Tree Borrows

Neven Villani,[1] Johannes Hostert,[2] Derek Dreyer,[3] Ralf Jung[2]

Rust Verification Workshop

2024-04-08

[1]ENS Paris-Saclay, Université Paris-Saclay

[2]ETH Zurich

[3]MPI-SWS

Strong guarantees for references

`&mut` → mutation, no aliasing

`&` → aliasing, no mutation[4]

[4] for non-interior-mutable types

Absence of aliasing + mutability allows optimizations

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

    *y = val;
}
```

Absence of aliasing + mutability allows optimizations

```
fn foo(y: &mut u64) {
    let val = *y;
  //*y = 42;

    *y = val;
}
```

Absence of aliasing + mutability allows optimizations

```
fn foo(y: &mut u64) {
    let val = *y;
  //*y = 42;

  //*y = val;
}
```

Absence of aliasing + mutability allows optimizations

```
fn foo(y: &mut u64) {
   //let val = *y;
   //*y = 42;

   //*y = val;
}
```

Absence of aliasing + mutability allows optimizations

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

    *y = val;
}
```

optimized

$$\Longrightarrow$$

```
fn foo(y: &mut u64) {
    //let val = *y;
    //*y = 42;

    //*y = val;
}
```

Absence of aliasing + mutability allows optimizations

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

optimized

$\Longrightarrow$

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

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

```rust
static mut X: u64 = 0;
```

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

```rust
static mut X: u64 = 0;

fn main() {
    foo(unsafe { &mut X });
}

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

```rust
static mut X: u64 = 0;

fn main() {
    foo(unsafe { &mut X });
}

fn foo(y: &mut u64) {
    let val = *y;
    *y = 42;
    println!("{}", unsafe { X }); // prints 42
    *y = val;
}
```

```rust
static mut X: u64 = 0;

fn main() {
    foo(unsafe { &mut X });
}

fn foo(y: &mut u64) {
  //let val = *y;
  //*y = 42;
    println!("{}", unsafe { X }); // prints 0
  //*y = val;
}
```

```rust
static mut X: u64 = 0;

fn main() {
    foo(unsafe { &mut X });
}

fn foo(y: &mut u64) {
  //let val = *y;
  //*y = 42;
    println!("{}", unsafe { X }); // prints 0
  //*y = val;
}
```

Optimization **changes observable behavior**…
Is the optimization incorrect?

It's not the optimization that is wrong, it's the code

Tree Borrows enforces aliasing rules by adding proof obligations to `unsafe` blocks.

Code that violates these rules is declared **Undefined Behavior**.

It's not the optimization that is wrong, it's the code

Tree Borrows enforces aliasing rules by adding proof obligations to `unsafe` blocks.

Code that violates these rules is declared **Undefined Behavior**.

> **Sounds familiar?**
>
> **Stacked Borrows** has the same purpose, Tree Borrows is its successor.

Stacked Borrows

Adds **extra state** to the abstract machine to track provenance.
Distinguishes pointers to the same location with a **tag**.

Stacked Borrows

Adds **extra state** to the abstract machine to track provenance.
Distinguishes pointers to the same location with a **tag**.

Uses a **stack** to store permissions.
Enforces that borrows are well-bracketed.

However, Stacked Borrows...

- does not handle two-phase borrows (gives up on any optimization)

However, Stacked Borrows...

- does not handle two-phase borrows (gives up on any optimization)

```
    vec.push(vec[0]);
//  ^^^ 1. implicit &mut in function arguments
//          ^^^^^^ 2. read-only operation before function
//                    entry does not invalidate the &mut
```

However, Stacked Borrows...

- does not handle two-phase borrows (gives up on any optimization)

```
    vec.push(vec[0]);
//  ^^^ 1. implicit &mut in function arguments
//            ^^^^^^ 2. read-only operation before function
//                      entry does not invalidate the &mut
```

- forbids common `unsafe` patterns (declared UB)

However, Stacked Borrows...

- does not handle two-phase borrows (gives up on any optimization)

```
    vec.push(vec[0]);
//  ^^^ 1. implicit &mut in function arguments
//            ^^^^^^ 2. read-only operation before function
//                      entry does not invalidate the &mut
```

- forbids common `unsafe` patterns (declared UB)

```
let from = data.as_ptr();
// SB inserts an implicit write, killing the raw pointer
let to = data.as_mut_ptr();
copy_nonoverlapping(from, to.add(1), 1); // UB
```

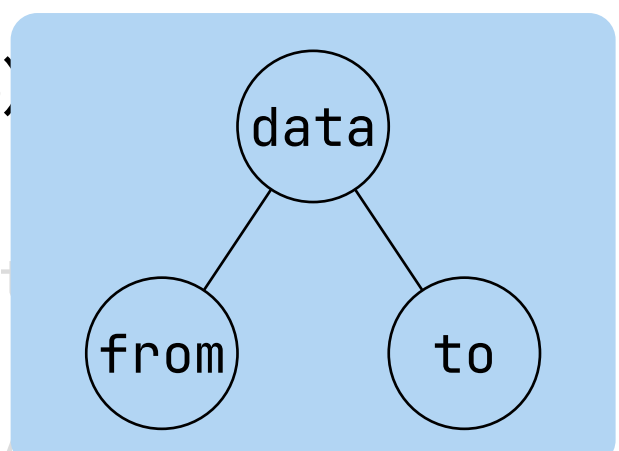
However, Stacked Borrows...

- does not handle two-phase borrows (gives up on any optimization)

```
    vec.push(vec[0]);
//  ^^^ 1. implicit &mut in function arguments
//            ^^^^^^ 2. read-only operation before function
//                      entry does not invalidate the &mut
```

- forbids common `unsafe` patterns (declared UB)

```
let from = data.as_ptr();
// SB inserts an implicit write, killing
let to = data.as_mut_ptr();
copy_nonoverlapping(from, to.add(1), 1);
```



However, Stacked Borrows...

- does not handle two-phase borrows (gives up on any optimization)

```
        vec.push(vec[0]);
//      ^^^ 1. implicit &mut in function arguments
//
//
```

- fo

