Tree Borrows

An aliasing model for Rust

Neven Villani, R. Jung, J. Hostert, D. Dreyer

ENS Paris-Saclay and MPI-SWS Saarbrücken

Rust

v1.0 2018
General-purpose
Safety- and performance-oriented

Not standardized, ongoing efforts

Features

Rich type system
→ Safety & Speed

ML-like

`unsafe`

C-like

Low-level primitives
→ Control

Purposes

Applications
Libraries
Systems

Tools

Libraries

Async, Algebra, UI, Cryptography, Datetime Data Structures, Parsing, Filesystem, Web, ...

Repo: `crates.io`

Must avoid UB

`cargo`
Package manager

Compilers

Rustc

Official

Fast, optimizing compiler.

Ignores UB

Miri

Sanitizer

Data races
Out of bounds
Uninitialized memory
Invalid layout
Aliasing conflicts: SB or TB

new!

Specifies and detects UB

What is Undefined Behavior ?

Common pattern:

- for expressivity and performance the language introduces low level primitives (`std::mem::transmute`, pointer arithmetic, `Obj.magic`, ...),
- misuse of these primitives can interfere with compiler invariants (garbage collection, well-formedness of typed values, uniqueness, ...),
- guaranteeing deterministic behavior is
  - too expensive (runtime bounds checks, type markers, ...)
  - not feasible (undecidable at compile time)
  - or otherwise undesirable (wasted optimization potential)

What is Undefined Behavior ?

Solution

Make it UB to misuse these constructs.
If a compiler invariant is violated by a language primitive, the compiler can do literally anything.

UB as a contract

Deal between the programmer and the compiler: these primitives are dangerous, only use them if you really know what you are doing.

`unsafe`

Already a selling point of Rust: `unsafe` is explicit.
UB can only occur as a result of well-delimited blocks.

Common examples

Unchecked out-of-bounds accesses

Rust

```
let v = { let x = [0]; unsafe { x.get_unchecked(1) } };
```

C

```
int* x = malloc(sizeof(int)); int v = x[1];
```

OCaml

```
let v = (let x = [| 0 |] in Array.unsafe_get x 1)
```

Common examples

Dereferencing null

Rust

```
let v = unsafe { *(0 as *const usize as *mut usize) }
```

C

```
int v = *(int*)0
```

OCaml

```
let v = (let x: int ref = Obj.magic 0 in !x)
```

Common examples

Constructing an invalid value

Rust

```rust
let x: bool = unsafe { std::mem::transmute(2) };
```

C

```c
bool x = ((union { int i; bool b }){ .i = 2 }).b;
```

OCaml

```ocaml
let x: bool = Obj.magic 2
```

Pointer types in Rust

```
// Raw pointers (unsafe)
*const T
*mut T
// (*const T ⊏ *mut T)

// References (safe)
&'a T // shared and immutable
&'a mut T // unique and mutable
// (&'a T ⊏ &'a mut T)
// ('a ⊂ 'b ⇒ &'a T ⊏ &'b T)
// ('a ⊂ 'b ⇒ &'a mut T ⊏ &'b mut T)

// Wrappers
UnsafeCell, Box, Unique, ...
```

Pointer types in Rust

Just like

```
// x: &mut bool
*x = 4;
```

is a type error (mismatched types `bool` and `u8`),

```
// x: &u8
*x = 4;
```

is also a type error (`&_` does not support assignment), and so is

```
// n: u8
let p = (&mut n, &mut n);
```

(impossible to satisfy lifetime constraints).
Mutability and uniqueness are part of the type!
Can we exploit that?

Is &mut really unique ?

```
let data: u64 = 0;
let r0: &mut u64 = &mut x;

let x: &mut u64 = unsafe { &mut *(r0 as *mut u64) };
let y: &mut u64 = unsafe { &mut *(r0 as *mut u64) };
*x += 1;
*y += 1;
```

Clearly `x` and `y` alias, even though they are both unrelated `&mut`.

Is &mut really unique ?

```
let data: u64 = 0;
let r0: &mut u64 = &mut x;

let x: &mut u64 = unsafe { &mut *(r0 as *mut u64) };
let y: &mut u64 = unsafe { &mut *(r0 as *mut u64) };
*x += 1;
*y += 1;
```

Clearly `x` and `y` alias, even though they are both unrelated `&mut`.

`unsafe` can violate compiler invariants

`unsafe` code can violate uniqueness (and well-formedness) guarantees, so the compiler cannot rely on them for optimizations.

