mutably, it still tolerates shared read-only access. In other words, it does not have a **Write** permission yet, but can obtain one in the future. This is exactly how our **Future Write** permission is set to behave.

Rather than limiting this behavior to 2-phase borrows only, we choose in Tree Borrows to make all mutable borrows behave in a uniform way: all mutable references will wait until their first write access to claim their **Write** permission, and will allow shared read-only access in the meantime. If not for this, a lot of code currently being written would not be accepted, such as some code that follows the following pattern

```
let ptr = vec.as_mut_ptr(); // More generally:
if vec.len() > 0 {          // - some mutable reborrow
    do_stuff(ptr)           // - use base pointer immutably
}                           // - then use the reborrow
```

A concrete example currently in use in the standard library test suite:

```

let mut x = 2;
let xref = &mut x;
let xraw = &mut *xref as *mut _; // create a mutable reborrow
let xshr = &*xref;
assert_eq!(*xshr, 2); // read-only usage of the base pointer
unsafe {
    *xraw = 4; // usage of the mutable reborrow
}
assert_eq!(x, 4);

```

Extending the behavior of 2-phase borrows to all mutable borrows also serves towards our goal of allowing all reorderings of read-only operations: it lets us exchange reads through mutable (but not used mutably) references with each other and with reborrows of other mutable references.

4.2.3 Initialization

The initial permissions depend of the type of borrow. Recall from ?? that raw pointers do not receive new permissions and instead use the same tag and permissions as their parent pointer. This leaves mutable and shared references for which we must determine an initial permission.

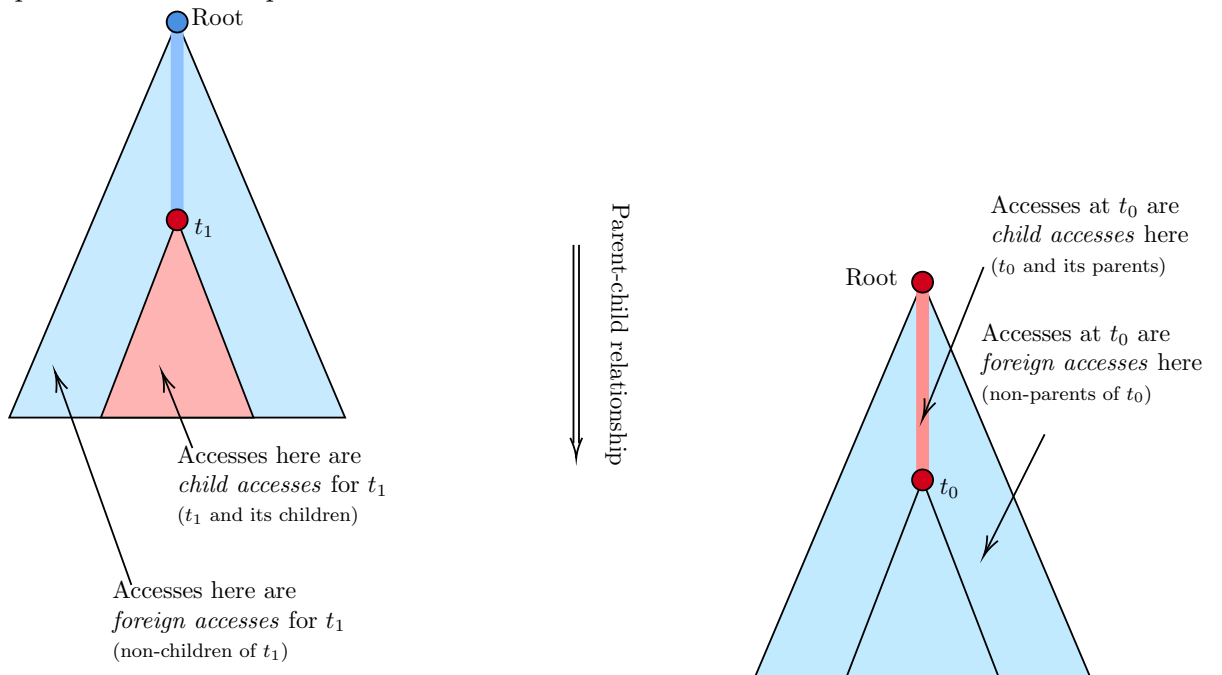
All references obtain a **Read** permission initially. This is required because we perform a fake read access upon reborrowing a reference, and it asserts that all references are dereferenceable (they are tagged **dereferenceable** in LLVM).

In addition, mutable references obtain a **Future Write** permission upon initialization making them **Reserved**, while shared references receive **!Write** which makes them **Frozen**.

4.3 Updates

We now describe how permissions are updated after an access is performed. The update process is roughly as follows.