```
let from = data.as_ptr();
// SB inserts an implicit write, killing
let to = data.as_mut_ptr();
copy_nonoverlapping(from, to.add(1), 1);
```

The stack is too rigid to represent the exact relationship

Stacked Borrows ⇝ Tree Borrows

Stack is not precise enough.

Use a **tree** instead → accurate tracking of pointer ancestry

Stacked Borrows ⇝ Tree Borrows

Stack is not precise enough.

Use a **tree** instead → accurate tracking of pointer ancestry

Results in

- accurate handling of two-phase borrows
- more permitted patterns
- simpler rules, fewer exceptions

Design constraints

Enough UB

- strict enough that interesting **optimizations** are possible
  $\rightarrow$ guided by desirable optimizations, and expected UB
  $\rightarrow$ *formalized in Coq, ongoing work to prove correctness*

Design constraints

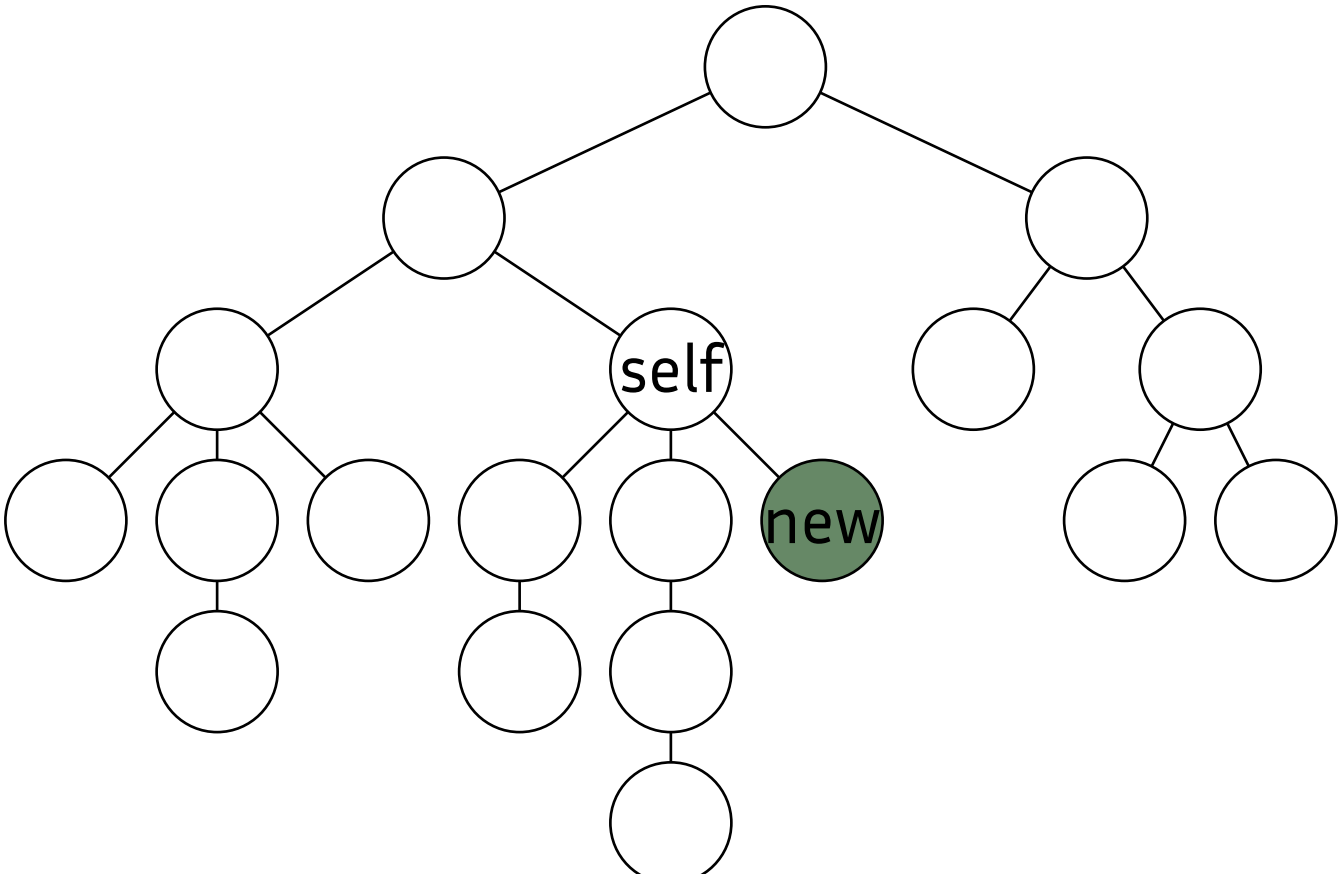
Enough UB

- strict enough that interesting **optimizations** are possible
  → guided by desirable optimizations, and expected UB
  → *formalized in Coq, ongoing work to prove correctness*

Not too much

- permissive enough that **existing libraries** are correct
  → guided by common patterns, complaints about Stacked Borrows
