Tree Borrows

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

Advisor: **Derek Dreyer** · MPI for Software Systems
Advisor: **Ralf Jung** · ETH Zürich

1 Introduction

While the purpose of type systems and typing information in programs is usually presented as primarily a matter of safety (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, can have static dispatch of generic functions, 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. For example a guarantee of immutability of the data behind a pointer enables optimizations that would not be valid if the value changed from one read to the next. Unfortunately, `unsafe` code can break these assumptions by allowing at compile-time accesses through raw pointers that do not respect uniqueness. This makes a number of desirable optimizations invalid, and it is also an indicator of subtle bugs.

We aim to define an aliasing model for Rust, which is a runtime semantics that restores some of the assumptions made impossible in the presence of `unsafe` by declaring a certain number of patterns as Undefined Behavior (UB), namely patterns that violate desirable assumptions. The compiler can then assume that no UB occurs, which rules out all programs that violate the required assumptions and enables back the associated optimizations.

1.1 Motivating example

As a first concrete example, consider the following function:

```rust
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;
}
```

The type declaration `x:&mut u64` indicates that `x` is a mutable reference to a 64-bit unsigned integer. The type of references in Rust includes mutability information, and mutable references `&mut u64` are a strict supertype of references `&u64`.

Because mutable references in safe Rust are unique, `x` and `y` must point to disjoint regions of memory (the type checker must be able to prove that any new mutable reference created is disjoint from any other reference still in use). 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 fewer operations (two fewer loads) by deleting the second line on which `*x` is read, without modifying behavior.

However there exists in Rust the unsafe keyword which allows the programmer to bypass certain compiler checks by extending the 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; // One raw pointer,
    let x: &mut u64 = unsafe { &mut *data_ptr }; // converted into two mutable
    let y: &mut u64 = unsafe { &mut *data_ptr }; // references to the same memory.
    let result = example1(x, y); // Invocation with non-disjoint x and 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 where the second read of *x is removed will instead return 42. The optimization shown here is thus not unconditionally valid: violating the uniqueness 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 the assumption of uniqueness 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 to exhibit Undefined Behavior, which as explained above allows optimizations to rule out such edge cases from their proof of validity. We call this semantics “Tree Borrows”, as a reference to the predecessor that it improves upon: “Stacked Borrows” [8].

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 ever 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.

A non-negotiable design decision of Rust is that programs that 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. At the bare minimum (but we aim much higher), safe code wrapped in an unsafe block but that does not actually use unsafe operations should not be 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 [1] and run existing test suites of various projects using this semantics. We found very few rejected instances, and all of them were

accepted as code that should have been written differently. Further testing is required, but for now we have no counter indication to the fact that the Tree Borrows semantics are in accordance with most idiomatic Rust code currently in use.

- **Have rules that are consistent and intuitive.** While it is difficult to measure this objectively, we argue that compared to its predecessor Stacked Borrows [8], Tree Borrows is more consistent in its handling of pointers. In particular a number of bug reports on the Github page of Miri (issues #1666, #1878, #2082, #2722) show that users misunderstand some details of the behavior of Stacked Borrows, and in Tree Borrows we have made the behavior of different kinds of pointers more consistent to try to minimize such misunderstandings. In particular Tree Borrows does not make a fundamental distinction between mutable and shared references, or between two-phase and standard reborrows. This is complemented by a pedagogical effort to write the description of Tree Borrows [13] in a style intended for a non-academic audience familiar with Rust. Early feedback is very positive.

- **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. Considering that the current version of Tree Borrows is not at all at the same standard of fine tuning performance than Stacked Borrows is, these results are encouraging.

- **Not disable existing optimizations.** An optimization that introduces UB in a program that did not contain any is invalid. If the aliasing model is not well enough tuned, this can make some standard optimizations no longer valid because the optimized version would contain UB. Stacked Borrows exhibits this behavior: the requirement of uniqueness is so strong as to make some reorderings of read-only operations invalid. Tree Borrows should preferrably not exhibit the same behavior.

1.2 Related work

We have already mentioned Stacked Borrows [8] which is the predecessor of Tree Borrows, and whose known inconsistencies have largely guided the design of Tree Borrows. The current implementation of Tree Borrows in Miri [1] coexists alongside Stacked Borrows with only a boolean command-line argument to switch from one to the other, which shows how similar they are in their purpose and interface.

Other previous projects have done similar work for C ([4] specifies Undefined Behavior, [9] defines an aliasing model, [11] proves the validity of reorderings that rely on aliasing guarantees) and LLVM ([10] proves optimizations), so the idea of using an aliasing model for optimizations is rather well-established.

The upcoming attempt at formalization is planned to use the Simuliris [3] framework.

2 Aliasing in Rust

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).

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 that they are guaranteed to satisfy in safe code, and should still preferably satisfy even in the presence of `unsafe` code.

This section can be skipped if the reader is familiar with the differences between mutable references, shared references, and raw pointers, as well as with the concept of two-phase borrows.

2.1 References and raw pointers

The default pointer type that Rust offers has more guarantees than its counterpart in most C-like languages: references are used in most situations where one would use a pointer in C. They are written `&` (as in `let x: &T = &t;`) or `&mut` (as in `let x: &mut T = &mut t;`) for immutable and mutable references respectively.

These references should usually satisfy "aliasing XOR mutability": for a given memory location, if a mutable reference exists there cannot also be other mutable or immutable references. These references give access to either unique write permissions or shared read permissions, but not both simultaneously. This rules out many common bugs such as race conditions or iterator invalidation.

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. Their use is made purposefully less straightforward than references through the fact that dereferencing raw pointers can be done only in blocks marked by the keyword `unsafe {...}`, which usually succeeds in making programmers resort to raw pointers only when they are absolutely necessary.

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

```
let mut data = 42u64;

// mutable reference from local variable
let some_mut_ref: &mut u64 = &mut data;

// shared reborrow of a mutable reference
let const_from_mut: &u64 = &*some_mut_ref;

// mutable reborrow of a mutable reference
let reborrow_mut: &mut u64 = &mut *some_mut_ref;

// mutable reborrow cast into raw pointer
let mut_ptr: *mut u64 = &mut *some_mut_ref as *mut u64;

// mutable reborrow from raw pointer, note the usage of unsafe when
// dereferencing a raw pointer
let ref_from_ptr: &mut u64 = unsafe { &mut *mut_ptr };
```

2.2 two-phase borrows

"two-phase borrows" are a special case of mutable borrows where requirements are relaxed in a way that often spares from having to introduce temporary variables and thinking about the order in which arguments of a function are computed.