In reaction to an access at t_0 , traverse the tree. For each t_1 node of the tree, if t_0 is a child of t_1 then the access is a child access at t_1 ; otherwise it is a foreign access at t_1 . Based on the type of access (both child/foreign and read/write) and some local information, determine the new permissions for the pointer.



4.3.1 Intuition on effects of accesses

Child accesses If **Active** is to represent mutable references, then it must allow child writes as well as child reads. Our interpretation of **Reserved** as a mutable reference that is not yet used mutably implies that it must be unaffected by child reads, but turn into **Active** on the first child write.

Frozen representing a non-interior-mutable shared reference, it must allow child reads. As child writes are forbidden on such references, we declare any child write on a **Frozen** to be UB.

Any child access is obviously UB on a **Disabled**.

Foreign accesses on Frozen Since shared references allow shared read access, **Frozen** must be unaffected by Foreign reads. As shared references on types without interior mutability assume that no other reference accesses the same data mutably, a **Frozen** must become **Disabled** upon a Foreign write.

4.3.2 Accesses outside of initial range

Tree Borrows is capable of handling pointers with unknown size as well as using a pointer to access data outside of the range it was reborrowed for.

One such case is

```
fn access_after_offset() { unsafe {  
    let data: [u64; 2] = [0, 1];  
    let fst = &mut data[0] as *mut u64;  
    let snd = fst.add(1);  
    ptr::swap(fst, snd);  
} }
```

Here the reborrow for `fst` only covers `data[0]`, but `snd` is then derived from `fst` and translated outside of its original range. As we still wish to check that even for these accesses no aliasing assumptions are violated, we track permissions even for locations outside of the range of the initial reborrow.

These permissions outside the range can be initialized lazily rather than as soon as the reborrow occurs, because it is costly to immediately add permissions on the entire allocation when they will most likely never be actually used by a child access.

We do not perform a read access on reborrow for locations outside of the range.

4.3.3 On whether to propagate loss of permissions

Consider the following pattern:

```
let x = &mut 42u64;  
// x: Reserved  
let y = &mut *x; // Child read for x  
// x: Reserved  
// |-- y: Reserved  
*y += 1; // Child write for x and y  
// x: Active  
// |-- y: Active  
let z = &mut *y; // Child read for x and y  
// x: Active  
// |-- y: Active  
//      |-- z: Reserved  
let _ = *x; // Child read for x; Foreign read for y and z
```

```
// The above setup produces the following tree:
// x: Active
// |--- y: Frozen
//     |--- z: Reserved
```

No UB occurs here, the model thus allows having in the tree a **Frozen** parent with a **Reserved** child. This looks like we have derived a mutable reference from a shared reference, which would be worrying.

We could resolve this concern by making **z** **Frozen** as well (i.e. requiring children of **Frozen** to be at least **Frozen** and children of **Disabled** to be at least **Disabled**), but in fact this would result in exactly the same UB as in the current model and thus non-**Frozen** children of **Frozen** are not a concern.

Indeed from the point of view of child accesses, **z** has already in practice lost its **Write** permission, since any child write for **z** is also a child write for **y** and is thus UB.

Even with the addition of Protectors, and child accesses no longer being the only kind of accesses that can cause UB, we can observe that the above pattern can only occur when the tag is unprotected.

Thus not propagating loss of permissions from parents to children (not forcing **Frozen** parents to also have **Frozen** children) can make pointers have more permissions in appearance (what their **ReadWritePerms** suggest) than they actually do (what accesses would cause UB), but never in a way that would affect detection of UB.

4.4 Summary

When creating a new pointer **z** from an existing **y**

- if **z** is a **Unpin** mutable reference
 - perform the effects of a read access through **y**
 - add a new child of **y** in the tree
 - give it the permissions **Reserved** = **Read** + **Future Write**
 - keep track of whether it has interior mutability or not
- if **z** is a non-interior-mutable shared reference
 - perform the effects of a read access through **y**
 - add a new child of **y** in the tree
 - give it the permissions **Frozen** = **Read** + **!Write**
- otherwise give **z** the same tag as **y**, they are indistinguishable from now on

When reading through a pointer **y**

- for all ancestors **x** of **y** (including **y**), this is a child read
 - assert that **x** has **Read** (i.e. is **Frozen** or **Reserved** or **Active**)
 - otherwise (if **x** is **Disabled**) this is UB
- for all non-ancestors **z** of **y** (excluding **y**), this is a foreign read
 - turn **Write** into **!Write** (i.e. **Active** → **Frozen**); this is UB if **z** is protected
 - if **z** is protected, turn **Future Write** into **!Write** (i.e. **Reserved** → **Frozen**)

