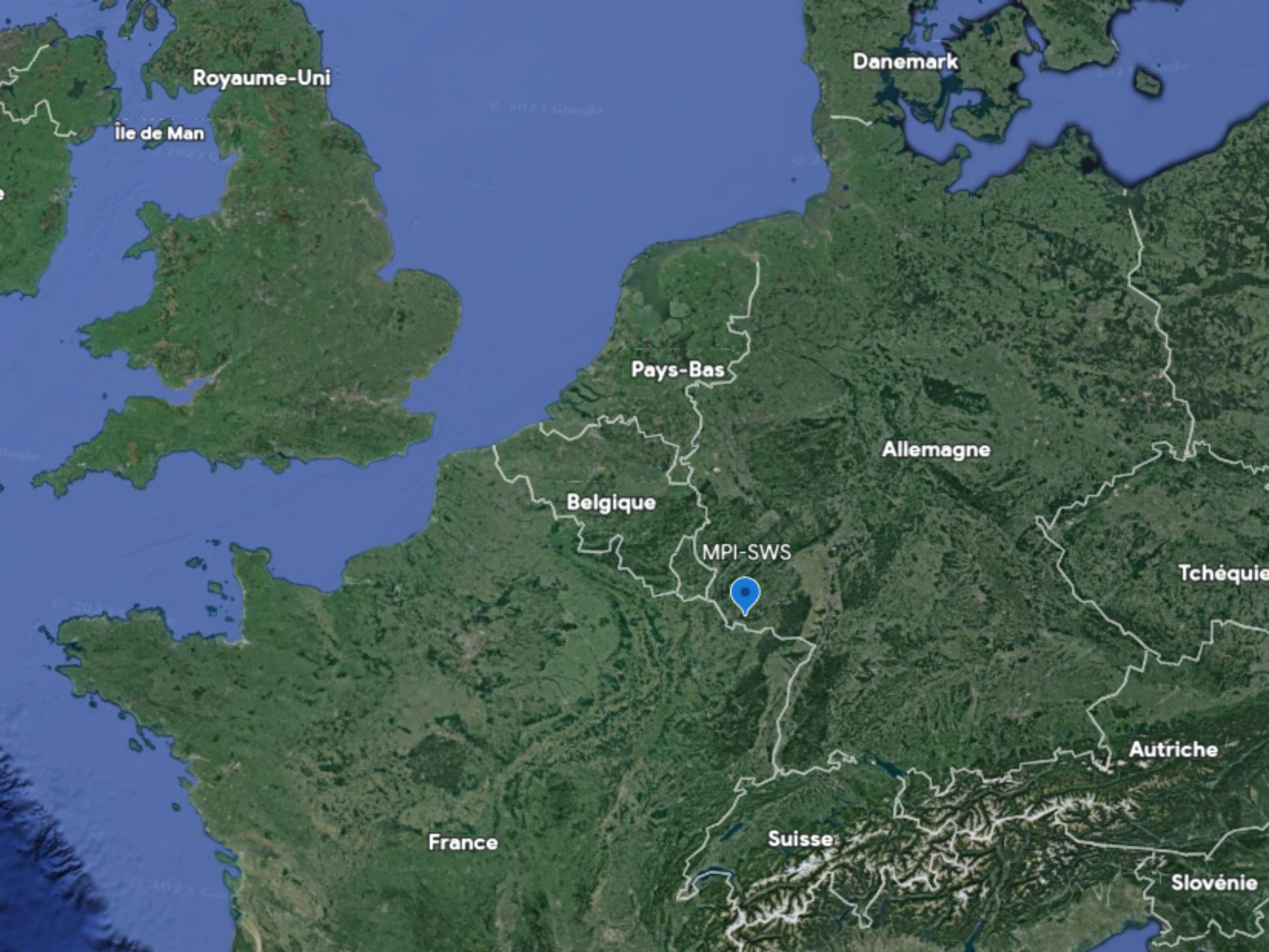
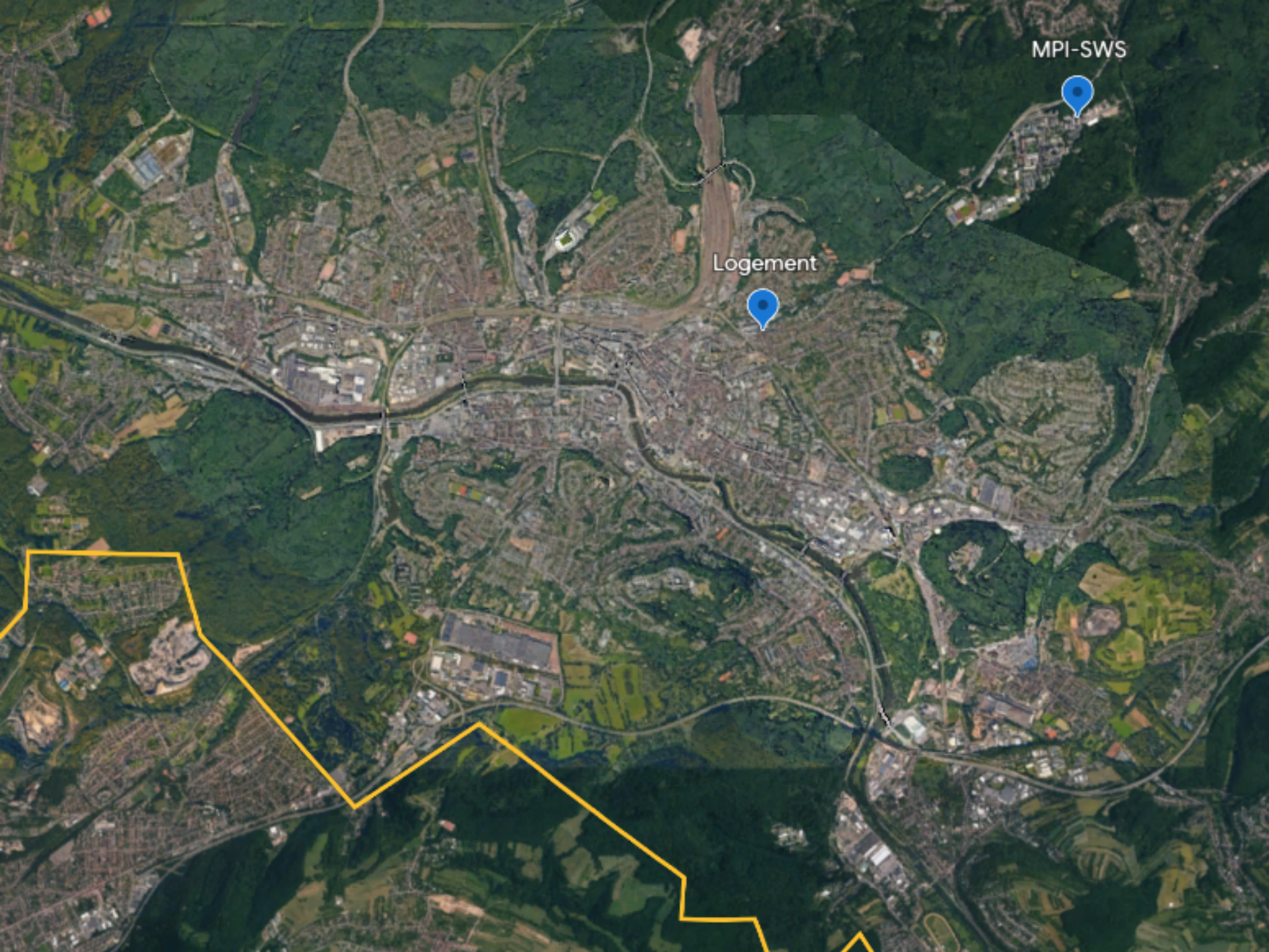