When a mutable reference is passed as a function argument, there is a guarantee that it will not actually be used mutably before function entry. Thus the compiler can tolerate read-only accesses until function entry. The following code features one such two-phase 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;
// two-phase borrow for x_bor begins
let arg1 = ...;
let arg2 = ...;
// two-phase borrow for x_bor ends and actual mutable borrow begins
X::method(x_bor, arg1, arg2, ...);
```

As a concrete example, consider

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

which the Borrow Checker accepts. In the absence of two-phase borrows at all, one would have to write

```
v: &mut Vec<usize>
{ let l = v.len();
  v.push(l); }
```

More generally, the existence of two-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 the compiler this behavior is only present for function arguments that are implicitly reborrowed (no &mut appears in the source code), but since it executes at runtime Tree Borrows can make a finer analysis and apply this behavior to all mutable borrows.

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 even 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) }; // cast &mut -> *mut -> &mut
    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 undecideable 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; // This raises a compilation error, but this code
        *z += 1;         // is never actually executed.
    }
    *y += 1;
}
```

One should keep in mind that the role of the Borrow Checker is quite different from that of Tree Borrows.

The Borrow Checker

- must run at compile time;
- must prove that *every* code path is sound;
- cannot be complete due to undecidable properties and must thus have an unsound opt-in escape hatch (`unsafe`).

Meanwhile Tree Borrows

- has additional information that comes with executing at runtime;
- only needs to check if a single provided code path is sound;
- must within these constraints be both sound and complete.

The purpose of Tree Borrows is not to mirror 1-to-1 the behavior of the Borrow Checker, the only hard constraint beyond consistency and simplicity is that in exclusively-safe code, Tree Borrows must accept at least all the programs that the Borrow Checker accepts. A much more fair comparison is between Tree Borrows and Stacked Borrows.

3.2 Stacked Borrows

Stacked Borrows already adresses the two issues we have raised above with why the Borrow Checker is not the right tool for this job: Stacked Borrows executes at runtime and it handles `unsafe` code as well. There are however a few aspects in which Stacked Borrows behaves in ways that have been argued to be too strict or too unpredictable. All of them are documented in various GitHub issues or internal discussions in the Rust community.

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.

```rust
// 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; // Swapped this read...
    let _val = *x; // ... with this one
}
```

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. This is considered to be problematic on grounds of Issue [5], and is a pattern that was found in several existing projects when Stacked Borrows was first released.

```rust
// 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) }; // created but never used
    let _val = *y;
}
```

Proper implementation of two-phase borrows. As stated in the Rustc Development Guide [2], two-phase borrows should act as shared references during their “reservation phase”.

This is not how Stacked Borrows defines them: in Stacked Borrows, two-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 two-phase borrows adds to the confusion: this makes programs suddently become UB if we remove some `&mut*` or if we inline some functions. We wish for Tree Borrows's handling of two-phase borrows to be more strict and more consistent in order to allow optimizations and reduce confusion.

```rust
// 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, // Only this line differs from the previous example
        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. This causes problems outlined in Issues [6] and [7]. As an example the following code is rejected by Stacked Borrows, but is a common pattern:

```rust
// 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 };
}
```

More generally this excessive strictness of Stacked Borrows prevents the very common case of using a raw pointer to the first element and a size to represent an array: `(*mut T, usize)` ≃ `&[T]`

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 on each byte of memory. Conflicting aliases manifest themselves in the form of attempted accesses with insufficient permissions: if a pointer does not allow write accesses it means that writing through it would violate the immutability assumptions of other pointers. If a pointer does not allow reading it means that reading through this pointer would violate the uniqueness assumption of other pointers.

This naturally introduces to possible causes for UB:

1. UB is raised when an access is done through a pointer with insufficient permissions (e.g. a read-only pointer was written through),
2. UB is raised when a pointer loses a permission that it should have kept for longer (e.g. a unique pointer prematurely loses uniqueness).

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*. While the permissions dictate the accesses that are currently allowed, the tree structure defines how the permissions will evolve over time.

4.1.1 Additions to the tree

Pointers are represented by their *tag* (a natural integer that can be copied but not forged), so the true model uses two successive mappings of pointers to tags then tags to permissions. The model allows for several pointers to have the same tag, they are then considered identical from the point of view of the aliasing model. Each tag is associated with one node of the borrow tree. The structure and contents of the borrow tree define

- the *parent* tag and *child* tags;
- the permission that each tag has over each location (byte) of the allocation.

If several pointers share the same tag, then they have the same node in the borrow tree, and all updates are described in terms of which nodes are affected and on which memory range.

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`;
- or a new fresh tag will be created, associated with `y`, and recorded in the tree stucture 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 Section 4.3, creation of a child tag `y` from a parent tag `x` counts as a read access through `y`. Among other things, this action ensures that `y` 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`.

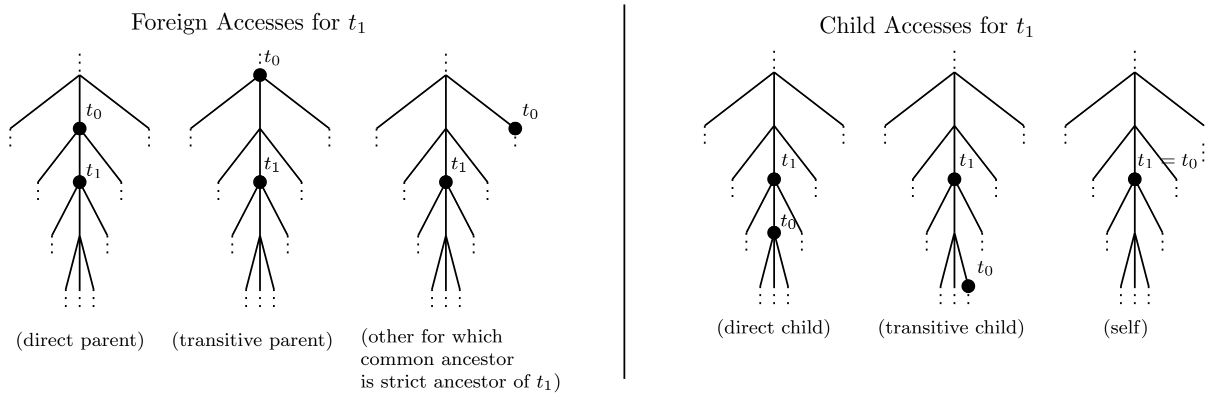

Figure 1: Accesses are classified in two categories depending on their relative position to the current tag. An access done through a (transitive and inclusive) child tag is a child access. An access done through a non-child tag (strict ancestor or non-comparable) is a foreign access.

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

To help the description of the permission update mechanism described in Section 4.3, we introduce here some terminology.

Consider a tag `t0` through which an access was performed, and a tag `t1` from the same allocation whose permissions will be updated. From the point of view of `t1` we call an access through `t0` a *child access* if `t0` is a transitive child of `t1` (including `t1` itself). In all other cases — which include strict transitive parents of `t1` as well as all pointers who do not share a branch with `t1` — we call an access through `t0` a *foreign access*. These two kinds of accesses are shown in Figure 1.

With the above tree structure fixed, Tree Borrows is parameterized by the following information:

- **how many permissions** there are (this is dictated by how many kinds of pointers exist and the states they can be in: shared or mutable, live or dead, interior mutable or not, ...)
- how each permission reacts to **child accesses** (this defines which operations are allowed by each pointer, e.g. shared pointers allow reading but not writing, deactivated pointers allow neither)
- how each permission reacts to **foreign accesses** (this defines to what extent a pointer requires uniqueness, e.g. shared pointers tolerate foreign reads, mutable pointers do not)

In other words, Tree Borrows is the conjunction of the borrow tree update mechanism with a finite automaton having for states the permissions and for transitions the kinds of accesses. The rest of this document is devoted to fine-tuning these parameters to achieve the desired amount of UB, based on both positive and negative examples.

4.2 Pointer permissions

4.2.1 Available permission combinations

Analysis of the different kinds of pointers leads us to choosing to give pointers a permission among the following:

```
enum ReadWritePerms {
    Reserved, // Represents a two-phase borrow during its reservation phase
    Active, // Represents an activated (written to) mutable reference
    Frozen, // Represents a shared (immutable) reference
    Disabled, // Represents a dead reference
}
```

We choose to introduce `Reserved` permissions as a way for Tree Borrows to natively handle two-phase borrows. `Reserved` does not directly allow write accesses through this pointer, but it keeps a right to obtain such write permissions later in the execution.

4.2.2 two-phase for all borrows

As we have explained, `Reserved` represents mutable references with a two-phase borrow not yet activated.

Rather than limiting this behavior to two-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 (become `Active`), 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

```
                               // More generally:
let ptr = vec.as_mut_ptr(); // - some mutable reborrow
if vec.len() > 0 {          // - use base pointer immutably
   do_stuff(ptr)            // - 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 two-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 Section 4.1 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.

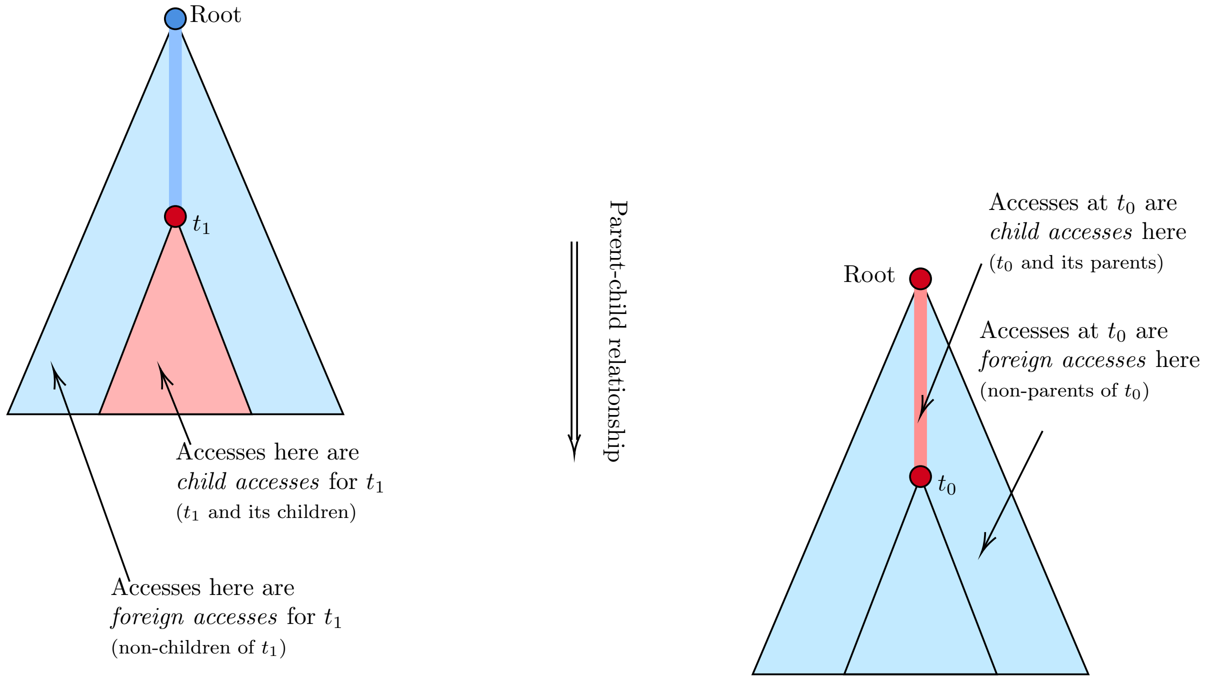
Root
t_1
Accesses here are *child accesses* for t_1 (t_1 and its children)
Accesses here are *foreign accesses* for t_1 (non-children of t_1)
Parent-child relationship
Accesses at t_0 are *child accesses* here (t_0 and its parents)
Root
Accesses at t_0 are *foreign accesses* here (non-parents of t_0)
t_0

Figure 2: Two opposite points of view for the effects of an access. Left: how a given tag reacts to accesses at various positions. Right: how an access at a given position affects other tags. The foreign/child access naming follows the left convention which is easier for describing the update of permissions, but the right convention is closer to the actual implementation and better explains some optimizations.

All references allow read accesses 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 are allowed to eventually write, so we initalized them to be `Reserved`, while shared references are initialized as `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 `t0`, traverse the tree. For each `t1` node of the tree, if `t0` is a child of `t1` then the access is a child access at `t1`; otherwise it is a foreign access at `t1`. 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 much 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` since dead references do not allow accesses.

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.

Foreign accesses on Active Several mutable references cannot coexist with each other or with shared references, so an `Active` must not remain `Active` upon a foreign access.

If another mutable reference is accessed, which corresponds to a foreign write, then this `Active` must lose all permissions and become `Disabled`.

According to the Borrow Checker, an `Active` loses all permissions on a foreign read:

```rust
// Example 3.A.1
fn main() {
    let base = &mut 42u64;
    let rmut = &mut *base;
    // base: Reserved
    // |-- rmut: Reserved
    *rmut += 1; // Child write for both base and rmut
    // base: Active
    // |-- rmut: Active
    let _val = *base; // Child read for base; foreign read for rmut
    // base: Active
    // |-- rmut: ???
    let _val = *rmut; // Compilation error
    // According to the Borrow Checker, rmut is no longer readable
}
```

However we want Tree Borrows to be suited for proving the validity of reordering any two adjacent read accesses, which means in particular that reordering two reads should not introduce new UB.

The following piece of code is accepted:

```rust
// Example 3.A.2
fn main() {
    let base = &mut 42u64;
    let rmut = &mut *base;
    // base: Reserved
    // |-- rmut: Reserved
    *rmut += 1; // Child write for both base and rmut
    // base: Active
    // |-- rmut: Active
    let _val = *rmut; // Child read for both base and rmut
    // base: Active
    // |-- rmut: Active
    let _val = *base; // Child read for base; foreign read for rmut
    // base: Active
    // |-- rmut: ???
}
```

but swapping the two last reads from *Example 3.A.2* would produce *Example 3.A.1* above, which would be UB if `Active` were to become `Disabled` on a foreign read. We thus choose to make `Active` become `Frozen` instead, which means that in both examples above `???` should be `Frozen` and UB occurs in neither.