When writing through a pointer `y`

- for all ancestors `x` of `y` (including `y`), this is a child write
 - turn `Future Write` into `Write` (i.e. `Reserved` -> `Active`)
 - it is UB to encounter `!Write` (either `Disabled` or `Frozen`)
- for all non-ancestors `z` of `y` (excluding `y`), this is a foreign write
 - if `z` is protected this is always UB; otherwise
 - if `z` is `Reserved` and has interior mutability it is unchanged; otherwise
 - turn `Write` and ‘`Future Write`’ into `!Write` as well as `Read` into `!Read` (i.e. `Reserved` -> `Disabled` and `Active` -> `Disabled`)

5 Tree Borrows implemented

5.1 Testing the Rust Standard Library

We used the method described in `github:rust-lang/miri-test-stdlib` to evaluate on the Standard Library that Tree Borrows does not declare “too much UB”, in other words that the code that is currently in use is not considered UB according to Tree Borrows.

We found two tests that were rejected by Tree Borrows, both were instances of the following pattern:

```
let mut root = 6u8;           // Base pointer: Active
let mref = &mut root;         // Direct child: Reserved
let ptr = mref as *mut u8;    // Uses the same tag as mref
// Write to ptr makes it Active
*ptr = 0;
// Parent read from the point of view of ptr makes it Frozen
assert_eq!(root, 0);
// Attempted write is rejected because Frozen forbids writes
*ptr = 0;
```

This pattern was previously accepted by Stacked Borrows, but has now been determined to be arguably a violation of uniqueness that should not have been accepted in the first place as well as easy to patch. The fix (which simply consists of replacing `&mut root` with `addr_of_mut!(root)`) was accepted in `github:rust-lang/rust/pull/107954`.

Other widely used libraries such as `rand` and `tokio` already satisfy the requirements of Tree Borrows, suggesting that most codebases will need few to no changes if they are to enforce the Tree Borrows semantics.

5.2 Performance concerns and optimizations

Tree Borrows is slower than Stacked Borrows.

The semantics require that on every access, every tag of the allocation have its permissions updated on the corresponding range. On allocations with many reborrows this can easily lead to a naive implementation of Tree Borrows being slower than Stacked Borrows by an arbitrarily large multiplicative factor.

Fortunately Tree Borrows has access to easy optimizations that allow it to have an execution time in the same order of magnitude as that of Stacked Borrows on realistic code samples. We observe that in practice the slowdown from Stacked Borrows to Tree Borrows is mostly less than $\times 2$ on benchmarks that are specifically designed to stress the borrow tracker, and about $\times 1.3$ on general tests that don’t particularly attempt to push the borrow tracker to its limits.

Note further that the benchmarks that follow compare a very optimized implementation of Stacked Borrows with a Tree Borrows implementation that was only optimized up to the point that it would not waste too much time. Fine-tuning the garbage collector of unused tags and implementing a cache are likely to improve the performance of Tree Borrows as they have already done for Stacked Borrows.

Project	Test	Runs	SB	TB	Factor
Miri	slice-get-unchecked	5	0.56s	4.15s	x7.41
	mse	5	0.67s	1.42s	x2.12
	serde1	5	1.53s	2.60s	x1.70
	serde2	5	3.19s	5.16s	x1.62
	unicode	5	1.27s	1.99s	x1.57
	backtraces	5	4.01s	5.81s	x1.45
Stdlib	core	1	4m31s	9m15s	x2.05
	alloc	1	4m45s	5m54s	x1.24
	std/time	1	15.1s	17.5s	x1.16
Regex	lib	1	13.3s	20.6s	x1.54
Hashbrown	lib	1	31.5s	38.3s	x1.21
Tokio	lib	1	40.7s	45.3s	x1.11
Rand	lib	1	1m24s	1m31s	x1.08

- Miri

Source `github:rust-lang/miri`

Command `MIRIFLAGS="" ./miri bench`

- Miri-test-stdlib

Source `github:rust-lang/miri-test-stdlib`

Command `MIRIFLAGS="" ./run-test.sh core --lib --tests`

Command `MIRIFLAGS="" ./run-test.sh alloc --lib --tests`

Command `MIRIFLAGS="-Zmiri-disable-isolation"`

`./run-test.sh std --lib --tests -- time::`

- Tokio

Source `github:tokio-rs/tokio`

Command `MIRIFLAGS="-Zmiri-disable-isolation -Zmiri-tag-raw-pointers"`

`cargo +nightly miri test --features full --lib`

- Rand

Source `github:rust-random/rand`

Command `MIRIFLAGS="" cargo +nightly miri test --lib`

- Hashbrown

Source `github:rust-lang/hashbrown`

Command `MIRIFLAGS="" cargo +nightly miri test --lib`

- Regex

Source `github:rust-lang/regex`

Command `MIRIFLAGS="" cargo +nightly miri test --lib -- --skip encode_decode`

6 Future work

6.1 Formalization

6.2 Proving optimizations

References