Royaume-Uni
Île de Man
Danemark
Pays-Bas
Allemagne
Belgique
MPI-SWS
Tchéquie
Autriche
France
Suisse
Slovénie


MPI-SWS
Logement

Tree Borrows

An aliasing model for Rust

Neven Villani, Ralf Jung, Derek 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 UB?

Undefined Behavior (UB) arises

- in languages that have both high- and low-level components
  (C, C++, OCaml, Java, Rust: yes)
  (Assembly, Python: no)
- as a necessity from the existence of
  - functions that bypass type systems
    (`Obj.magic`, `std::mem::transmute`, `void*`)
  - unpredictable interaction between language primitives
    (multithreading, pointer arithmetic)
  - interaction with other languages
    (FFI, inline assembly)

Fundamental tradeoff:
more UB = more optimizations = less predictability
Why not Stacked Borrows? Too much UB.
For Tree Borrows: try less UB (at the cost of some optimizations)

Rust's type system

```rust
let t: T = v;
//          ^ value
//       ^ type
//  ^ variable
//  ^^^^^^^^ variable binding

fn pow2(n: u8) -> u128 {
    2.ipow(n)
}

fn main() {
    let n: u8 = 42;
    let m: u128 = pow2(n);
    println!("2^{n} = {m}");
}
```

Rust's type system

```
// Primitives
f32, f64, u8, i8, u16, i16, u32, i32,
u64, i64, u128, i128, usize, isize, bool
// Products
struct Point {
    x: f64,
    y: f64,

type Triplet<T> = (T, T, T);
type Array3<T> = [T; 3];
// Sums
enum Shape<T> {
    Circle(Point, f64),
    Square(Point, f64),
    Triangle(Array3<Point>),
    Other(T),
}
```

Rust's type system

```
// 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)
```

Rust's type system

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?

A motivating example for Aliasing UB

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

Tree Borrows: specification and detection of pointer aliasing UB

Starting observation

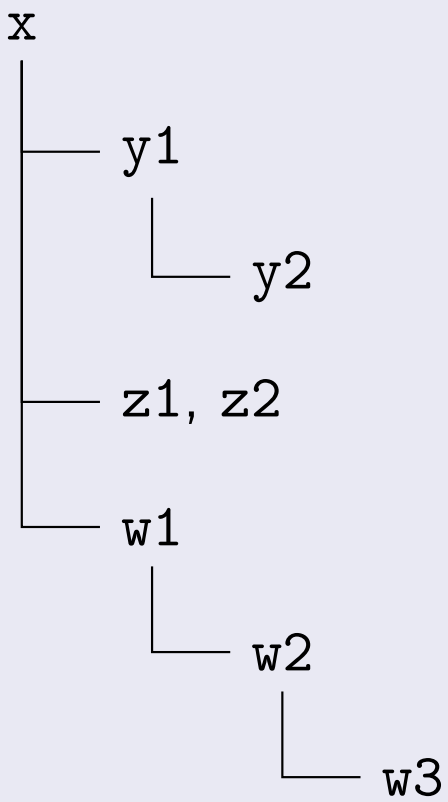
Proper usage of pointers (lifetime inclusion and inheritance of mutability) follows a tree discipline.

