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:

```
1  fn example1(x: &mut u64, y: &mut u64) -> u64 {
2      let xval = *x; // First read of *x
3      *y = xval + 1;
4      let xval = *x; // Second (redundant ?) read of *x
5      return xval;
6  }
```

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 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;
    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 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 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. We thus conclude that the Tree Borrows semantics are in accordance with most idiomatic Rust code currently in use.
- **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 introduced in [3], Tree Borrows is more consistent in its handling of pointers. In particular a number

of bug reports on the Github page of Miri [1] (issues #1666, #1878, #2082, #2722) 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. 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 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. There 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:

```
1 let mut data = 42u64;
2 let some_mut_ref: &mut u64 = &mut data;
3 let const_from_mut: &u64 = &*some_mut_ref;
4 let reborrow_mut: &mut u64 = &mut *some_mut_ref;
5 let mut_ptr: *mut u64 = &mut *some_mut_ref as *mut u64;
6 let ref_from_ptr: &mut u64 = unsafe { &mut *mut_ptr };
```

2.2 2-phased borrows

"2-phased 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 2-phased borrow:

```
1 impl X {
2     fn method(&mut self, ...) { ... }
3 }
```

```
4
5  x: &mut X
6  x.method(arg1, arg2, ...);
```

where the method call desugars to approximately

```
1  let x_bor: &mut *x;
2  // 2-phased borrow for x_bor begins
3  let arg1 = ...;
4  let arg2 = ...;
5  // 2-phased borrow for x_bor ends and actual mutable borrow begins
6  X::method(x_bor, arg1, arg2, ...);
```

As a concrete example, consider

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

which the Borrow Checker accepts. In the absence of 2-phased borrows at all, one would have to write

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

More generally, the existence of 2-phased 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

```
1  // Rejected by the Borrow Checker.
2  // Expected to be UB.
3  fn alternate_writes() {
4      let x = &mut 0u64;
5      let y = &mut *x;
6      let z = &mut *x;
7      *y += 1;
8      *z += 1;
9  }
```

```
10
11 // Accepted by the Borrow Checker.
12 // Expected to be UB.
13 fn alternate_writes_raw() {
14     let x = &mut 0u64;
15     let y = unsafe { &mut *(x as *mut u64) };
16     let z = unsafe { &mut *(x as *mut u64) };
17     *y += 1;
18     *z += 1;
19 }
```

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

```
1  // Rejected by the Borrow Checker.
2  // Expected to not be UB.
3  fn unreachable_borrow() {
4      let x = &mut 0u64;
5      let y = &mut *x;
6      if false {
7          let z = &mut *x;
8          *z += 1;
9      }
10     *y += 1;
11 }
```

3.2 Stacked Borrows

Stacked Borrows already adresses 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.

```
1  // UB according to Stacked Borrows.
2  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-phased borrows. As stated in [2], 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(
```

```
18          &mut *x,
19          unsafe { *xraw += 1; },
20      );
21  }
22
23  fn print(x: &mut u8, _: ()) {
24      println!("{x}");
25  }
```

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:

```
1  // UB according to Stacked Borrows.
2  // Should not be UB according to Tree Borrows.
3  fn offset_outside_reborrowed_range() {
4      let mut data = [0u8, 1, 2];
5      let x1 = (&mut data[1]) as *mut u8;
6      unsafe { *(x1.add(1)) = 3 };
7  }
```

