# 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...

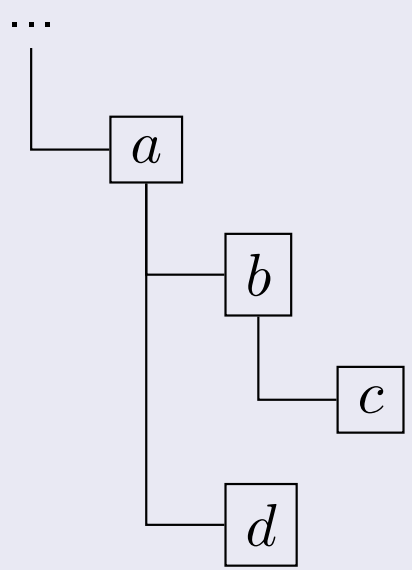
- more optimizations (stronger assumptions)
- less safety (especially if rules are vague)

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

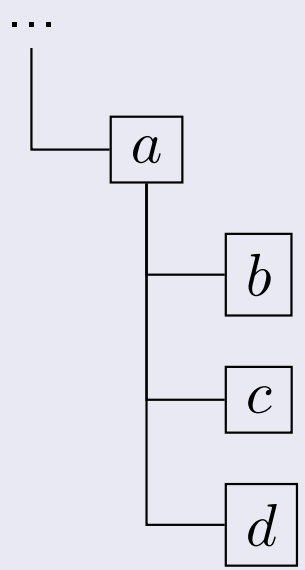
# Information loss in the stack

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

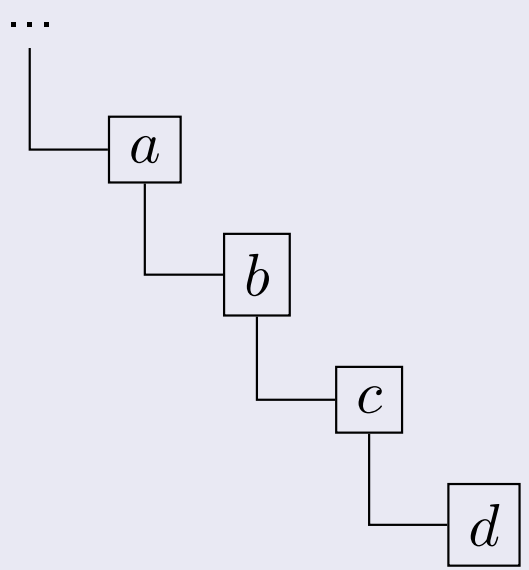
```
let a = &...;
let b = &*a;
let c = &*b;
let d = &*a;
```




```
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 ?

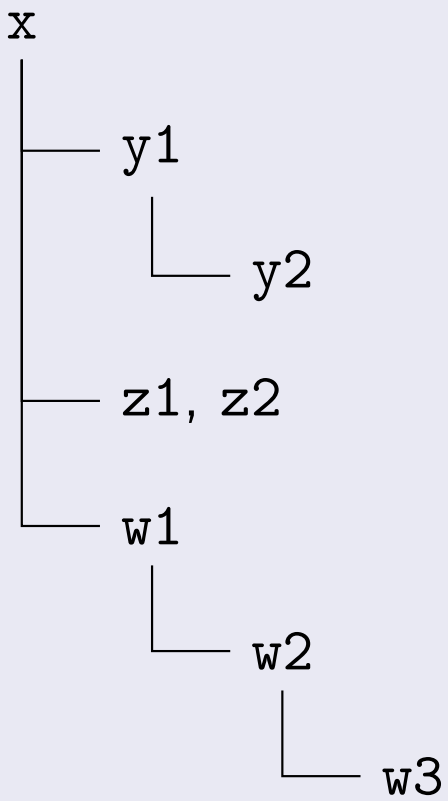
LLVM and Rust specifications: "other references/pointers"
Suggests that two pointers to the same data are "different".

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;
}
```

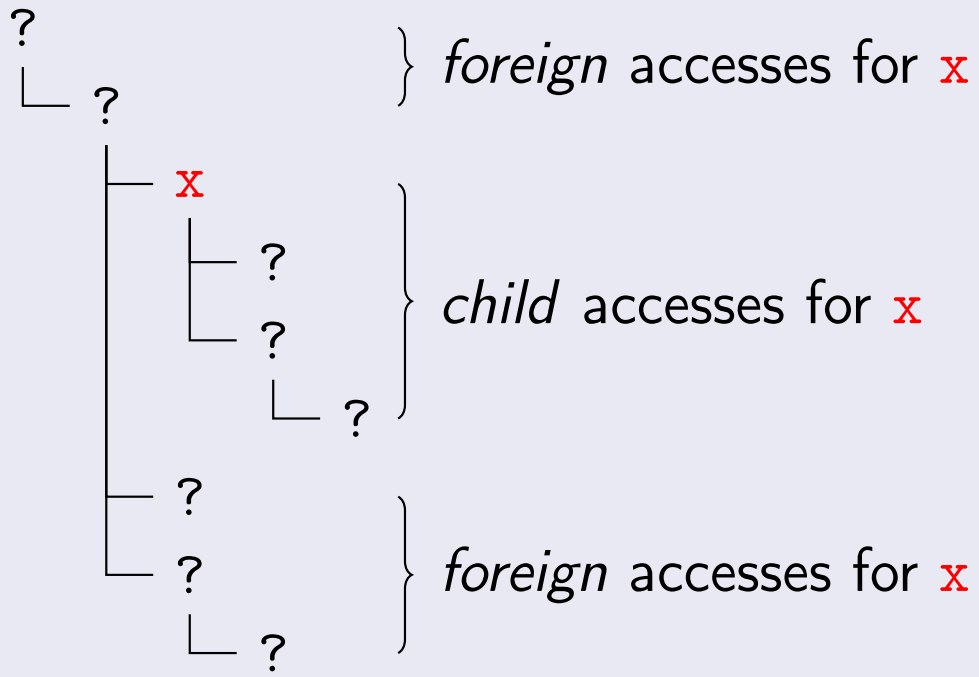
# What's in the tree ?

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, $2 \times 2$ kinds of accesses



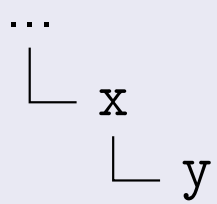

each *read* or *write*.

## "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 is invalidated by foreign accesses
- ...

# Active, Frozen, Disabled

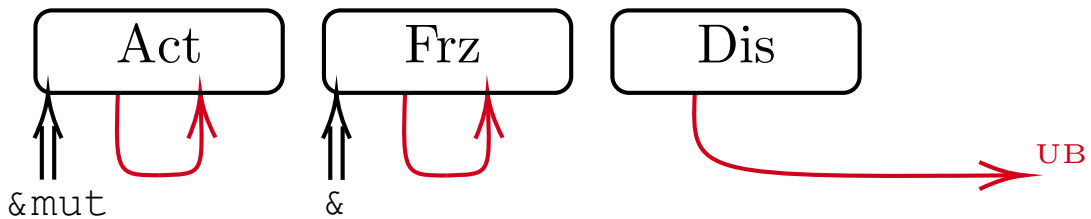
Core triplet of permissions to represent

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

## Child read : must allow reading

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

child read



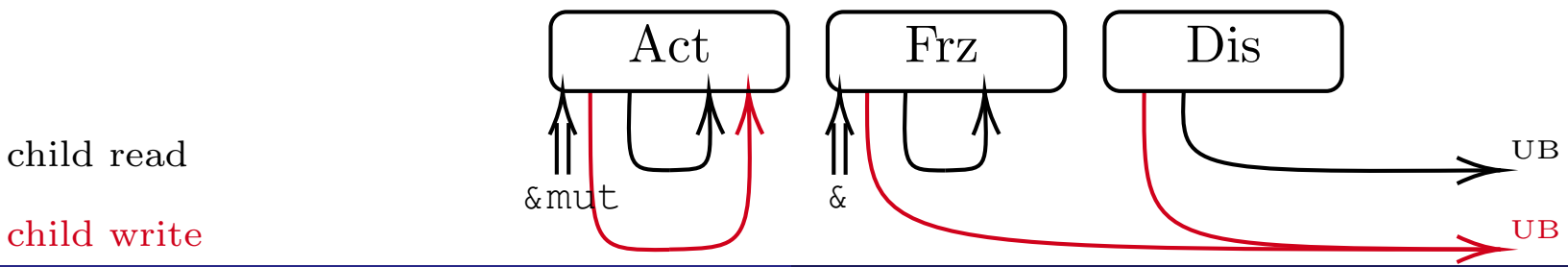
# Active, Frozen, Disabled

Core triplet of permissions to represent

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

## Child write: must allow writing

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



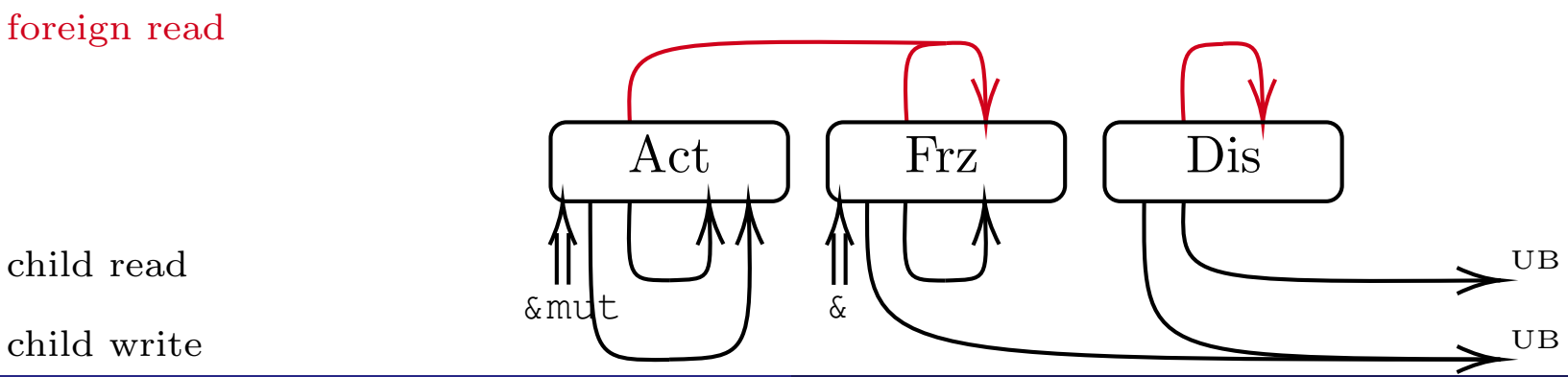
# Active, Frozen, Disabled

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`



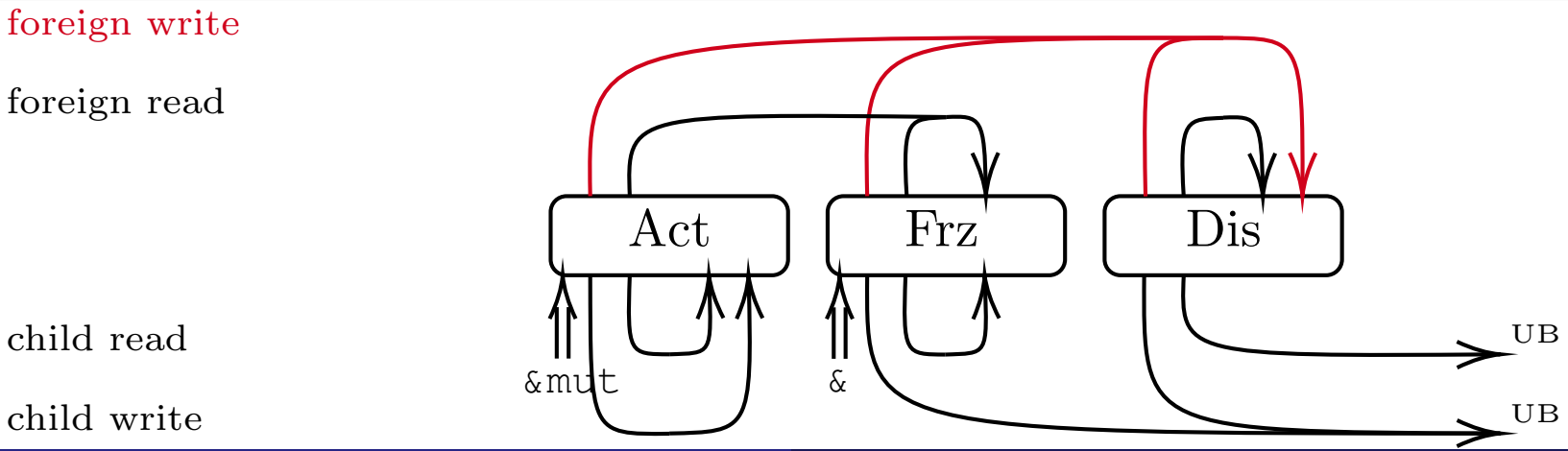
# Active, Frozen, Disabled

Core triplet of permissions to represent

- unique mutable references: `Active`,
- shared immutable references: `Frozen`,
- lifetime ended: `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 (following subsections fix them)

- ✗ two-phase borrows not handled yet
- ✗ too permissive for `noalias` and `dereferenceable`
- ✗ `UnsafeCell` needs special handling

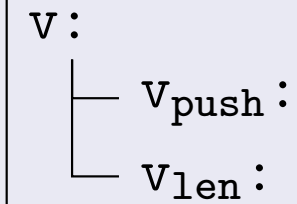
# Not all mutable references can be `Active`

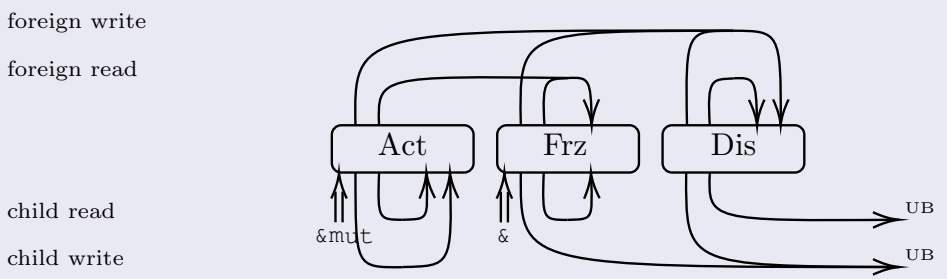
## Two-phase borrows

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()
     );
 }
```

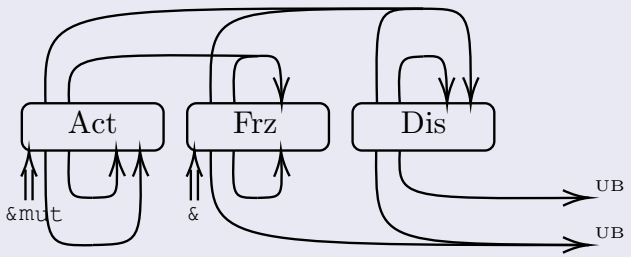
```
v: Active
├─ vpush:
└─ vlen:
```

foreign write

foreign read

child read

child write



# 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` ← reborrow

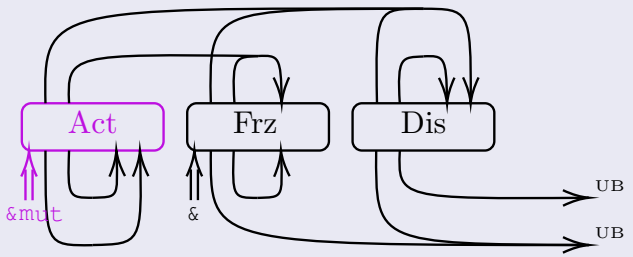
- `v_push: Active`
- `v_len:`

foreign write

foreign read

child read

child write



# Not all mutable references can be `Active`

## Two-phase borrows

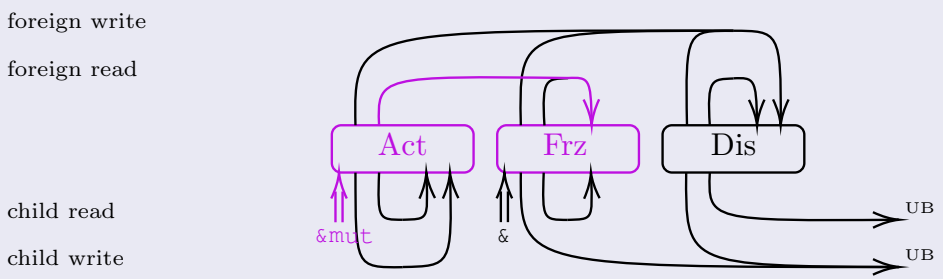
Mutable reborrows in function arguments tolerate shared reborrows until function entry.

## Core triplet: unsound

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

`v: Active` ← reborrow

- `v_push: Frozen`
- `v_len: Frozen` ← read



# Not all mutable references can be `Active`

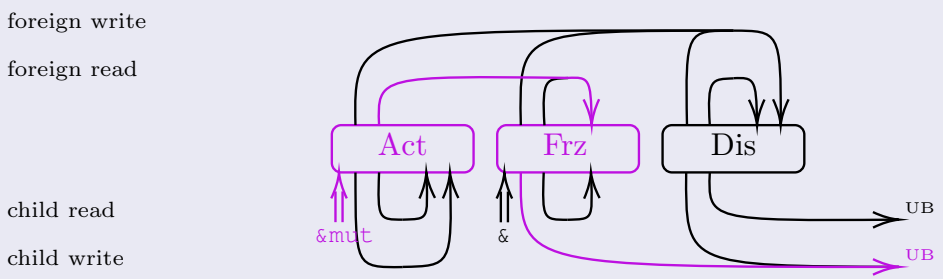
## Two-phase borrows

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: Frozen   ← write (UB)
└─ v_len: Frozen
```

