Tree Borrows

Neven Villani

Feb. 2022

Tree Borrows

This document describes the design of Tree Borrows, as implemented in Miri under `src/borrow_tracker/tree_borrows`.

0. Preliminaries

In order to enable certain optimizations it is needed to reject code that would be incompatible with them. These incompatible patterns are declared violations of the aliasing discipline assumed by the compiler. In order to define and encode such aliasing discipline, Tree Borrows tracks at all times the permissions associated with each pointer for each memory location. When a pointer is used in a way that it has insufficient permissions for, this is considered Undefined Behavior.

These permissions are dynamic during the execution of the program, and are updated based on the various pointer creations and accesses that occur.

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.

1. Tree Structure

1.1. Creation of a new Node

For each memory location, we maintain a *borrow tree* which contains roughly the following data: (details in `tree.rs`)

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

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

All pointers tracked by Tree Borrows have an associated tag. Several pointers can share the same tag.

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 structure as a child of the tag of `x`. This new tag will have its own associated permissions.

Moreover for the purposes of what we will discuss in **3. Updates**, creation of a child tag from a parent tag `x` counts as a Read access through `x`. Among other things, this action ensures that `x` actually has permission to access the locations it is being reborrowed on.

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

- mutable references `&mut` whose underlying type is `!Unpin`;
- shared references `&` whose underlying type has interior mutability;
- raw pointers `*mut` and `*const`.

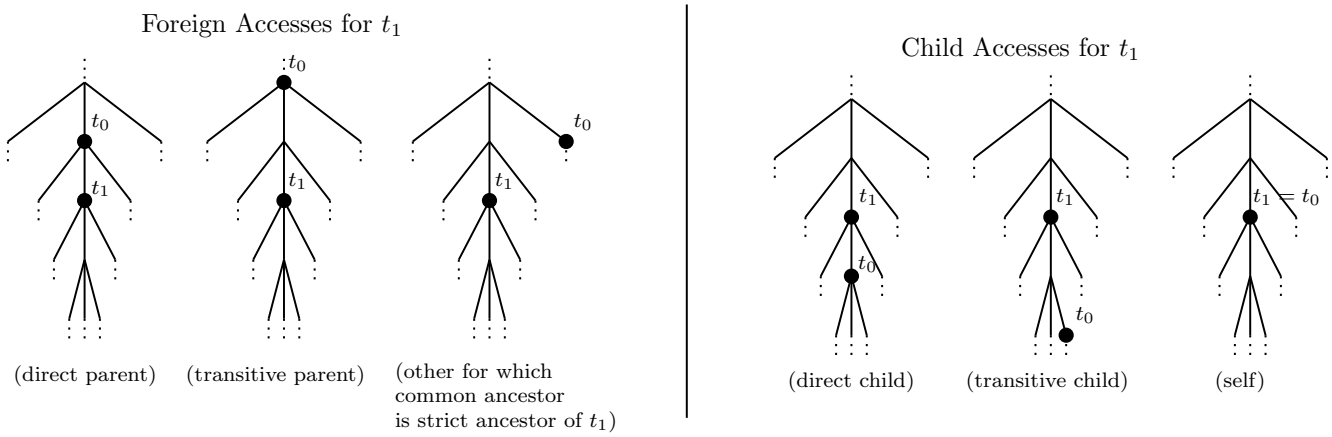
Those kinds of pointers share the property that they do not satisfy the rule of "aliasing XOR mutability", and must thus be handled by Tree Borrows differently from other more restricted pointers.

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.

1.2. Navigating the Tree

For the purposes of the permission update mechanism described in **3. Updates**, 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*.



2. Pointer Permissions

2.1. Available Permission Combinations

Pointers can have the following permissions:

- Read permisions: `Read` or `!Read`
- Write permissions: `Write` or `Future Write` or `!Write`

We choose to introduce `Future Write` permissions as a way for Tree Borrows to natively handle 2-phase borrows, as is explained in **2.2. The Purpose of Future Write**. This does not directly allow Write accesses through this pointer, but it reserves a right to obtain such `Write` permissions later in the execution.