A motivating example for Aliasing UB

```
fn foo(x: &mut u64) {
    let val = *x;
    *x = 42;
    opaque();
    *x = val;
}
```

optimized into

```
fn foo(x: &mut u64) {
    opaque();
}
```

Well-typedness of any program that calls `foo` implies uniqueness of `x` during the execution of `foo`: opaque cannot mutate `x`!

A motivating example for Aliasing UB

```
fn foo(x: &mut u64) {
    let val = *x;
    *x = 42;
    opaque();
    *x = val;
}
```

optimized into

```
fn foo(x: &mut u64) {
    opaque();
}
```

Well-typedness of any program that calls `foo` implies uniqueness of `x` during the execution of `foo`: opaque cannot mutate `x`!
...except if the user uses `unsafe` to violate uniqueness

A motivating example for Aliasing UB

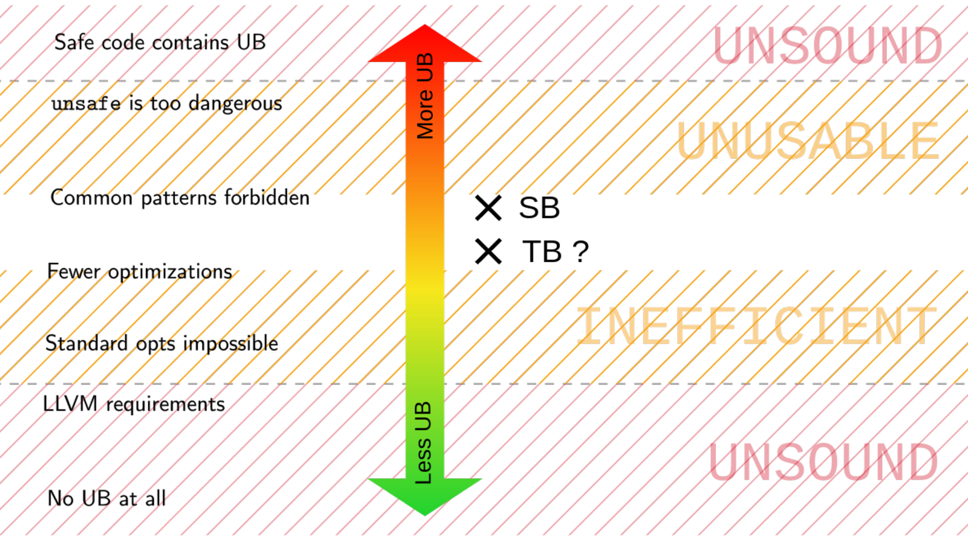
```
fn foo(x: &mut u64) {
    let val = *x;
    *x = 42;
    opaque();
    *x = val;
}
```

optimized into

```
fn foo(x: &mut u64) {
    opaque();
}
```

Well-typedness of any program that calls `foo` implies uniqueness of `x` during the execution of `foo`: opaque cannot mutate `x`!
...except if the user uses `unsafe` to violate uniqueness
...which we are going to assume does not happen: violating uniqueness is UB!

How much UB is enough ? Too much ?



Tree Borrows: specification and detection of pointer aliasing UB

Starting observation

Proper usage of pointers (lifetime inclusion and inheritance of mutability) follows a tree discipline.

- when pointer dies, so do its children
- when pointer requires uniqueness, remove other branches

Key ideas

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

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

- x
  - y1
    - y2
  - z1, z2
  - w1
    - w2
      - w3

```
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;
}
```

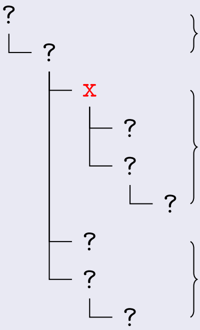
What's in the tree ?

Each pointer is given a tag

Tree Borrows tracks:

- **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



foreign accesses for x

child accesses for x

foreign accesses for x

each *read* or *write*.

Kinds of accesses: examples

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

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

```
...
└ x
  ├ y
  └ z
```

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:

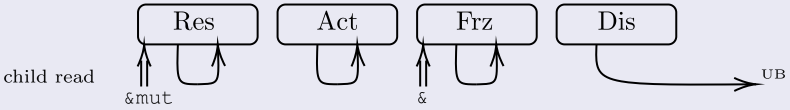
- pointer allows mutation
  $\Rightarrow$ permission allows child writes
- pointer guarantees uniqueness
  $\Rightarrow$ permission is invalidated by foreign accesses
- ...