Foreign accesses on Reserved `Reserved` is a more special case, and its behavior is guided by the following examples (and more in Appendix C):

```rust
// Example 3.R.1: Foreign read (standard two-phase borrow example)
// This must not be UB
fn main() {
    let mut x = vec![];
    // x: Reserved
    x.push( // two-phase borrow starts here for x' implicitly reborrowed from x
        // x: Reserved
        // |-- x': Active
        x.len() // Foreign read for x'
        // After this, x' must still be writeable inside Vec::push
        // thus a foreign read must not affect Reserved tags.
    );
}

// Example 3.R.2: Foreign write
// This should be UB
fn main() {
    let mut x = 2;
    let mut xref = &mut x;
    // x: Reserved
    // |-- xref: Reserved
    *x = 3; // Child write for x; foreign write for xref
    // x: Active
    // |-- xref: ???
    *xref = 4; // If this is not UB then we are able to alternate writes
              // between two references that each claim exclusive access.
              // This is bad, so xref must no longer have write permissions.
              // The data has been mutated, so xref also can't claim shared
              // read-only access. Therefore foreign writes must make Reserved
              // turn into Disabled, which causes UB in this example as desired.
    *x = 3;
}
```

These suggest that a `Reserved` should tolerate foreign reads (stays `Reserved`) but not foreign writes (becomes `Disabled`). We have already established that a `Reserved` allows child reads (stays `Reserved`) and must be changed to `Active` before the first child write.

4.3.2 Interpretation of transitions using an ordering

From the conclusions of Section 4.3.1 we can observe that all transitions follow the order `Reserved < Active < Frozen < Disabled`, which corresponds to the permissions during a "normal" lifetime of a mutable reference: it is `Reserved` upon creation to accomodate for two-phase borrows, then eventually becomes `Active` on the first child write. The next foreign read marks the end of its period of exclusive access and it can now only be accessed immutably by becoming `Frozen`. Eventually its lifetime ends altogether as a different branch claims exclusive access, it becomes `Disabled` and will remain so forever.

A possible interpretation of the established transitions would be the following:

- in reaction to a child access, the state will advance forward according to `<` until it has obtained the required read/write permissions for the access to be performed
  - `Reserved`, `Active`, and `Frozen` already allow reading, so a child read does not affect them;
  - `Active` also allows writing, so it is unaffected child Write;
  - `Reserved` does not directly allow writing, but it can advance to `Active` to obtain the missing permission;
  - `Frozen` for a write and `Disabled` for either a read or a write have no way of obtaining the required permissions, since all reachable states are also missing these permissions, this produces UB.

- in reaction to a foreign access, the state will advance forward according to `<` until it has lost the incompatible read/write permissions
  - `Disabled` having no permissions to lose, it has both many looping transitions and many incoming transitions;
  - `Frozen` is the destination for an `Active` that needs to lose its write permission;
  - `Frozen` and `Reserved` are compatible with a foreign read, hence the loops.

In addition, notice that every state that has an incoming transition for any kind of access also has a loop to itself for the same kind of access. For example a foreign read access on `Active` leads to `Frozen`, which is stable under foreign reads. Similarly a child write on Reserved leads to `Active` which is stable under child writes. In other words, all transitions induced by accesses are idempotent.

This observation is an easy consequence of the "advance until permissions are compatible" interpretation of the transitions: once a state has been reached with compatible permissions, applying the same access again will be a no-op because the state is obviously already compatible. This idempotency of all accesses opens the door for optimizations: if the consequences of a certain kind of access have already been applied to some subtree of the borrow tree, said subtree can be skipped entirely from the tree traversal if the next access is also of the same kind. This optimization has been implemented and yields very noticeable improvements (x2 to x5 speedup).

4.3.3 Protectors

The compiler optimizations that Tree Borrows should justify include LLVM optimizations, thus Tree Borrows must not allow violations of assumptions made by LLVM. Among these assumptions is `noalias` which specifies to what extent a pointer passed as an argument to a function aliases with other pointers. In Rust, mutable and shared references in function arguments are both marked `noalias`.

> `noalias`
> This indicates that memory locations accessed via pointer values based on the argument are not also accessed, during the execution of the function, via pointer values not based on the argument. This guarantee only holds for memory locations that are modified, by any means, during the execution of the function.
> — from the LLVM Language Reference Manual [12]

This can be reworded in language closer to Tree Borrows:

> An access via pointer values based on the argument is a **child access**.
> An access via pointer values **not** based on the argument is a **foreign access**.
> If a tag is marked noalias then during the execution of the function there must not be both child and foreign accesses relative to this tag if at least one of them is a write access.

This makes several pieces of code that would be allowed according to the model so far be UB according to LLVM. Any code that is UB according to LLVM must also be UB according to Rust. Examples of such code are shown in Appendix B.

To remedy this and properly detect this kind of UB, we introduce *Protectors*, named after their Stacked Borrows equivalent. Upon function entry, we add a Protector to every reference passed as argument, which is now declared *protected*. Protected pointers behave slightly differently from unprotected pointers. Protectors are removed when the function returns.

This is implemented by maintaining a `HashSet` containing call ids of functions that have not yet returned and their reference arguments, and querying this set on each transition to know if