  → *implemented in the Miri interpreter, checked against libraries*

Tracking relationships

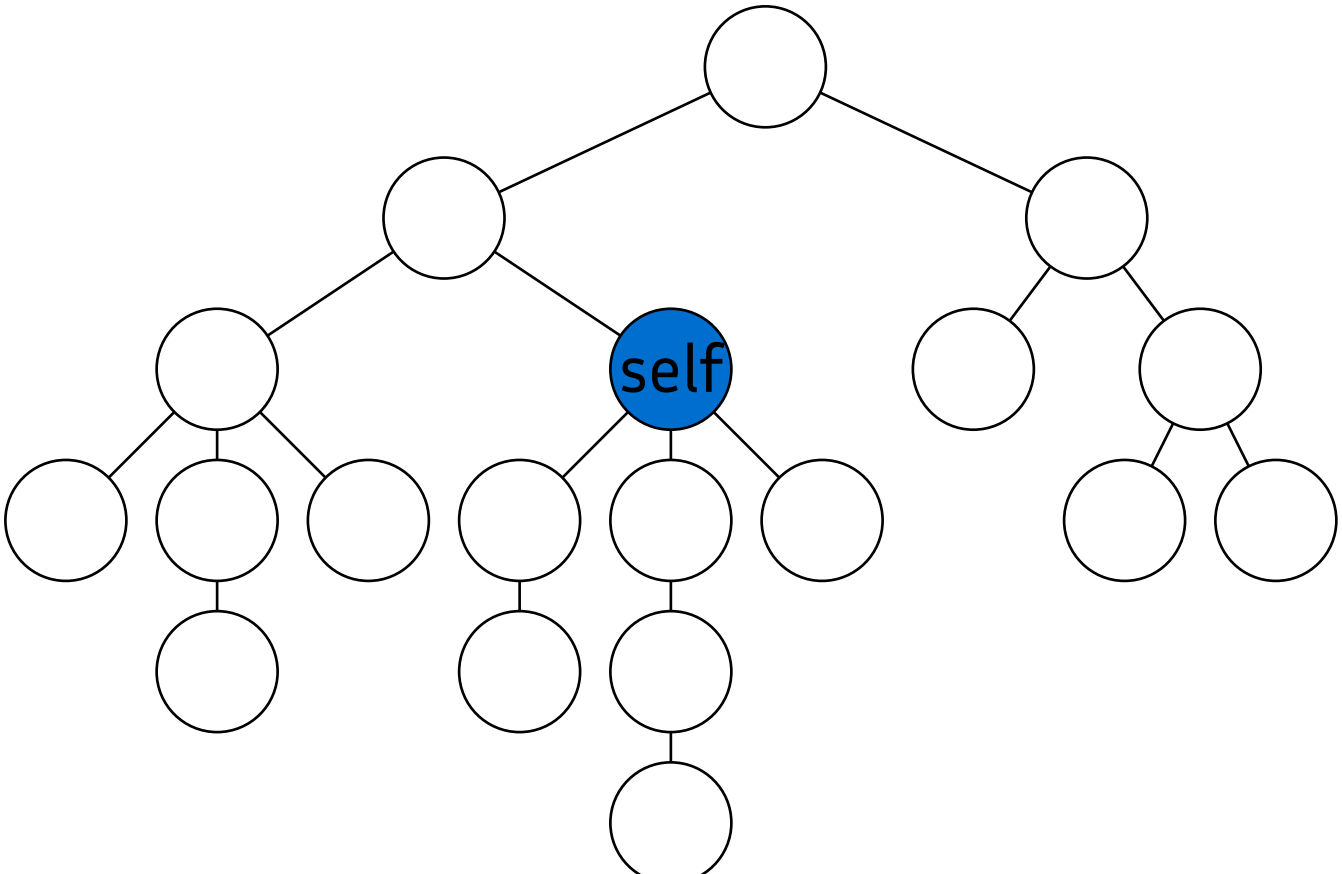


reborrows

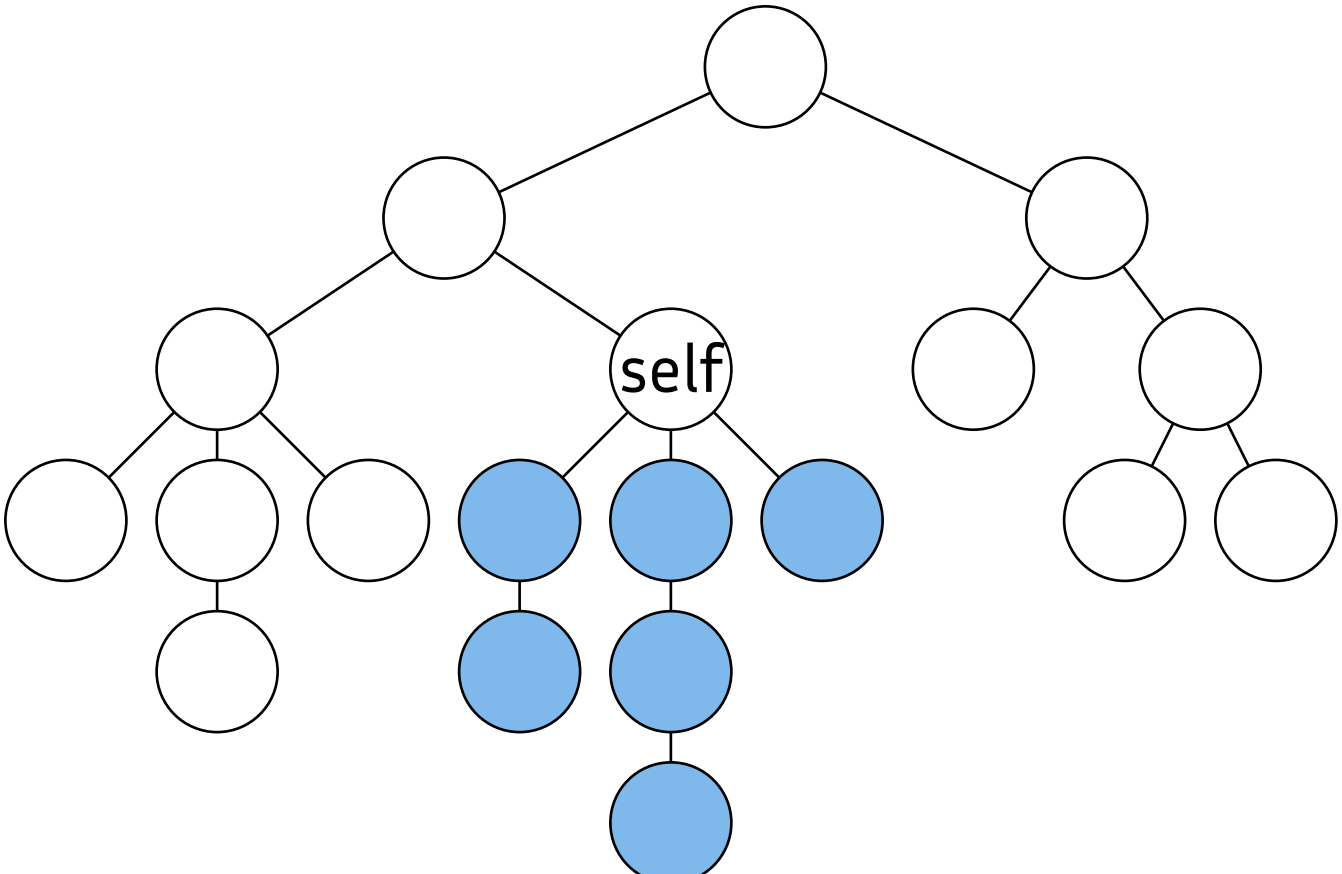
create

immediate children

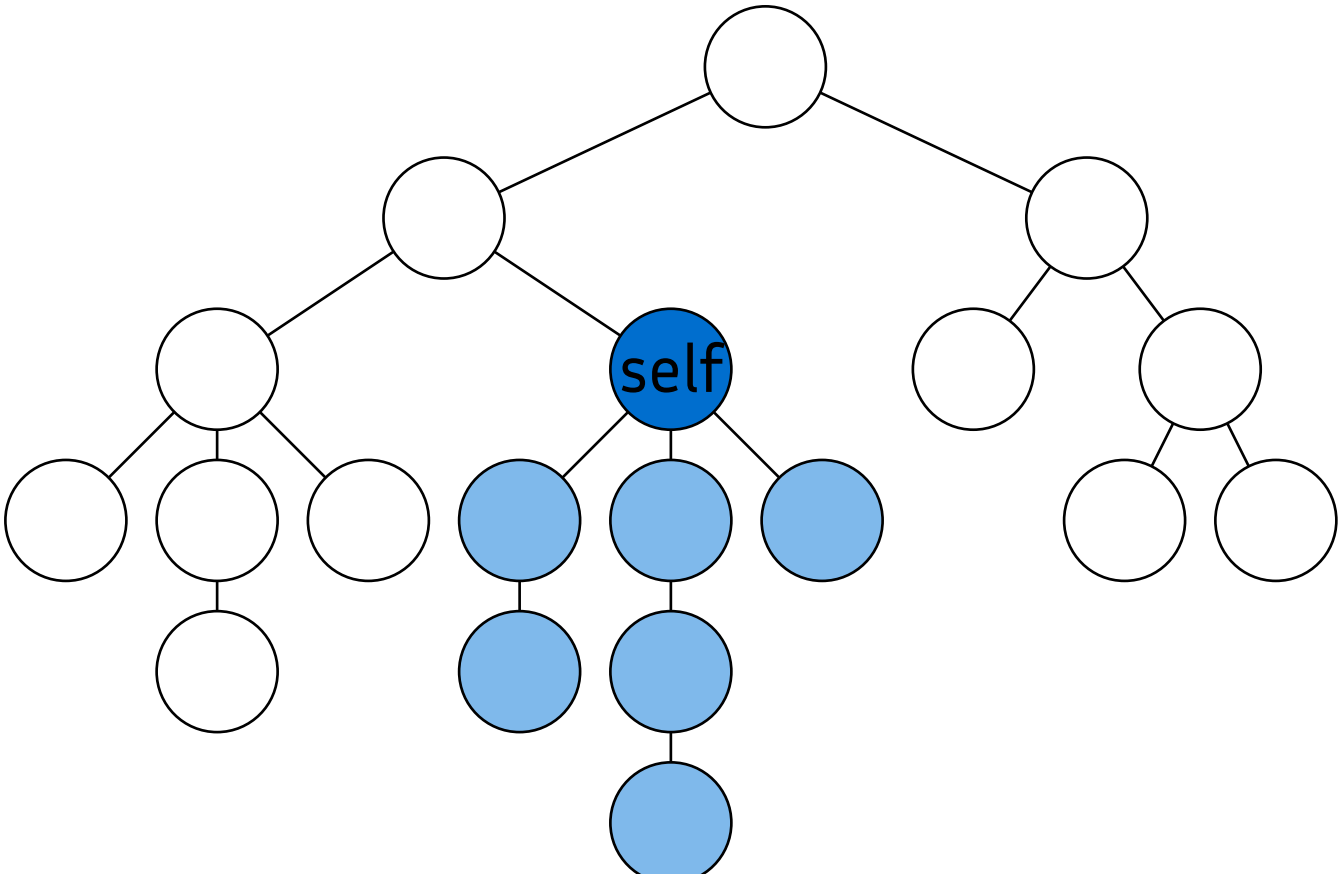
```
let new = &*self;
```



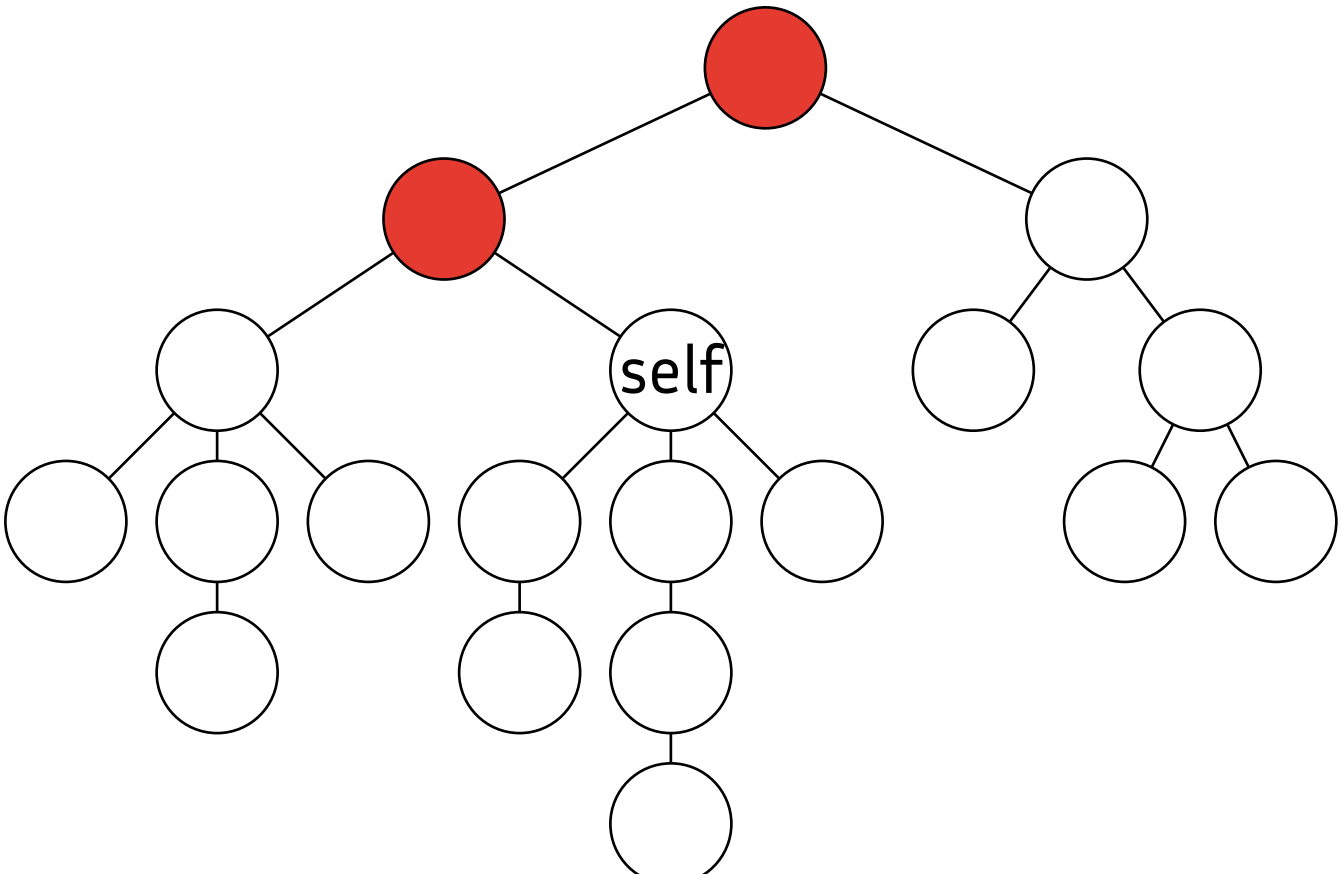
self & strict children
→ children



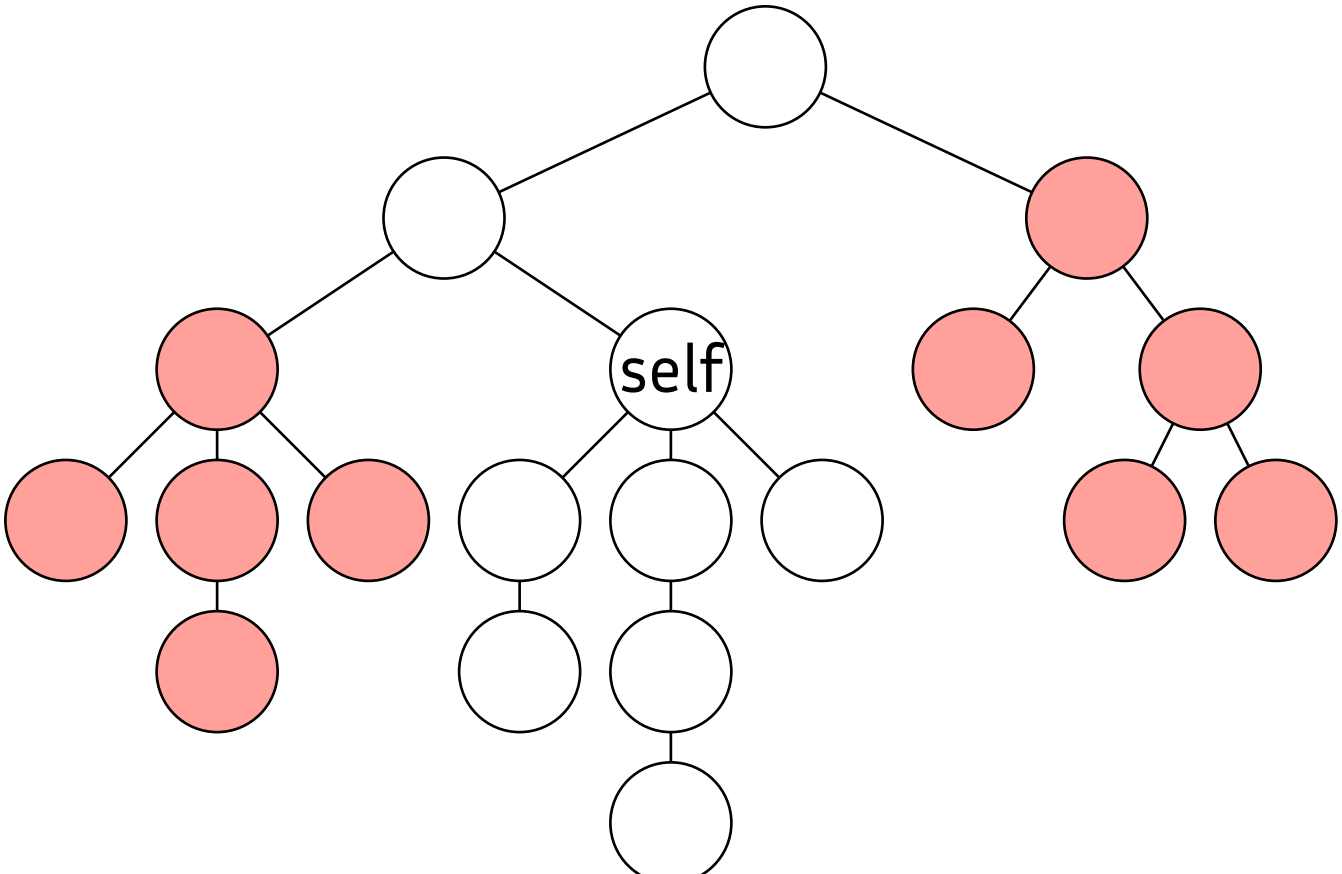
self & strict children
→ children



