Draft: Tree Borrows

y86-dev

May 4, 2022

This document is a draft. It is still under active development and might contain errors. Many important sections are missing or incomplete.

Abstract

This document details an extension of Stacked Borrows. It uses a tree structure instead of the stack to reason about the soundness of accessing pointers.

1 Overview

Tree Borrows (TB) is heavily based on the work of [2]. As such, the reader is expected to know the general concepts introduced there.

1.1 Goals

TB tries to simplify and improve upon the stack approach by using a tree instead. Moreover it also aims to solve the issues that arose regarding mutable aliasing [1].

We hope to

- track interior mutability better.
- define when mutable aliasing is allowed.
- keep most of the existing behavior.

1.2 Current Issues

- TB does not include untagged pointers, so ptr-int-ptr round trips will always produce pointers with invalid tags. This should not really be a problem, because iirc (y86-dev) non provenance respecting code is going to be UB at some point. This of course poses some problems for code abusing provenance (e.g. XOR linked lists), so maybe we need to add the `Untagged` pointer-id like SB has today.
- TB needs type information at allocation time. I (y86-dev) gave a quick glance at the source code and did find easy access to the type at allocation time. Maybe it is possible to postpone such a type lookup until the first use of the location.
- Creating a `T`, transmuting it to `UnsafeCell<T>` will not allow `UnsafeCell`-like access to `T` (because the original tag of the location will be `Unique`).

1.3 Tags

TB also tags locations and pointers with so called *tags*. In addition to the tags from SB, TB introduces new tags: `SharedImmutRead` and `Two-Phase-Unique` . TB does not make use of the `Disabled` tag, which leaves us with these tags:

	no aliasing	allows aliasing
permits writes	`Unique`	`SharedReadWrite`
read only	`SharedImmutRead`	`SharedReadOnly`
two phase	`Two-Phase-Unique`	

Before using a two phase borrow (read/write/retag) remove it from the tree and create a `Unique` /`SharedReadWrite` with the same id in its place. This is also a deviation from existing behavior, in SB a retag does not change the two-phase status of a borrow.

1.3.1 Tag Invariants

- `Unique` : While this pointer tag exists in the borrow tree, all writes to the location need to be carried out by this pointer or any pointer derived from it.
- `SharedImmutRead` : While this pointer tag exists in the borrow tree, the location is not written to.

All tags are invalidated (the borrow tree will be cleared/deleted), when their location is `!Copy` and moved.

1.3.2 Tag Tree Invariants

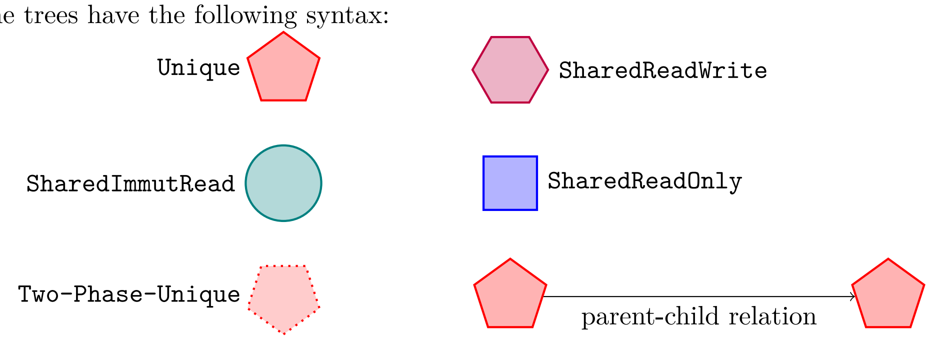
To uphold the invariants mentioned before, these invariants will be enforced upon the tree:
For every

- `Unique` :
  - other branches contain no `Unique` /`SharedReadWrite` pointing to the same location.
  - no `SharedImmutRead` exists that points to the same location.
  - no ancestor is a `SharedImmutRead`
- `SharedImmutRead` : other branches and descendants contain no `Unique` /`SharedReadWrite` pointing to the same location.
- `SharedReadWrite` : no ancestor is a `SharedImmutRead`
- `Two-Phase-Unique` : has no descendants