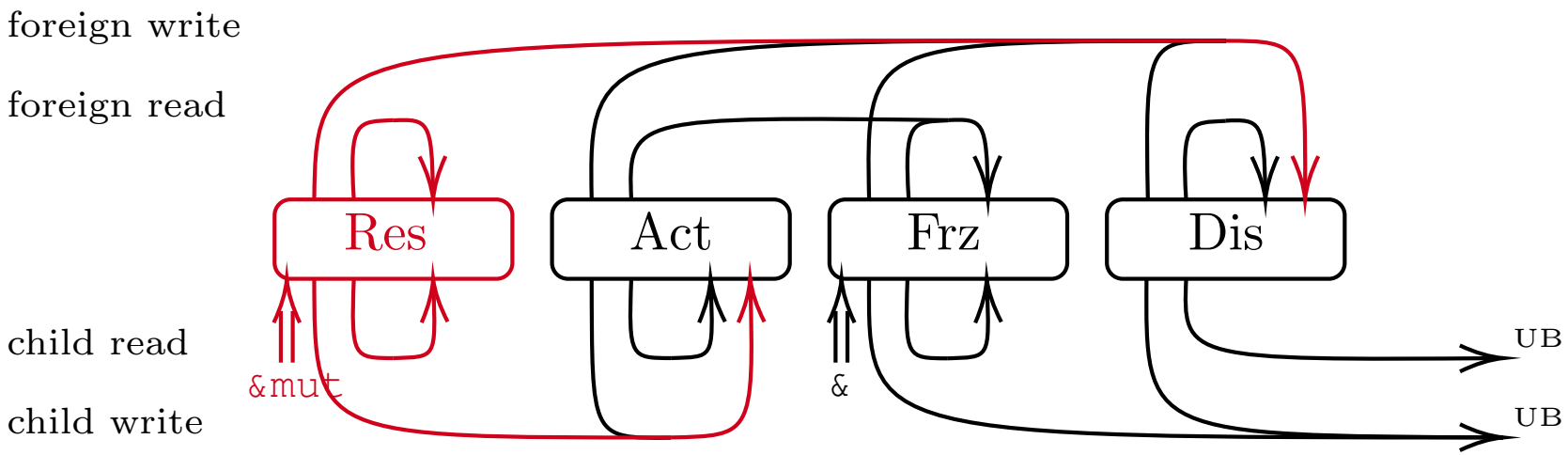


# New permission: `Reserved`

## Intuition

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

Mutable references not yet written to
⇒ behave like `Frozen` until the first child write
⇒ can coexist with each other and with `Frozen`

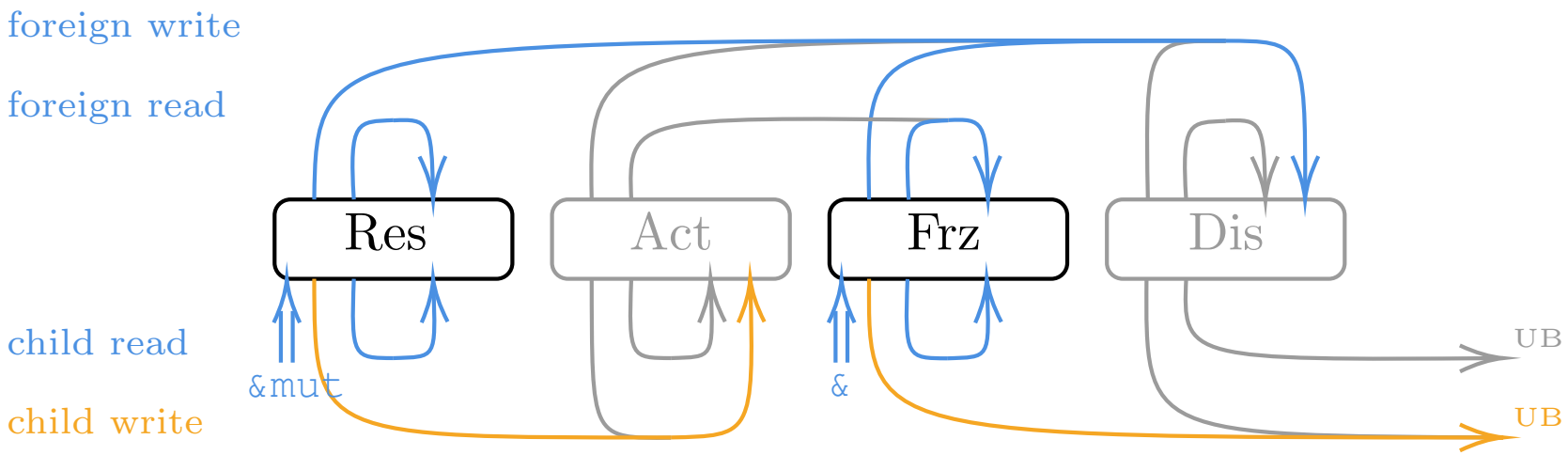


# New permission: `Reserved`

## Intuition

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

Mutable references not yet written to
⇒ behave like `Frozen` until the first child write
⇒ can coexist with each other and with `Frozen`



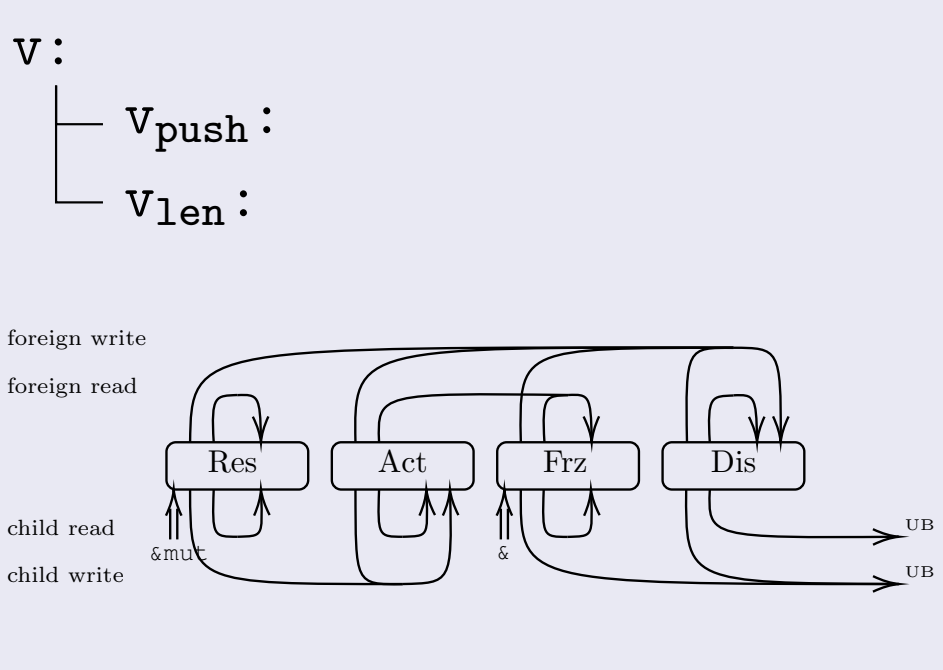
# Reserved in action

## Two-phase borrows

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

## Core triplet + Reserved: fixed

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



# Reserved in action

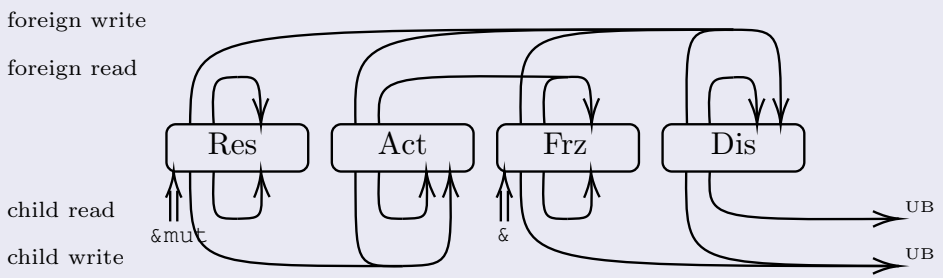
## Two-phase borrows

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()
     );
 }
```

```
v: Active
├─ vpush:
└─ vlen:
```



# Reserved in action

## Two-phase borrows

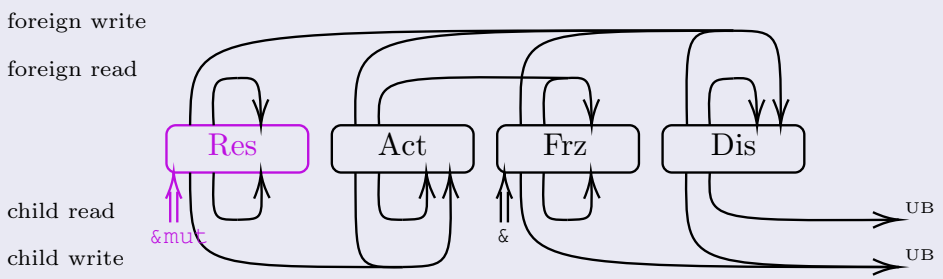
Mutable reborrows in function arguments tolerate shared reborrows until function entry.

## Core triplet + Reserved: fixed

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

v: Active ← reborrow

- $v_{push}$: Reserved
- $v_{len}$:



# Reserved in action

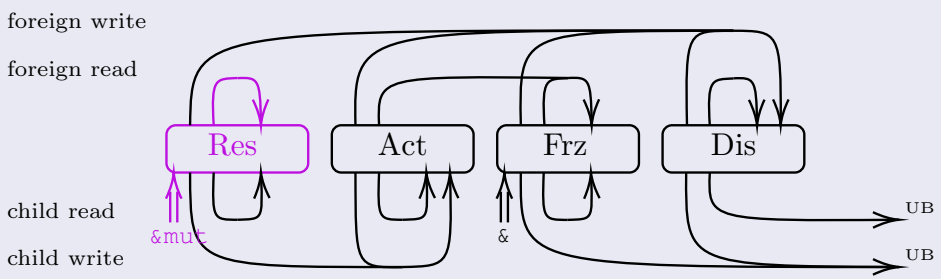
## Two-phase borrows

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()
     );
 }
```

```
v: Active                    ← reborrow
├─ v_push: Reserved
└─ v_len: Frozen             ← read
```



# Reserved in action

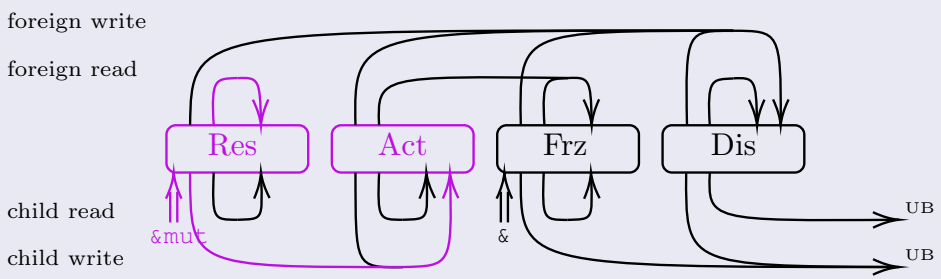
## Two-phase borrows

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()
>    );
 }
```

```
v: Active
├─ v_push: Active      ← write
└─ v_len: Disabled
```



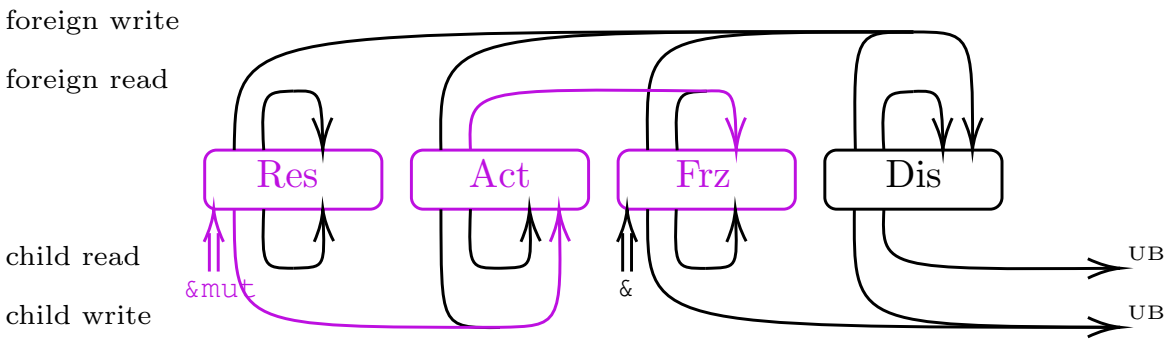
# Loss of permissions too early

## 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; // activation
    opaque(/* foreign read for x: noalias violation */);
}
```



# Protectors lock permissions

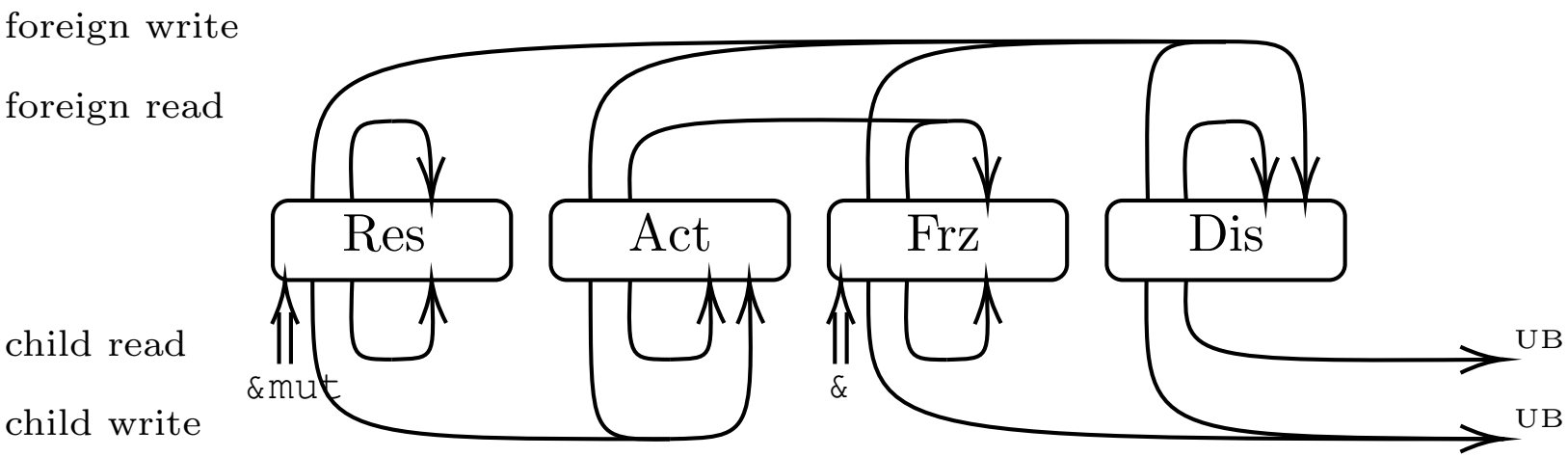
**Intuition**

`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.