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:

```
1  // Per-tag data consists of
2  // - permissions
3  // - local tree structure
4  struct Node {
5      // The pointer (identified by its tag) that this node represents
6      tag: Tag,
7      // The parent pointer; the root has no parent
8      parent: Option<Tag>,
9      // The child pointers
10     children: Vec<Tag>,
11     // The current permissions associated with this tag
12     permissions: ReadWritePerms,
13     /* 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 onwardS `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 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`.

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 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*.

4.2 Pointer permissions

4.2.1 Available permission combinations

Pointers can have the following permissions:

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

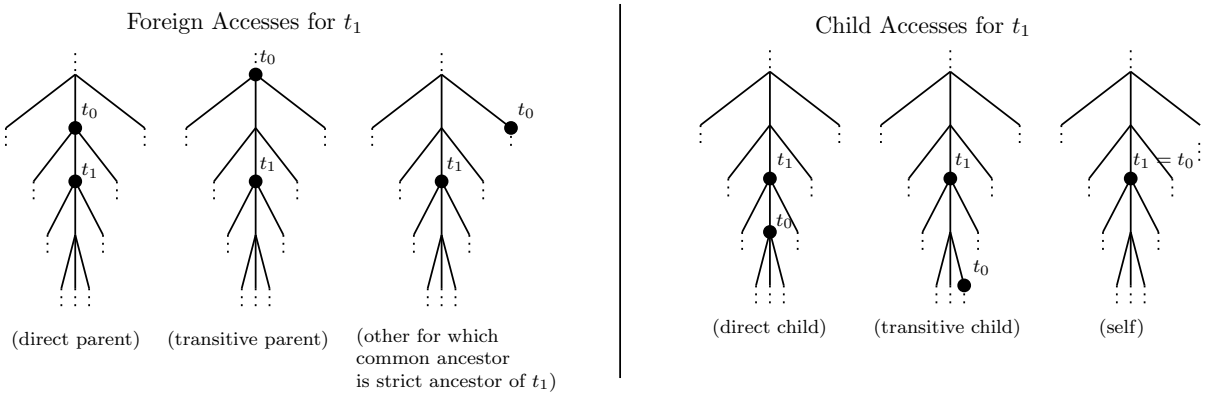

Figure 1: Accesses are classified in two categories depending on their relative position to the current tag. An access done through a (transitive and inclusive) child tag is a child access. An access done through a non-child tag (strict ancestor or non-comparable) is a foreign access.

We choose to introduce `Future Write` permissions as a way for Tree Borrows to natively handle 2-phased borrows. 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 is for a pointer whose lifetime has ended;
- a `Frozen` tag is for a shared references without interior mutability;
- an `Active` tag is for an activated mutable reference;
- a `Reserved` tag is for a 2-phased borrowed mutable reference.

4.2.2 2-phase for all borrows

As we have explained, `Reserved` represents mutable references with a 2-phased borrow not yet activated.

Rather than limiting this behavior to 2-phased 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

```
                          // 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 2-phased 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.

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 `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`.

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:

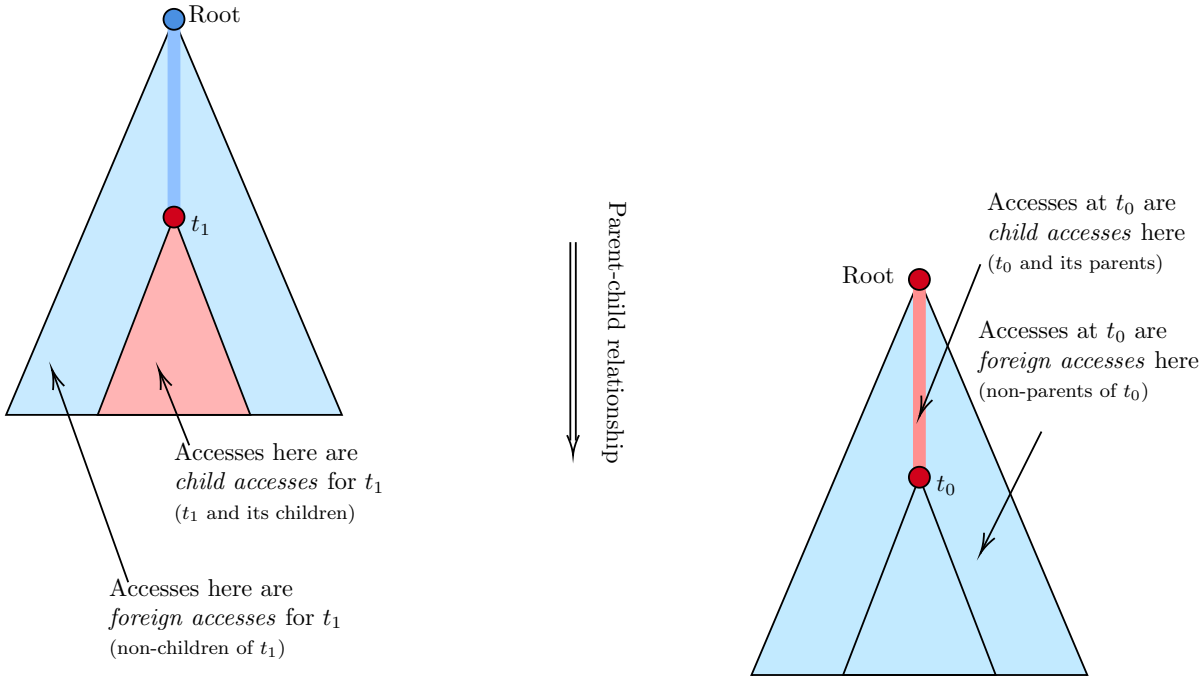

Figure 2: Two opposite points of view for the effects of an access. Left: how a given tag reacts to accesses at various positions. Right: how an access at a given position affects other tags. The foreign/child access naming follows the left convention which is easier for describing the update of permissions,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

```
1  // Example 3.A.1
2  fn main() {
3      let base = &mut 42u64;
4      let rmut = &mut *base;
5      // base: Reserved
6      // |-- rmut: Reserved
7      *rmut += 1; // Child write for both base and rmut
8      // base: Active
9      // |-- rmut: Active
10     let _val = *base; // Child read for base; foreign read for rmut
11     // base: Active
12     // |-- rmut: ???
13     let _val = *rmut; // Compilation error
14     // According to the Borrow Checker, rmut is no longer readable
15 }
```

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:

```
1  // Example 3.A.2
2  fn main() {
3      let base = &mut 42u64;
4      let rmut = &mut *base;
5      // base: Reserved
6      // |-- rmut: Reserved
7      *rmut += 1; // Child write for both base and rmut
8      // base: Active
9      // |-- rmut: Active
10     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:

```
// Example 3.R.1: Foreign read (standard 2-phased borrow example)
// This must not be UB
fn main() {
    let mut x = vec![];
    // x: Reserved
    x.push( // 2-phased 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.1': Foreign read outside 2-phased borrow
// This is a very common pattern in the standard library, it must NOT be UB.
// This also justifies why we need to use Reserved for all mutable references,
// and not exclusively for 2-phased borrows.
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);
}
```

```
// 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;
}