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

A Tree of pointers




```
let x = &mut 0u64; // initial borrow

let y1 = &mut *x; // mutable reborrow
let y2 = &*y1; // shared reborrow

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

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

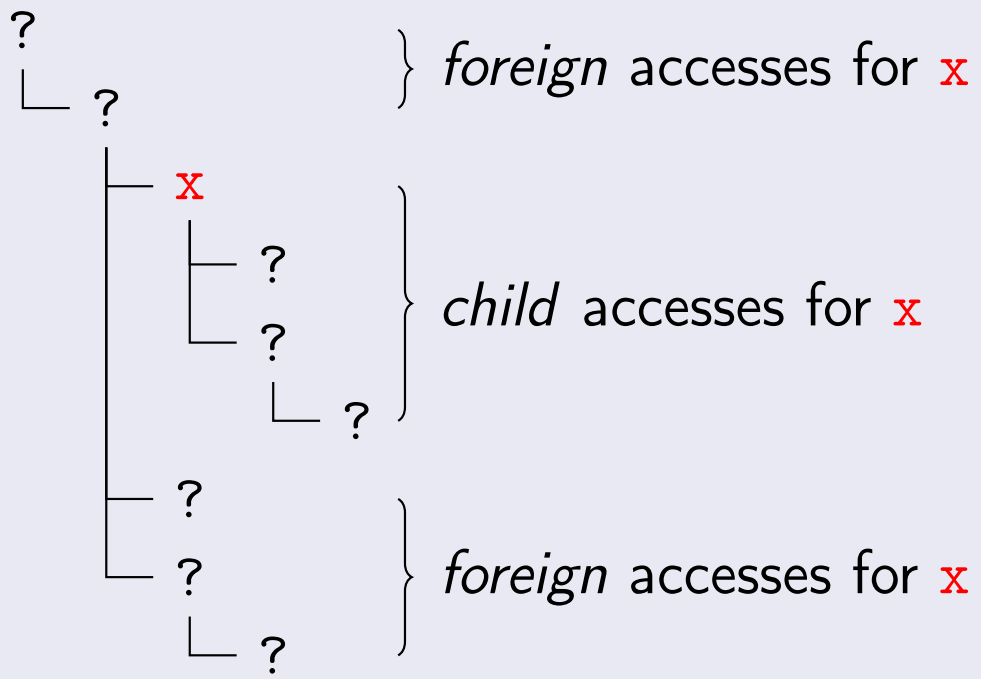
What's in the tree ?

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)

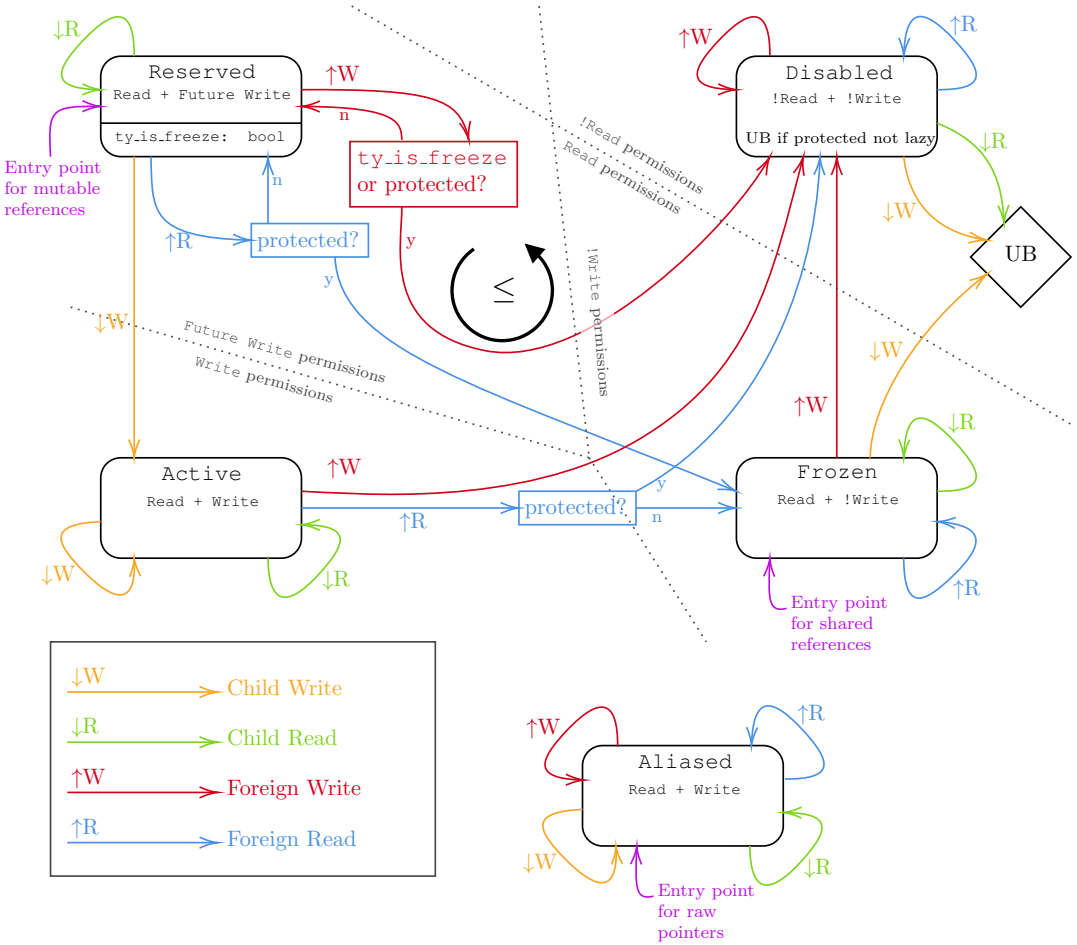
One pointer, 2×2 kinds of accesses



each *read* or *write*.

Tracking permissions

State machine on $\Sigma = \{\text{Foreign}, \text{Child}\} \times \{\text{Rd}, \text{Wr}\} + \{\text{Ret}\}$ and
$Q = \{\texttt{Res}, \texttt{ResIM}, \texttt{Act}, \texttt{Frz}, \texttt{Dis}, \texttt{Als}\} \times \{\text{Lazy}, \text{Init}\} \times \{\text{Prot}, \text{Unprot}\}$



- Independent execution on every (pointer id, byte),
- Entry point depends on the pointer kind and the creation context,
- Infinite stream of accesses, rejects any finite prefix that contains UB.

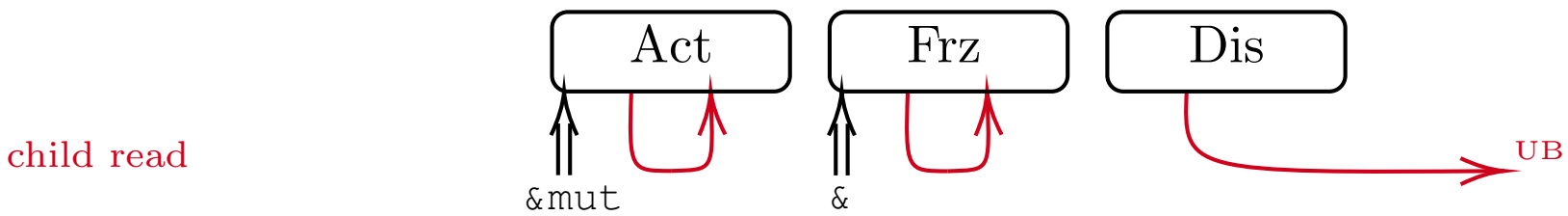
Active, Frozen, Disabled (simplified)

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



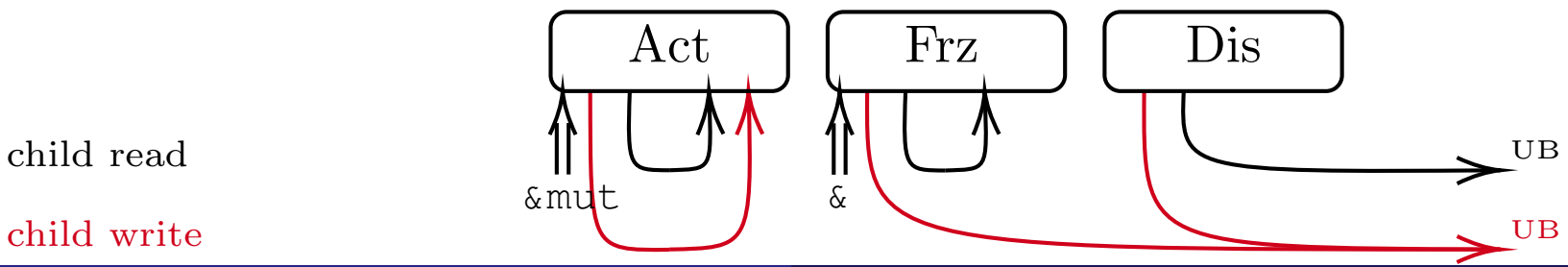
Active, Frozen, Disabled (simplified)

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



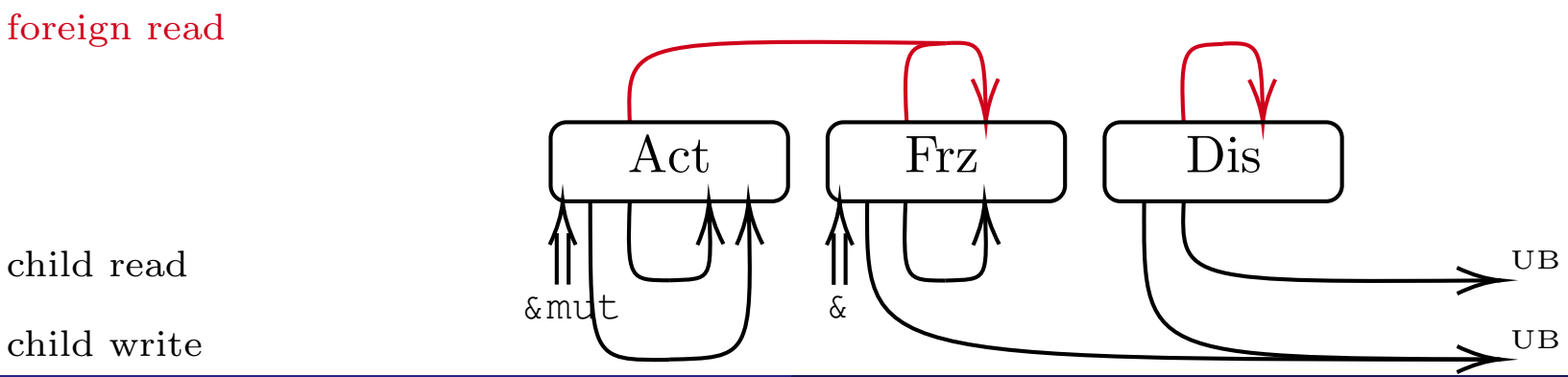
Active, Frozen, Disabled (simplified)

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`



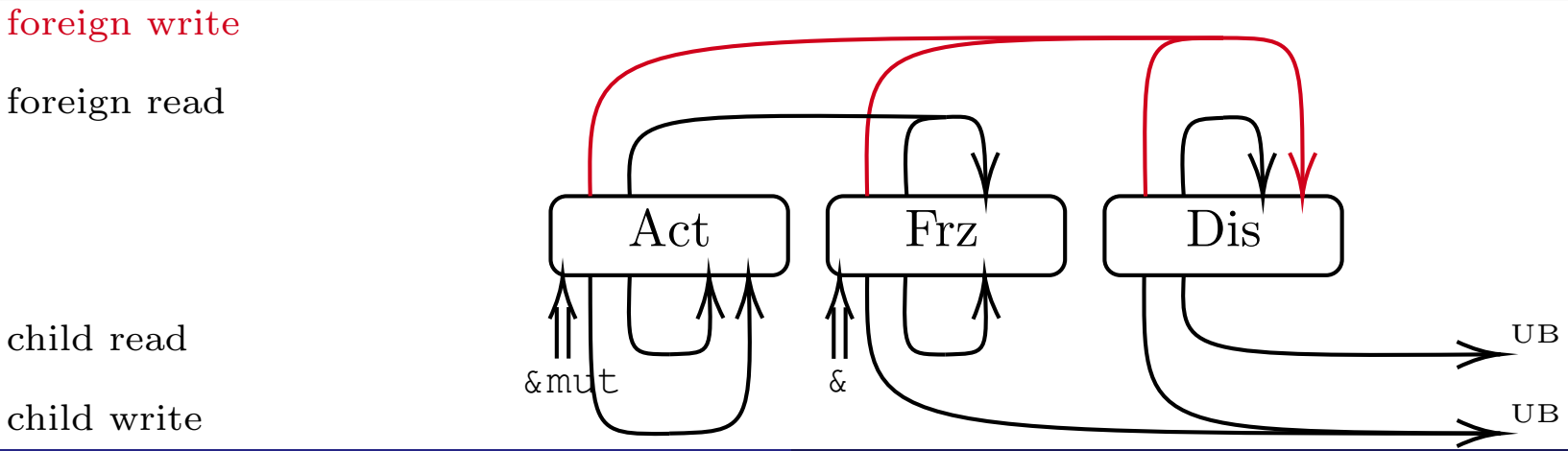
Active, Frozen, Disabled (simplified)

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`



This simplified model already enforces

- ✓ `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

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

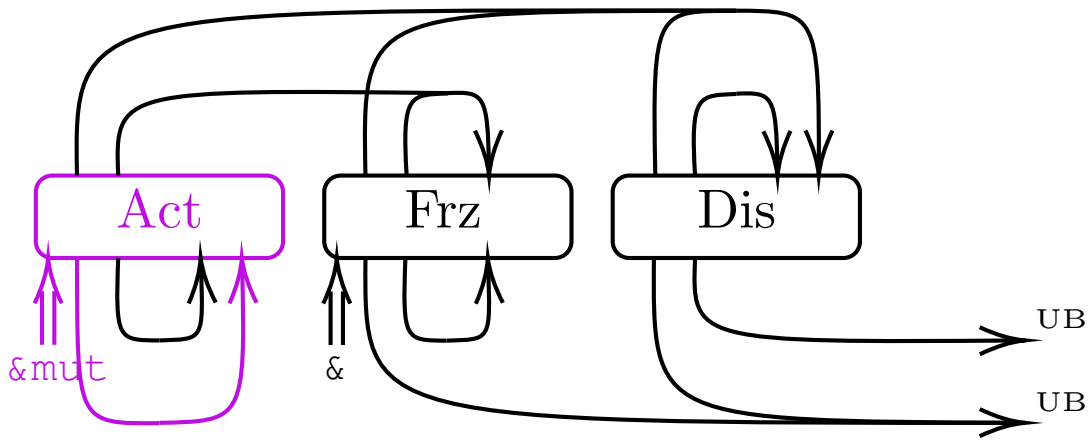
```
let x = &mut ...;
let y = &mut ...;
*x = 42; // (optimization: move down ?)
*y = 19; // is this a foreign write ? if not

