Tree Borrows

An aliasing model for Rust

Neven Villani, Ralf Jung, Derek Dreyer

ENS Paris-Saclay and MPI-SWS Saarbrücken

A motivating example

See code example: `examples/rw-elim`

A motivating example

See code example: `examples/rw-elim`

optimizations rely on type information
+ `unsafe` can bypass the type system
= many optimizations are unsound in the presence of `unsafe`

Do we have to give up on all these optimizations ?

Is my optimization unsound ?
No, it's the client that is UB.

UB: "Undefined Behavior"

- semantics of a program that contains UB: any behavior
- equivalently: the compiler can assume that UB does not occur

The compiler is allowed to "miscompile" programs that contain UB.

In this case: pointer aliasing UB.
(Other kinds: uninitialized memory, data races, dangling pointers, invalid values, ...)

The role of pointer aliasing UB

(Tree|Stacked) Borrows

- **define** an operational semantics with UB of reborrows and memory accesses
- **detect** violations of the aliasing discipline it dictates

in order to

- **enable compiler optimizations** by ruling out aliasing patterns
  - remove redundant loads and stores
  - permute noninterfering operations
- **justify LLVM attributes** on pointers that `rustc` emits
  - `noalias`: added by rustc on references, means that the data is not being mutated through several different pointers
  - `dereferenceable`: added by rustc on references, means that the pointer is not null or dangling or otherwise invalid to dereference

Motivating example: with Miri

Miri (`github:rust-lang/miri`) is

- a Rust interpreter
- that detects UB

Back to `examples/rw-elim`: run with Miri

Basics of Stacked Borrows (SB)

Starting observation

Proper usage of mutable references follows a stack discipline.

Key ideas

- per-location tracking of pointers
- use a stack to store pointer identifiers
- on each reborrow a new identifier is pushed to the stack
- a pointer can be used if its identifier is in the stack
- using of a pointer pops everything above it (more recent)

An example SB execution

```
let x = &mut 0; // [x]
let y = &mut *x; // [x, y] (reborrow of x into y)
let z = &mut *x; // [x, z] (usage of x pops y)
                 // (reborrow of x into z)
*y = 42; // y is not in the stack, UB !
*z = 57;
```

SB too strict ?

Many optimizations are possible again, but...
UB in `tokio`, `pyo3`, `rkyv`, `eyre`, `ndarray`, `arrayvec`, `slotmap`, `nalgebra`, `json`, ...

Enforcing SB would break too much backwards compatibility, so right now the compiler cannot apply any SB-enabled optimizations.

SB too strict ?

Many optimizations are possible again, but...
UB in `tokio`, `pyo3`, `rkyv`, `eyre`, `ndarray`, `arrayvec`, `slotmap`, `nalgebra`, `json`, ...

Enforcing SB would break too much backwards compatibility, so right now the compiler cannot apply any SB-enabled optimizations.

Essential tradeoff

More UB is...

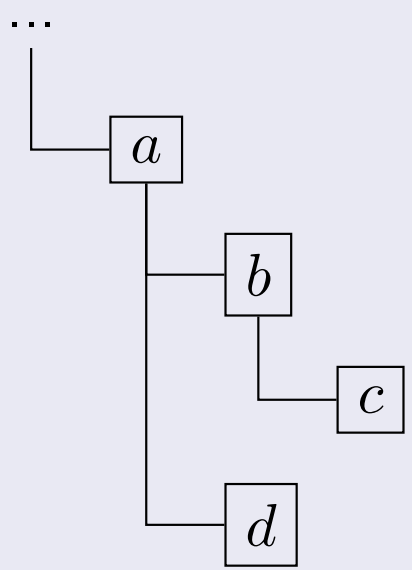
- more optimizations (stronger assumptions)
- less safety (especially if rules are vague)

UB is the responsibility of the user, so
too much UB makes users unhappy.