Tree Borrows uses the following combinations of permissions:

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

They are to be interpreted in the following way:

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

2.2. The Purpose of Future Write

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

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

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

where the method call desugars to approximately

```
let x_bor: &mut *x;
                    -----
let arg1 = ...;         |--- special zone within which x is still accessible,
let arg2 = ...;         |    but only immutably
                    -----
X::method(x_bor, arg1, arg2, ...);
```

As a concrete example, consider

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

which the Borrow Checker accepts despite rejecting the following

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

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

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

```
                                    // More generally:
let ptr = vec.as_mut_ptr();         // - some implicit or explicit mutable reborrow
if vec.len() > 0 {                  // - use the base pointer immutably
    ... // some code that uses ptr  // - then use the reborrow or a pointer derived from it
}
```

2.3. Initialization and Progress

The initial permissions depend of the type of borrow. Recall from **1. Creation of a new Node** that raw pointers do not receive new permissions, which leaves mutable and shared references.

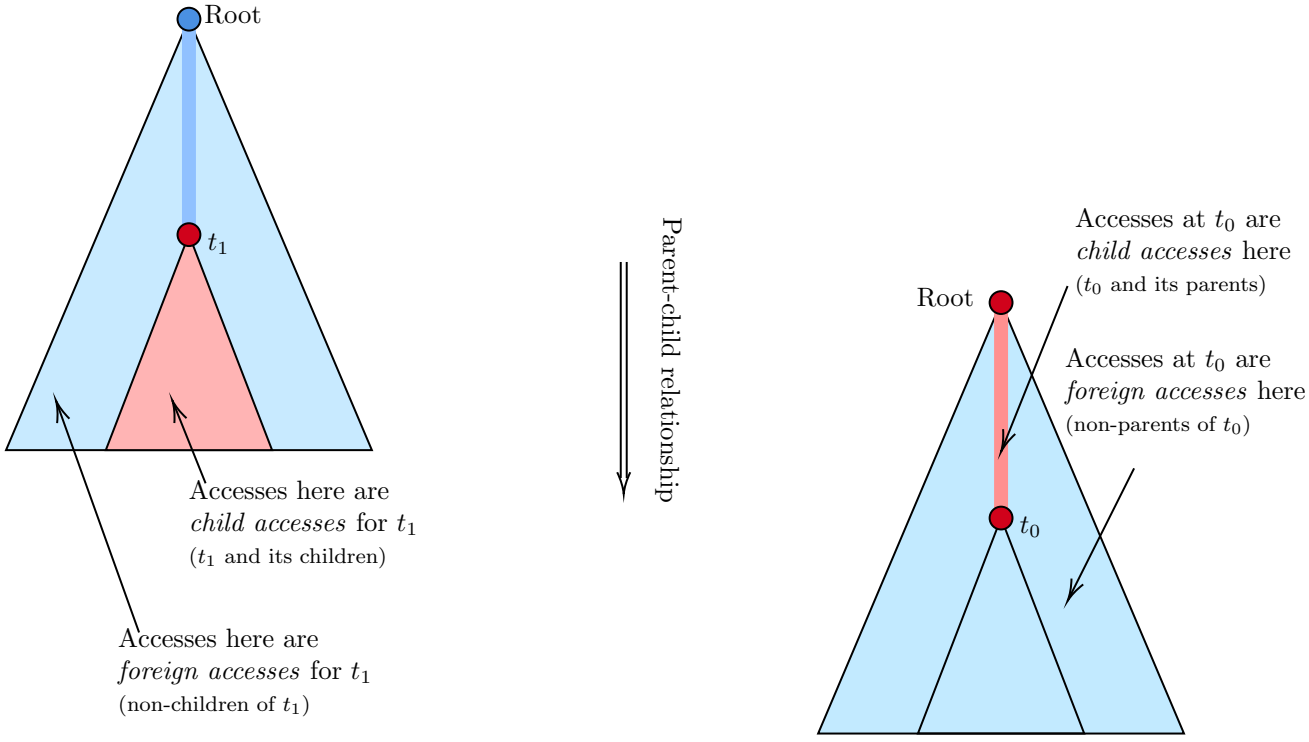
All references obtain a `Read` permission initially.

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

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.