Reserved, Active, Frozen, Disabled

Basic permissions to represent

- two phase borrowed (mutable in the future): `Reserved`,
- unique mutable references: `Active`,
- shared immutable references: `Frozen`,
- lifetime ended: `Disabled`.

Child read : must allow reading

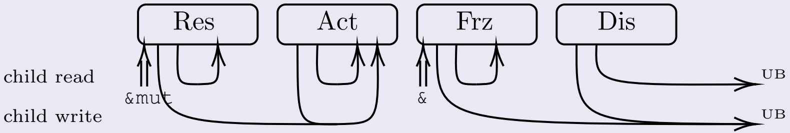


Reserved, Active, Frozen, Disabled

Basic permissions to represent

- two phase borrowed (mutable in the future): `Reserved`,
- unique mutable references: `Active`,
- shared immutable references: `Frozen`,
- lifetime ended: `Disabled`.

Child write: must allow writing

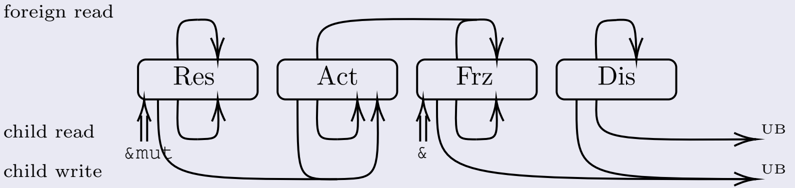


Reserved, Active, Frozen, Disabled

Basic permissions to represent

- two phase borrowed (mutable in the future): `Reserved`,
- unique mutable references: `Active`,
- shared immutable references: `Frozen`,
- lifetime ended: `Disabled`.

Foreign read : no longer unique

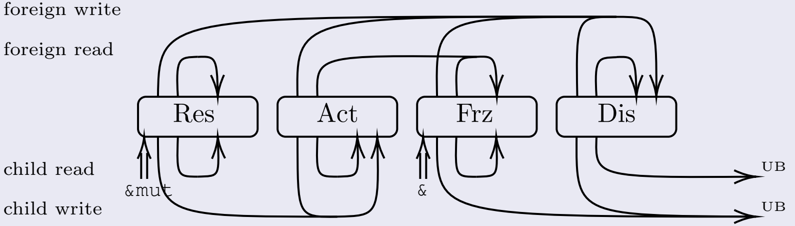


Reserved, Active, Frozen, Disabled

Basic permissions to represent

- two phase borrowed (mutable in the future): `Reserved`,
- unique mutable references: `Active`,
- shared immutable references: `Frozen`,
- lifetime ended: `Disabled`.

Foreign write: no longer immutable



Loss of permissions too early

LLVM `noalias` (in TB terms)

No foreign access during the same function call as a child access if at least one is a write.

1 \ 2	↑R	↑W	↓R	↓W
↑R				×
↑W			×	×
↓R		×		
↓W	×	×		

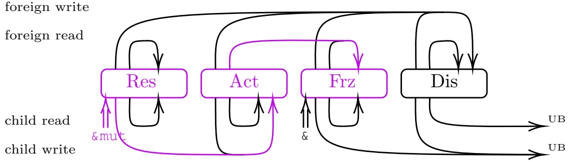
Loss of permissions too early

LLVM `noalias` (in TB terms)

No foreign access during the same function call as a child access if at least one is a write.

Previous model: unsound

```
fn write(x: &mut u64) {
    *x = 42; // activation
    opaque(/* foreign read for x: noalias violation */);
}
```



Protectors lock permissions

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.

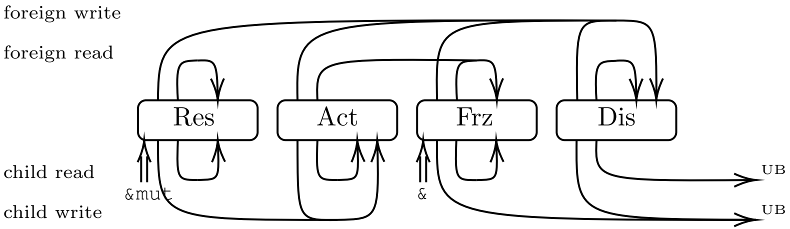
$1 \setminus 2$	$\uparrow$R	$\uparrow$W	$\downarrow$R	$\downarrow$W
$\uparrow$R				$\times$
$\uparrow$W			$\approx$	$\approx$
$\downarrow$R		$\times$		
$\downarrow$W	$\times$	$\times$		