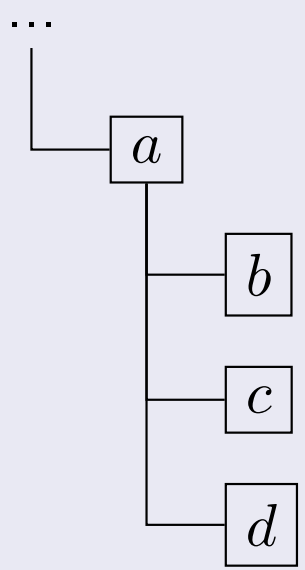
Information loss in the stack

$$[..., a, b, c, d]$$

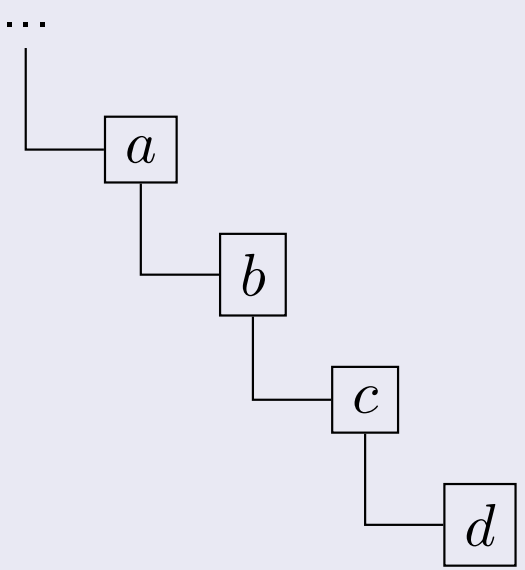
```
let a = &...;
let b = &*a;
let c = &*b;
let d = &*a;
```




```
let a = &...;
let b = &*a;
let c = &*a;
let d = &*a;
```




```
let a = &...;
let b = &*a;
let c = &*b;
let d = &*c;
```



~~Stacked~~ Tree Borrows (TB)

Starting observation

Proper usage of mutable references follows a stack discipline.

Key ideas

- per-location tracking of pointers
- use a stack to store pointer identifiers
- on each reborrow a new identifier is pushed to the top of the stack
- a pointer can be used if its identifier is in the stack
- using of a pointer pops everything above it (more recent)

~~Stacked~~ Tree Borrows (TB)

Starting observation

Proper usage of all pointers follows a tree discipline.

Key ideas

- per-location tracking of pointers
- use a tree to store pointer identifiers
- on each reborrow a new identifier is added as a leaf of the tree
- each pointer has permissions
- a pointer can be used if its permission allows it (to be defined)
- using a pointer makes incompatible (to be defined) pointers lose permissions

When are pointers different ?

LLVM and Rust specifications: “other references/pointers”
Suggests that two pointers to the same data are “different”.

When are pointers different ?

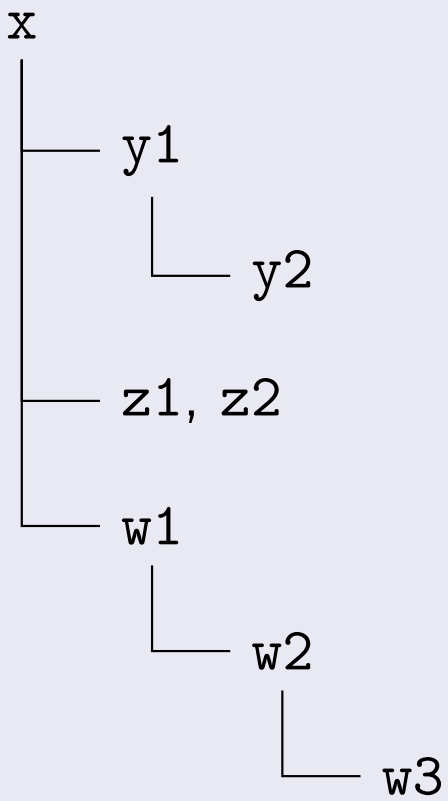
LLVM and Rust specifications: “other references/pointers”
Suggests that two pointers to the same data are “different”.

A pointer in our semantics is:

```
struct Pointer {
    address: usize,
    size: usize,
    tag: usize, // <- added specifically for TB/SB
}
```

Two pointers to the same data are not equal for TB/SB if they have different tags.

A Tree of pointers




```
let x = &mut 0u64;

let y1 = &mut *x;
let y2 = &*y1;

let z1 = &*x;
let z2 = z1 as *const u64;

foo(x);
fn foo(w2: &mut u64) {
    let w3 = &*w2;
}
```

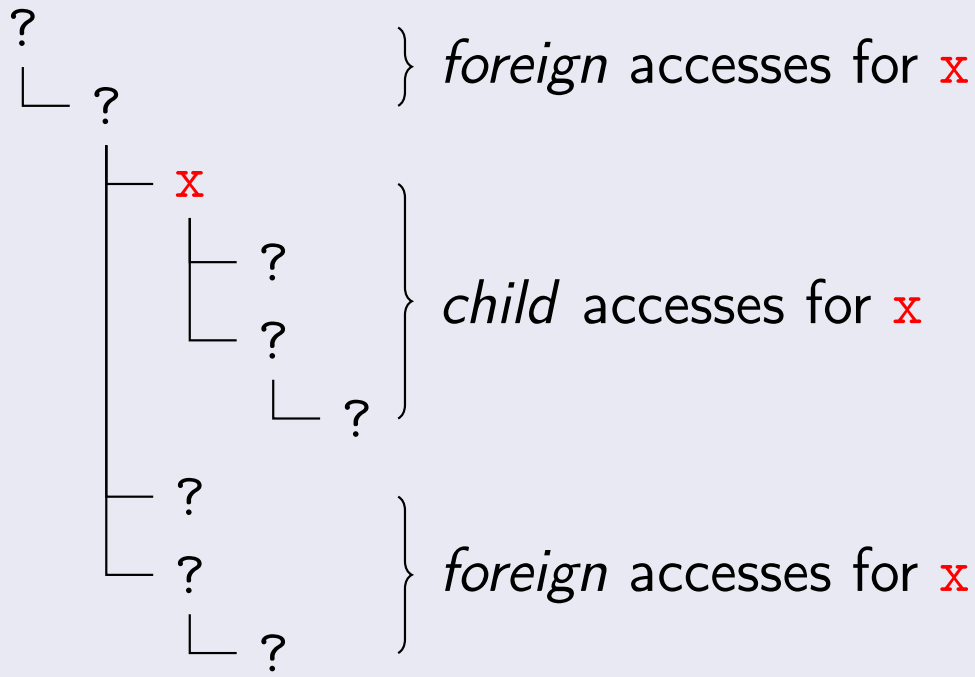
What's in the tree ?