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:

```
// 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:

```
// 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:

```
// Example 3.R.1: Foreign Read (standard 2-phase borrow example)
// This must not be UB
fn main() {
    let mut x = vec![];
    // x: Reserved
    x.push( // 2-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
```

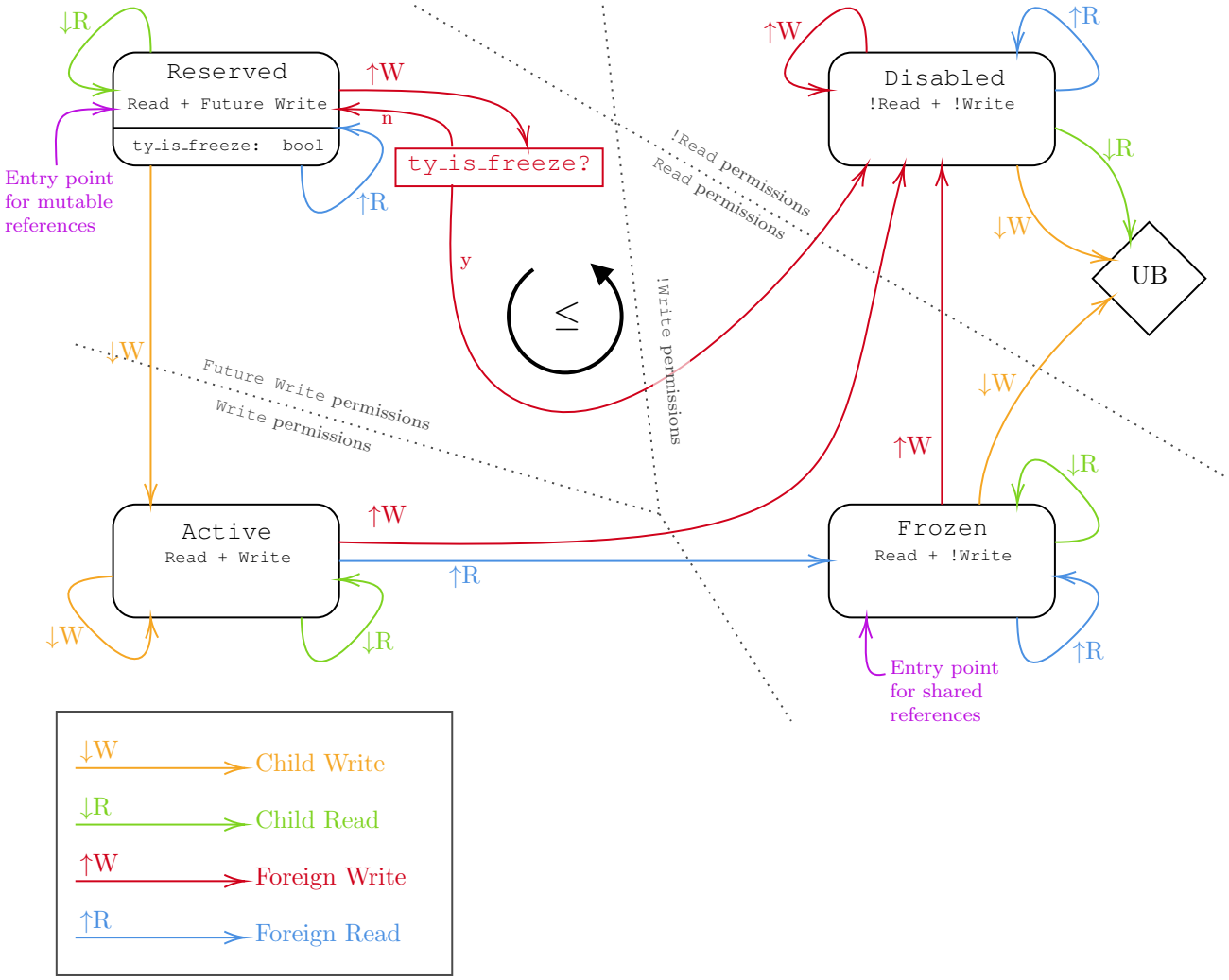
```
        // thus a Foreign Read must not affect Reserved tags.
    );
}

// Example 3.R.1': Foreign Read outside 2-phase 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-phase 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-phase 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-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.

3.2. Automata of Permissions

The full automata of how permissions react to accesses can be seen in *Figure ???*. For now this is nothing more than the aggregation of everything established in **3.1**.



A possible interpretation of these transitions:

- 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.

3.4. 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 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:

```rust
// 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)
    }
}
```

```rust
// 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 ???*.