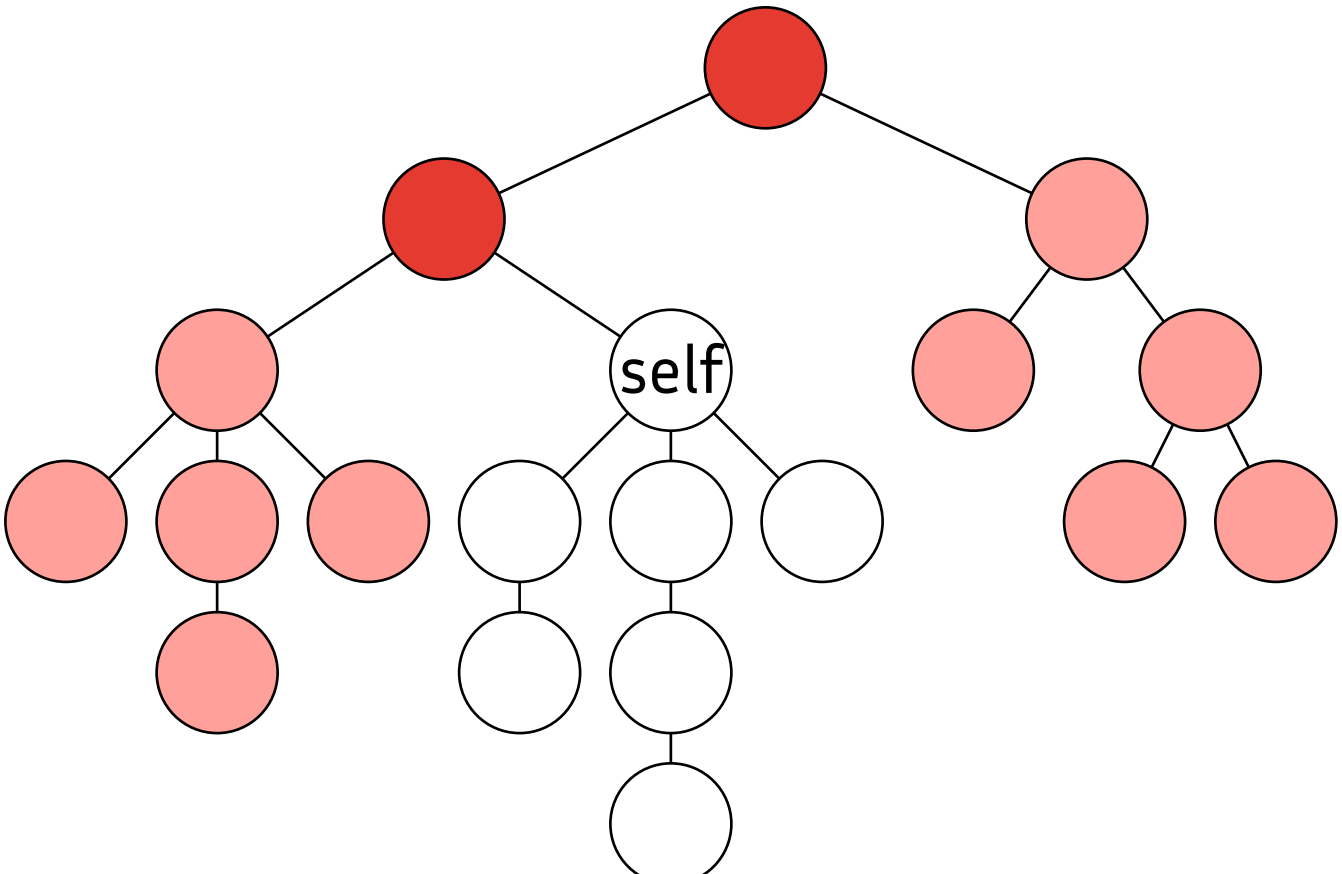
self & strict children
→ children



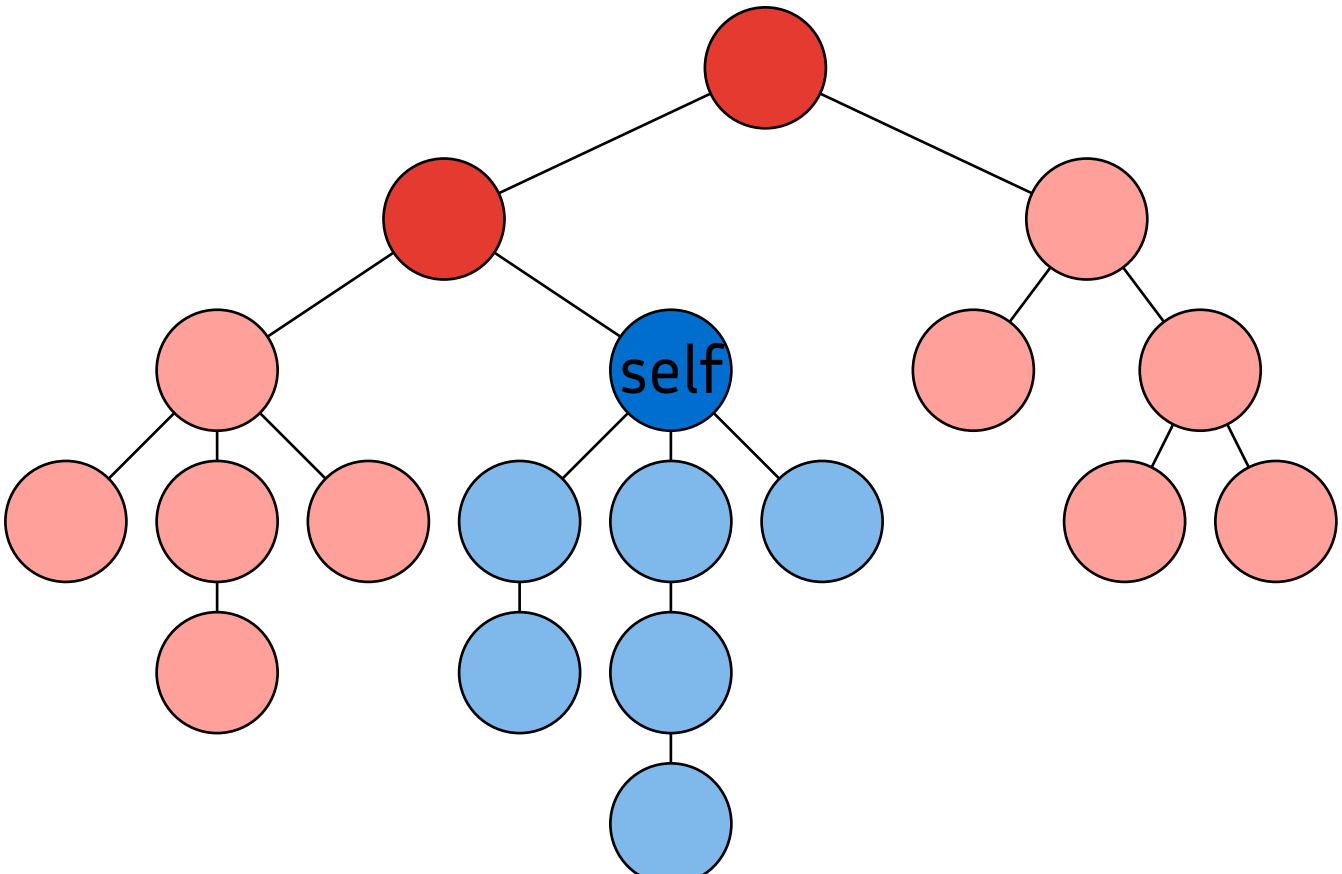
parents & cousins
→ foreign



parents & cousins
→ foreign



parents & cousins
→ foreign



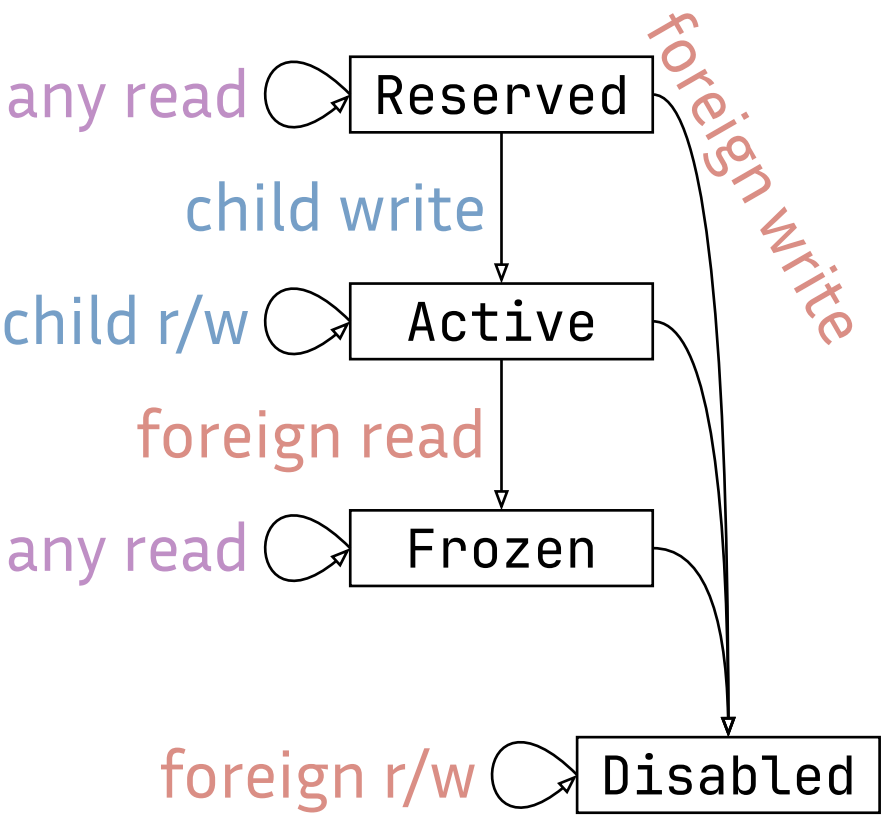
self & strict children
→ children

parents & cousins
→ foreign

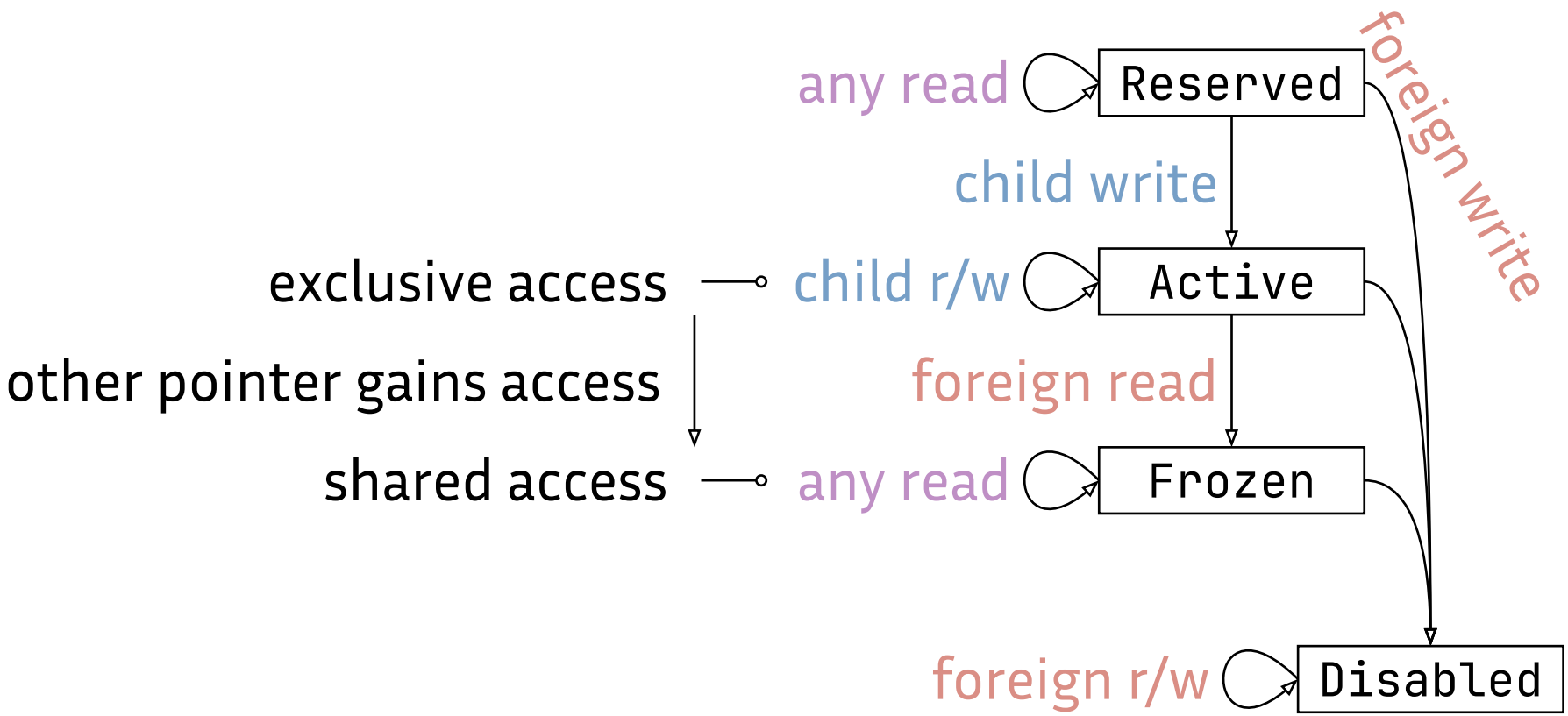
State machine

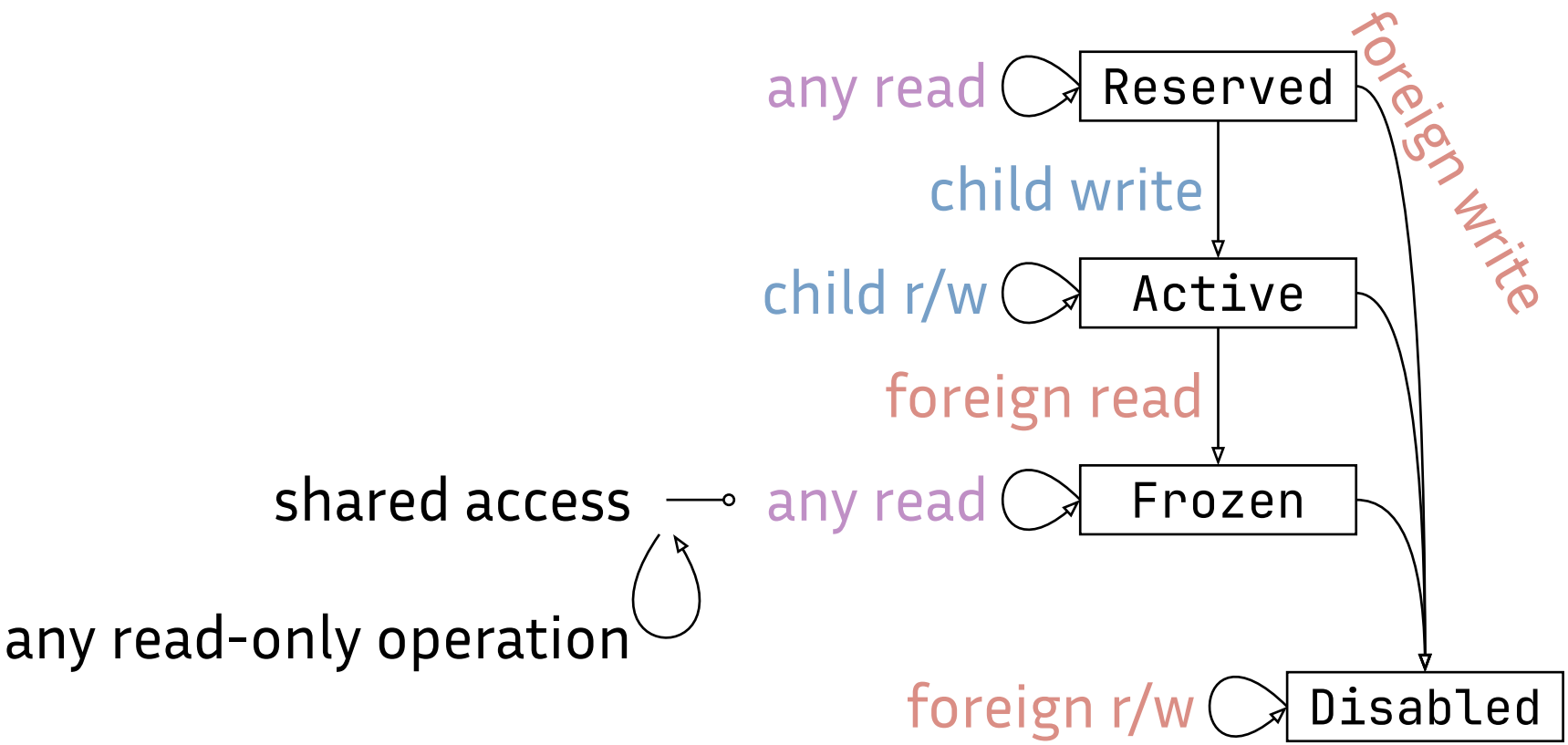