*x = 57;
```

foreign write

foreign read

child read

child write



Swap write-write

✓ Base model

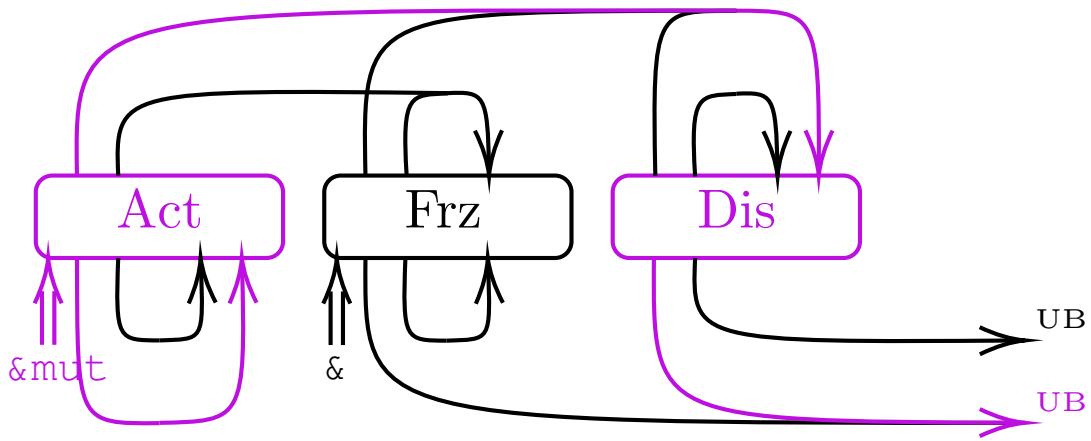
```
let x = &mut ...;
let y = &mut ...;
*x = 42; // (optimization: move down ?)
*y = 19; // is this a foreign write ? if yes