Each pointer is given a tag

(Tree|Stacked) Borrows track:

- **permission**: per tag, per location;
- **hierarchy** between tags;
- accesses are done through a tag:
  - **require permissions** of the tag
    (UB if the permissions are insufficient)
  - **update permissions** of other tags
    (UB if the modification is forbidden)

One pointer, 2×2 kinds of accesses



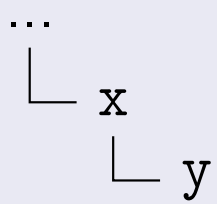
each *read* or *write*.

"pointers based on..."

LLVM specification: "pointer `y` is based on pointer `x`"
$\simeq$ TB: "`y` is a child of `x`".

Kinds of accesses: examples

```
let x = &mut ...;
let y = &mut *x;

*x = 1; // Write access; foreign for y; child for x.
let _ = *y; // Read access; child for y; child for x.
```



Summary

- pointers identified by a tag;
- tags are stored in a tree structure;
  - reborrows create fresh tags,
  - new tag is a child of the reborrowed tag
- each tag has per-location permissions;
  - permissions allow or reject *child accesses* (done through child tags)
  - permissions evolve in response to *foreign accesses* (done through non-child tags).

How many permissions ?

In short: one permission per "kind of pointer"

- (interior) mutability,
- lifetime information,
- creation context,
- ...

Guarantees required of pointers determine behavior of permissions:

- pointer allows mutation
  $\Rightarrow$ permission allows child writes
- pointer guarantees uniqueness
  $\Rightarrow$ permission prevents foreign accesses
- ...

Active, Frozen, Disabled

Core triplet of permissions to represent

- unique mutable references: `Active`,
- shared immutable references: `Frozen`,
- lifetime ended: `Disabled`.

Child read: must allow reading

- `Active` → `Active`
- `Frozen` → `Frozen`
- `Disabled` → UB

Child write: must allow writing

- `Active` → `Active`
- `Frozen` → UB
- `Disabled` → UB

Active, Frozen, Disabled

Core triplet of permissions to represent

- unique mutable references: `Active`,
- shared immutable references: `Frozen`,
- lifetime ended: `Disabled`.

Foreign read: no longer unique

- `Active` → `Frozen`
- `Frozen` → `Frozen`
- `Disabled` → `Disabled`

Foreign write: no longer immutable

- `Active` → `Disabled`
- `Frozen` → `Disabled`
- `Disabled` → `Disabled`

Parallel to the borrow checker

Similarities

- ✓ `Active` (`&mut`) readable and writeable
- ✓ `Frozen` (`&`) and all their children are only readable
- ✓ data behind `Active` (`&mut`) is owned exclusively
- ✓ data behind `Frozen` (`&`) is immutable

Differences (OK to be more permissive than the borrow checker)

- ✗ `Active` (`&mut`) demoted to `Frozen` (`&`)
- ✗ several `Active` (`&mut`) can coexist if never written to

Unsoundness (two following subsections fix them)

- ✗ two-phase borrows not handled yet
- ✗ too permissive for `noalias` and `dereferenceable`

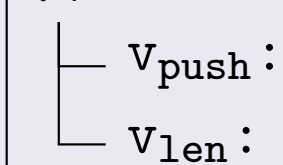
Not all mutable references can be `Active`

Two-phase borrows

Mutable reborrows in function arguments tolerate shared reborrows until function entry.

Core triplet: unsound

```
fn main() {
    let mut v =
        vec![1usize];
    v.push(
        v.len()
    );
}
```



Not all mutable references can be `Active`

Two-phase borrows

Mutable reborrows in function arguments tolerate shared reborrows until function entry.