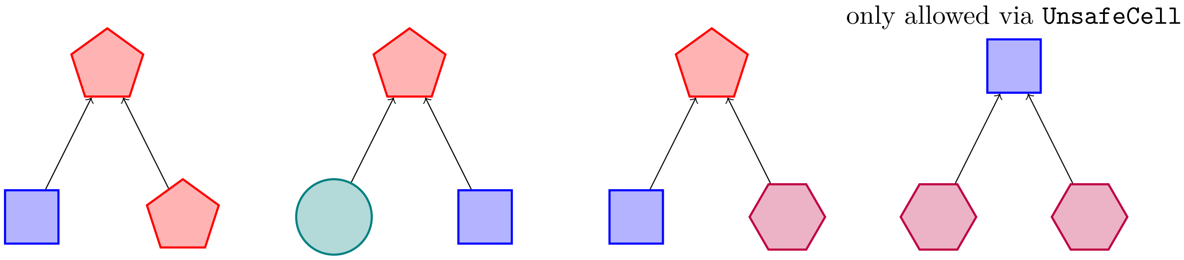
And of course it has to be a tree (no cycles and each node has only one outgoing connection).

1.3.3 Tree Figure Description

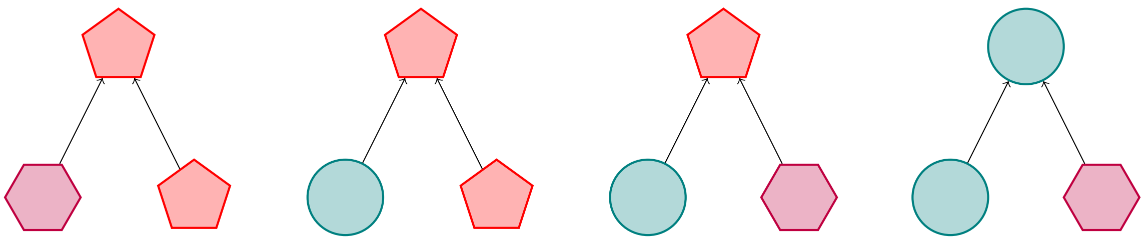
The trees have the following syntax:



1.3.4 Examples of Valid Trees



1.3.5 Examples of Invalid Trees



1.4 Tree Operations

1.4.1 Allocating

Each location will be tracked by one tree. When creating an allocation the root node of the tree is created. Its tag is determined by the nature of the allocation[1]:
When T is not UnsafeCell/UnsafeAliasCell:

- let x: T = ...; $\implies$ SharedImmutRead
- let mut x: T = ...; $\implies$ Unique
- static x: T = ...; $\implies$ SharedImmutRead
- static mut x: T = ...; $\implies$ SharedReadWrite

UnsafeCell:

- let x: UnsafeCell<T> = ...; $\implies$ SharedReadOnly
- let mut x: UnsafeCell<T> = ...; $\implies$ SharedReadOnly
- static x: UnsafeCell<T> = ...; $\implies$ SharedReadOnly
- static mut x: UnsafeCell<T> = ...; $\implies$ SharedReadWrite

UnsafeAliasCell:

- let x: UnsafeAliasCell<T> = ...; $\implies$ SharedReadWrite
- let mut x: UnsafeAliasCell<T> = ...; $\implies$ SharedReadWrite
- static x: UnsafeAliasCell<T> = ...; $\implies$ SharedReadWrite
- static mut x: UnsafeAliasCell<T> = ...; $\implies$ SharedReadWrite

[1] What about Box and other smart pointers? We also need to track allocations done by the allocator. They should receive the same treatment as let mut x: T = ...;

When `T` contains an `UnsafeCell`/`UnsafeAliasCell` then we allocate each value contained separately, allocate memory for the type as if it did not contain any `Cells` and then move each field to its respective position. When moving the contents of a location, we also move the tag to the new location. This results in us finely tracking values stored in `UnsafeCell`/`UnsafeAliasCell` even within enums and unions [2].

1.4.2 Read Access

When reading we need to ensure that the tag still exists. Additional care needs to be taken, to ensure our variants are still upheld.

Reading a pointer with tag `Unique` will change all derived `Unique` ↦ `SharedImmutRead` and all derived `SharedReadWrite` ↦ `SharedReadOnly` . This ensures that two reads of the same `Unique` pointer will result in the same value, if that pointer is never used to write in between (deriving a new pointer with write permission counts as a write access to the original pointer).

Reading a pointer with a different tag does not require any modification of the tree. Only a check for existence of the pointer tag.