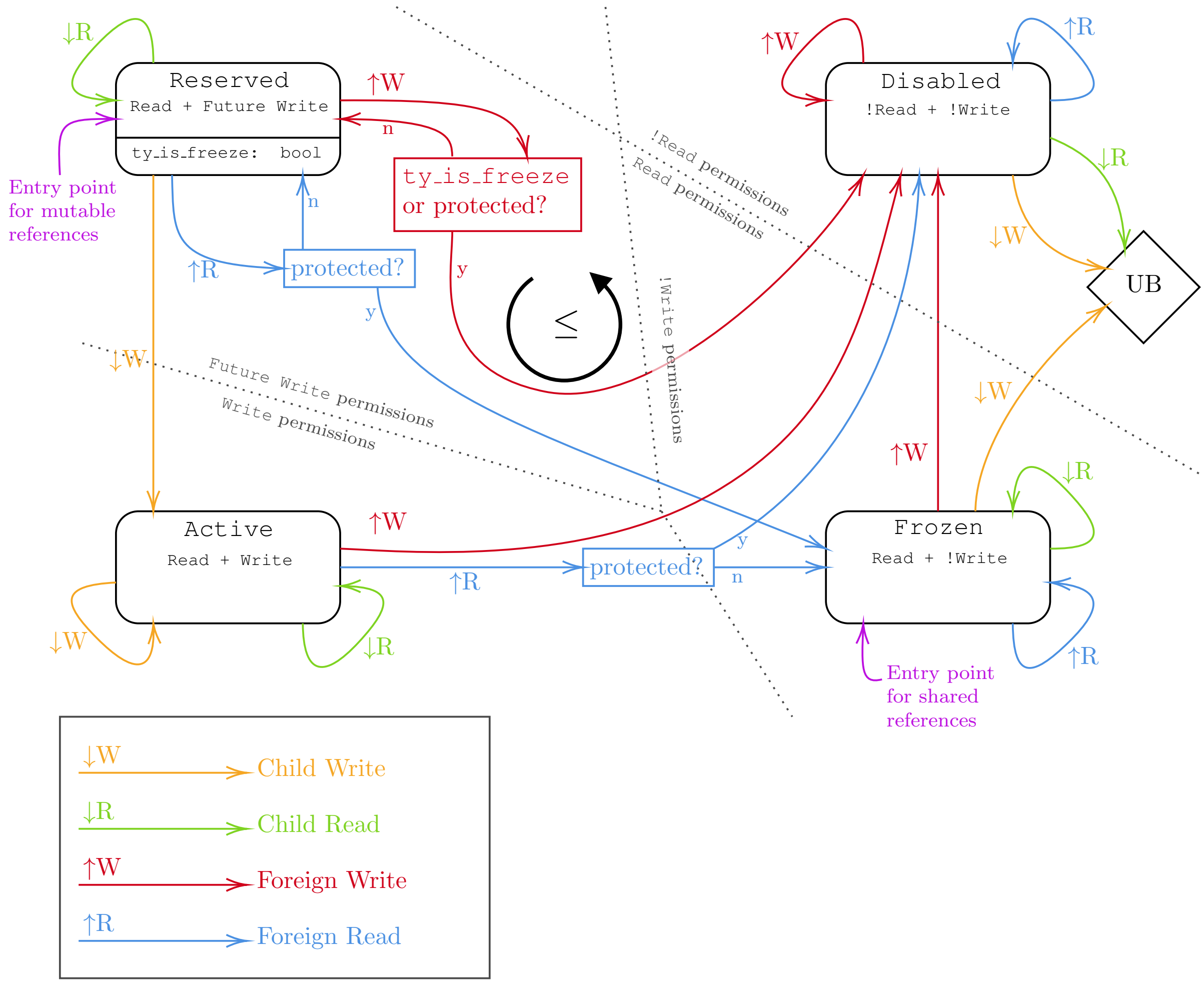

Figure 3: State machine for the update of permissions including protectors

this tag's permissions should follow the protected or unprotected version of the transitions. In order to not declare too much UB, we also add to each state a boolean field accessed, which is initially false and becomes true on the first child access.

We apply the following rules to pointers that have an active protector (when the protector is removed upon function exit, these rules no longer apply):

- any protected tag becoming Disabled is UB;
- Reserved is affected by foreign reads and writes regardless of interior mutability, it always becomes Frozen after a foreign read and Disabled after a foreign write;
- Active becomes Disabled instead of Frozen on a foreign read.

The transitions including protectors can be seen in the state automaton of Figure 3.