Core triplet: unsound

```
 fn main() {
>    let mut v =
>        vec![1usize];
     v.push(
         v.len()
     );
 }
```

```
v: Active
├─ v_push :
└─ v_len :
```

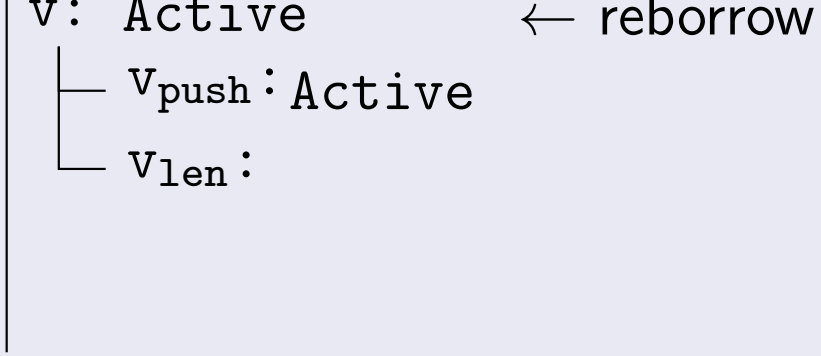
Not all mutable references can be `Active`

Two-phase borrows

Mutable reborrows in function arguments tolerate shared reborrows until function entry.

Core triplet: unsound

```
 fn main() {
     let mut v =
         vec![1usize];
>    v.push(
         v.len()
     );
 }
```



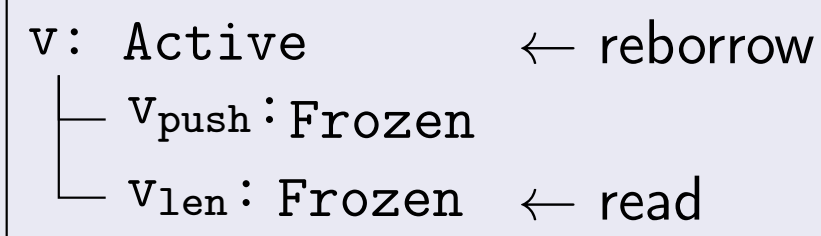
Not all mutable references can be `Active`

Two-phase borrows

Mutable reborrows in function arguments tolerate shared reborrows until function entry.

Core triplet: unsound

```
 fn main() {
     let mut v =
         vec![1usize];
     v.push(
>        v.len()
     );
 }
```



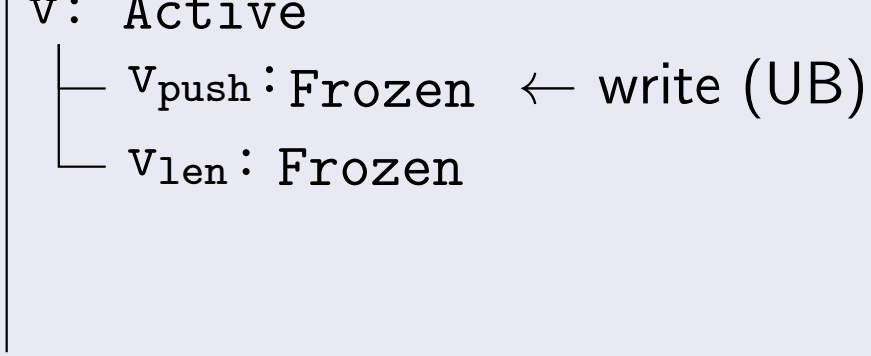
Not all mutable references can be `Active`

Two-phase borrows

Mutable reborrows in function arguments tolerate shared reborrows until function entry.

Core triplet: unsound

```
 fn main() {
     let mut v =
         vec![1usize];
>    v.push(
>        v.len()
>    );
 }
```



New permission: `Reserved`

Intuition

An `&mut` not yet written to is not different from a `&`.

A mutable reference not yet written to

- `Reserved` +child read → `Reserved`
- `Reserved` +foreign read → `Reserved`
- `Reserved` +foreign write → `Disabled`
- `Reserved` +child write → `Active`

⇒ behaves as a `Frozen` until the first child write

⇒ can coexist with each other and with `Frozen`

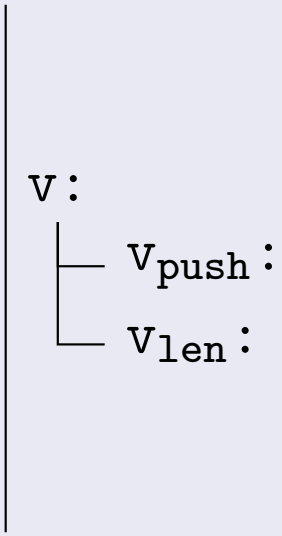
Reserved in action

Two-phase borrows

Mutable reborrows in function arguments tolerate shared reborrows until function entry.

Core triplet + Reserved: fixed

```
fn main() {
    let mut v =
        vec![1usize];
    v.push(
        v.len()
    );
}
```



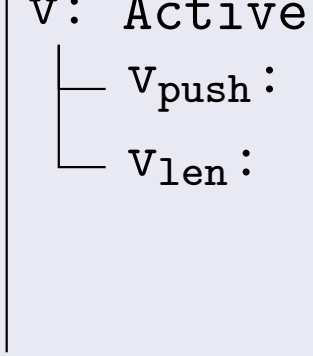
Reserved in action

Two-phase borrows

Mutable reborrows in function arguments tolerate shared reborrows until function entry.