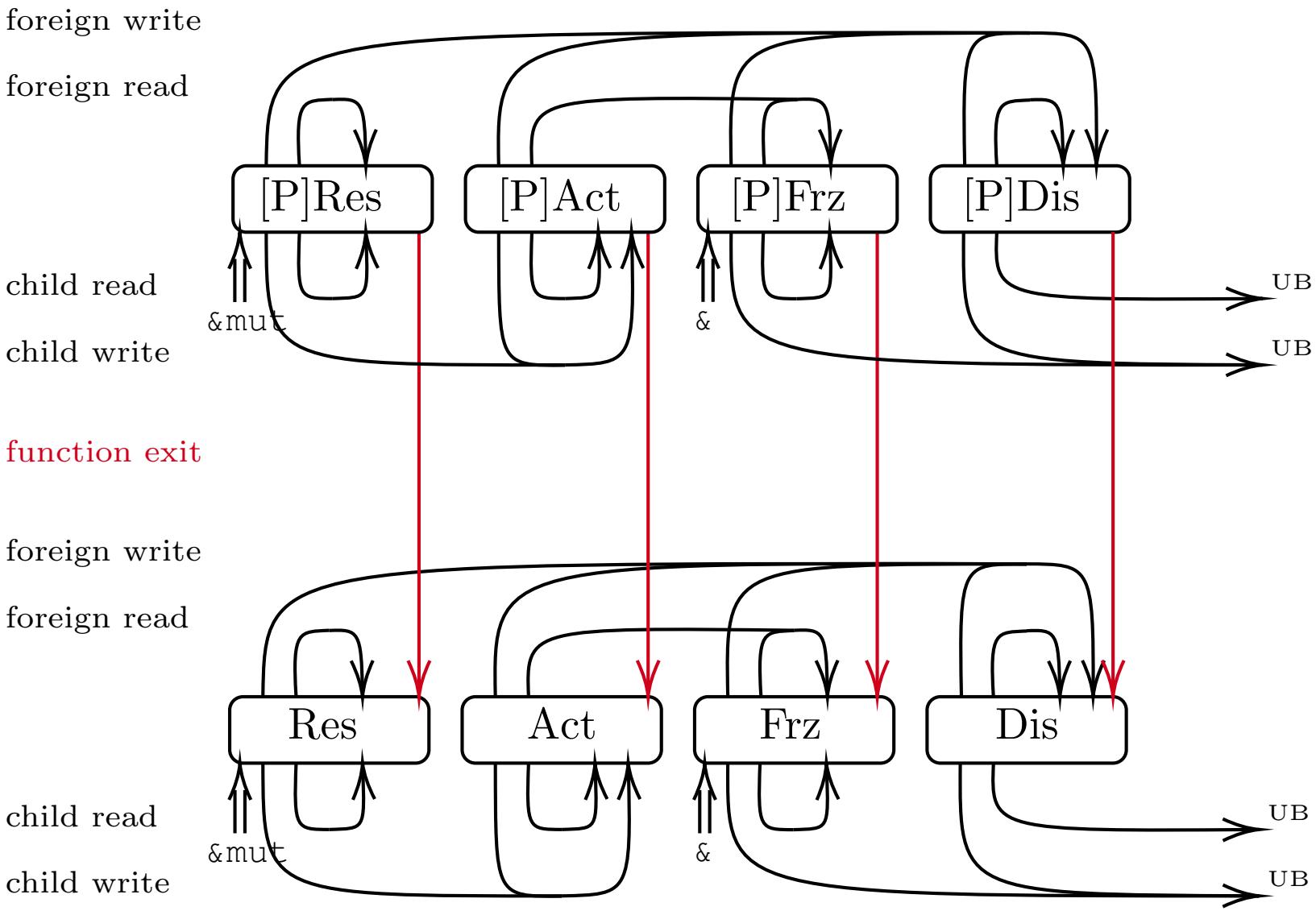
Concept adapted from Stacked Borrows: protectors.

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

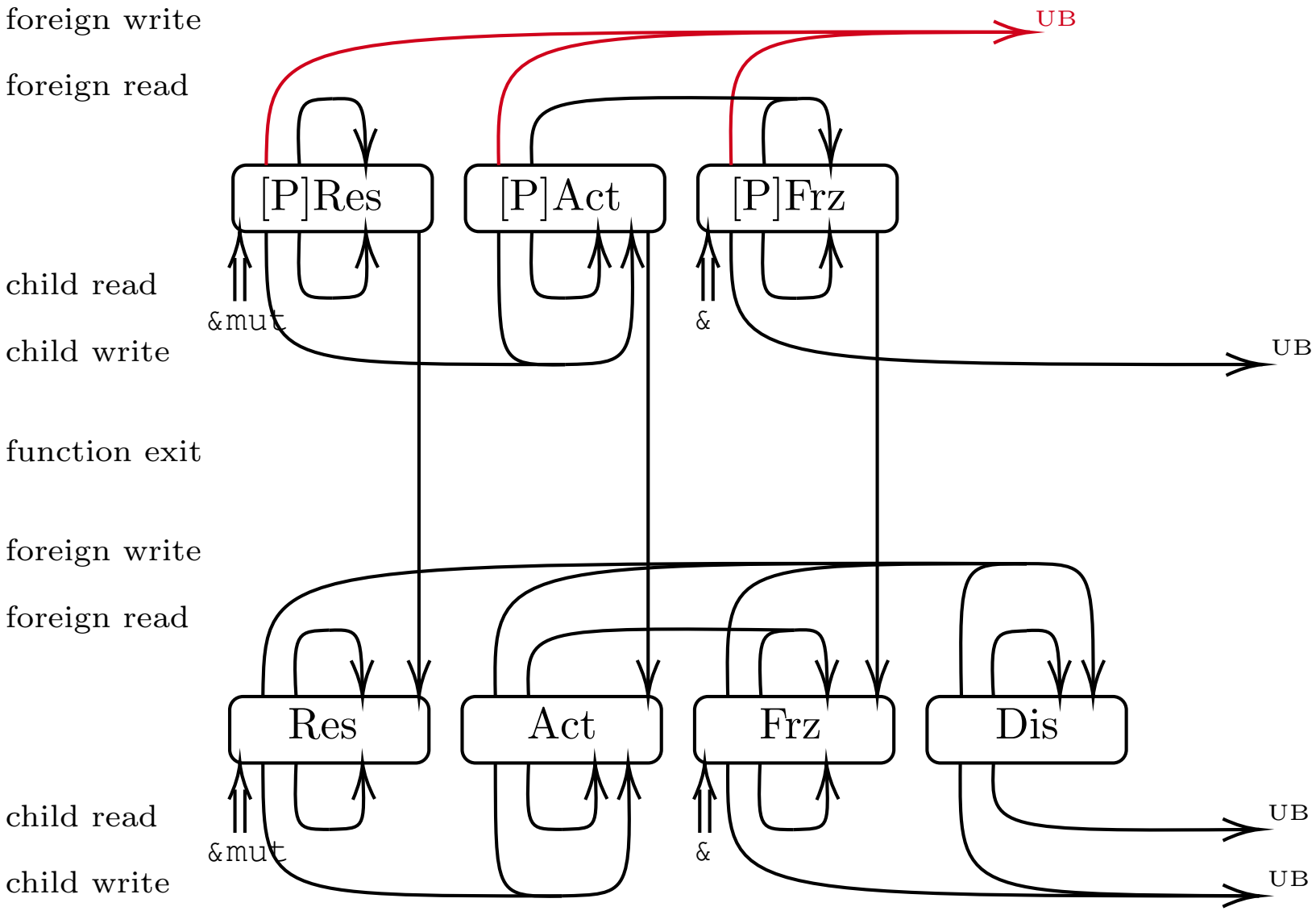
# Protectors lock permissions



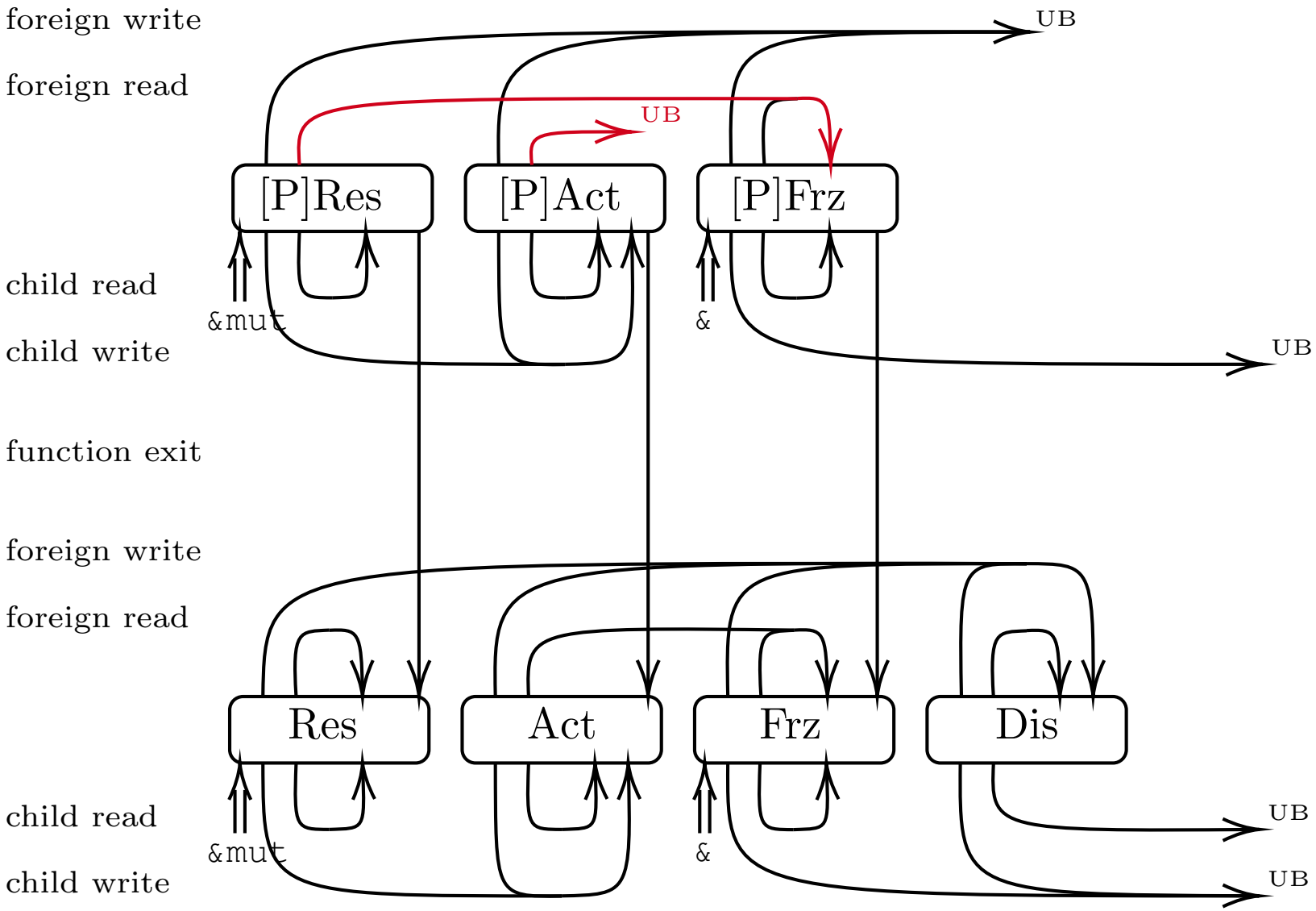
# Protectors lock permissions



# Protectors lock permissions



# Protectors lock permissions



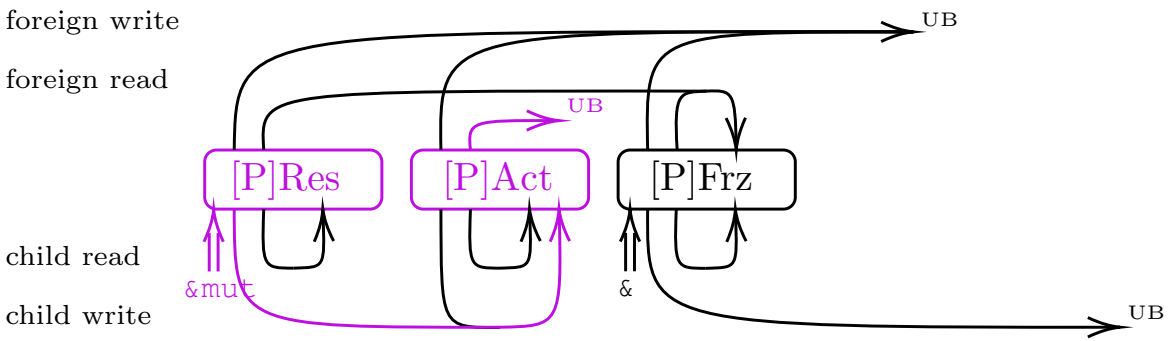
# Protectors lock permissions

## LLVM `noalias` (in TB terms)

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

## Core triplet + `Reserved` + 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-read'-read → read'-write-read | ✓ | ✓* |

all are possible for references without interior mutability
(`&mut` if a write is involved, both `&` and `&mut` if read-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) | ✓* | ✗ | ← SB only |
| ▷ Swap write-call-write → write-write-call | ✓* | ✓* | |
| Swap write-call → call-write | ✓* | ✓* | |
| ▷ Swap write-read'-read → read'-write-read | ✓ | ✓* | ← SB only |

all are possible for references without interior mutability
(`&mut` if a write is involved, both `&` and `&mut` if read-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-read'-read → read'-write-read | ✓ | ✓* |

← TB only

all are possible for references without interior mutability
(`&mut` if a write is involved, both `&` and `&mut` if read-only)
*: only for protected references

# Swap write-write

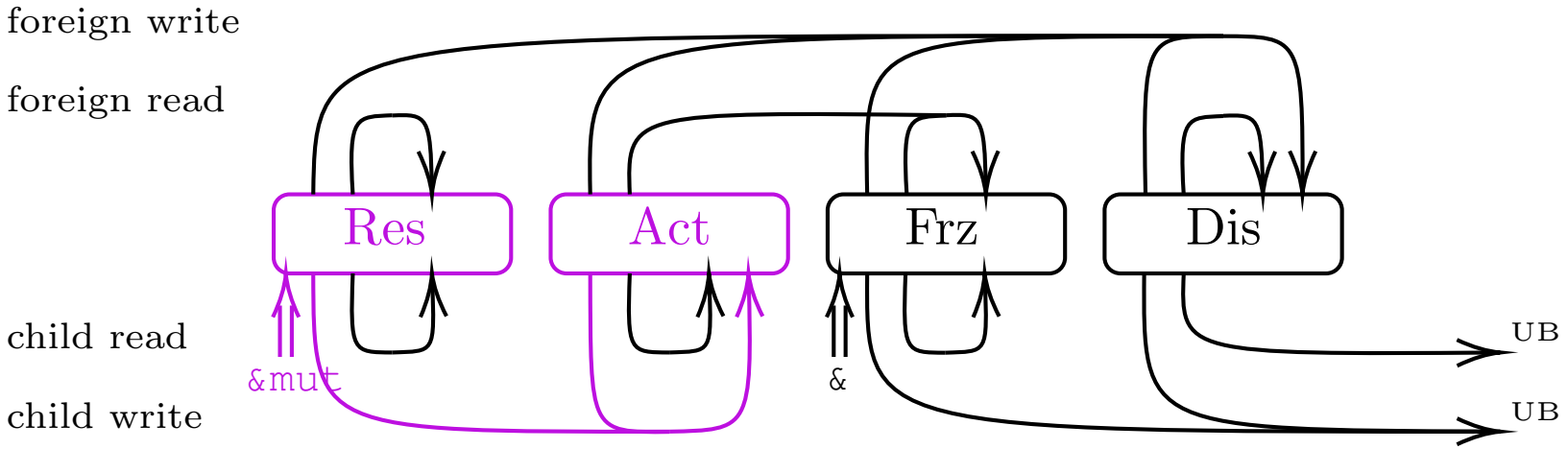
## ✓ Base model

```
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

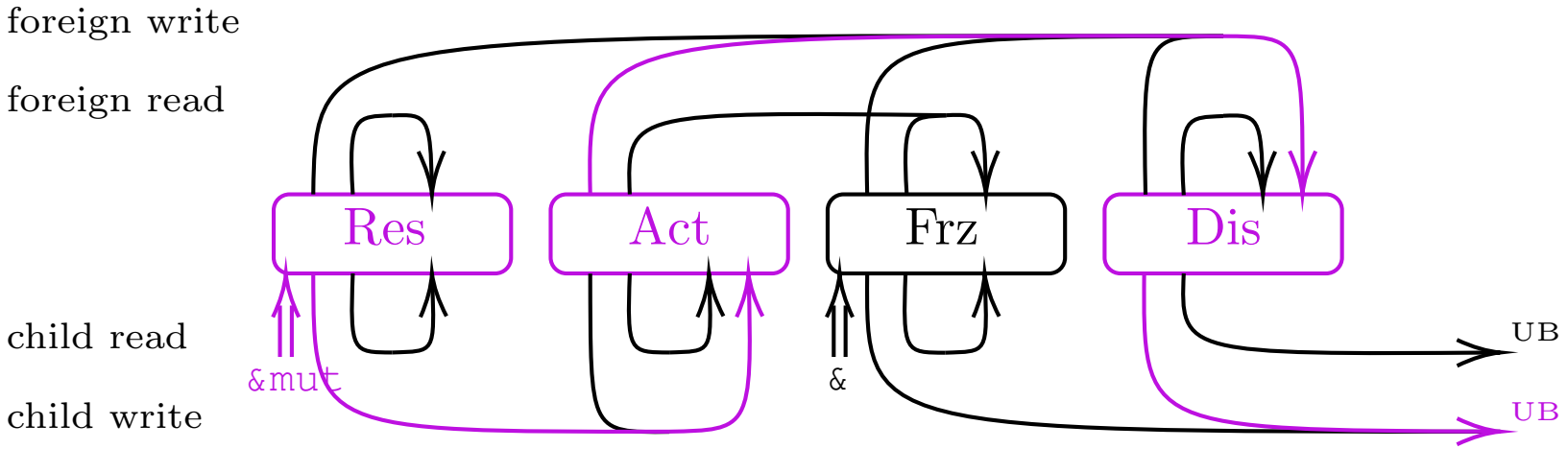
```
let x = &mut ...;
let y = &mut ...;
*x = 42; // (optimization: move down ?)
*y = 19; // is this a foreign write ? if not