1.4.3 Write Access

Writing to a pointer with tag `Unique` will remove all derived `Unique` , `SharedImmutRead` and `Two-Phase-Unique` pointers, inheriting any of their `SharedReadOnly` children. All derived `SharedReadWrite` will be turned into `SharedReadOnly` . This re-asserts the uniqueness of the pointer (removing all derived aliasing pointers/turning them into read only) and prevents future reads using a `SharedImmutRead` (because that pointer would then observe a written-to value) and prevents future activation of any `Two-Phase-Unique` .

Writing to a pointer with tag `SharedReadWrite` does not require any any modification of the tree. Only a check for existence of the pointer tag.

1.4.4 New Pointer Creation

When casting a reference to a raw pointer (`Unique` / `SharedImmutRead` ↦ `SharedReadWrite` / `SharedReadOnly` , or `Unique` ↦ `Two-Phase-Unique`) or reborrowing a reference, treat it the same as a write/read access of that pointer.

When deriving a pointer with `Unique` / `SharedImmutRead` tag from a pointer with `SharedReadWrite` / `SharedReadOnly` tag, we might need to modify the tree:

- `SharedReadWrite` ↦ `Unique` :
    1. check if the pointer tag exists
    2. remove all descendants with `Unique` , `SharedReadWrite` , `SharedImmutRead` , `Two-Phase-Unique` tag and inherit their children.
    3. recurs the tree upwards to the root, exploring any side branches (all branches where the `SharedReadWrite` is not a descendant) and remove all descendants with `Unique` , `SharedReadWrite` , `SharedImmutRead` , `Two-Phase-Unique` tag and inherit their children to the branching ancestor.
- `*` ↦ `SharedImmutRead`
    1. check if the pointer tag exists
    2. replace all `Unique` / `SharedReadWrite` descendants with `SharedImmutRead` /`SharedReadOnly`
    3. recurs the tree upwards to the root, exploring any side branches (all branches where the `*` is not a descendant) and replace all `Unique` / `SharedReadWrite` descendants with `SharedImmutRead` / `SharedReadOnly` .

There are only three legal ways to turn `*const T` into `*mut T`:

- the `get()` function of `UnsafeCell`

[2] I (y86-dev) have no idea if the type can be known at allocation time in miri. I glanced at the source code and was not able to find easy access to the type information at the creation site of the `Stacks` struct in current SB.

- the `get()` function of `UnsafeAliasCell`
- pointer casting `*const T` to `*mut T`, if the original raw pointer that was created was `*mut T` (so round tripping `*mut T` $\mapsto$ `*const T` $\mapsto$`*mut T` is allowed).

1.4.5 Moving

When moving a value, the trees for the locations will be removed. This prevents read/write after move.

1.5 Comparison With the Stack

1.5.1 `Unique` and `SharedImmutRead`

When only `SharedImmutRead` and `Unique` are used, the tree behaves exactly as the stack would. When only using `Unique` , there can only exist one branch containing `Unique` at any one time. Creating a new branch will either extend or remove the previous branch.

When also dealing with `SharedImmutRead` , the tree may "freeze" when a `SharedImmutRead` is created, turning all `Unique` that are below the parent pointer into `SharedImmutRead` 's as well. When a `Unique` above the parent is used (all the `Unique` 's above the parent will still remain) then all pointers below it will be removed, leaving us again with a single branch of `Unique` 's.

This is almost exactly the same behavior as the stack, only giving more information about the relation of two `SharedImmutRead` 's (in the stack you cannot know if they are derived from each other or from a different pointer further down the stack).

1.5.2 `SharedReadOnly` and `SharedReadWrite`

When creating `SharedReadOnly` and `SharedReadWrite` the tree will also branch (like when using `SharedImmutRead`), but this time one can write using `SharedReadWrite` pointers and observe locations that are written to with `SharedReadOnly` .

The invariants of `SharedImmutRead` and `Unique` are still upheld, so when using one, `SharedReadWrite` will be transformed into `SharedReadOnly` and further writes will be denied. `SharedReadOnly` will not be disturbed by anything (except location moves of `!Copy` data) and thus one is allowed to read the memory even if there exist any other pointers.

The Stack has to add some non-stack operations to support these pointers, inserting below already added elements and replacing `Unique` with `Disabled` to still allow access via `SharedReadWrite` when the parent `Unique` is invalidated.

1.5.3 `Two-Phase-Unique`

A `Two-Phase-Unique` behaves almost exactly as a `SharedImmutRead` , when a `SharedReadWrite` or `Unique` is created, any `Two-Phase-Unique` is removed and will not allow future activation. On activation (TB also activates on retag) it is replaced by a `Unique` and counts as a write-use of the parent pointer, thus invalidating any aliasing `SharedImmutRead` 's.