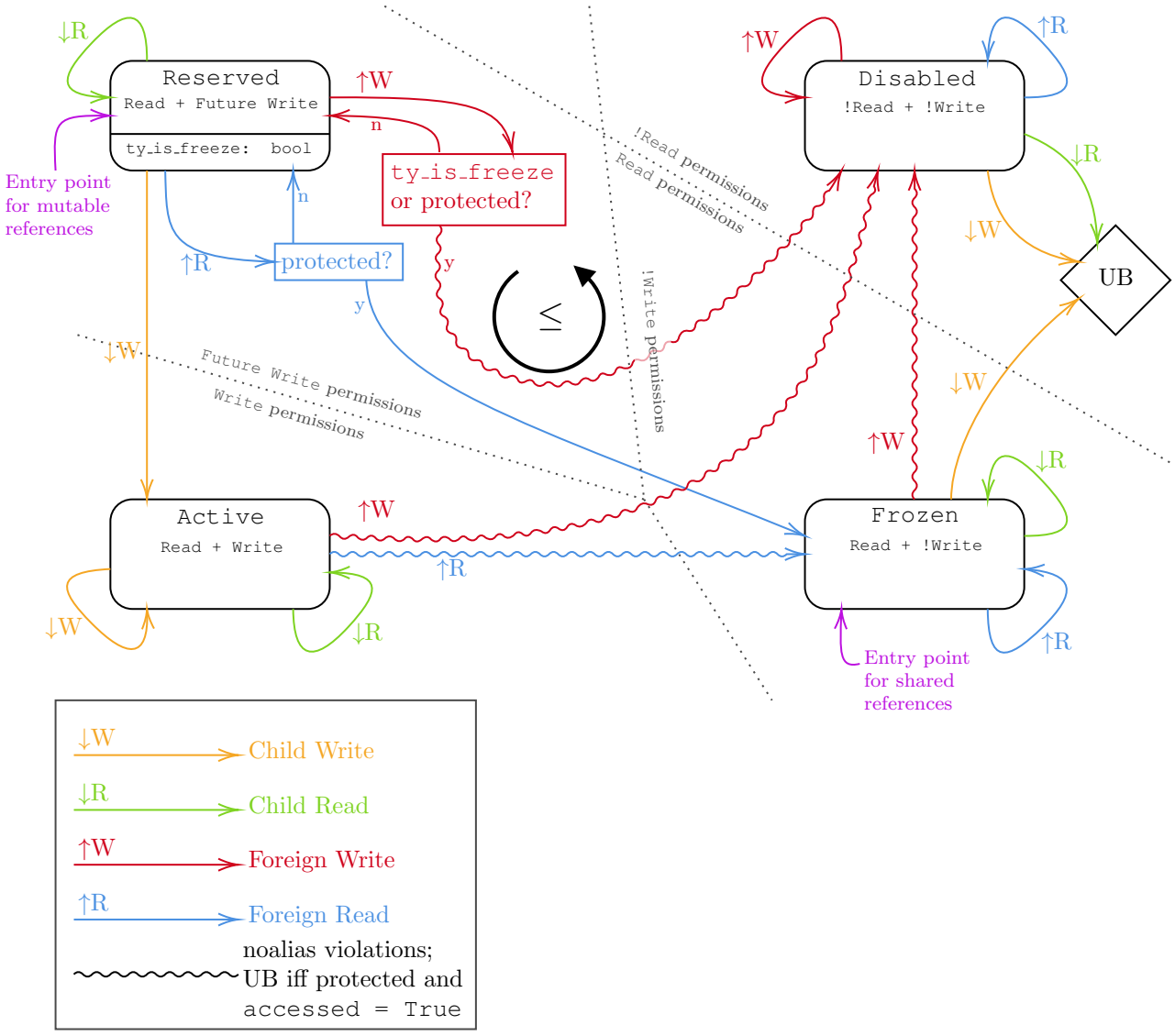


Figure 1: **Model including Protectors** Most of the transitions have been left untouched, except how `Reserved` behaves under Foreign accesses. Additionally it is UB to lose `Write` or `Read` permissions.

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

- `Reserved -<Foreign Read + unprotected>-> Reserved -<Foreign Read + protected>-> Frozen`, and
- `Reserved -<Foreign Write + unprotected>-> 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.

3.3. 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 main() { 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.

When a location is accessed

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
```

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. 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`)