*x = 57;
```



# Swap write-write

## ✓ Base model

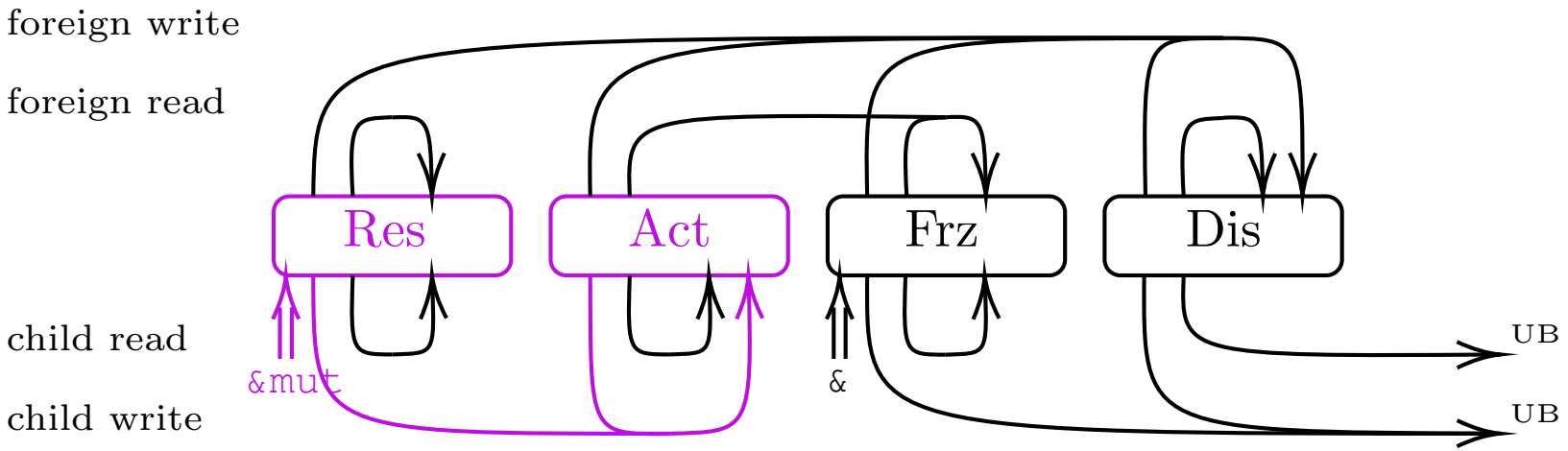
```
let x = &mut ...;
let y = &mut ...;
*x = 42; // (optimization: move down ?)
*y = 19; // is this a foreign write ? if yes

*x = 57;
```



# Swap write-write

## ✓ Base model

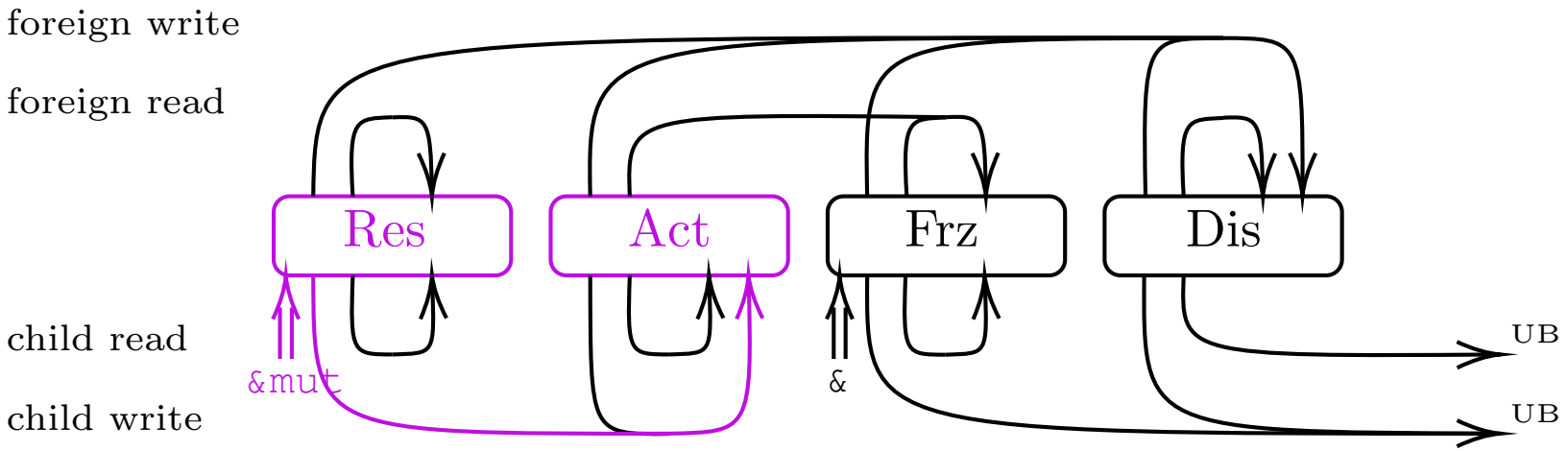
```
let x = &mut ...;
let y = &mut ...;

*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

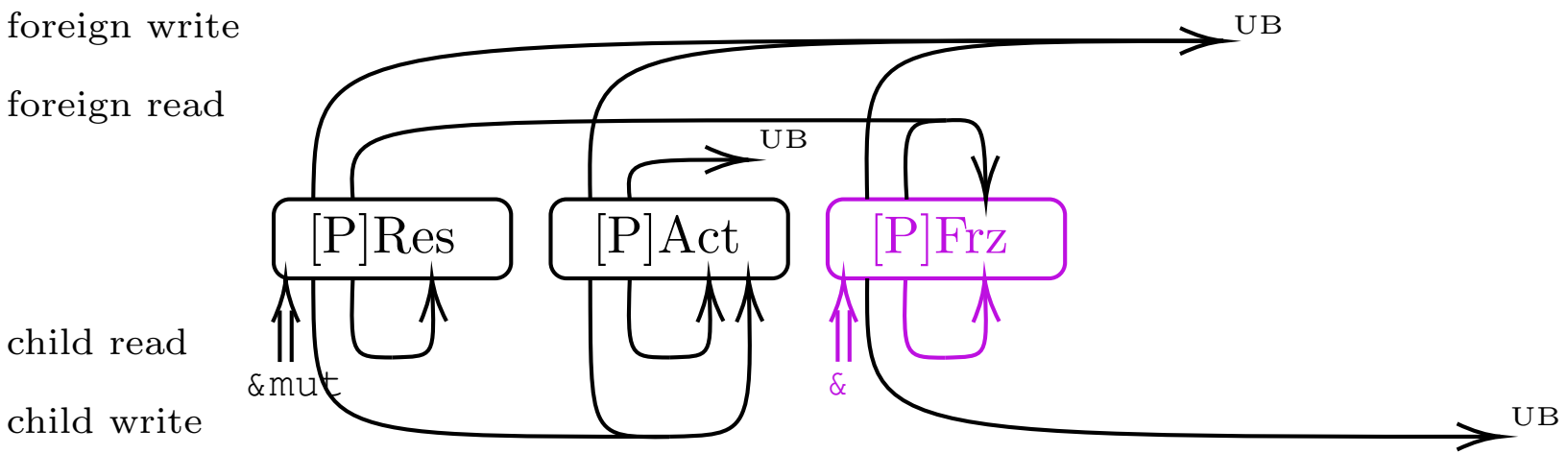
## ✓ Base model

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

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

# Insert speculative read

## ✓ Base model

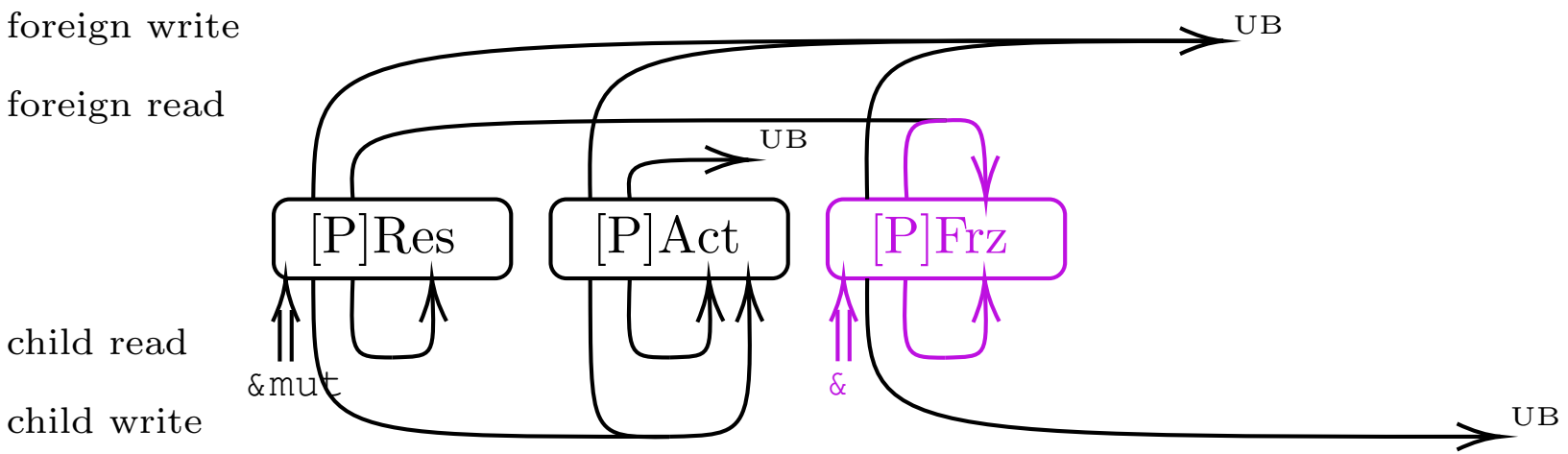
```
fn read(x: &u64) -> u64 {

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



# Insert speculative read

## ✓ Base model

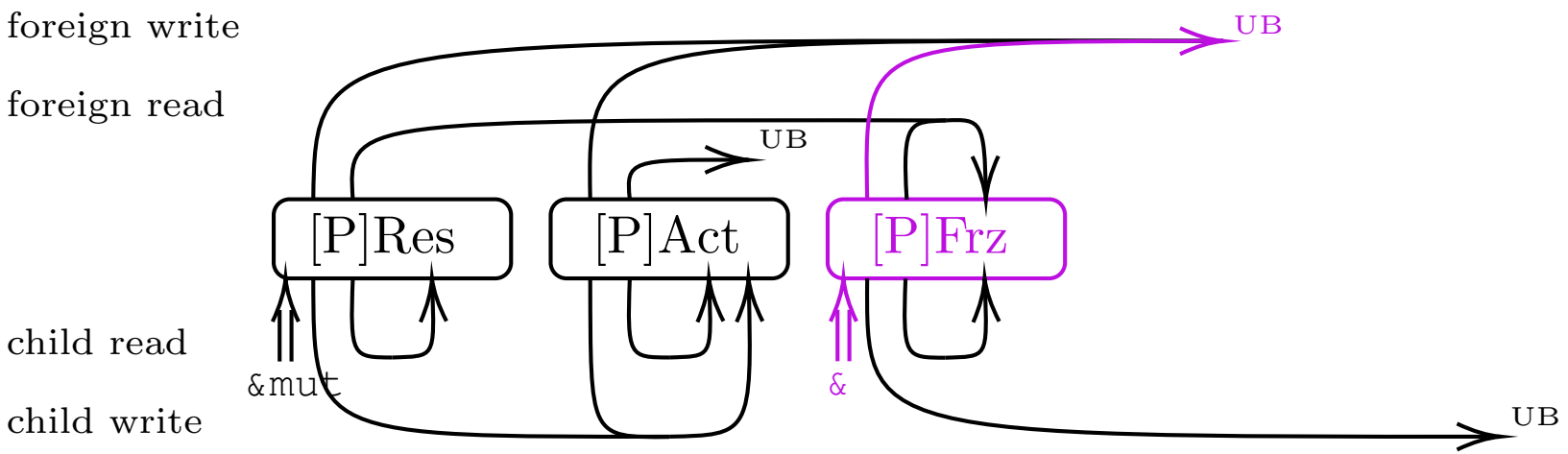
```
fn read(x: &u64) -> u64 {