// Example 3.R.2': 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 2-phased 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.
        }
    }
}
```

Ordering of states From the above 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 2-phased 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.

4.3.2 Automata of Permissions

The full automata of how permissions react to accesses can be seen in Figure 3. For now this is nothing more than the aggregation of everything established previously in Section 4.3.1.

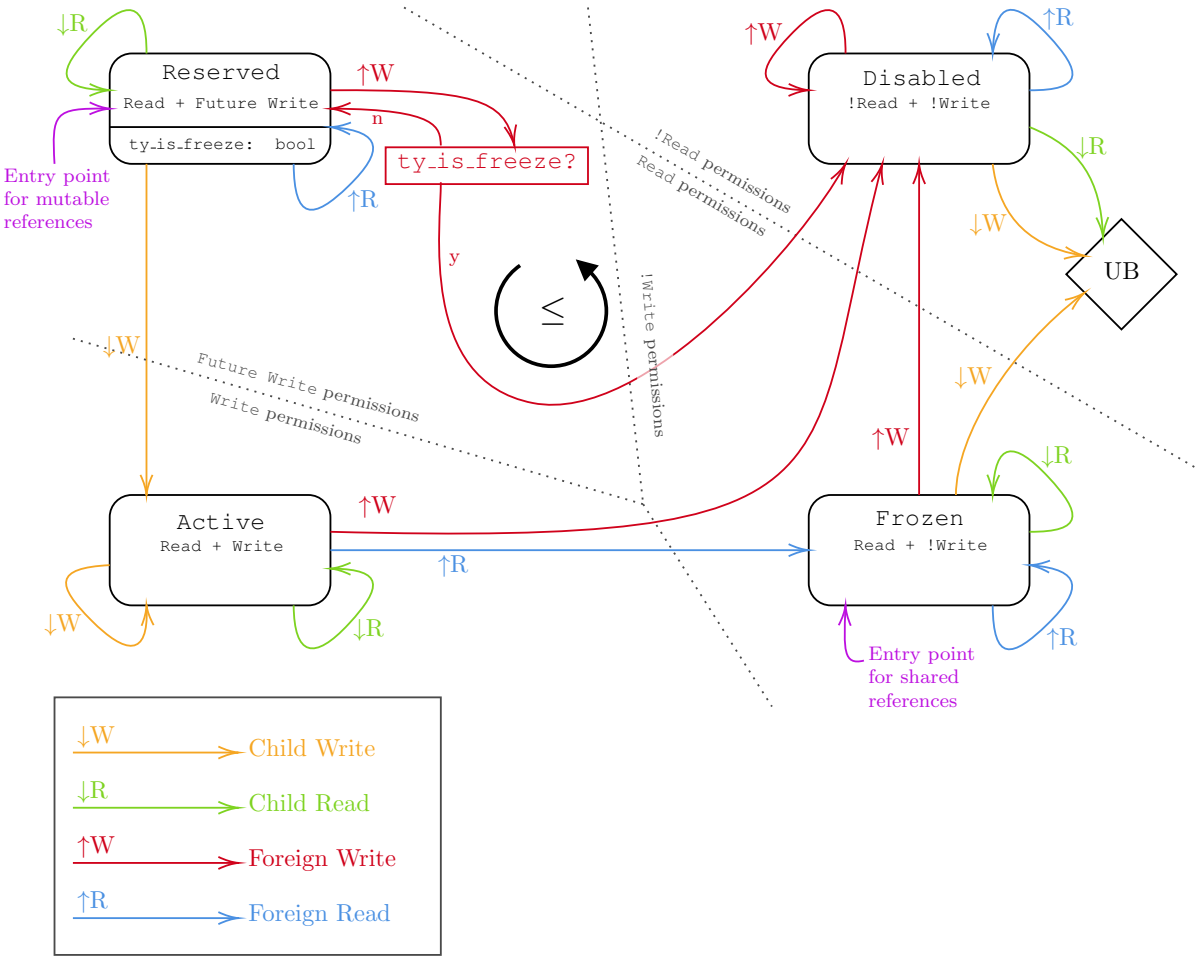

Figure 3: State machine for the update of permissions. This version is not complete yet.

A possible interpretation of these 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 or write permissions for the access to be performed
  - `Reserved`, `Active`, and `Frozen` already have their `Read` permissions, so a child read does not affect them;
  - `Active` also has `Write` permissions, so it is unaffected child Write;
  - `Reserved` does not have `Write` (only `Future write`), but it can advance to `Active` to obtain the missing permission;
  - `Frozen` for a `Write` and `Disabled` for both `Read` and `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 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`'s `!Write` and `Reserved`'s `Future Write` are compatible with a foreign read, hence the loops;
  - as a unique exception to everything being incompatible with a foreign write, a `Reserved` can stay `Reserved` if it has interior mutability, as explained previously by *Example 3.R.2'*.

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. 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.

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

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 for this tag if at least one of them is a write access.

This means that the following pieces of code must be Undefined Behavior, but with the current model not all of them are:

```
// Example 3.N.1
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 --
        // (With the current model no UB occurs)
    }
}

// Example 3.N.2
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 --
        // (With the current model no UB occurs)
    }
}

// Example 3.N.3
fn main() {
    let data = &mut 42u64;
    let y = data as *mut u64;
    let x = &*data;
    foreign_write_before_read(x, y);
    fn foreign_write_before_read(x: &u64, y: *mut u64) {
        // x: Frozen [noalias]
        unsafe { *y += 1; } // Foreign write for x
        // x: Disabled [noalias]
        let _ = *x; // Child read for x which has !Write
        // /!\ Combined with the previous foreign write this is a noalias violation
        // This is already (correctly) UB in the current model
        // -- UB must occur before this point --
    }
```

```
}

// Example 3.N.4
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 --
        // (With the current model no UB occurs)
    }
}
```

Only in *Example 3.N.3* is there indeed UB. Here is what the other examples teach us:

- (*Example 3.N.4*) a `Frozen` becoming `Disabled` indicates the presence of a foreign write (only possible cause of the transition). 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`;
- (*Example 3.N.2*) an `Active` 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;
- (*Example 3.N.1*) after a foreign read has occured, `Reserved` must no longer allow child writes. We model this by making it `Frozen`, which means that the next attempted child write would 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 foreign write, `Reserved` becomes `Disabled` which means that the next attempted child read or write would be UB.

In terms of read and write permissions, this means that compared to the previous model without protectors, additional UB occurs on any transition that causes `Read -> !Read` or `Write -> !Write`.

Once the function has returned, the pointer is no longer subjected to `noalias` and it resumes reacting "normally" to accesses.

To handle this new addition to the model we introduce *Protectors*. Upon function entry, we add a Protector to every reference passed as argument (Implementation: simply maintain a `HashSet` containing call ids of functions that have not yet returned and their reference arguments, and query this set on each transition to know if 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.

Being protected thus modifies the behavior of pointers in the way described in Figure 4.

Note: we observed earlier that for every kind of access, the corresponding transitions are idempotent. The only transitions that have changed and could violate the observation would be

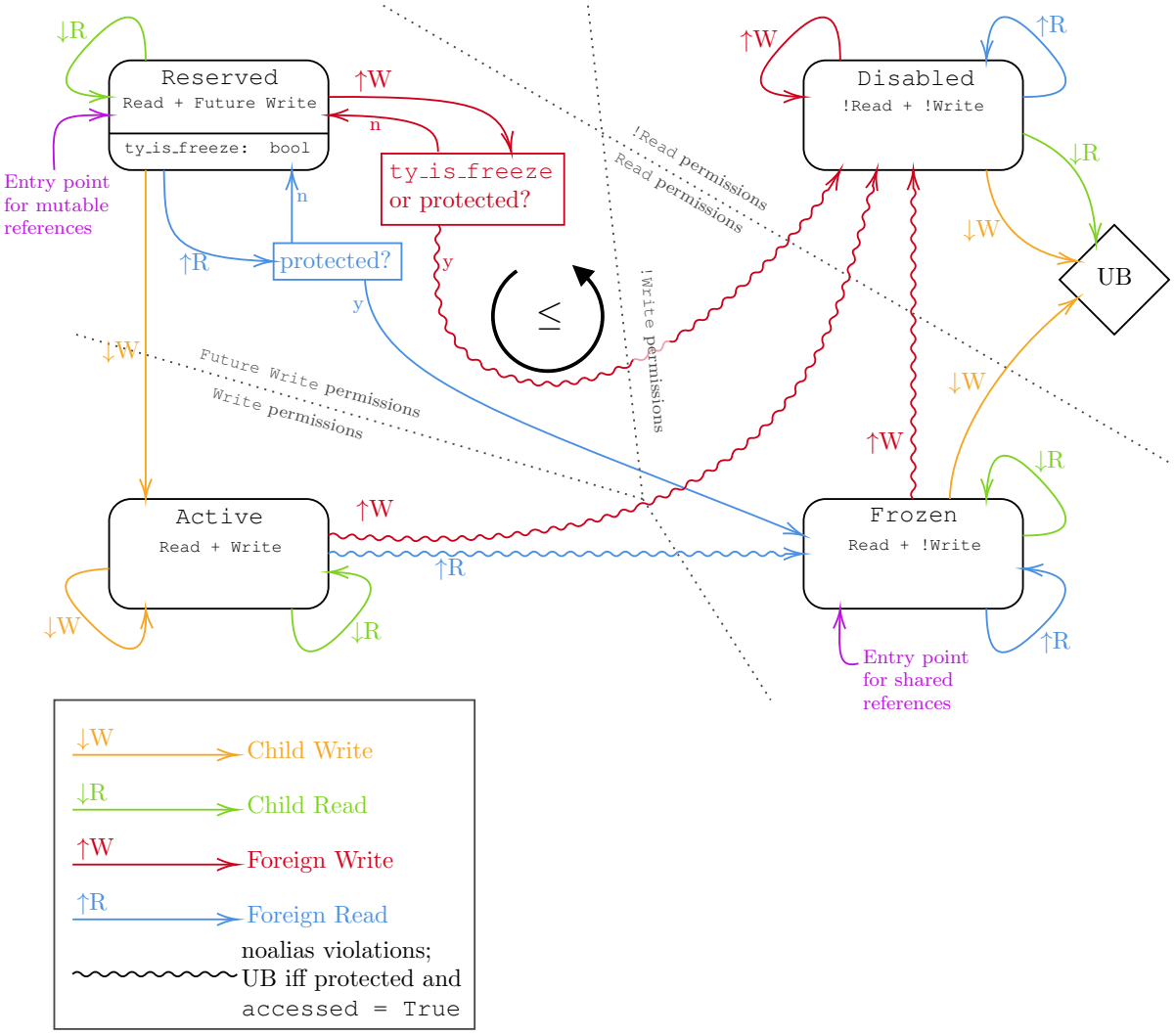

Figure 4: State machine for the update of permissions including protectors

- `Reserved -<foreign read>-> Reserved -<foreign read + protected>-> Frozen`, and
- `Reserved -<foreign write>-> Reserved -<foreign write + protected>-> Disabled`,

but for these to happen would require that a protector be added after the tag has already been subjected to an access, which is not something that occurs in our model. We can thus still rely on the effect of every access being idempotent.

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;
    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.

4.3.5 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 x2 on benchmarks that are specifically designed to stress the borrow tracker, 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.

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

We show a sketch of how to use Tree Borrows to prove some reordering-based optimization

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

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

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.

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.

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] 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`.