$\times$: should be UB
$\approx$: should be UB earlier

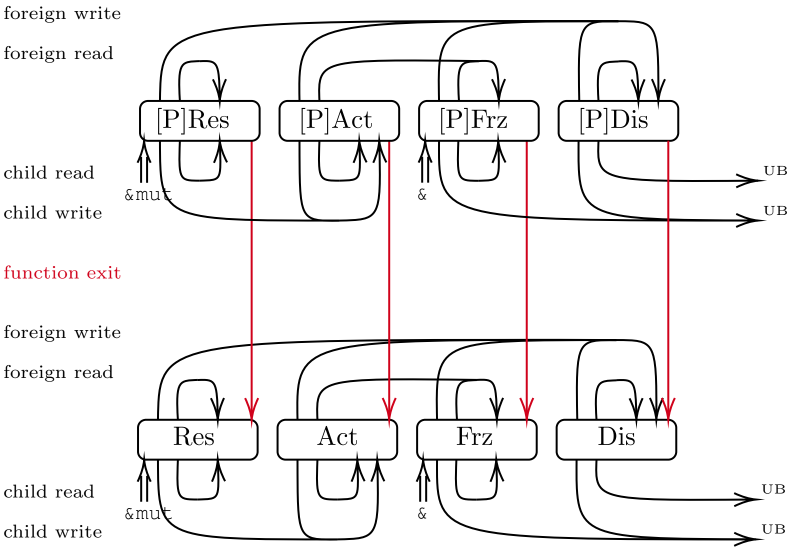
Concept adapted from Stacked Borrows: protectors.

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

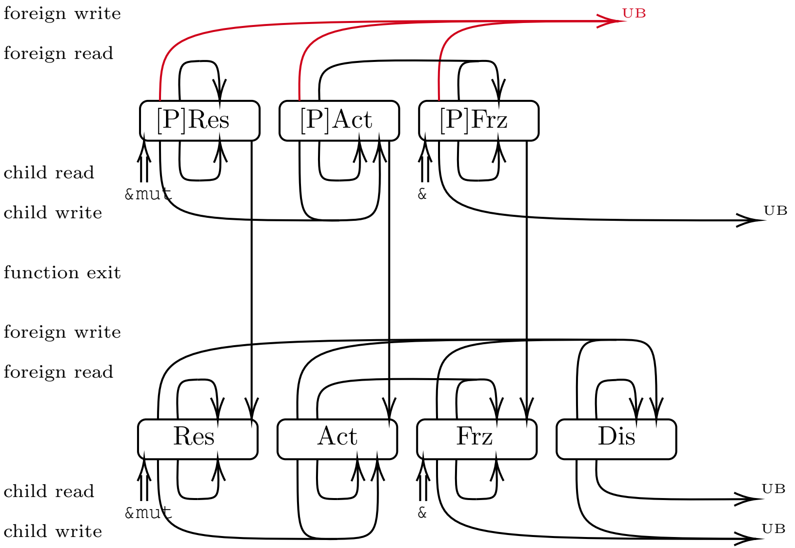
Protectors lock permissions



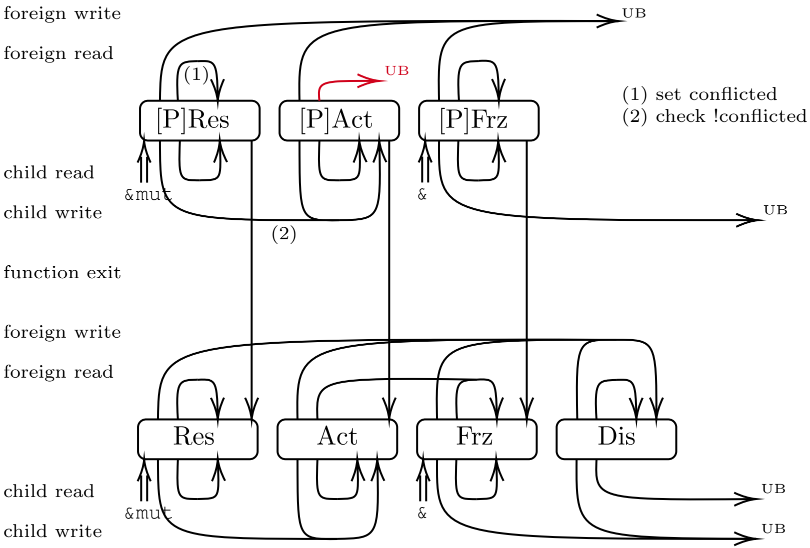
Protectors lock permissions



Protectors lock permissions



Protectors lock permissions



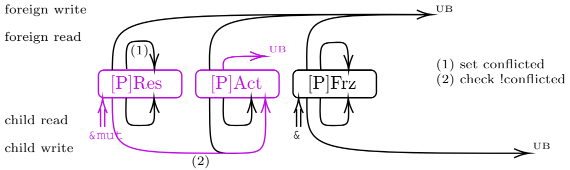
Protectors lock permissions

LLVM `noalias` (in TB terms)