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



# Insert speculative read

## ✓ Base model

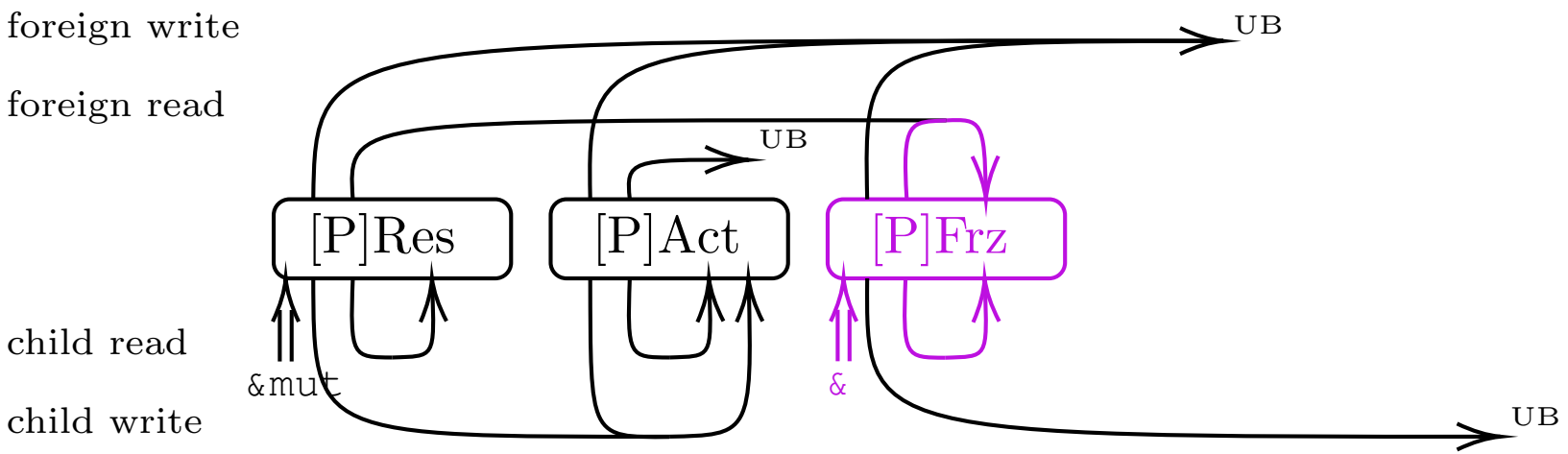
```
fn read(x: &u64) -> u64 {

    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

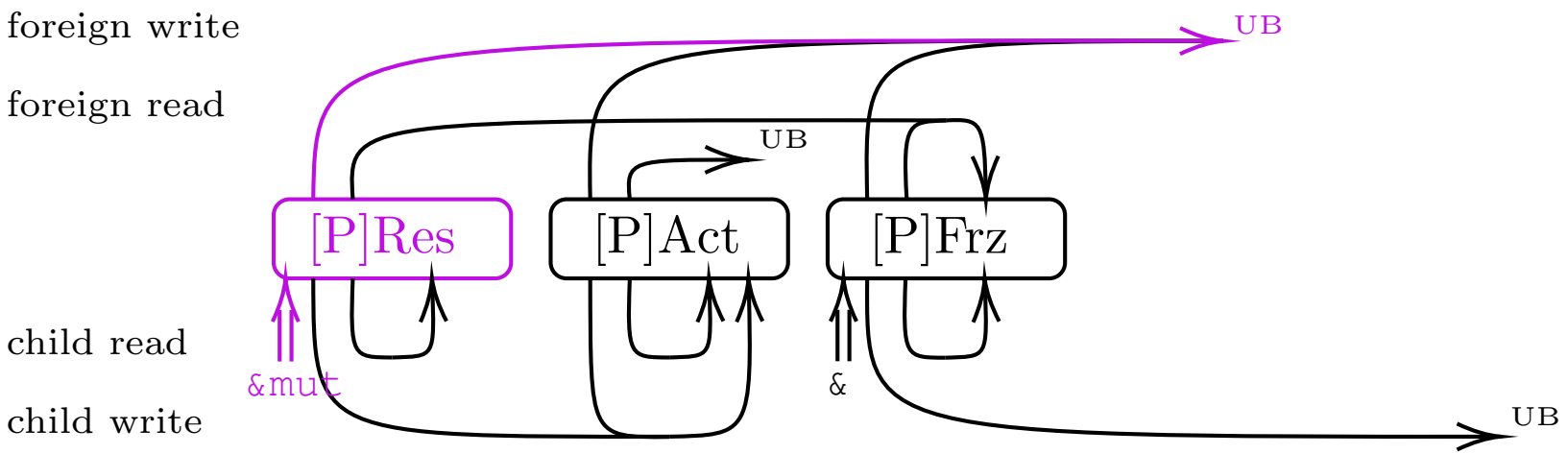
## ✗ Base model

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

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

# Insert speculative write

## ✗ Base model

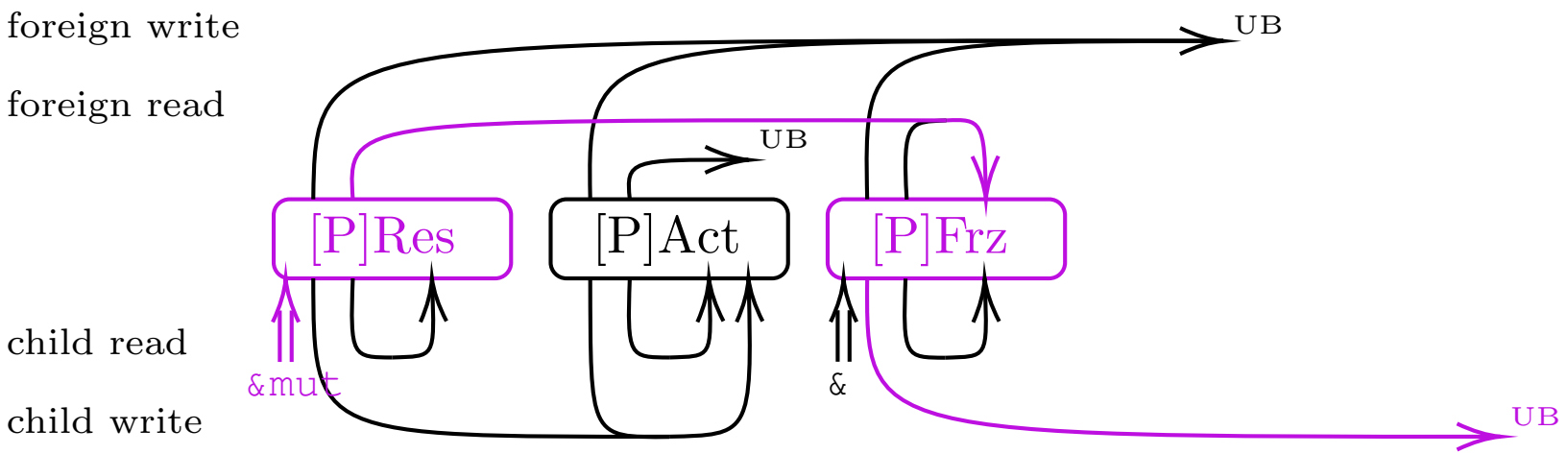
```
fn foo(x: &mut u64) {

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



# Insert speculative write

## ✗ Base model

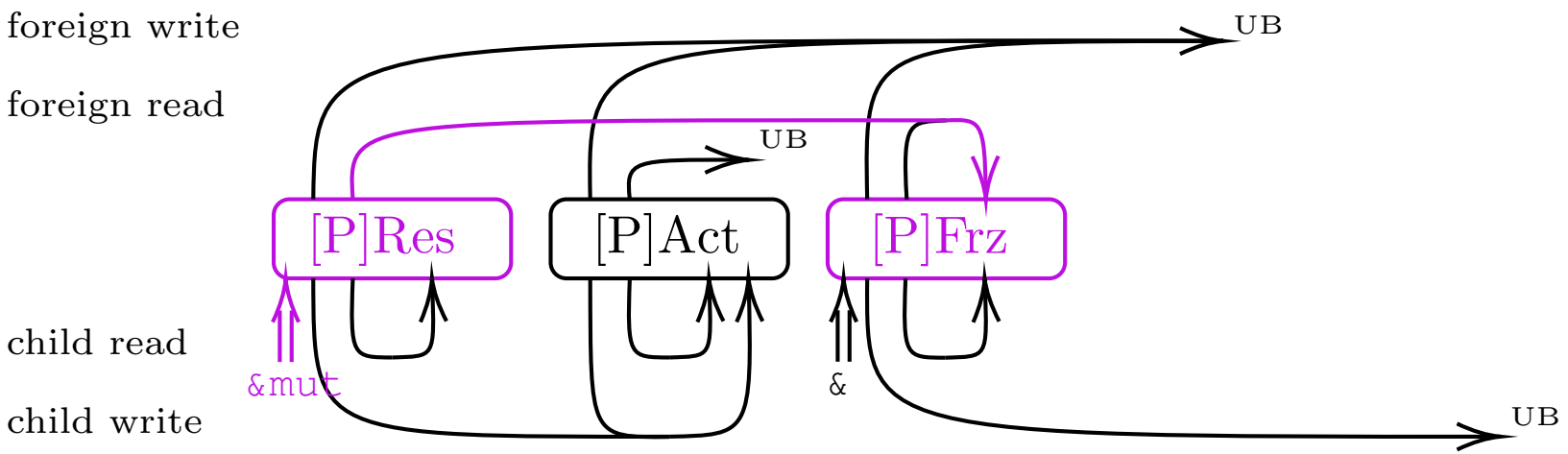
```
fn foo(x: &mut u64) {

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



# Insert speculative write

## ✗ Base model

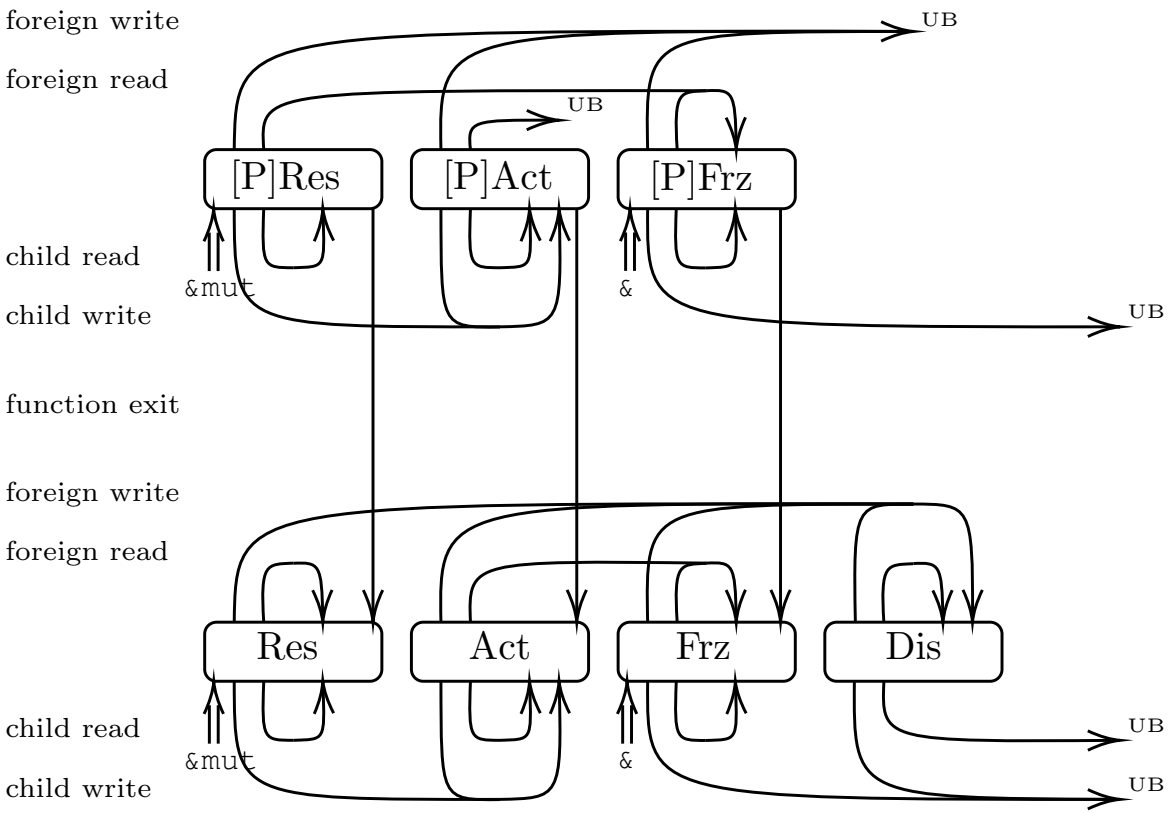
```
fn foo(x: &mut u64) {

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



# Insert speculative write: strengthening

## Possible strengthening

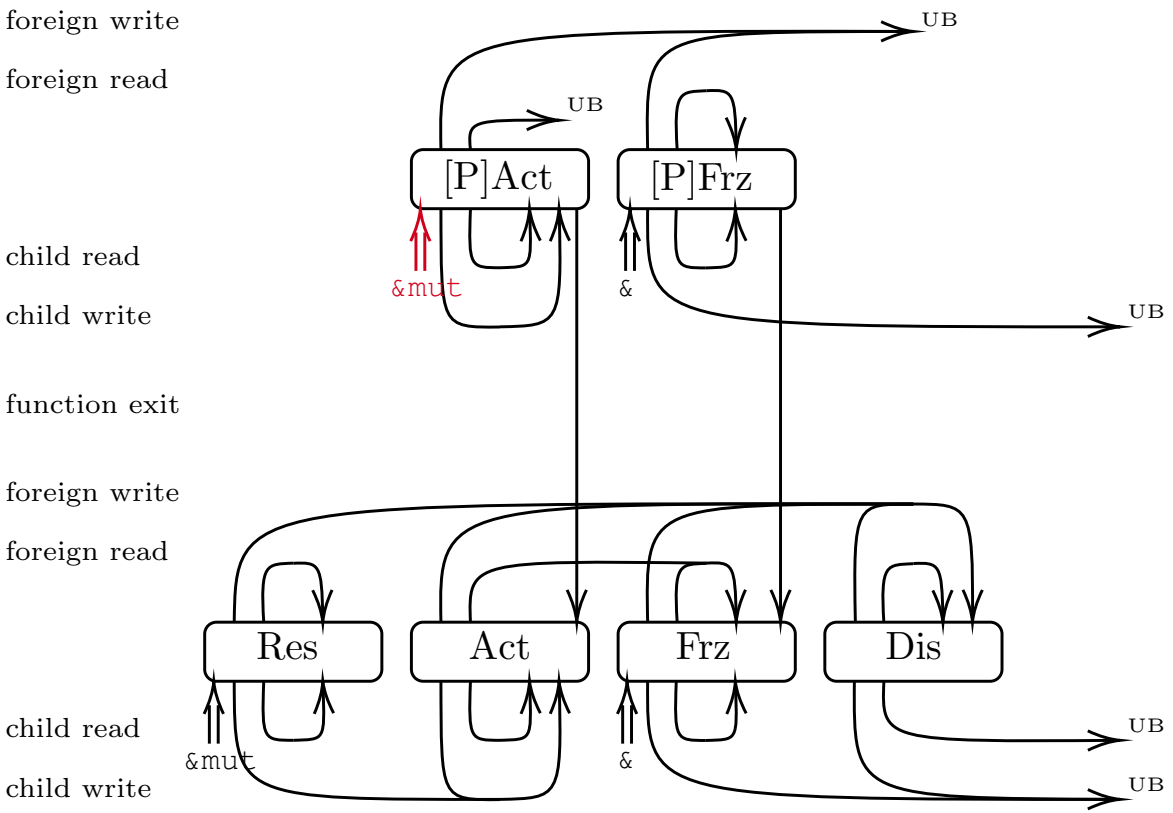
Write to mutable references of function entry



# Insert speculative write: strengthening

## Possible strengthening

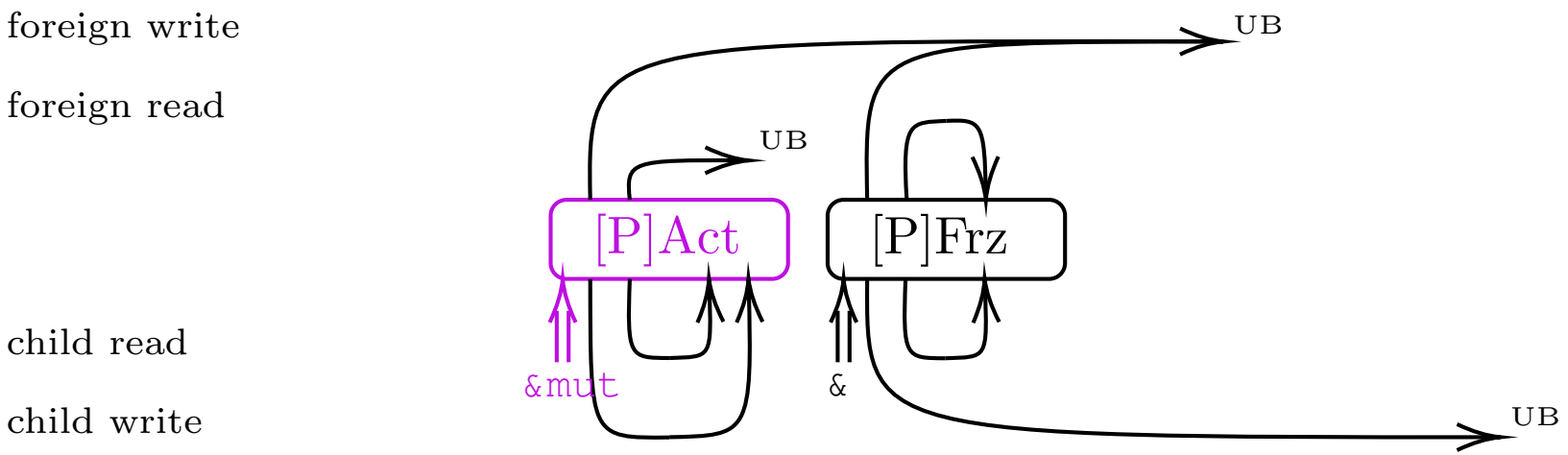
Write to mutable references of function entry