Core triplet + Reserved: fixed

```
 fn main() {
>    let mut v =
>        vec![1usize];
     v.push(
         v.len()
     );
 }
```



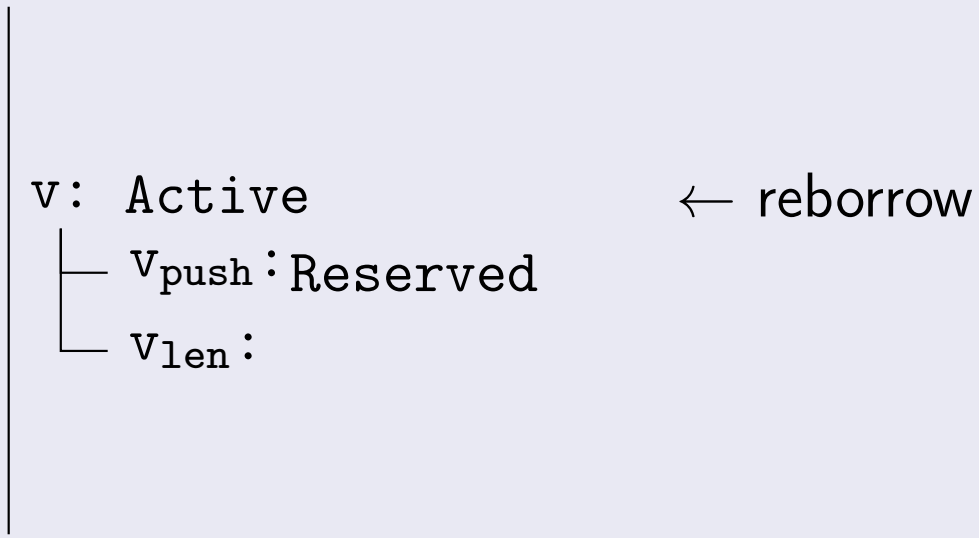
Reserved in action

Two-phase borrows

Mutable reborrows in function arguments tolerate shared reborrows until function entry.

Core triplet + Reserved: fixed

```
 fn main() {
     let mut v =
         vec![1usize];
>    v.push(
         v.len()
     );
 }
```



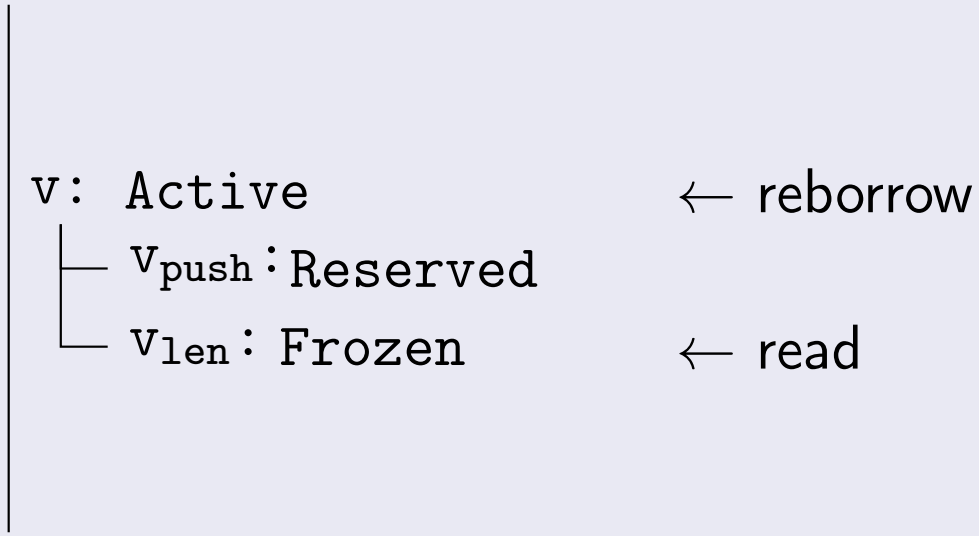
Reserved in action

Two-phase borrows

Mutable reborrows in function arguments tolerate shared reborrows until function entry.

Core triplet + Reserved: fixed

```
 fn main() {
     let mut v =
         vec![1usize];
     v.push(
>        v.len()
     );
 }
```



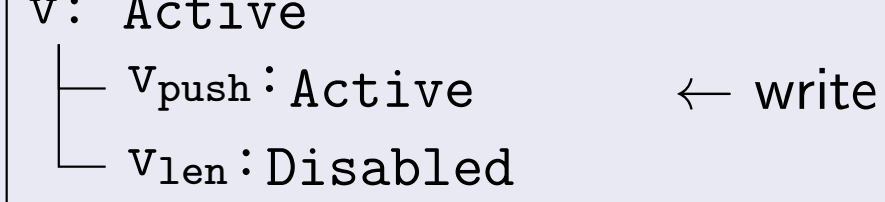
Reserved in action

Two-phase borrows

Mutable reborrows in function arguments tolerate shared reborrows until function entry.