SB maps two-phase borrows to `SharedReadWrite` , requiring additional information from MIR when a two-phase borrow is activated.[3]

1.6 Protectors

Protectors are still needed in TB, as they solve the same problem that is also present in SB.

2 Rationale

this section does not yet contain a formal proof.

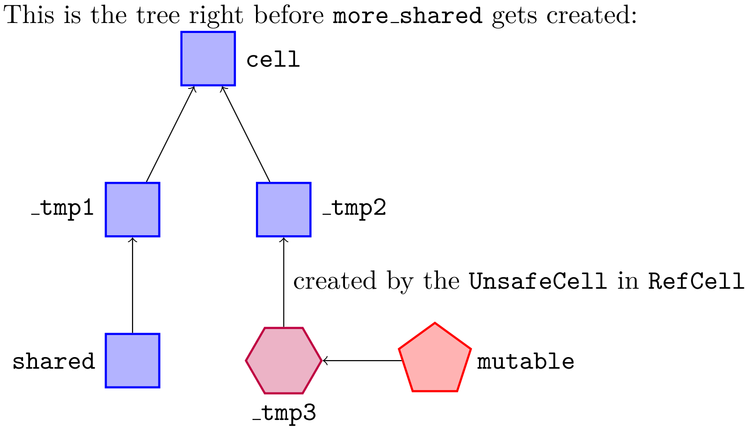
[3] Needs verification

2.1 Examples from the SB paper

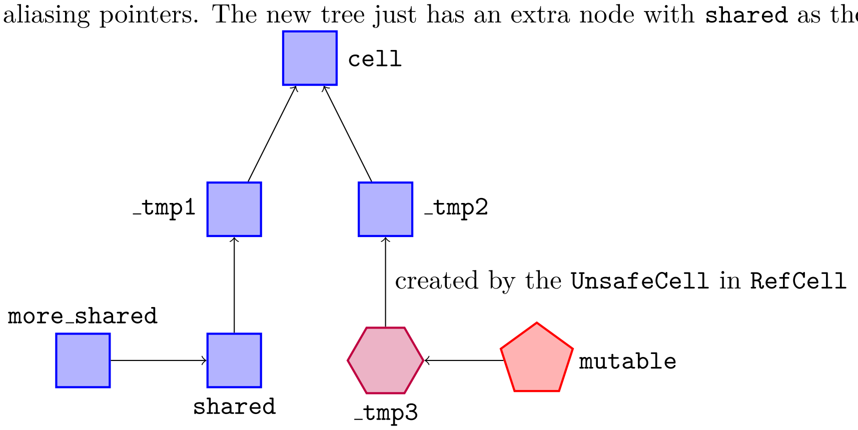
2.1.1 Evil RefCell

```
fn evil_ref_cell (shared : &RefCell<i32>, mutable : &mut i32) {
    retag [fn] shared; retag [fn] mutable;
    let more_shared = &*shared;
    *mutable = 23;
}
fn main() {
    let cell = RefCell::new(42);
    evil_ref_cell(&cell, &mut *cell.borrow_mut().unwrap());
}
```

This is the tree right before more_shared gets created:



When more_shared gets created, we reborrow the SharedReadOnly , which specifically allows mutation via aliasing pointers. The new tree just has an extra node with shared as the parent:

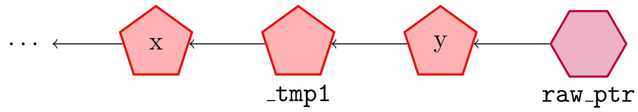


2.1.2 Bad Raw Pointer Pattern

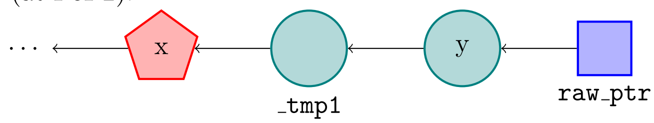
```
fn make_raw (y: &mut i32) -> * mut i32 { retag [fn] y; y as *mut i32 }

fn bad_pattern (x: &mut i32) {
    retag [fn] x;
    let raw_ptr = make_raw(x);
    // Point of interest 1
    let val1 = *x;
    // Point of interest 2
    let val2 = unsafe { *raw_ptr };
}
```

After PoI 1 the borrow tree will look like this:



We then access `x` for reading, thus revoking write access from all descendants. The tree now looks like this (at PoI 2):



When we now try to read from `raw_ptr`, then this is still allowed, because it still exists in the tree.