# Insert speculative write: strengthening

## ✓ Strengthened model

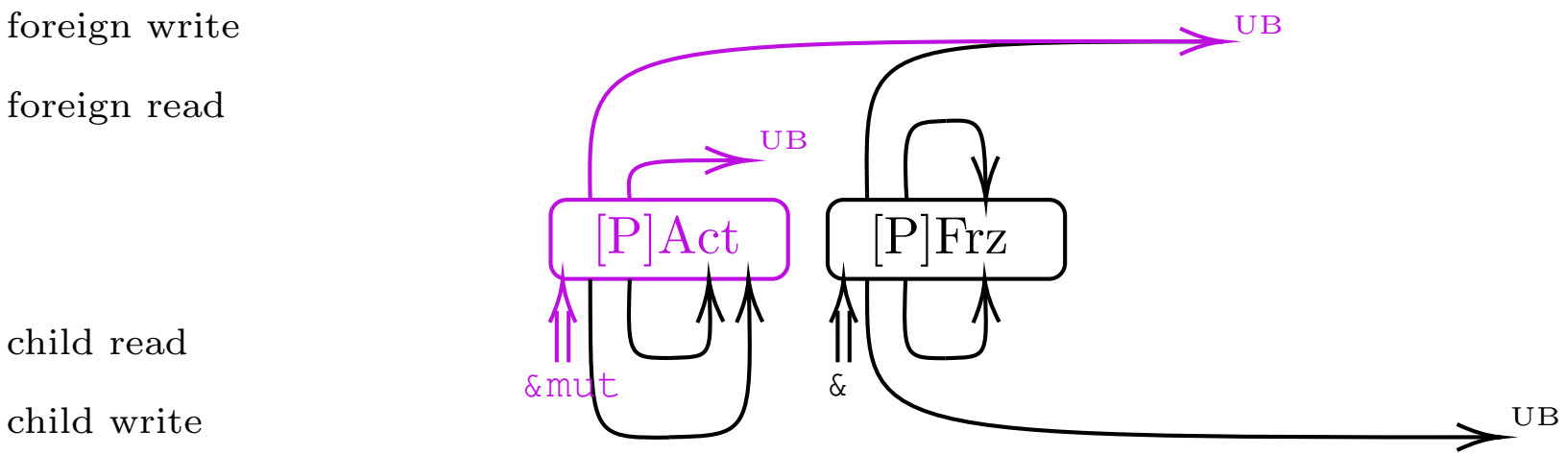
```
fn foo(x: &mut u64) {

    opaque(/* contains foreign access ? if none */);
    *x = 42;
}
```



# Insert speculative write: strengthening

## ✓ Strengthened model

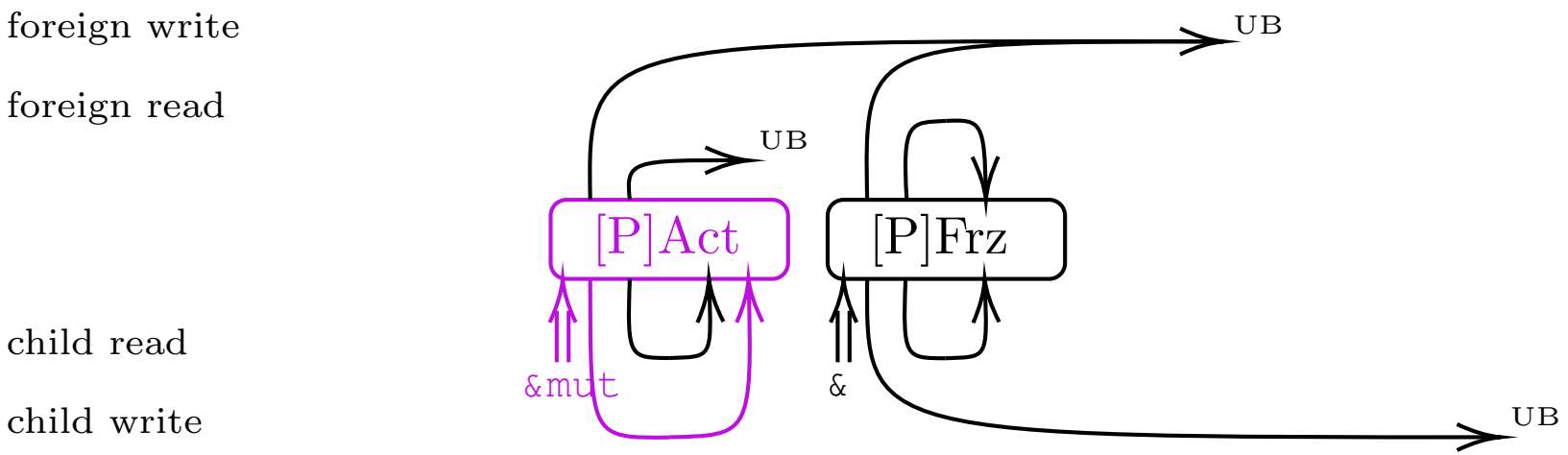
```
fn foo(x: &mut u64) {

    opaque(/* contains foreign access ? if any  */);
    *x = 42;
}
```



# Insert speculative write: strengthening

## ✓ Strengthened model

```
fn foo(x: &mut u64) {
    *x = 42;
    opaque(/* assume no foreign access */);

}
```



# 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

- `&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

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

## ✗ Common pattern

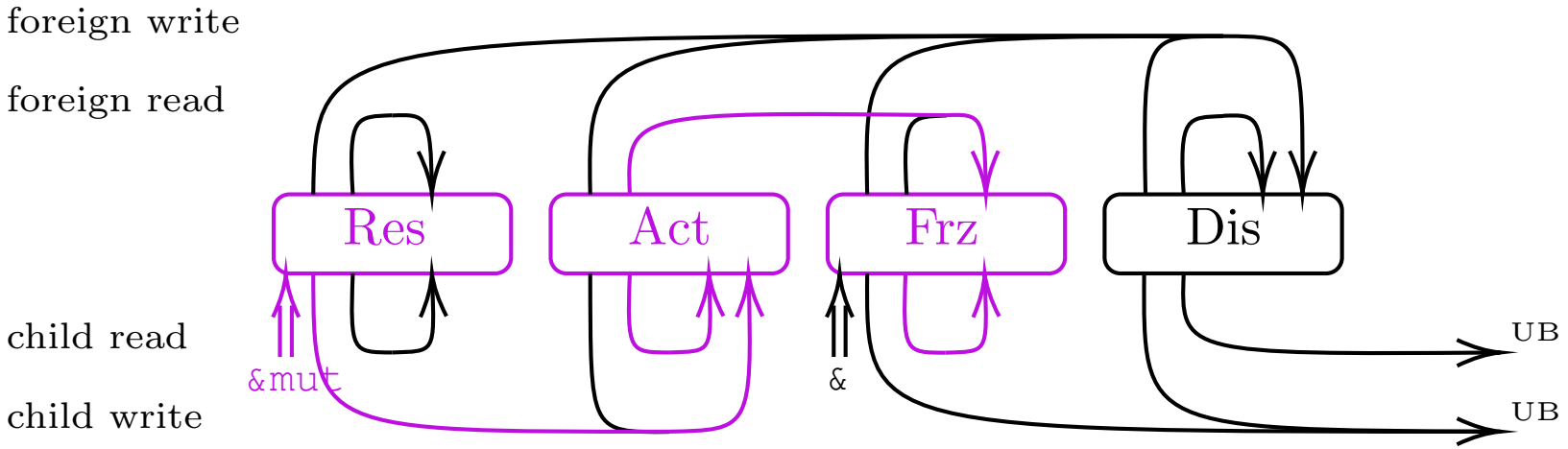
```
let raw = buf.as_mut_ptr();
let shr = buf.as_ptr().add(1);
copy_nonoverlapping(shr, raw, 1);
```

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

# Swap write-read'-read → read'-write-read

## ✗ Base model

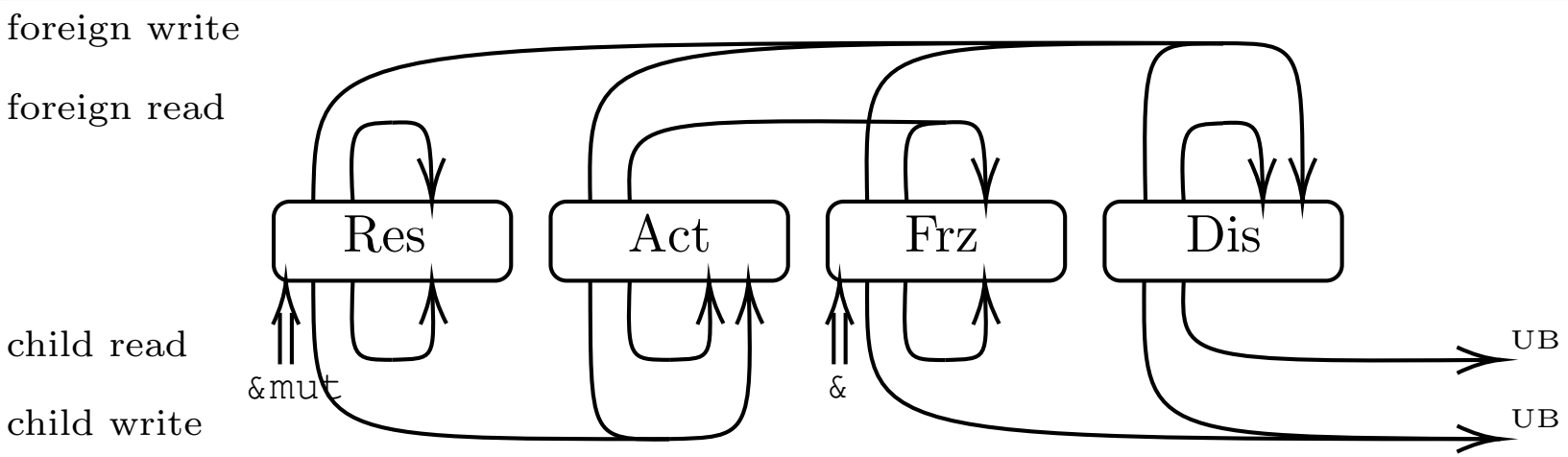
```
let x = &mut ...;
*x = 42; // (optimization: move down ?)
let y = &...;
let vy = *y; // is this read foreign ? maybe

let vx = *x;
```



# Swap write-read'-read → write-read-read': strengthening

## Possible strengthening

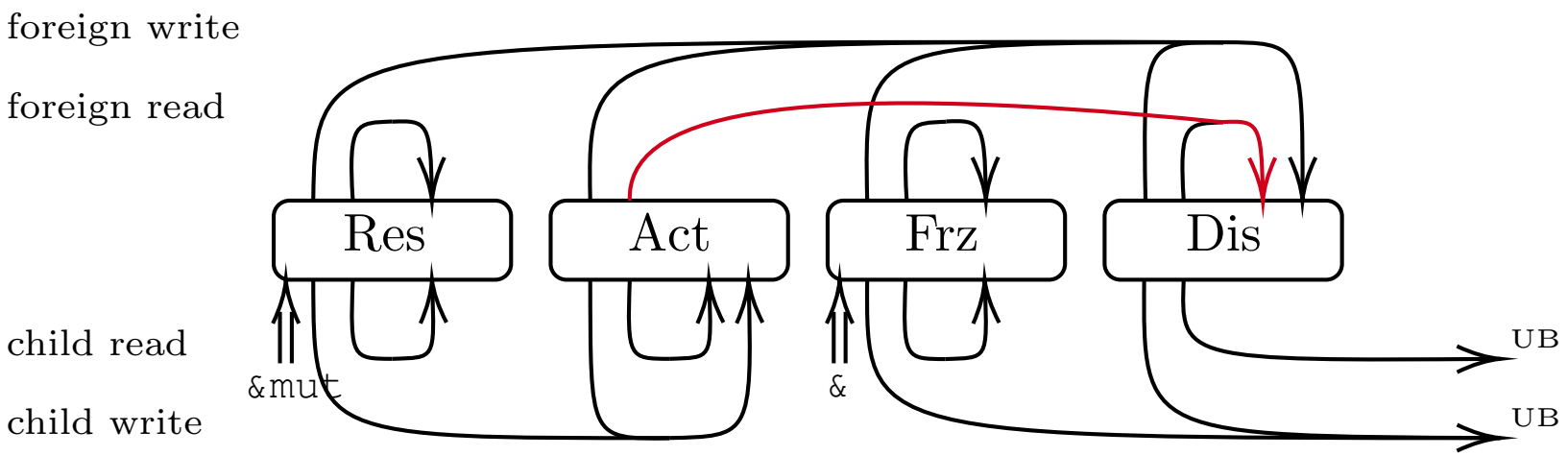
Foreign read makes `Active` become `Disabled` (rather than `Frozen`)



# Swap write-read'-read → write-read-read': strengthening

## Possible strengthening

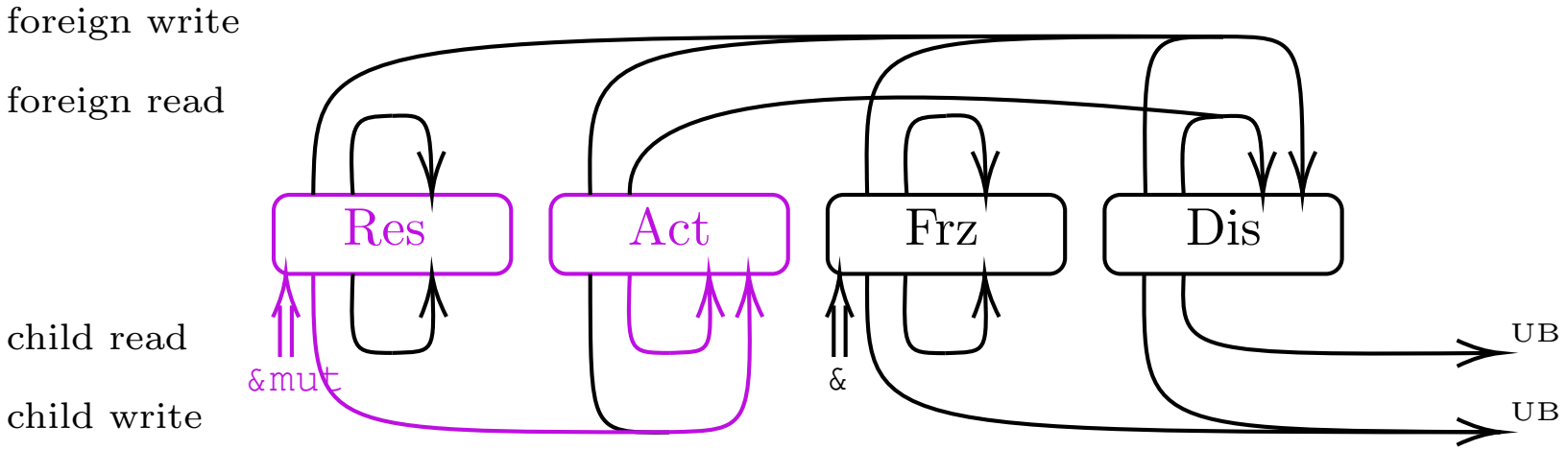
Foreign read makes `Active` become `Disabled` (rather than `Frozen`)



# Swap write-read'-read → write-read-read': strengthening

## ✓ Strengthened model

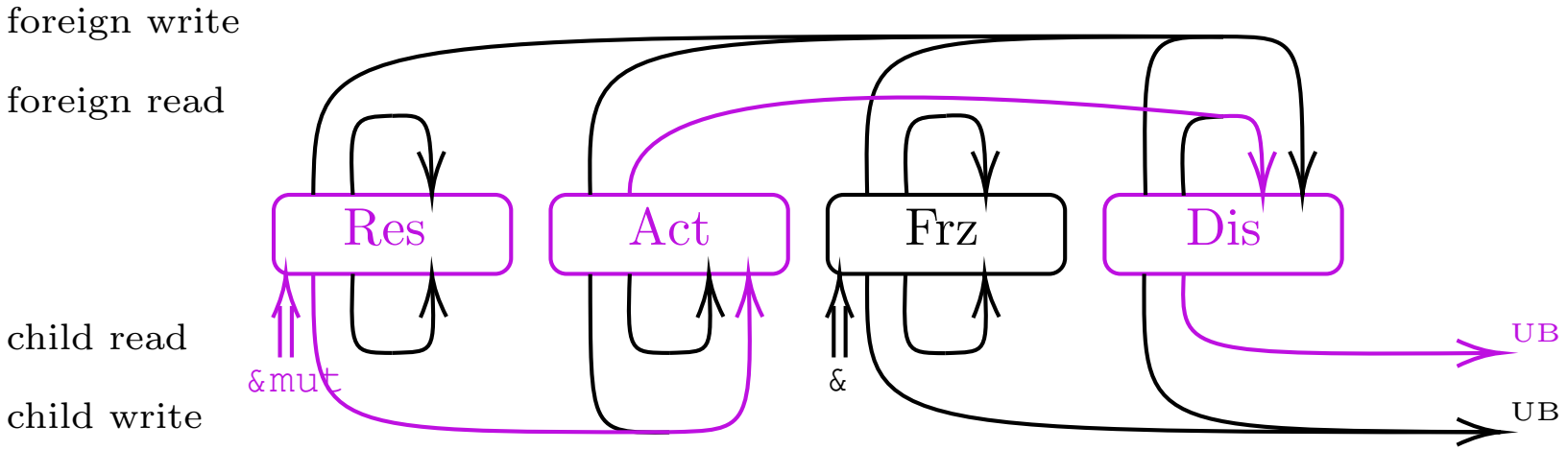
```
let x = &mut ...;
*x = 42; // (optimization: move down ?)
let y = &...;
let vy = *y; // is this read foreign ? if not

let vx = *x;
```