4.3.4 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; // only reborrowed for [0]
    let snd = fst.add(1); // used on [1], which was not reborrowed
    ptr::swap(fst, snd);
} }
```

Here the reborrow for `fst` only covers `data[0]`, but `snd` is then derived from `fst` and offset 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.

5 Tree Borrows implemented

All behavior described here was implemented in Miri and is available since its recent merge through the `-Zmiri-tree-borrows` flag. Further testing and improvement of diagnostics are scheduled.

5.1 Testing the Rust Standard Library

We used the method described in `github:rust-lang/miri-test-libstd` 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. It is also easy to patch: the fix (which simply consists of replacing `&mut root as *mut _` 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. Checking more thoroughly that this is actually the case is one of the ambitions of our future work.

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 in Figure 4 that in practice the slowdown from Stacked Borrows to Tree Borrows is mostly less than x2 on benchmarks that are specifically designed to stress the borrow tracker,

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
	`zip-equal`	5	???	???	???
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
Memchr	`lib`	1	2.1s	2.2s	x1.05
Slotmap	`lib`	1	Error	18m15s	x?

Figure 4: Benchmarks comparing the execution time of various widely used test suites for Stacked Borrows (SB) or Tree Borrows (TB). The increase in computation time is shown to be rarely a lot more than a factor 1.5. Some of these projects use Stacked Borrows as part of their continuous integration pipeline, so switching to Tree Borrows is shown to be reasonable in terms of additional computational requirement. The Appendix A lists the commands and sources that were used for each of these benchmarks.

and about x1.3 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.

The `zip-equal` benchmark above was added after we discovered a pathological case in the test suite of the http crate. We were able to fix the issue (that came from an improper implementation of the tag garbage collector) and dramatically improve performance on this specific pattern (the bug caused a quadratic blowup of the runtime, which we removed).

6 Future work

6.1 Evaluating Tree Borrows on more real-world code

Merely evaluating Tree Borrows on the examples shown in Section 5.1 is not sufficient for a complete picture of how Tree Borrows fits in the Rust ecosystem.

For this, we can rely on the `crater` tool and the crates.io repository. `crates.io` is the main official repository of Rust libraries. Most Rust features not part of the standard library are published to `crates.io`, and it even contains some better-than-stdlib libraries. In its quality of official collection of user-written Rust code, `crates.io` is often used to detect bugs in the compiler early: since all code published to `crates.io` compiles and passes tests, a simple way to catch backwards incompatibilities in new versions of the compiler is to run it on the entire codebase of `crates.io`. The tool that automates running tests on all of `crates.io` is called `crater`.

Miri comes with a fine-tuned fork of `crater`, which Ben Kimock hosts, and using it we obtained the data shown in 5. Using this data we can evaluate in advance how difficult it would

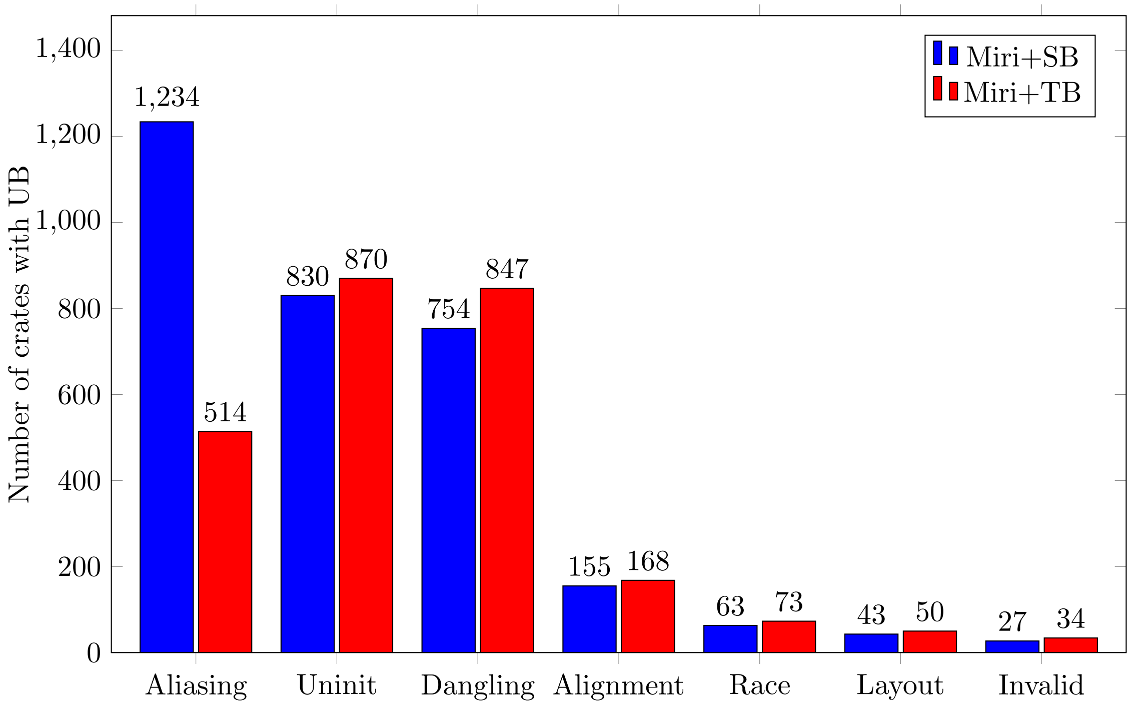

Figure 5: As of May 2023 (when the logs were obtained), total number of crates on `crates.io` (out of 100k) whose test suite contains at least one instance of each kind of UB detected by Miri.

be for the Rust ecosystem to adapt to a new kind of UB: more crates that contain code that Miri considers UB would equate more compiler breakage in the future if `rustc` actually implements Miri-based optimizations.

In Figure 5, meaning of columns:

- Aliasing: aliasing UB detected by Stacked Borrows ("Miri+SB") or Tree Borrows ("Miri+TB");
- Uninit: read of uninitialized memory;
- Dangling: dangling or null raw pointer used to create a `Box` or `&mut` (should be non-dangling), or dereferencing a dangling pointer;
- Alignment: unaligned raw pointer used to create a `Box` or `&mut` (should be aligned), or dereferencing an unaligned pointer;
- Race: racing accesses to the same data in two threads;
- Layout: violations of calling conventions, attempting to deallocate C heap memory which does not match the Rust heap memory layout, zero-size allocation, ...;
- Invalid: otherwise invalid value constructed (e.g. a `bool` that is not `0` or `1`, enum discriminant that does not exist, unreadable vtable, ...).

Methodology We have access to miri.saethlin.dev which records the output of Miri with SB on every (100k+) crate on `crates.io`. We also have an analogue dataset for Miri with TB.

The dataset consists of a series of records:

- a crate name;

- a crate version;
- an output among:
  - “Success”, if no error occured;
  - “Timeout”, if the crate exceeded its alloted amount of time to run test;
  - “Error”, if an environment or runtime error occured (e.g. calling a function not supported by Miri);
  - “UB: [list of instances of UB]”, if the crate contains UB in at least one test.

A crate can contain several instances of UB if different tests each contain UB, but if a single test contains several kinds of UB Miri will only be able to report the first one that occurs. Each (name, version) pair uniquely identifies a crate.

In order to reduce noise and ensure a fair comparison between Miri+SB and Miri+TB, the dataset was cleaned in the following ways:

- Since the Miri+SB and Miri+TB versions were not run at exactly the same time, we first restrict the dataset to only (name, version) pairs that are common across both. This ensures that SB and TB are tested on exactly the same code.
- Crates that timeout in one but not the other are excluded, since it is possible that UB would have occured if the test had ran slightly longer.
- Crates that result in a runtime error in one but not the other are also excluded, since it may be due to being run with different configurations.

For the remaining crates, which were all run on both SB and TB and executed all their tests without timing out, we count whether at least one of their tests contains an instance of each kind of UB. If a crate contains one instance of Aliasing UB and one instance of Uninit UB it will be counted in both columns, but if it contains two instances of Aliasing UB then it will only count once towards that number. This reduces the impact of crates that have a lot of tests that all contain exactly the same instance of UB due to calling a common helper function that contains UB.

Interpretation Currently all the crates that appear in this count compile withour error and execute correctly under the official compiler `rustc`. Miri however is more strict than `rustc` and considers the amount of crates shown to contain UB of some sort. Therefore each column measures in some sense the amount of backwards incompatibility that would come from the compiler actually performing the related optimizations.

In the specific case of Stacked and Tree Borrows, we can see that Tree Borrows causes roughly half as many crates to fail than Stacked Borrows, which means that implementing optimizations allowed by Tree Borrows would have much less risk of breaking existing code than optimizations allowed by Stacked Borrows.

Anectotal evidence from a handful of popular crates suggests that this effect is even more noticeable when authors are actually careful about UB: in the popular crates that contain UB under SB and have actively tried to fix it but failed (`tokio`, `pyo3`, `rkyv`, `eyre`, `ndarray`, `arrayvec`, `slotmap`, `nalgebra`, `json`), all of them were found to be free of UB under TB.

UB Shadowing The reader may be surprised to see that while Miri+TB has much less Aliasing UB than Miri+SB, it has slightly more of every other kind of UB. There is a very simple explanation for that, simply by virtue of how Miri reports UB.

In a given test, Miri executes the code until it detects UB, and when it does it aborts the test and outputs an error message. Therefore if a single test contains several instances of UB Miri will only report the first one. Some errors related to e.g. accessing uninitialized memory happen by simple coincidence to also be aliasing errors, or to be preceded chronologically by aliasing errors. This occurs frequently for at least the following reasons: (a) programmers who are not careful about uninitialized memory probably tend to also not be careful about aliasing, and (b) programs that manipulate uninitialized memory definitely use `unsafe` code and probably use that memory in nontrivial ways, which make aliasing errors more likely.

There is thus an increased likelyhood in a test for which Miri+SB reports Aliasing UB that it also contains Uninit UB, but the Uninit UB is “shadowed” by the Aliasing UB that occurs first. If Miri+TB no longer reports Aliasing UB for that specific test, it still contains Uninit UB. Thus with the move from Miri+SB to Miri+TB, the following happens:

- some tests that contain exclusively SB Aliasing UB are removed from the count entirely,
- some tests that contain SB Aliasing UB and also another kind of UB are recategorized under only said other kind.

This accounts for the entirety of the increased counts of every kind of UB other than Aliasing.

By being more permissive, Tree Borrows thus allows some tests that were thought to contain Aliasing UB to be more accurately recategorized under other kinds of UB.

The significance of this should not be underestimated: not only does Miri+TB still contain much fewer instances of UB than Miri+SB overall, it may also encourage more crate authors to actually fix the remaining UB. Indeed SB has a reputation for being too strict, and for years now it has been known that SB as it currently is would not become the standard. Programmers that encounter an SB error message may thus tend to automatically dismiss it as a problem for later, or something that will never become the standard anyway. On the other hand it is much more well-known that uninitialized memory could be dangerous immediately, so an Aliasing+Uninit bug is more likely to be fixed if it is classified as Uninit than as Aliasing.

6.2 Proving optimizations

We show a sketch of how to use Tree Borrows to prove some reordering-based optimization. Part of our future work will be an attempt to formalize and generalize the proof that follows as well as others.