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

With protectors: fixed

```
fn write(x: &mut u64) { // with protector
    *x = 42; // activation
    opaque(/* foreign read for x: noalias violation */);
}
```



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
Swap call-read → read-call (speculative)	✓	✓
Swap read-call → call-read	✓*	✓*
▷ Swap read-read' → read'-read	✓*	✓
Swap call-write → write-call (speculative)	✓*	✗
▷ Swap write-call-write → write-write-call	✓*	✓*
Swap write-call → call-write	✓*	✓*
▷ Swap write-write'-read → write'-write-read	✓	✓*

*: only for protected references

Some standard optimizations

Possible in...	SB	TB	
Swap call-read → read-call (speculative)	✓	✓	
Swap read-call → call-read	✓*	✓*	
▷ Swap read-read' → read'-read	✓*	✓	
Swap call-write → write-call (speculative)	✓*	✗	← SB only
▷ Swap write-call-write → write-write-call	✓*	✓*	
Swap write-call → call-write	✓*	✓*	
▷ Swap write-write'-read → write'-write-read	✓	✓*	← SB only

*: only for protected references

Some standard optimizations

Possible in...	SB	TB
Swap call-read → read-call (speculative)	✓	✓
Swap read-call → call-read	✓*	✓*
▷ Swap read-read' → read'-read	✓*	✓
Swap call-write → write-call (speculative)	✓*	✗
▷ Swap write-call-write → write-write-call	✓*	✓*
Swap write-call → call-write	✓*	✓*
▷ Swap write-write'-read → write'-write-read	✓	✓*

← TB only

*: only for protected references

Swap write-write

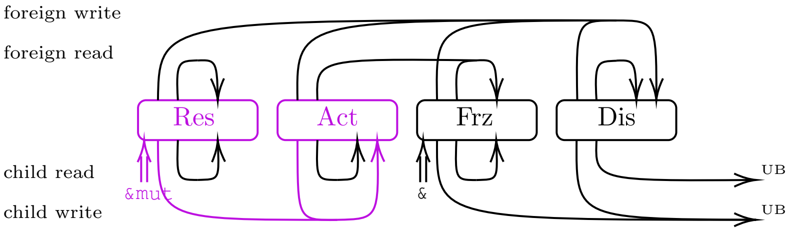
✓ Base model

```
let x = &mut ...;
let y = &mut ...;
*x = 42; // (optimization: move down ?)
*y = 19; // is this a foreign write ?

*x = 57;
```

Swap write-write

✓ Base model

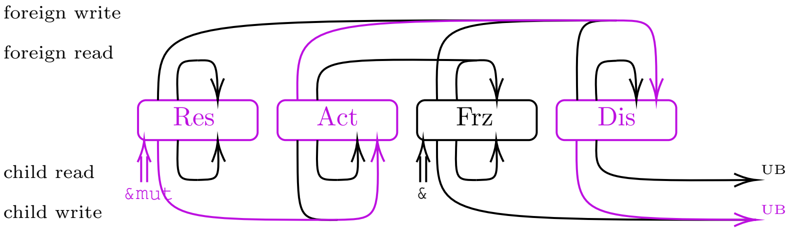
```
let x = &mut ...;
let y = &mut ...;
*x = 42; // (optimization: move down ?)
*y = 19; // is this a foreign write ? if yes

*x = 57;
```



Swap write-write

✓ Base model

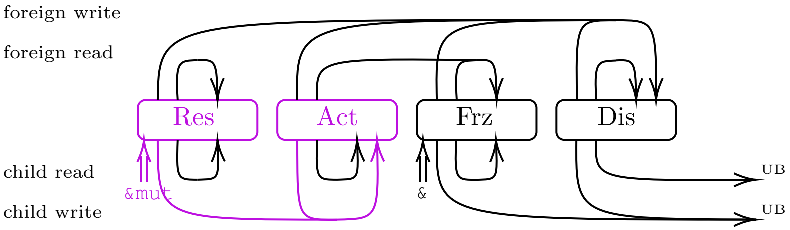
```
let x = &mut ...;
let y = &mut ...;
*x = 42; // (optimization: move down ?)
*y = 19; // is this a foreign write ? if not

*x = 57;
```