# Swap write-read'-read → write-read-read': strengthening

## ✓ Strengthened model

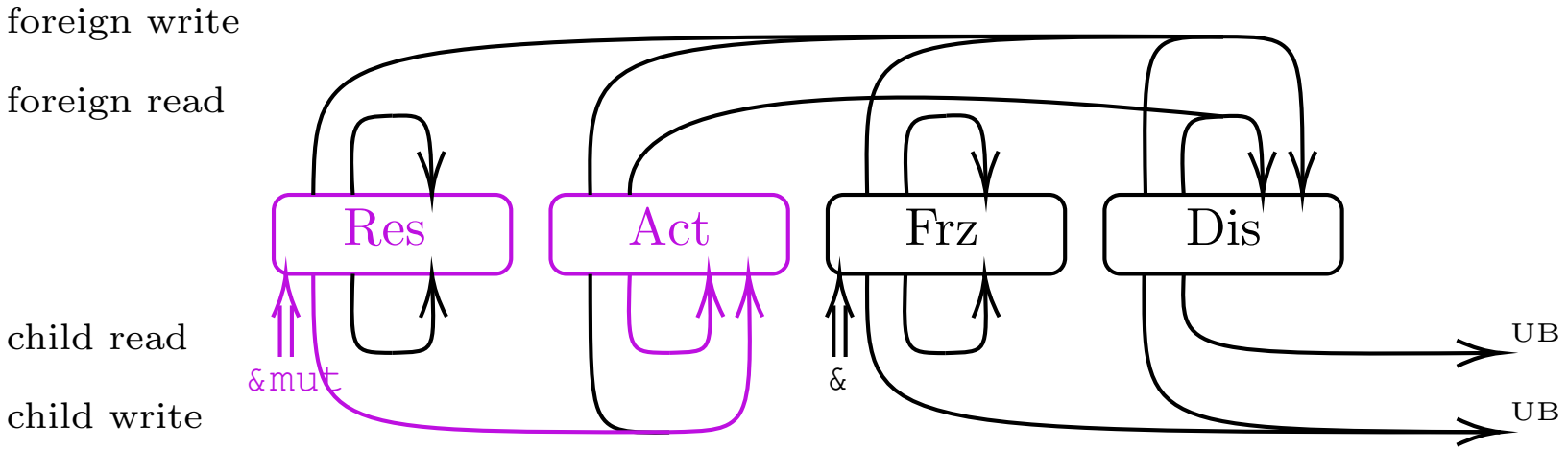
```
let x = &mut ...;
*x = 42; // (optimization: move down ?)
let y = &...;
let vy = *y; // is this read foreign ? if yes

let vx = *x;
```



# Swap write-read'-read → write-read-read': strengthening

## ✓ Strengthened model

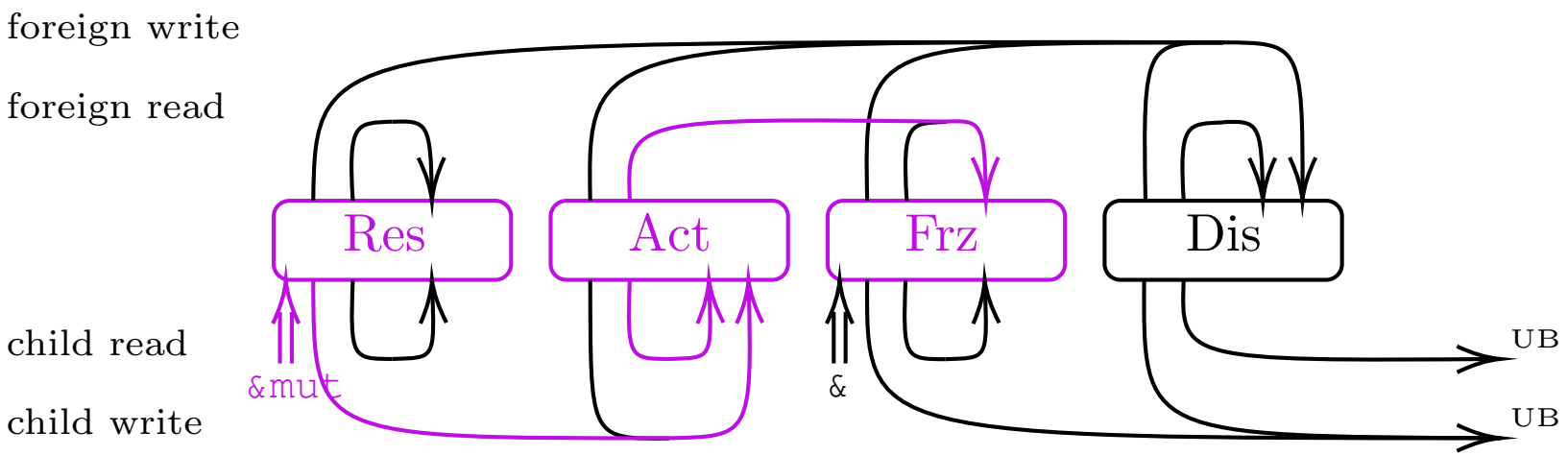
```
let x = &mut ...;

let y = &...;
let vy = *y; // read assumed not foreign
*x = 42;
let vx = *x;
```



# Swap write-read'-read → write-read-read': blocker

## ✓ Swap read-read

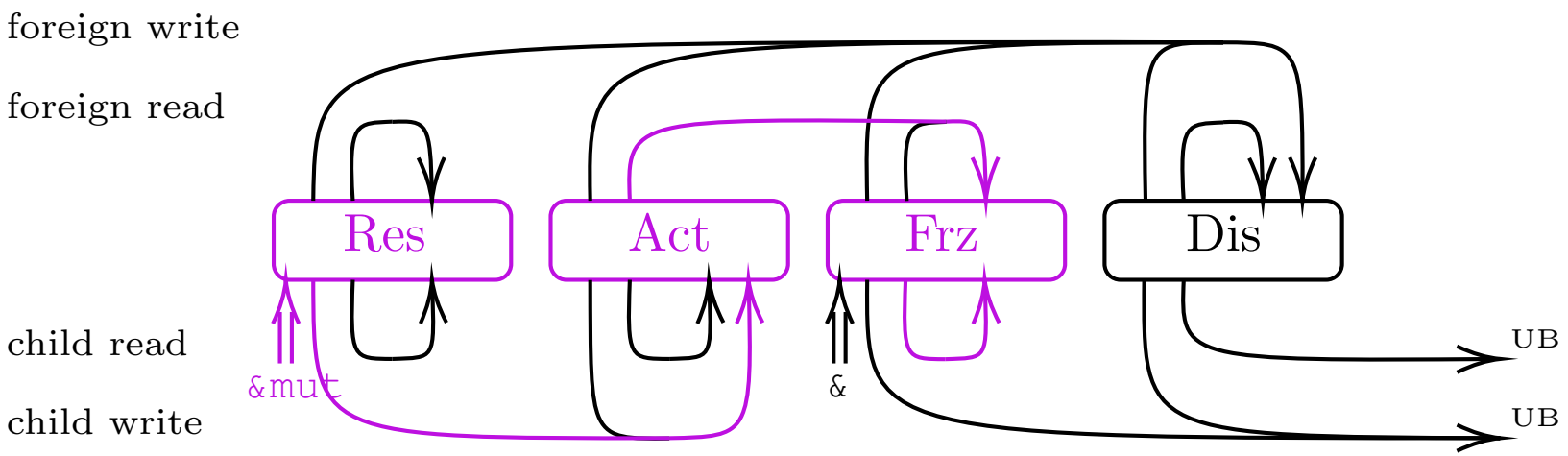
```
let y = &mut *x;
*y = 42;
let vy = *y; // (optimization: move down ?)
let vx = *x; // foreign read
```



# Swap write-read'-read → write-read-read': blocker

## ✓ Swap read-read

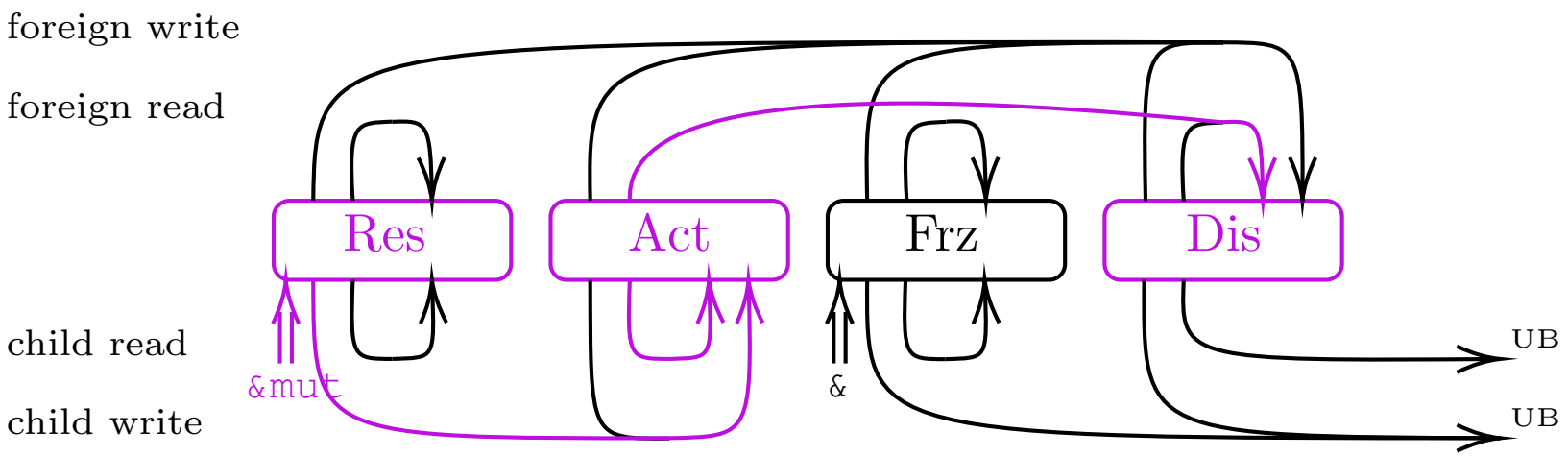
```
let y = &mut *x;
*y = 42;

let vx = *x; // foreign read
let vy = *y;
```



# Swap write-read'-read → write-read-read': blocker

## ✗ Swap read-read: strengthened model

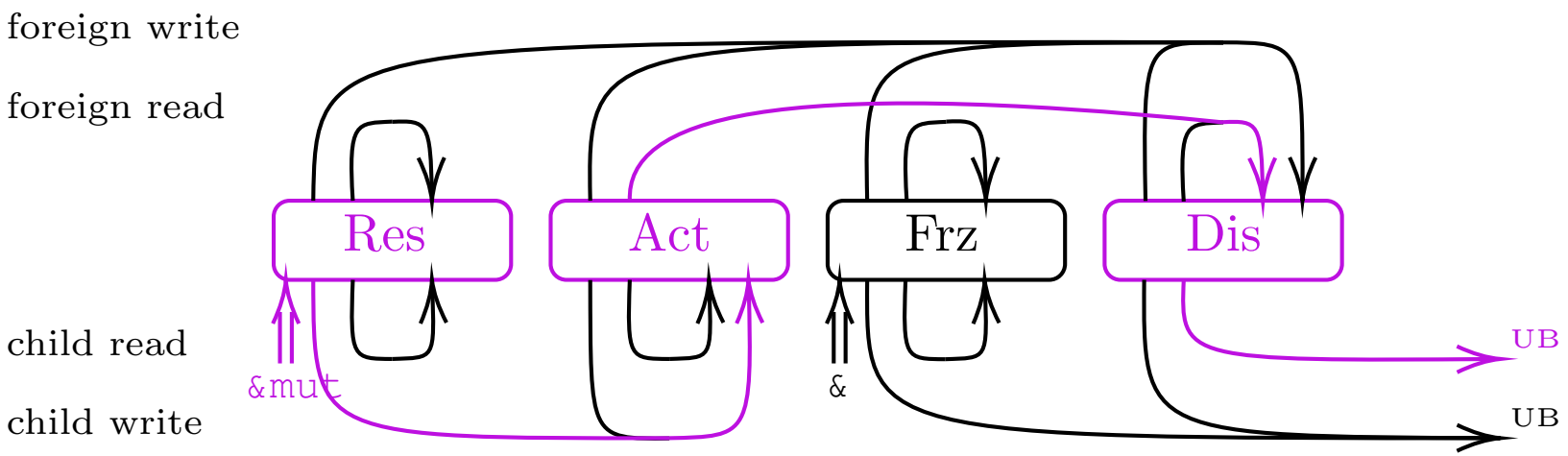
```
let y = &mut *x;
*y = 42;
let vy = *y; // (optimization: move down ?)
let vx = *x; // foreign read
```



# Swap write-read'-read → write-read-read': blocker

## ✗ Swap read-read: strengthened model

```
let y = &mut *x;
*y = 42;

let vx = *x; // foreign read
let vy = *y;
```

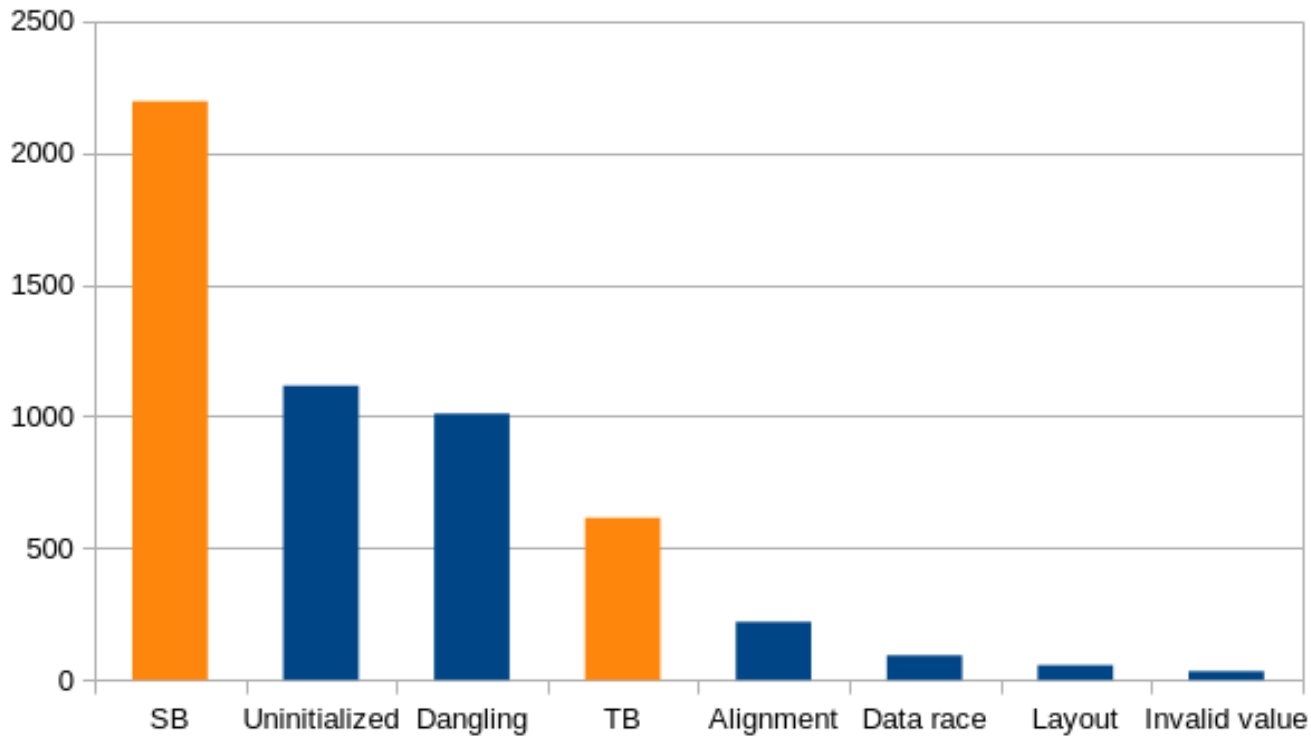


# Summary

- 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)