```
fn example2_unopt(x: &u64) -> u64 {
    let val = *x;
    g(); // arbitrary (possibly unsafe) unknown code modeled by a function call
    val
}

fn example2_opt(x: &u64) -> u64 {
    g();
    *x // This optimization is a read/call reordering + value propagation
}
```

For the above reordering to be valid, the following needs to hold. For any definition of `g`, for any context C, if C[`example2_unopt`] does not exhibit UB then C[`example2_opt`] also does not exhibit UB. We call C[`example2_unopt`] the source, and C[`example2_opt`] the target.

Proof sketch: it is sufficient to prove that if `example2_unopt` and `example2_opt` are executed with the same memory state and initial borrow tree for which `example2_unopt` does not trigger UB then

- `example2_opt` does not trigger UB, and
- they both modify the memory in the same way, and
- they return the same value, and

- they result in the same final borrow tree.

Indeed if neither triggers UB immediately and they result in the same memory state and borrow tree then the two versions of the function will be indistinguishable without UB in the source.

We mark the following notable points of the code:

```
fn example2_unopt(x: &u64) -> u64 {
    // s0
    let val = *x;
    // s1
    g();
    // s2
    val
    // s3
}

fn example2_opt(x: &u64) -> u64 {
    // t0
    g();
    // t1
    *x
    // t3
}
```

and denote $T(x)$ and $M(x)$ the tree and memory at the execution point x.

Let l the location that `x` points to: $M(s_0)[l]$ is the value of `*x` at s_0.

We first show that `g` must not perform a write to l. We have created a fresh tag p_x for `x` and no child tag of p_x has been passed to `g`, thus any access that occurs during `g` is a foreign access for p_x. Since p_x is protected, any foreign write would be UB, thus `g` can be assumed not to perform any write access to l. Therefore the value at l in unchanged during the execution of `g` in the source. This proves that both functions return the same value.

Assuming $M(s_0) = M(t_0)$ since the two functions are to be executed in the same context, and since no write occurs at all between s_0 and s_1 we thus obtain $M(s_1) = M(t_0)$. In the source and the target the whole memory is identical when `g` is called, thus `g` executes in the same way in both. Since no memory modification occurs between s_2 and s_3 or between t_1 and t_3, we obtain that $M(s_3) = M(t_3)$.

What remains is to show that the source and target result in the same borrow tree and that the target does not have UB. Since $T(t_0)$ refines $T(s_1)$, no UB occurs in the target during the execution of `g`. We then examine what happens to all borrows for the location:

- p_x in the source is subjected to a child read then zero or more child reads during `g`. It does not matter the order, the final permission of p_x will be the same in the target, i.e. `Frozen`;

- there are no children of p_x;

- non-children of p_x that already existed before the execution of the function are subjected in the target to the operations in `g` then a read of t_x instead of the opposite in the source. We easily check that in the state machine all transitions induced by read accesses (both child and foreign) commute with each other and thus the final permissions are the same in the source and in the target.

- non-children of p_x that were created during the execution of `g` are not protected when `*x` is read in the target and they are not `Active` because creating an `Active` requires a write. The read is thus a no-op.

We have thus shown that in the absence of UB in the source, $T(s_0) = T(t_0) \implies T(s_3) = T(t_3)$ and thus the source and target have the same behavior, and moving a read access down across unknown code within a function is a valid optimization.

6.3 Formalization

Using the Simuliris framework [3] (Simuliris is derived from Iris which is derived from Coq) we hope to be able to formally prove the above optimization and others. This requires first formalizing Tree Borrows, and one difficulty is expected to be the handling of parallelism: since Tree Borrows requires but does not directly implement detection of race conditions (race conditions are UB, so the optimizations must not introduce race conditions), the formalization will need to either consider only a sequential language or include a race condition detection algorithm from prior work.

References

[1] The Rust Project Developers. Miri. `https://github.com/rust-lang/miri/`, 2016. Accessed: 2023-03-14.

[2] The Rust Project Developers. Rust Compiler Development Guide. `https://rustc-dev-guide.rust-lang.org/borrow_check/two_phase_borrows.html?highlight=temporary%20mutable#two-phase-borrows`, 2018. Accessed: 2023-03-14.

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

[4] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of c. *ACM SIGPLAN Notices*, 50:336–345, 06 2015. `doi:10.1145/2813885.2737979`.

[5] Ralf Jung. Rust Unsafe Code Guidelines - Issue 133: Asserting uniqueness too early. `https://github.com/rust-lang/unsafe-code-guidelines/issues/133`, 2019. Accessed: 2023-03-27.

[6] Ralf Jung. Rust Unsafe Code Guidelines - Issue 134: Raw pointer usable only for T too strict. `https://github.com/rust-lang/unsafe-code-guidelines/issues/134`, 2019. Accessed: 2023-03-27.

[7] Ralf Jung. Rust Unsafe Code Guidelines - Issue 276: Stacked Borrows cannot properly handle extern type. `https://github.com/rust-lang/unsafe-code-guidelines/issues/276`, 2020. Accessed: 2023-03-27.

[8] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: an aliasing model for rust. *Proceedings of the ACM on Programming Languages*, 4:1–32, 12 2019. `doi:10.1145/3371109`.

[9] Robbert Krebbers. Aliasing restrictions of c11 formalized in coq. pages 50–65, 12 2013. `doi:10.1007/978-3-319-03545-1_4`.

[10] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno Lopes. Reconciling high-level optimizations and low-level code in llvm. *Proceedings of the ACM on Programming Languages*, 2:1–28, 10 2018. `doi:10.1145/3276495`.

[11] Xavier Leroy and Sandrine Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41, 07 2008. `doi:10.1007/s10817-008-9099-0`.

[12] LLVM Project. LLVM Language Reference Manual. `https://llvm.org/docl/LangRef.html`, 2001. Accessed: 2023-03-27.

[13] Neven Villani. Tree Borrows - A new aliasing model for Rust. `https://perso.crans.org/vanille/treebor`, 2023. Accessed: 2023-03-27.

A Benchmarks for Section 5.2

The benchmarks from Figure 4 were executed with the following commands. By default Stacked Borrows is used, and Tree Borrows is selected by appending `-Zmiri-tree-borrows` to the list of `MIRIFLAGS`.

- Miri
  These benchmarks are designed specifically to exhibit pathological behaviors of Stacked Borrows. It happens that many of these are also pathological cases for Tree Borrows, in particular very narrow trees (stack-like, obtained by chaining many reborrows) and very wide trees (obtained by reborrowing many times from the same pointer). It is within expectations that Tree Borrows would perform worse in comparison to Stacked Borrows on these tests than on other tests.

  Source `github:rust-lang/miri`

  Command `MIRIFLAGS="" ./miri bench`

- Miri-test-libstd
  These tests constitute the OS-independent test suites of the Rust standard library. They are already used to detect regressions in either Miri or Rustc, but this check currently uses Stacked Borrows and not Tree Borrows.

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

  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
  This crate is a framework for writing asynchronous code. Since it uses `unsafe` and parallelism in nontrivial ways, detecting UB is important.

  Source `github:tokio-rs/tokio`

  Command `MIRIFLAGS="-Zmiri-disable-isolation -Zmiri-tag-raw-pointers"`
  `cargo +nightly miri test --features full --lib`

- Rand
  This crate is the most widely used crate for random number generation. Rand is historically relevant to one of the main issues [6] of Stacked Borrows' handling of out-of-range raw pointers.

  Source `github:rust-random/rand`

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

- Hashbrown
  This crate implements a high-performance hash map, which naturally involves aliasing and `unsafe`.

  Source `github:rust-lang/hashbrown`

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

- Regex
  As the name suggests this is a crate for parsing, compiling, and executing regular expressions. The test `encode_decode` is ignored because it takes too long for Miri to execute.

Source `github:rust-lang/regex`

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

- Memchr
  High-performance implementation of standard string search primitives. Historically had a conflict with Stacked Borrows as stated in Issue #58

  Source github:BurntSushi/memchr

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

- Slotmap
  General purpose maps. Currently has an open Issue #92 showing that some tests contain UB according to Stacked Borrows but not Tree Borrows.

  Source github:orlp/slotmap

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

B Requirements of protectors

To show why protectors are needed and why they require some alternative transitions, we show here examples of programs that are UB according to LLVM, but would not be UB according to Tree Borrows **if protected pointers behaved identically to unprotected pointers**. This is a continuation of the example shown in Section 4.3.3.

B.1 Increased requirements of Reserved

The following program justifies that after a foreign read has occured, Reserved must no longer allow foreign reads. Indeed if Reserved is unaffected by foreign reads then it allows a foreign read followed by a child write. We model this by making Reserved become Frozen on a foreign read, which means that the next attempted child write would correctly be UB.

For the same reason Reserved must not stay Reserved after a foreign write even if it has interior mutability: we declare that under a foreign write, Reserved becomes Disabled regardless of interior mutability which means that the next attempted child read or write would be UB.