*x = 57;
```

foreign write

foreign read

child read

child write



Swap write-write

✓ Base model

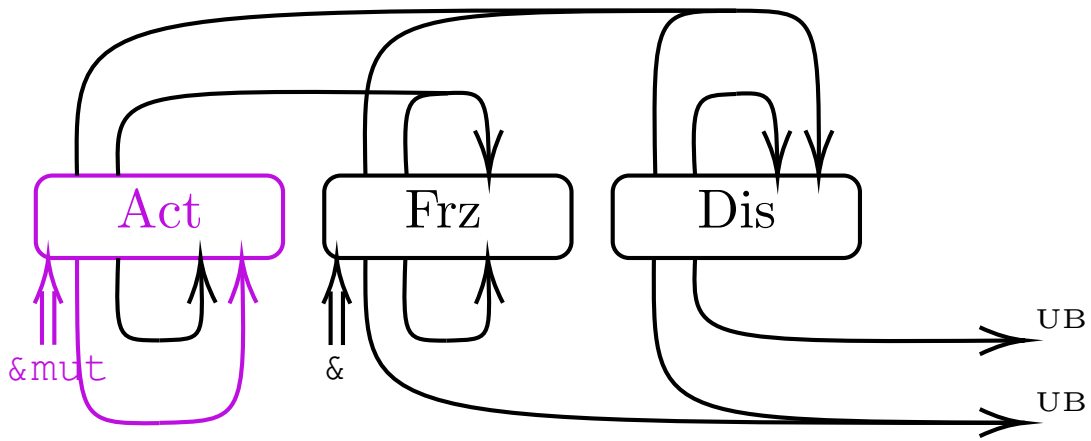
```
let x = &mut ...;
let y = &mut ...;