Swap write-write

✓ Base model

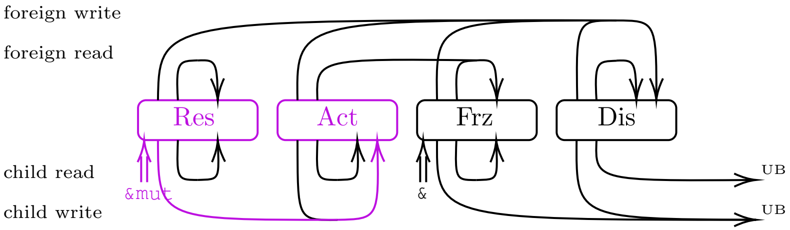
```
let x = &mut ...;
let y = &mut ...;

*y = 19; // assumed not to be a foreign write
*x = 42;
*x = 57;
```



Swap write-write

✓ Base model

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

*y = 19; // assumed not to be a foreign write

*x = 57;
```



Insert speculative read

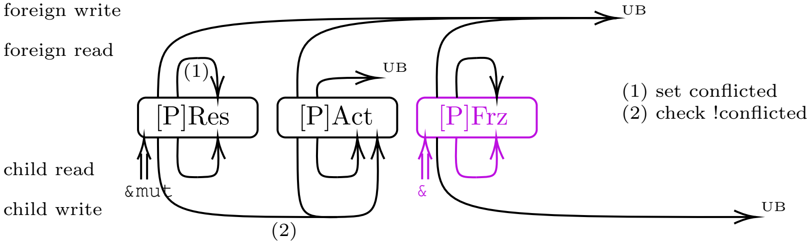
✓ Base model

```
fn read(x: &u64) -> u64 {

    opaque(/* contains foreign access ?             */);
    *x // (optimization: move up ?)
}
```

Insert speculative read

✓ Base model

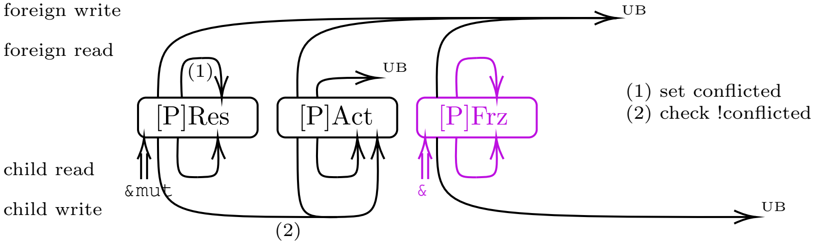
```
fn read(x: &u64) -> u64 {

    opaque(/* contains foreign access ? if none  */);
    *x // (optimization: move up ?)
}
```



Insert speculative read

✓ Base model

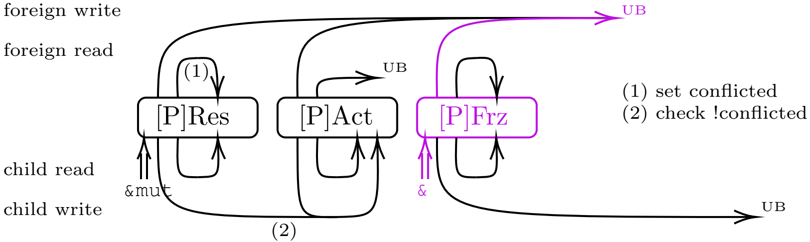
```
fn read(x: &u64) -> u64 {

    opaque(/* contains foreign access ? if read  */);
    *x // (optimization: move up ?)
}
```



Insert speculative read

✓ Base model

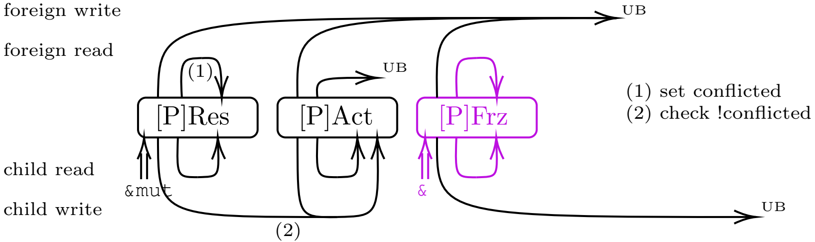
```
fn read(x: &u64) -> u64 {