```rust
fn main() {
    let data = &mut 42u64;
    let y = data as *const u64;
    let x = &mut *data;
    foreign_read_before_write(x, y);
    fn foreign_read_before_write(x: &mut u64, y: *const u64) {
        // x: Reserved [noalias]
        let _ = unsafe { *y }; // Foreign read for x
        // x: Reserved [noalias]
        *x += 1; // Child write for x
        // /!\ Combined with the previous foreign read this is a noalias violation
        // x: Active [noalias]
        // -- UB must occur before this point --
        // (Without protectors no UB occurs)
    }
}
```

B.2 Loss of read permissions

A Frozen pointer becoming Disabled is a symptom that a foreign write occured after a child read. Indeed a foreign write is only possible cause of the transition in question. If the location was also accessed through any child access, then these two accesses violate noalias, thus a transition Frozen -> Disabled should be UB on any accessed location. The same remark applies to a Reserved or Active becoming Disabled.

```rust
fn main() {
    let data = &mut 42u64;
    let y = data as *mut u64;
    let x = &*data;
    read_before_foreign_write(x, y);
    fn read_before_foreign_write(x: &u64, y: *mut u64) {
        // x: Frozen [noalias]
        let _ = *x; // Child read for x
        // x: Frozen [noalias]
        unsafe { *y += 1; } // Foreign write for x
        // /!\ Combined with the previous child read this is a noalias violation
        // x: Disabled [noalias]
        // -- UB must occur before this point --
        // (Without protectors no UB occurs)
    }
}
```

B.3 Loss of write permissions

An `Active` pointer becoming `Frozen` indicates the presence of a child write (requirement for the existence of an `Active`) and a foreign write (only possible cause of the transition). These two accesses violate `noalias`, thus a transition `Active -> Frozen` should be UB.

```
fn main() {
    let data = &mut 42u64;
    let y = data as *const u64;
    let x = &mut *data;
    write_before_foreign_read(x, y);
    fn write_before_foreign_read(x: &mut u64, y: *const u64) {
        // x: Reserved [noalias]
        *x += 1; // Child write for x
        // x: Active [noalias]
        let _ = unsafe { *y }; // Foreign read for x
        // /!\ Combined with the previous child write this is a noalias violation
        // x: Frozen [noalias]
        // -- UB must occur before this point --
        // (Without protectors no UB occurs)
    }
}
```

C Behavior of Reserved

In addition to the examples shown in Paragraph 4.3.1, about Reserved, some details of the behavior are guided by the observations that follow.

This pattern is used frequently in the standard library test suite, it would preferably not be UB. It consists of a read access through a raw pointer between the creation and access of a mutable reference, and it illustrates why Tree Borrows uses the Reserved permission for all mutable references and not just for two-phase borrows.

```rust
// Example: Foreign read outside two-phase borrow
// This should not be UB.
fn main() {
    let mut x = 2;
    let xref = &mut x;
    let xraw = &mut *xref as *mut _;
    let xshr = &*xref;
    // x: Reserved
    // |-- xref: Reserved
    //     |-- xraw: Reserved
    //     |-- xshr: Frozen
    assert_eq!(*xshr, 2); // This is a foreign read for xref and xraw
    unsafe { *xraw = 4; } // This is a child write for xref and xraw
                        // meaning it must still be writeable at this point,
                        // therefore the above foreign read must not have turned
                        // them Frozen.
    assert_eq!(x, 4);
}
```

This second pattern shows that mutable references that involve interior mutability must be exempt from the rule that a Reserved is disabled upon a foreign write. This is safe code, it must absolutely not be UB.

```rust
// Example: Foreign write with interior mutability
// This must not be UB
fn main() {
    use std::cell::Cell;
    trait Thing: Sized {
        fn do_the_thing(&mut self, _s: ());
    }

    let mut x = Cell::new(1);
    // x: Reserved
    x.do_the_thing({
        // A two-phase borrow starts here for x' implicitly reborrowed from x
        // x: Reserved
        // |-- x': Reserved
        x.set(3) // This is a foreign write for x'
        // x: Active
        // |-- x': ???
    })
    impl<T> Thing for Cell<T> {
        fn do_the_thing(&mut self, _s: ()) {
            // Function call starts, x' is implicitly reborrowed into x''
            // x: Active
            // |-- x': ???
            //     |-- x'': Reserved
            self.set(5): // x' must be readable and writeable
                        // This is a child write for x', which means
                        // that x' is now Active and was not previously Disabled
                        // or Frozen. Therefore x' was previously still Reserved,
                        // even though it was subjected to a foreign write.
        }
    }
}
```

D Summary of the model

Below is a summary of the points established in 4.1.1 and 4.3 as a full description of Tree Borrows.

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` on the reborrowed range
  - add a new child of `y` in the tree
  - give it the permissions `Reserved` (immediately on the reborrowed range, lazily on the rest of the allocation)
  - 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` on the reborrowed range
  - add a new child of `y` in the tree
  - give it the permissions `Frozen` (immediately on the reborrowed range, lazily on the rest of the allocation)
- 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 permissions (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 `Active` into `Frozen`; this is UB if `z` is protected
  - if `z` is protected, turn `Reserved` into `Frozen`

When writing through a pointer `y`

- for all ancestors `x` of `y` (including `y`), this is a child write
  - turn `Reserved` into `Active`
  - it is UB to encounter 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 all of `Reserved`, `Active`, and `Frozen` into `Disabled`.