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

foreign write

foreign read

child read

child write

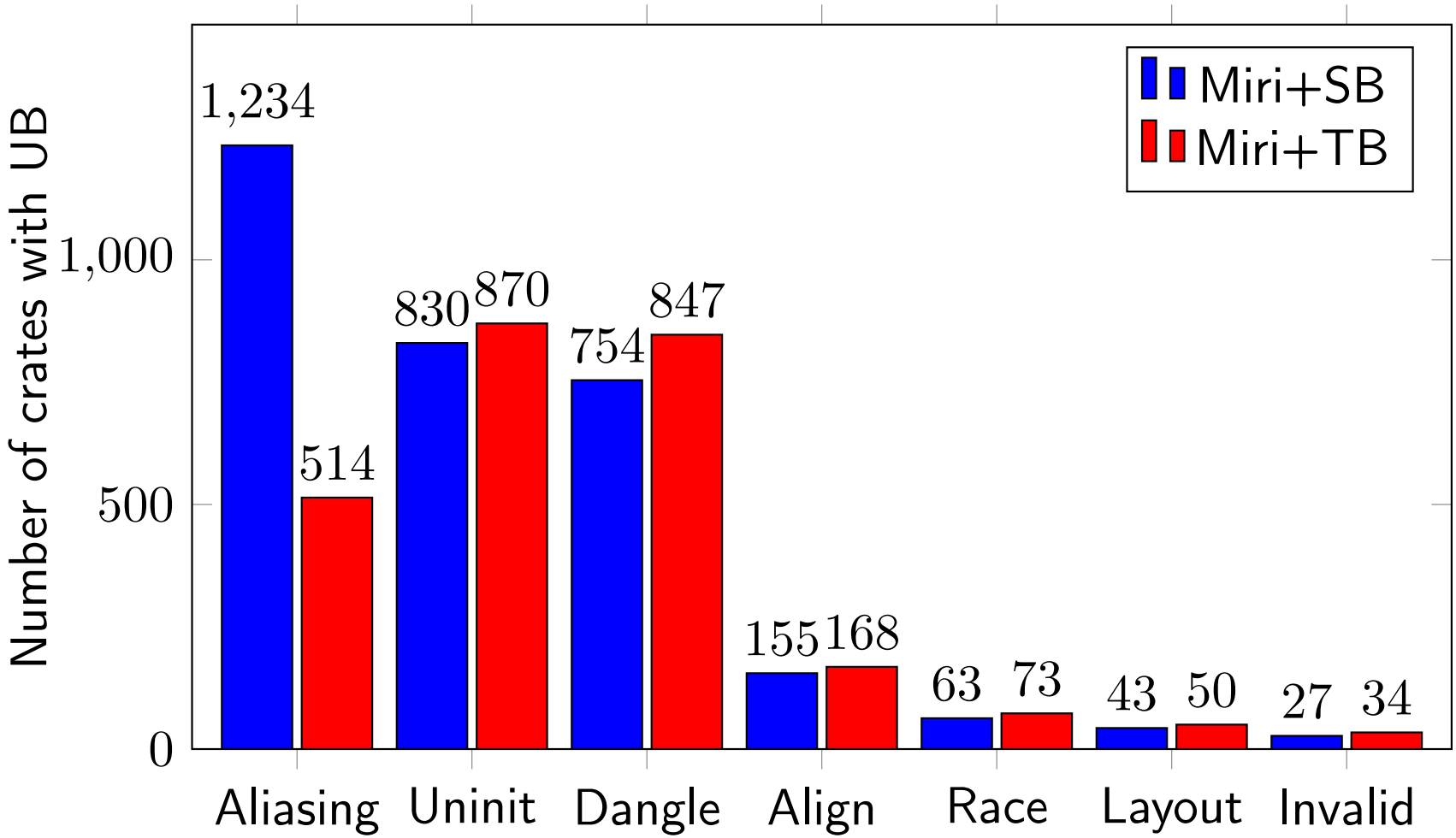


Some standard optimizations

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	✓	✓*

*: only for function arguments (stronger semantics not presented here)

Counting crates with UB



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 rare

Questions ?

TB also has...

- implementation in Miri (`github:rust-lang/miri`)
- performance improvements compared to the naive implementation
  - many tricks to trim tree traversals
  - lazy initialization for out-of-range accesses
- formalization in Coq, with many optimizations proven

Slides: `github:Vanille-N/tree-beamer/tree/ens`
Complementary material: `perso.crans.org/vanille/treebor`