Core triplet + Reserved: fixed

```
 fn main() {
     let mut v =
         vec![1usize];
>    v.push(
>        v.len()
>    );
 }
```



Protectors lock permissions until the end of the function

LLVM `noalias` (in TB terms)

No foreign access during the same function call as a child write.

Core triplet + `Reserved`: unsound

```
fn write(x: &mut u64) {
    *x = 42;  // x: Active
    opaque(/* foreign read for x */);
    // x: Frozen
    // `x` does not satisfy the requirements of `noalias`
}
```

Protectors lock permissions until the end of the function

Intuition

`noalias` requires exclusive access during the entire function call, so we remember the set of all functions that have not yet returned and enforce exclusivity for their arguments.

Concept adapted from Stacked Borrows: protectors.

- references get a protector on function entry
- protector lasts until the end of the call

While protected, behavior changes

- `Reserved` +foreign read → ~~`Reserved`~~ `Frozen`
- `Reserved/Active/Frozen` +foreign write → ~~`Disabled`~~ UB
- `Active` +foreign read → ~~`Frozen`~~ UB

Protectors lock permissions until the end of the function

LLVM `noalias` (in TB terms)

No foreign access during the same function call as a child write.

Core triplet + `Reserved` + protectors: fixed

```
fn write(x: &mut u64) {
    *x = 42; // x: [P] Active
    opaque(/* foreign read for x */);
    // x: [P] Frozen
    // UB: while x is protected,
    // Active -> Frozen is forbidden
}
```

Summary

- `Reserved`, `Active`, `Frozen`, `Disabled` represent different possible states of pointers.
- Interactions with child and foreign accesses enforce uniqueness/immutability guarantees.
- Protectors are added on function entry to strengthen these guarantees up to the requirements of `noalias`.

Some standard optimizations

Possible in...	SB	TB
Insert speculative read	✓	✓
Insert speculative write	✓	✗
Remove redundant read	✓	✓
Remove redundant write	✓	✓
Reorder read-write	✓	✗
Reorder read-write (fn args)	✓	✓
Reorder read-read	✗	✓
Reorder read-read (fn args)	✓	✓
Reorder write-write	✓	✓
Reorder write-write (fn args)	✓	✓
Reorder write-read	✓	✓
Reorder write-read (fn args)	✓	✓
Reorder reborrow-reborrow	✗	✓

Some standard optimizations

Possible in...	SB	TB	
Insert speculative read	✓	✓	
Insert speculative write	✓	✗	←
Remove redundant read	✓	✓	
Remove redundant write	✓	✓	
Reorder read-write	✓	✗	←
Reorder read-write (fn args)	✓	✓	
Reorder read-read	✗	✓	
Reorder read-read (fn args)	✓	✓	
Reorder write-write	✓	✓	
Reorder write-write (fn args)	✓	✓	
Reorder write-read	✓	✓	
Reorder write-read (fn args)	✓	✓	
Reorder reborrow-reborrow	✗	✓	

Some standard optimizations

Possible in...	SB	TB	
Insert speculative read	✓	✓	
Insert speculative write	✓	✗	
Remove redundant read	✓	✓	
Remove redundant write	✓	✓	
Reorder read-write	✓	✗	
Reorder read-write (fn args)	✓	✓	
Reorder read-read	✗	✓	←
Reorder read-read (fn args)	✓	✓	
Reorder write-write	✓	✓	
Reorder write-write (fn args)	✓	✓	
Reorder write-read	✓	✓	
Reorder write-read (fn args)	✓	✓	
Reorder reborrow-reborrow	✗	✓	←

✓ Reorder write-any (fn args)

```
fn write(x: &mut u64) {
    // x: [P] Reserved
    *x = 42; // (optimization: move down ?)
    // x: [P] Active
    opaque(/* maybe foreign read/write */);
    // x: [P] Active (Frozen|Disabled -> UB)


}
```

✓ Reorder write-any (fn args)

```
fn write(x: &mut u64) {
    // x: [P] Reserved


    opaque(/* assume no foreign read/write */);
    // x: [P] Reserved
    *x = 42;
    // x: [P] Active
}
```

✓ Insert speculative read

```
fn read(x: &u64) -> u64 {
    // x: [P] Frozen

    opaque(/* maybe foreign write */);
    // x: [P] Frozen (Disabled -> UB)
    *x // (optimization: move up ?)
}
```

✓ Insert speculative read

```
fn read(x: &u64) -> u64 {
    // x: [P] Frozen
    let val = *x;
    opaque(/* assume no foreign write */);
    // x: [P] Frozen
    val
}
```

Insert speculative write

Possible strengthening

Write to mutable references on function entry.