    opaque(/* contains foreign access ? if write */);
    *x // (optimization: move up ?)
}
```



Insert speculative read

✓ Base model

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



Insert speculative write

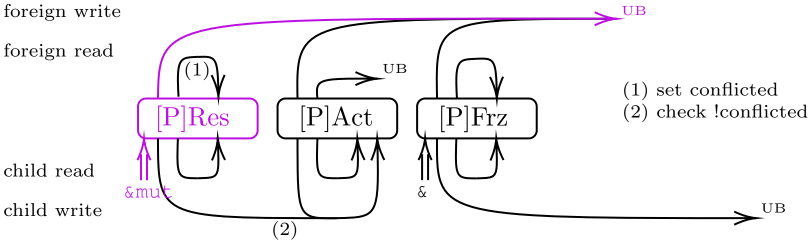
✗ Base model

```
fn foo(x: &mut u64) {

    opaque(/* contains foreign access ?                    */);
    *x = 42; // (optimization: move up ?)
}
```

Insert speculative write

✗ Base model

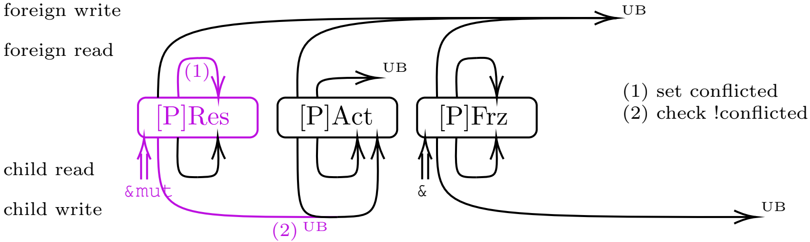
```
fn foo(x: &mut u64) {

    opaque(/* contains foreign access ? if write      */);
    *x = 42; // (optimization: move up ?)
}
```



Insert speculative write

✗ Base model

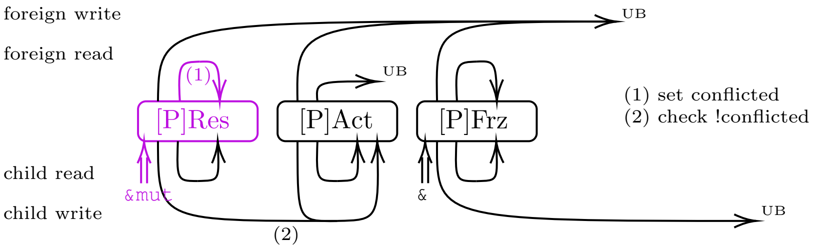
```
fn foo(x: &mut u64) {

    opaque(/* contains foreign access ? if read        */);
    *x = 42; // (optimization: move up ?)
}
```



Insert speculative write

✗ Base model

```
fn foo(x: &mut u64) {

    opaque(/* contains foreign access ? if read+loop */);
    *x = 42; // (optimization: move up ?)
}
```



Insert speculative write: blocker

as_mut_ptr: base model

- &mut [T] -> *mut T
- returns a Reserved child of the input

✓ Common pattern

```
let raw = buf.as_mut_ptr();
let shr = buf.as_ptr().add(1);
copy_nonoverlapping(shr, raw, 1);
```

```
buf: Active
  └─ ...
       └─ raw:  Reserved
```

Insert speculative write: blocker

as_mut_ptr: base model

- `&mut [T] -> *mut T`
- returns a `Reserved` child of the input

✓ Common pattern

```
let raw = buf.as_mut_ptr();
let shr = buf.as_ptr().add(1);
copy_nonoverlapping(shr, raw, 1);
```

```
buf: Active
 ├─ ...
 │   └─ raw:  Reserved
 └─ ...
     └─ shr:  Frozen
```

Insert speculative write: blocker

as_mut_ptr: base model

- `&mut [T] -> *mut T`
- returns a Reserved child of the input

✓ Common pattern

```
let raw = buf.as_mut_ptr();
let shr = buf.as_ptr().add(1);
copy_nonoverlapping(shr, raw, 1);
```

```
buf: Active
 ├─ ...
 │   └─ raw:  Active
 └─ ...
     └─ shr:  Disabled
```

Insert speculative write: blocker

as_mut_ptr: strengthened with speculative writes

- `&mut [T] -> *mut T`
- returns an `Active` child of the input

✗ Common pattern

```
let raw = buf.as_mut_ptr();
let shr = buf.as_ptr().add(1);
copy_nonoverlapping(shr, raw, 1);
```