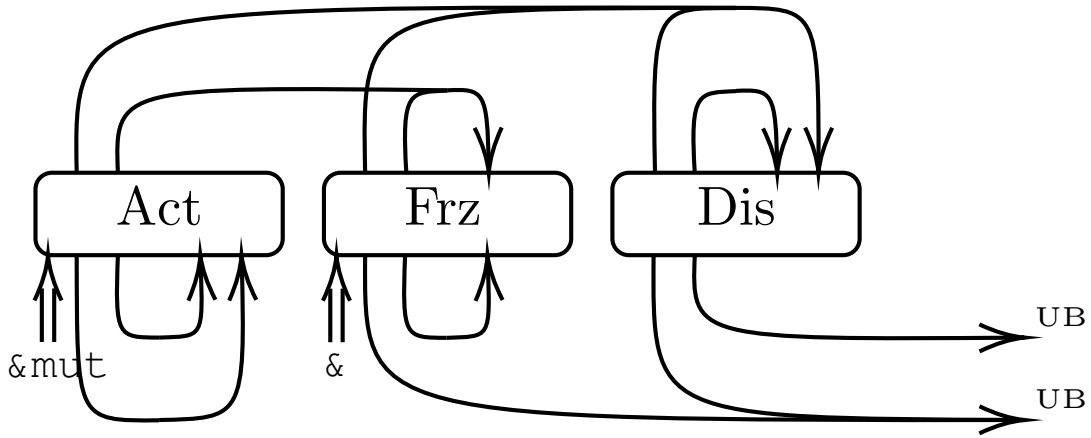
Core

foreign write

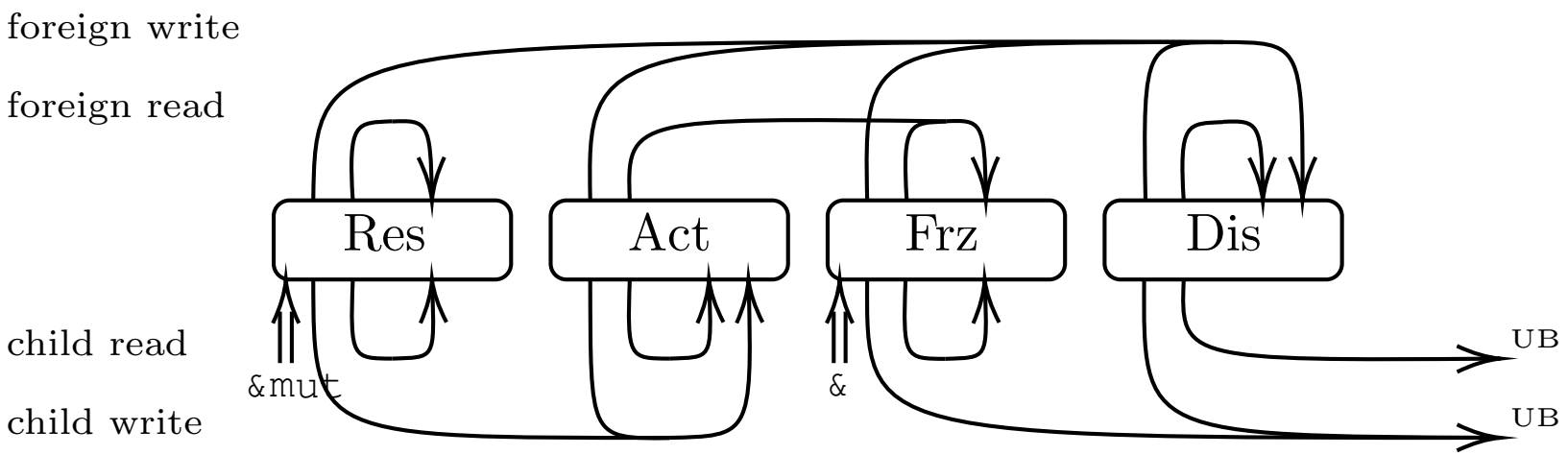
foreign read

child read

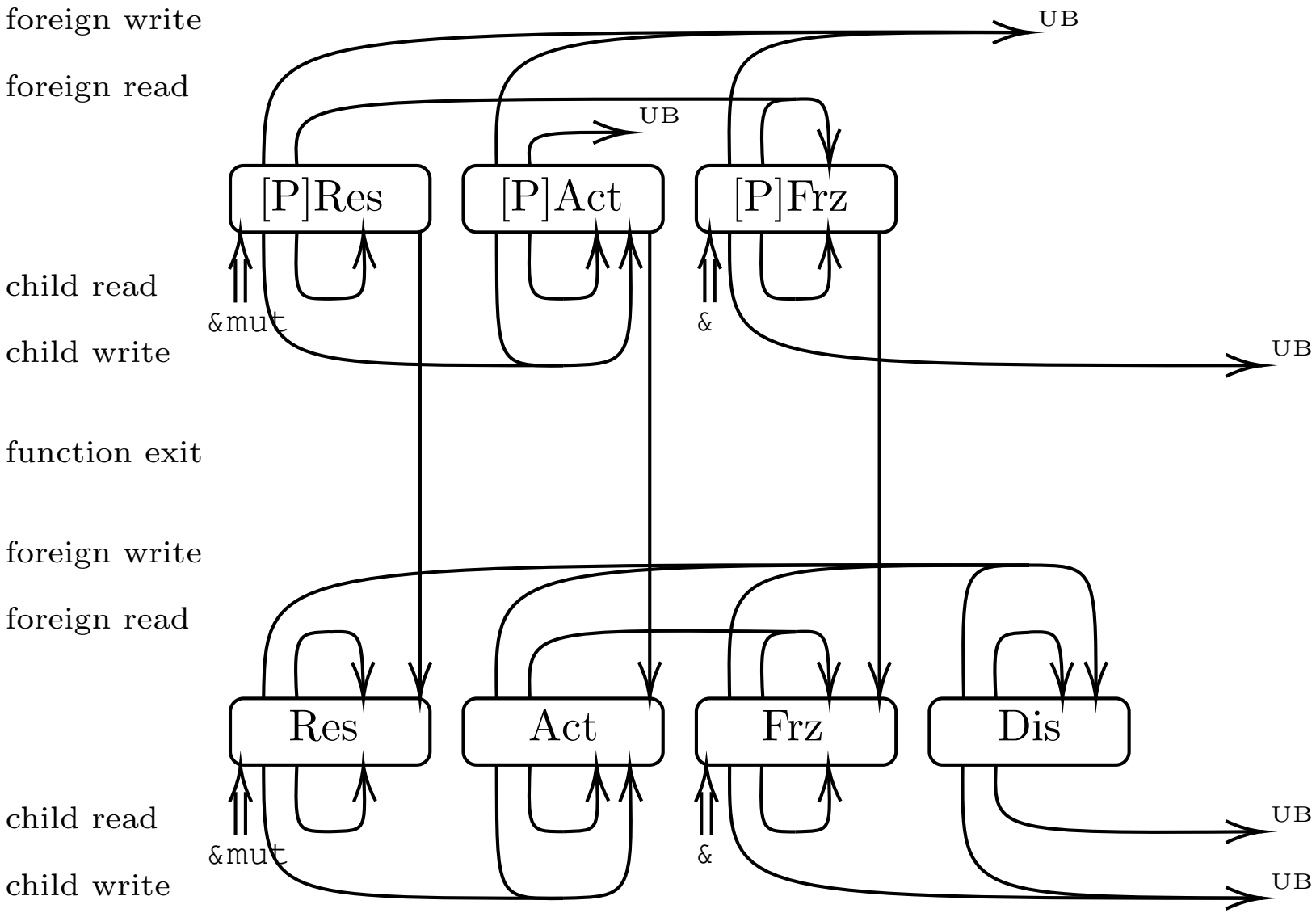
child write