✗ Base model

```
fn foo(x: &mut u64) {
    // x: [P] Reserved

    opaque(/* maybe foreign read/write */);
    // x: [P] Reserved|Frozen (Disabled -> UB)
    *x = 42; // (optimization: move up ?)
}
```

Insert speculative write

Possible strengthening

Write to mutable references on function entry.

✓ Strengthened model

```
fn foo(x: &mut u64) {
    // x: [P] Active

    opaque(/* maybe foreign read/write */);
    // x: [P] Active (Frozen|Disabled -> UB)
    *x = 42;
}
```

Insert speculative write

Possible strengthening

Write to mutable references on function entry.

✓ Strengthened model

```
fn foo(x: &mut u64) {
    // x: [P] Active
    *x = 42;
    opaque(/* assume no foreign read/write */);
    // x: [P] Active (Frozen|Disabled -> UB)

}
```

Insert speculative write

Possible strengthening

Write to mutable references on function entry.

✓ "as_mut_ptr" pattern

```
// as_mut_ptr: &mut [T] -> *mut T (does not actually write)
let raw = buffer.as_mut_ptr(); // creates raw: Reserved
let shr = buffer.as_ptr().add(1); // creates shr: Frozen
                                  // raw stays Reserved (foreign read)
copy_nonoverlapping(shr, raw, 1); // raw gets activated
```

Insert speculative write

Possible strengthening

Write to mutable references on function entry.

✗ "as_mut_ptr" pattern: strengthened model

```
// as_mut_ptr: &mut [T] -> *mut T (may speculatively write)
let raw = buffer.as_mut_ptr(); // creates raw: Active
let shr = buffer.as_ptr().add(1); // creates shr: Frozen
                                  // raw becomes Frozen (foreign read)
copy_nonoverlapping(shr, raw, 1); // write through Frozen: UB
```

Reorder write-read

Possible strengthening

Foreign read makes `Active` become `Disabled` (rather than `Frozen`)

✗ Base model

```
let x = &mut *y;
// x: Reserved
*x = 42; // (optimization: move down ?)
// x: Active
opaque(/* maybe foreign read/write */);
// x: Active|Frozen|Disabled


let _ = *x;
// x: Active|Frozen (Disabled -> UB)
```

Reorder write-read

Possible strengthening

Foreign read makes `Active` become `Disabled` (rather than `Frozen`)

✓ Strengthened model

```
let x = &mut *y;
// x: Reserved
*x = 42; // (optimization: move down ?)
// x: Active
opaque(/* maybe foreign read/write */);
// x: Active|Disabled


let _ = *x;
// x: Active (Disabled -> UB)
```

Reorder write-read

Possible strengthening

Foreign read makes `Active` become `Disabled` (rather than `Frozen`)

✓ Strengthened model

```
let x = &mut *y;
// x: Reserved


opaque(/* assume no foreign read/write */);
// x: Reserved
*x = 42;
// x: Active
let _ = *x;
// x: Active
```

Reorder write-read

Possible strengthening

Foreign read makes `Active` become `Disabled` (rather than `Frozen`)

✓ Reorder read-read

```
let x = &mut *z; // x: Reserved
*x = 42; // x: Active
let _ = *x; // x: Active (optimization: move down ?)
let _ = *z; // x: Frozen (foreign read)
```

Reorder write-read

Possible strengthening

Foreign read makes `Active` become `Disabled` (rather than `Frozen`)

✓ Reorder read-read

```
let x = &mut *z; // x: Reserved
*x = 42; // x: Active

let _ = *z; // x: Frozen (foreign read)
let _ = *x; // x: Frozen
```

Reorder write-read

Possible strengthening

Foreign read makes `Active` become `Disabled` (rather than `Frozen`)

✗ Reorder read-read: strengthened model

```
let x = &mut *z; // x: Reserved
*x = 42; // x: Active
let _ = *x; // x: Active (optimization: move down ?)
let _ = *z; // x: Disabled (foreign read)
```

Reorder write-read

Possible strengthening

Foreign read makes `Active` become `Disabled` (rather than `Frozen`)