```
buf: Active
  └─ ...
       └─ raw: Active
```

Insert speculative write: blocker

`as_mut_ptr`: strengthened with speculative writes

- `&mut [T] -> *mut T`
- returns an `Active` child of the input

✗ Common pattern

```
let raw = buf.as_mut_ptr();
let shr = buf.as_ptr().add(1);
copy_nonoverlapping(shr, raw, 1);
```

```
buf: Active
 ├─ ...
 │   └─ raw:  Frozen
 └─ ...
     └─ shr:  Frozen
```

A more formal approach

One big invariant:

- must be preserved by any step of the program that does not cause UB
- must provide sufficient hypotheses for the optimizations we want

A more formal approach

An excerpt from the invariant for protected activated mutable references:

$$\begin{aligned}\forall t \in T, l \in L. \exists p, c.(p, t, c) \in \sigma_t.TREES(l). \\ & p \neq \text{Disabled} \Rightarrow \\ & \sigma_s.\text{MEM}(l) \sim \sigma_t.\text{MEM}(l) \\ & \wedge p = \text{Active} \\ & \wedge \forall t' \in \text{Children}_{\sigma_t}(t) \ . \ t'.\text{PERM}[l] = \text{Disabled} \\ & \wedge \forall t' \in \text{Parents}_{\sigma_t}(t) \ . \ t'.\text{PERM}[l] = \text{Active} \\ & \wedge \forall t' \in \text{Uncles}_{\sigma_t}(t) \ . \ t'.\text{PERM}[l] \in \{\text{Disabled, Res InMut}\}\end{aligned}$$

What happens to this property when we do ...

...a foreign read ? a foreign write ? a reborrow ?

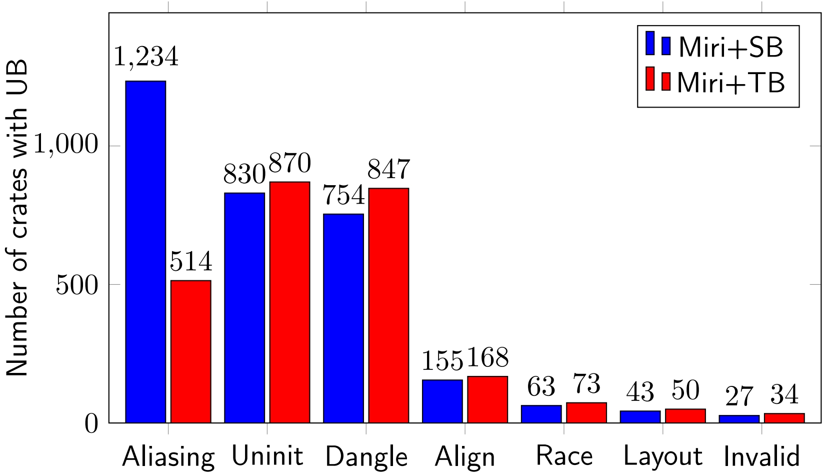
What does this allow ? $\rightarrow$ arbitrary child accesses

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

Data obtained with the help of Ben Kimock using github:saethlin/miri-tools



Summary

- Tree Borrows UB is much less common on `crates.io` than Stacked Borrows UB

  $\Rightarrow$ fulfills goal of being more permissive

Notable examples

`tokio`, `pyo3`, `rkyv`, `eyre`, `ndarray`, `arrayvec`, `slotmap`, `nalgebra`, `json`.

- patterns allowed by Stacked Borrows but forbidden by Tree Borrows are theoretically possible but have not been found in actual code

Reception

TB has existed for one year

- ⊕ consensus that TB is simpler and more predictable than SB,
- ⊕ cases where TB is more permissive than SB are welcome (e.g. `&Header` pattern),
- ⊕ cases where TB is less permissive are rare (no complaints yet),
- ⊖ fewer optimizations (expected),
- ⊖ controversial granularity of interior mutability,
- ⊖ slight performance regression in Miri.

Questions ?

TB also has...

- interactions between interior mutability and protectors/`Reserved`
- performance improvements
  - tricks to trim tree traversals
  - lazy initialization for out-of-range accesses
- formalization in Coq in progress

Don't hesitate to test your code with Miri and send us your interesting/unexpected cases of UB! (`github:rust-lang/miri`)

`MIRIFLAGS=-Zmiri-tree-borrows cargo +miri miri test`

Slides: `github:Vanille-N/tree-beamer/tree/lmf`

Complementary material: `perso.crans.org/vanille/treebor`