2.2 More Examples

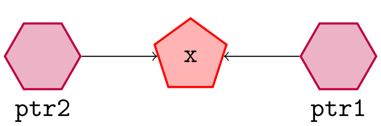
2.2.1 Mutable Pointer Reborrow

```
fn main() { unsafe {
    let mut x = 0;
    let ptr1 = &mut x as *mut i32;
    // PoI 1
    let ptr2 = &mut x as *mut i32;
    // PoI 2
    *ptr1 = 42;
//   ^^^^^ error: tag not in the borrow tree
    *ptr2 = 60;
} }
```

2.2.2 SharedReadWrite Reborrow

Creating raw pointers directly without temporary mutable references is ok:

```
fn main() { unsafe {
    let mut x = 0;
    let ptr1 = addr_of_mut!(x);
    // PoI 1
    let ptr2 = addr_of_mut!(x);
    // PoI 2
    *ptr1 = 42;
    *ptr2 = 60;
} }
```



3 Unresolved Patterns

This section goes over some examples which exhibit not yet finalized behavior.

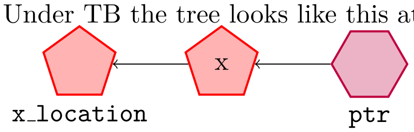
3.1 Deviations From SB/Miri

3.1.1 Allow reading from `*mut T` after writes using the parent `&mut T`

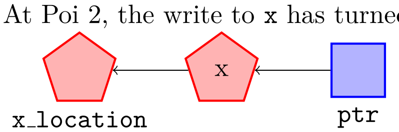
TB does not consider this as UB, but miri does:

```
fn main() { unsafe {
    let mut x_location = 0;
    let x = &mut x;
    let ptr = &mut *x as *mut i32;
    // PoI 1
    *x += 42;
    // PoI 2
    println!("{:?}", *ptr);
    //               ^^^^ miri errors here: trying to reborrow <untagged> for SharedReadOnly
    // permission at alloc1283[0x0], but that tag does not exist in the borrow stack for this location
} }
```

Under TB the tree looks like this at Poi 1:



At Poi 2, the write to `x` has turned the `SharedReadWrite` into a `SharedReadOnly` :



This would then allow reading from that pointer.
In favor of this behavior:

- natural and consistent behavior of aliasing pointers (if you were to cast ptr to `*const i32` then this would also allow later reads).
- enforces the uniqueness of the write ability of `&mut`

Against this behavior:

- deviating from current SB and miri behavior.
- disallows this pattern (SB/miri is ok with this):

```
fn main() { unsafe {
    let mut x = 0;
    let x = &mut x;
    let ptr1 = &mut *x as *mut i32;
    let ptr2 = ptr1;
    *ptr1 += 20;
    *ptr2 += 1;
    println!("{:?}", *x);
    //               ^^ TB note: this read to an Unique pointer revokes
    //                           the write access of all derived pointers.
    *ptr1 += 1;
//  ^^^^^ TB error: tried to write using a SharedReadOnly pointer derived from x.
    *ptr2 += 20;
    println!("{:?}", *x);
} }
```

3.1.2 Use/Write after move

TB considers this UB, but miri does not:

```rust
fn main() { unsafe {
    let mut x = "Hello world!".to_owned();
    println!("addr of x: {:p}", &x);
    let ptr = &mut x as *mut String;
    let y = x;
    println!("addr of y: {:p}", &y);
    println!("ptr points to {ptr:p}");
    println!("{}", *ptr);
    //             ^^^^ TB error: tried to read using SharedReadWrite <tag>,
    //                            but that tag points to a location without a borrow tree.
} }
```

In favor of this behavior:

- behavior reflecting the borrow checker.
- allow future optimization (e.g. reuse of stack memory)

Against this behavior:

- deviating from current SB and miri behavior.

4 Open Questions

While writing this document we came across the following questions and did not find an immediate answer to them:

- Is it allowed to create multiple two-phase borrows to the same location at the same time and then only activate one?

References

[1] Mutable aliasing – zulip discussion. `https://rust-lang.zulipchat.com/#narrow/stream/213817-t-lang/topic/aliased.20mutability`.

[2] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: An aliasing model for rust. `https://plv.mpi-sws.org/rustbelt/stacked-borrows/paper.pdf` [Accessed on 2022-04-27].