✗ Reorder read-read: strengthened model

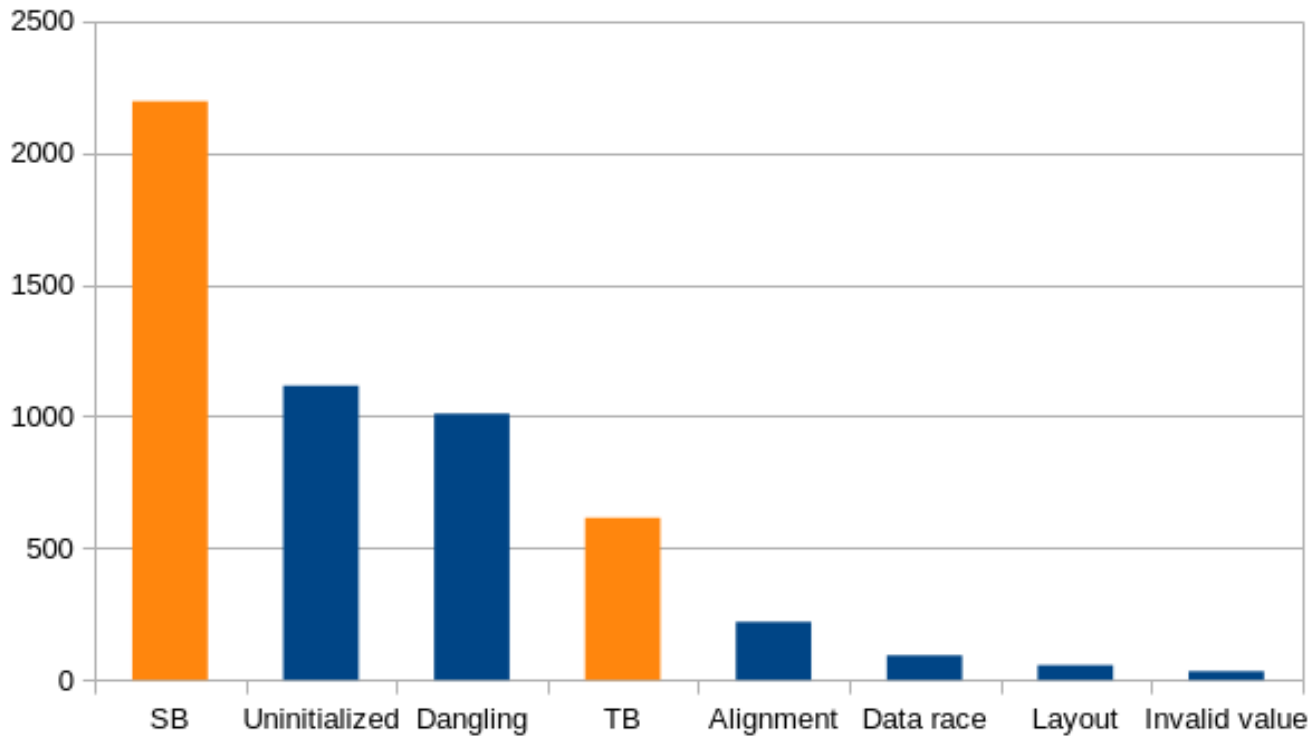
```
let x = &mut *z; // x: Reserved
*x = 42; // x: Active

let _ = *z; // x: Disabled (foreign read)
let _ = *x; // Access through Disabled: UB!
```

Summary

- TB allows read reorderings (SB does not)
- TB allows speculative reads (SB as well)
- TB forbids speculative writes (SB allows them)
  - the model can be strengthened to justify these optimizations...
  - ...at the cost of common patterns.

Counting crates with UB (1/2)

Data obtained using `github:saethlin/miri-tools`



Number of crates on `crates.io` with at least one test that contains UB, for each kind of UB detected by Miri.

Counting crates with UB (2/2)

Kind	SB	TB	Notes
Protector invalidations	70	58	(1)
Protector deallocations	12	12	
Accesses without permissions	998	545	(2)
Accesses outside range	903	0	(3)
Wildcard pointers	213	—	(4)

Number of crates that contain UB, for subclasses of UB defined by SB and TB. From 97 851 crates, of which 3 808 contain UB of any kind

(1) now allowed: `Reserved -> Frozen`
(2) see: `as_mut_ptr`
(3) not included: accesses in wrong allocation
(4) not handled by TB

Notable examples

Accesses outside range

UB in `tokio`, `pyo3`, `rkyv`, `eyre`, `ndarray`, ...
according to SB but not TB

Invalidations by mutable reborrows ("`as_mut_ptr`" pattern)

UB in `arrayvec`, `slotmap`, `nalgebra`, `json`, ...
according to SB but not TB

Summary

- Tree Borrows UB is much less common on `crates.io` than Stacked Borrows UB
  $\Rightarrow$ fulfills goal of being more permissive
- elimination of out-of-bounds UB
  $\Rightarrow$ blocker in SB for many popular crates
- patterns allowed by Stacked Borrows but forbidden by Tree Borrows are rare

Questions ?

TB also has...

- tweaked rules for interactions between interior mutability and protectors/`Reserved`
- performance improvements compared to the naive implementation
  - many tricks to trim tree traversals
  - lazy initialization for out-of-range accesses
- ongoing attempt at formalization in Coq

Don't hesitate to test your code with Miri and send us your interesting/unexpected cases of UB!
Slides and examples on `github:Vanille-N/tree-beamer`
Complementary material: `perso.crans.org/vanille/treebor`