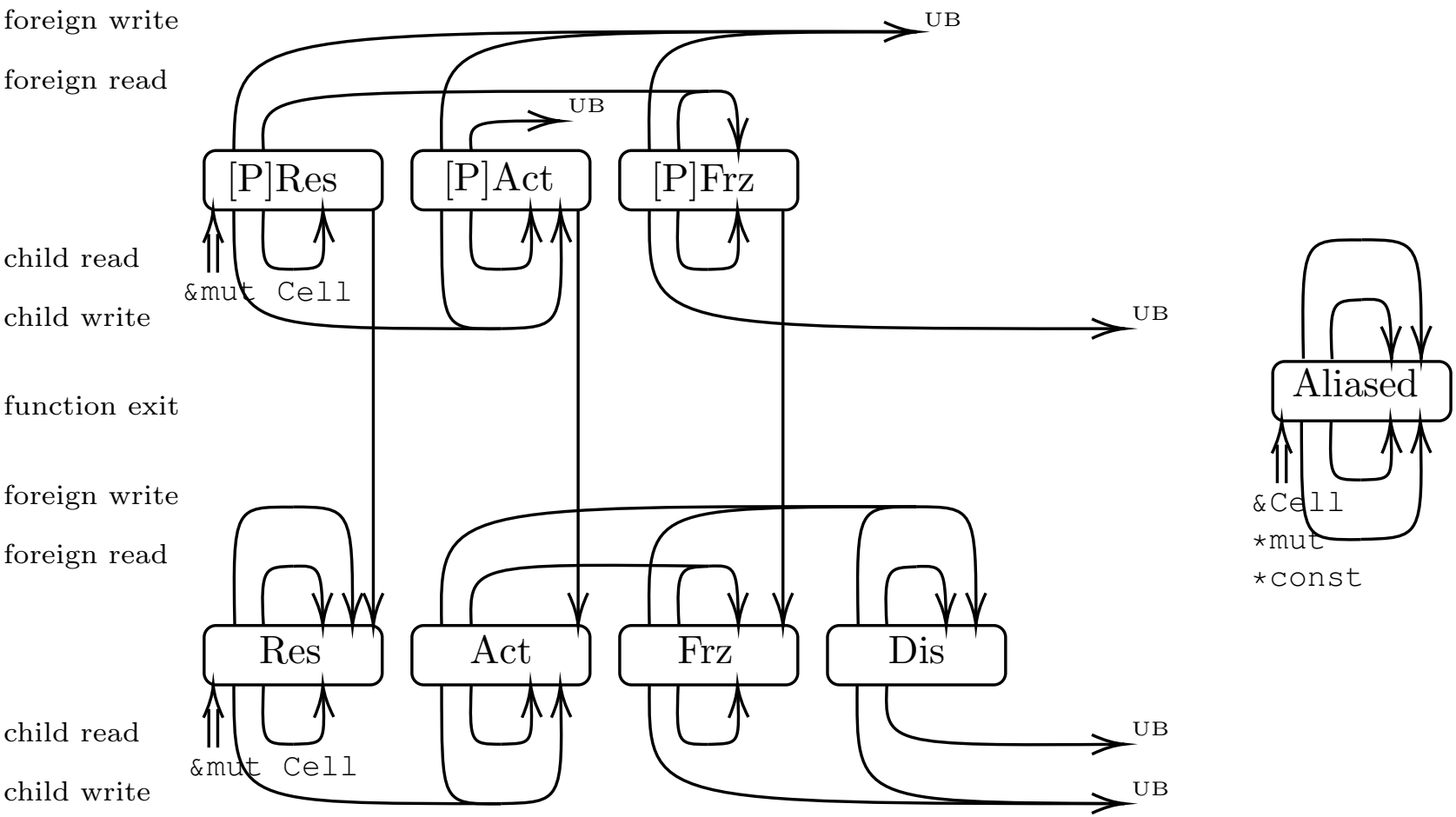
Base


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

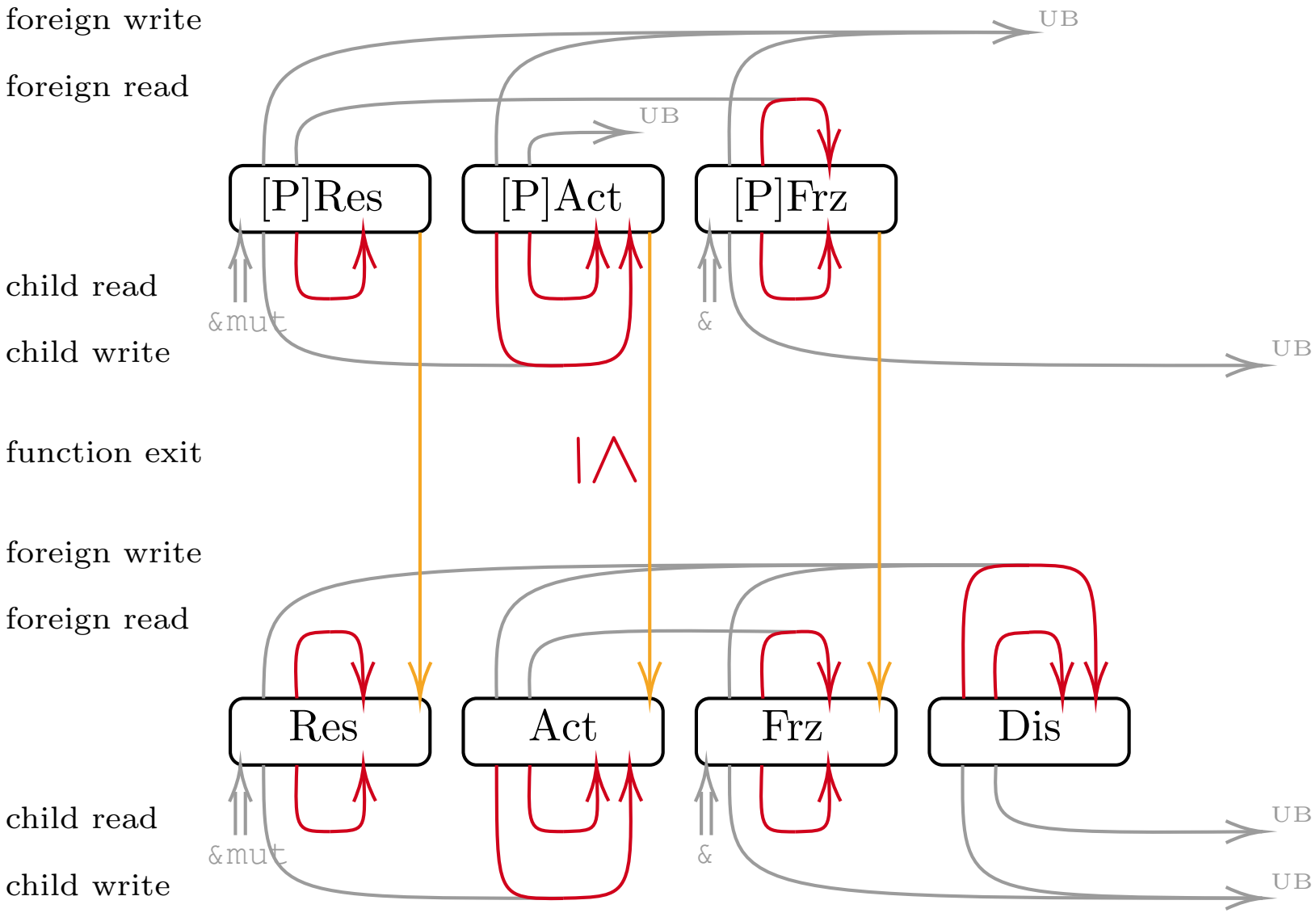
Base + protected



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