Tests run by Ben Kimock 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. (100k most recently downloaded)

# 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

Slides and examples on
`github:Vanille-N/tree-beamer/tree/eth`
Complementary material: `perso.crans.org/vanille/treebor`

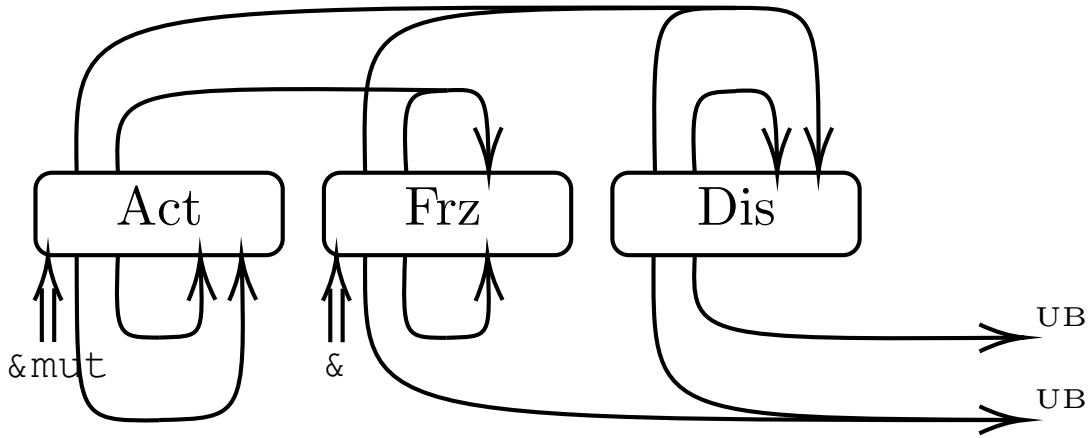
# Core

foreign write

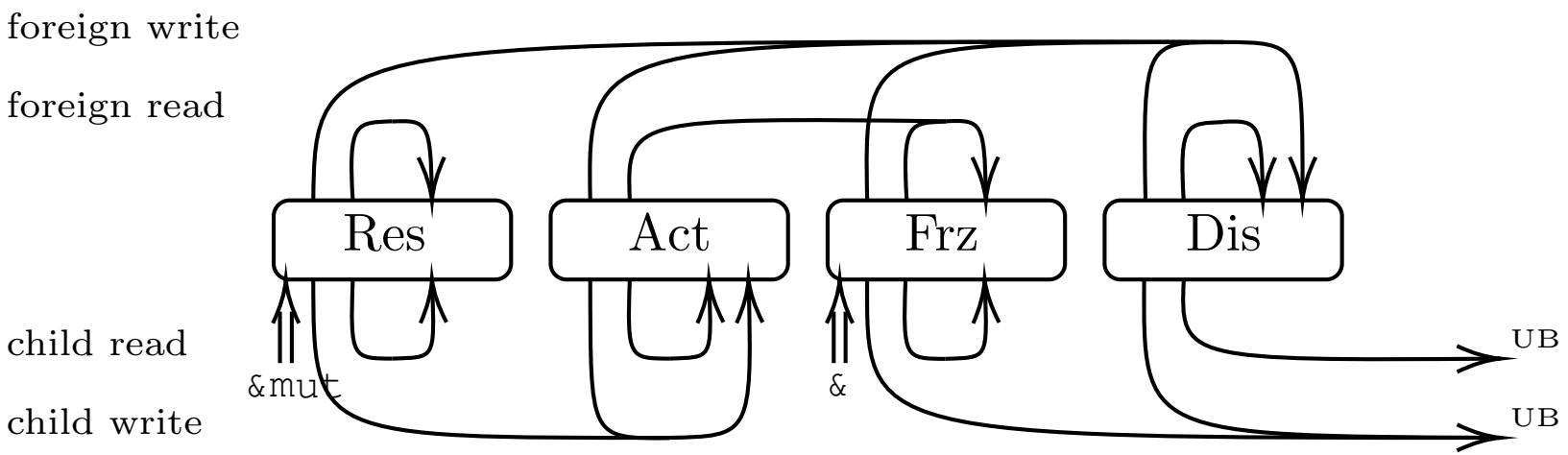
foreign read

child read

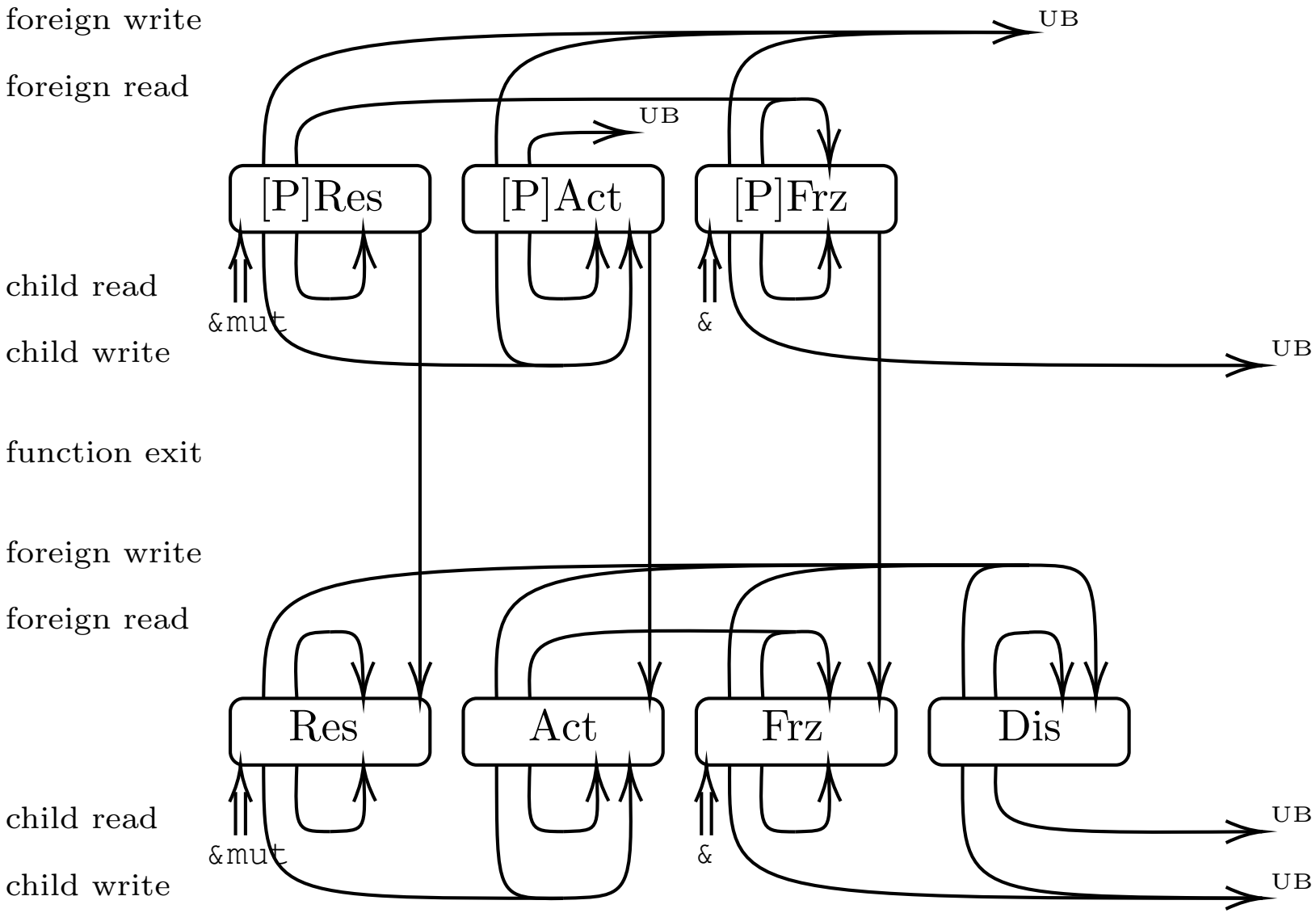
child write



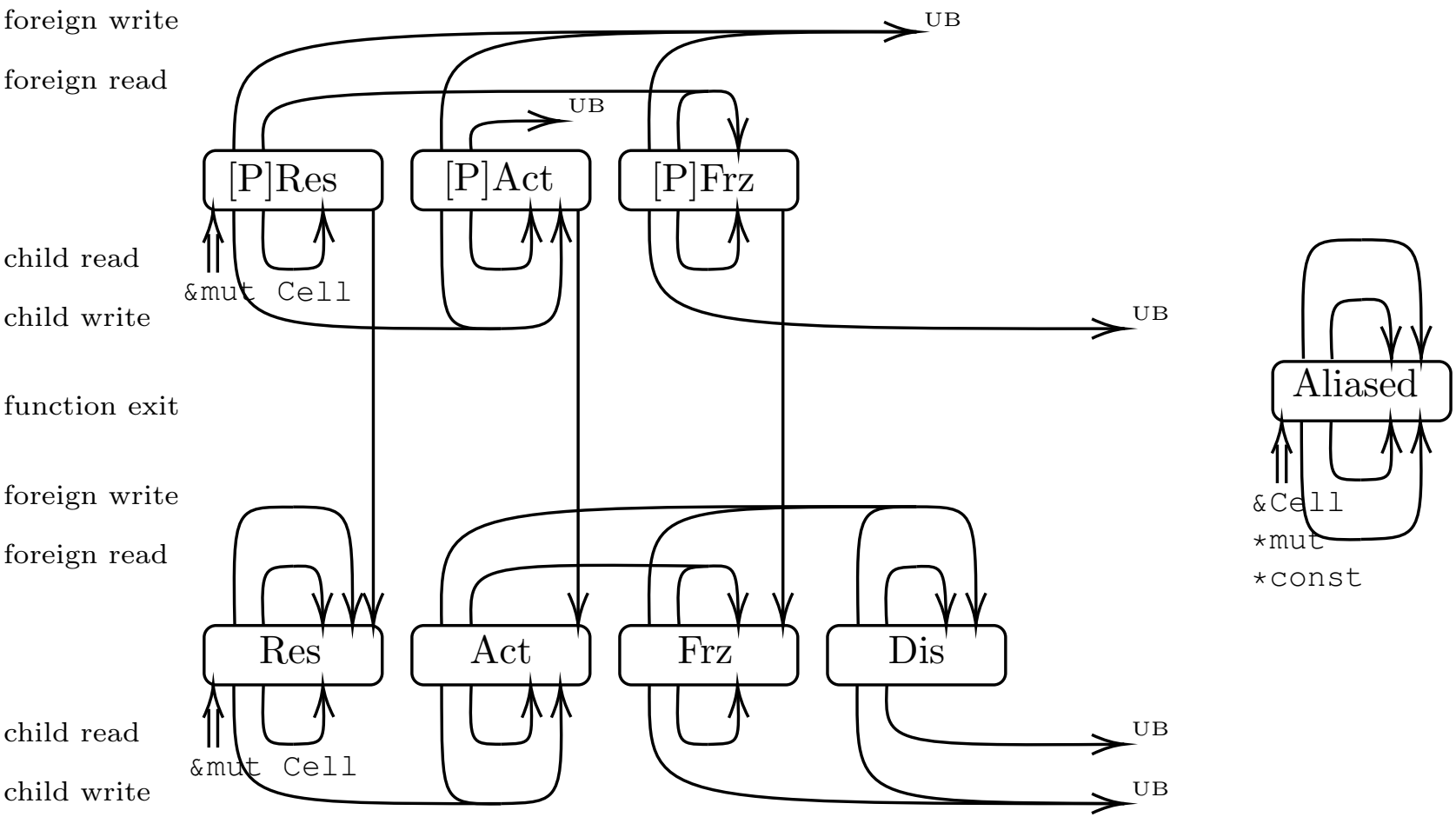
# Base


foreign write
foreign read
child read
child write
Res
Act
Frz
Dis
&mut
&
UB
UB

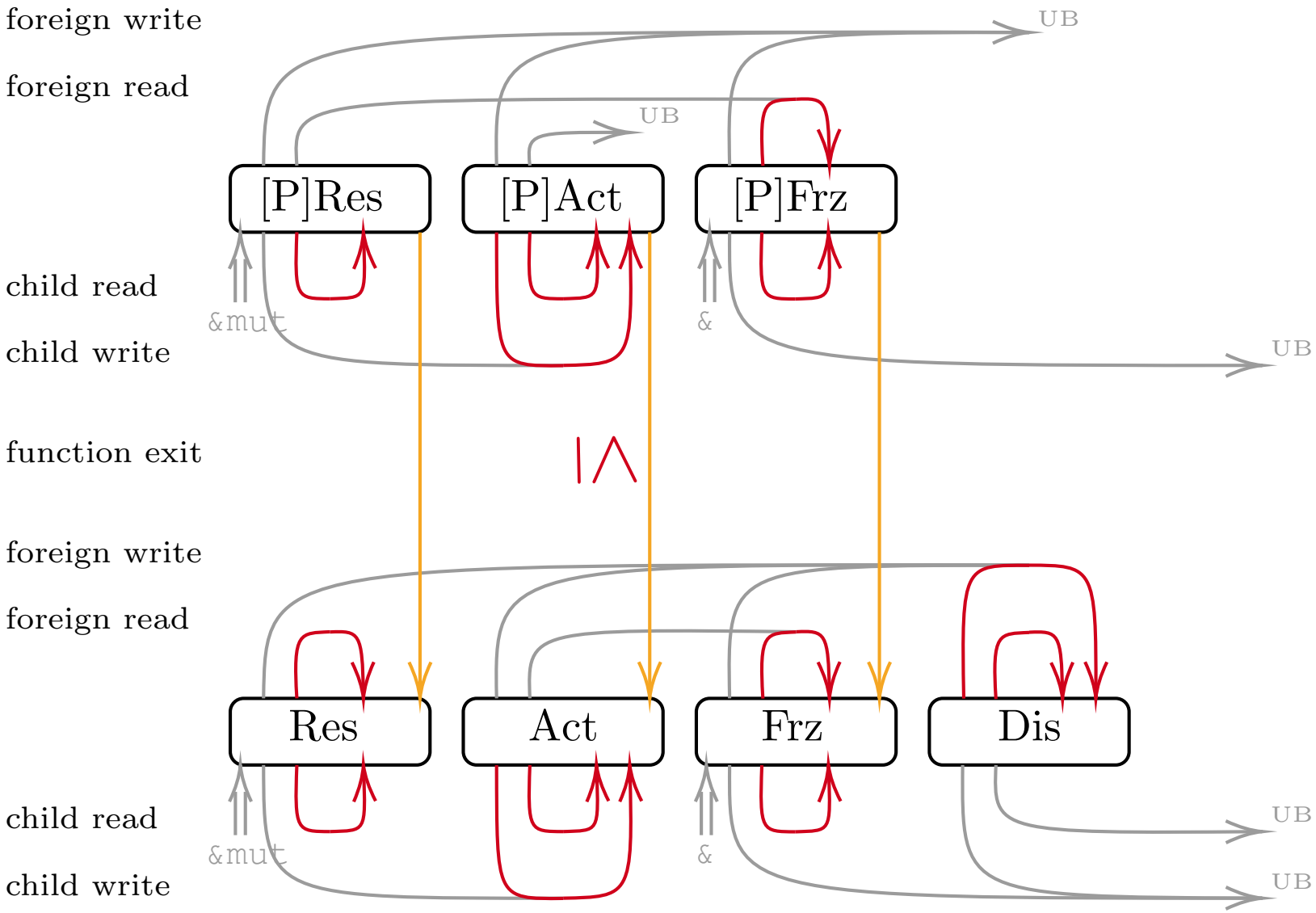
# Base + protected



# Base + cell


foreign write
foreign read
child read
child write
function exit
foreign write
foreign read
child read
child write
UB
UB
UB
UB
UB
[P]Res
[P]Act
[P]Frz
&mut Cell
Res
Act
Frz
Dis
&mut Cell
Aliased
&Cell
*mut
*const

# Intuition: all accesses are idempotent



# Intuition: reordering of reads