Per-location permission

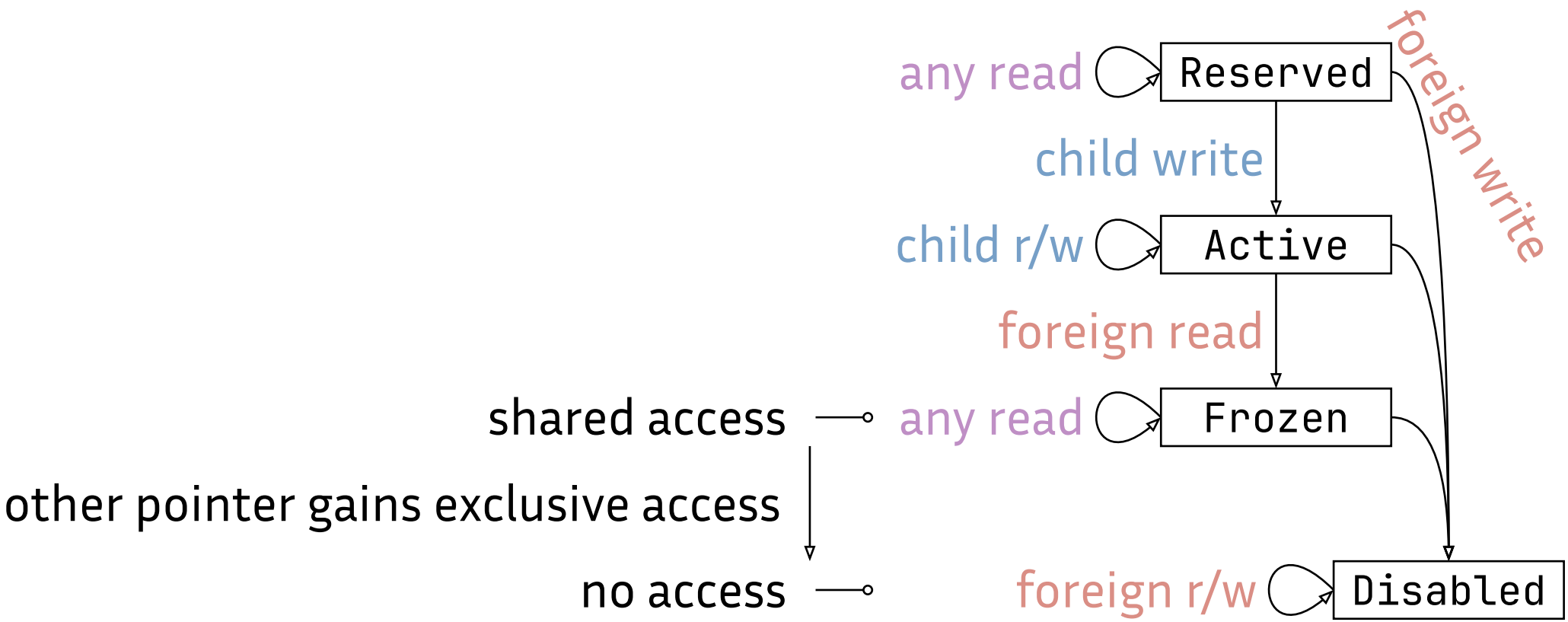
After creation each pointer experiences a sequence of child/foreign read/write accesses and gains/loses permissions in consequence


any read
Reserved
foreign write
child write
child r/w
Active
foreign read
any read
Frozen
foreign r/w
Disabled


&mut not yet written to
write to it
&mut already written
any read
child write
child r/w
foreign read
any read
foreign r/w
foreign write
Reserved
Active
Frozen
Disabled


any read
Reserved
foreign write
child write
exclusive access
child r/w
Active
other pointer gains access
foreign read
shared access
any read
Frozen
foreign r/w
Disabled


any read
Reserved
foreign write
child write
child r/w
Active
foreign read
shared access
any read
Frozen
any read-only operation
foreign r/w
Disabled

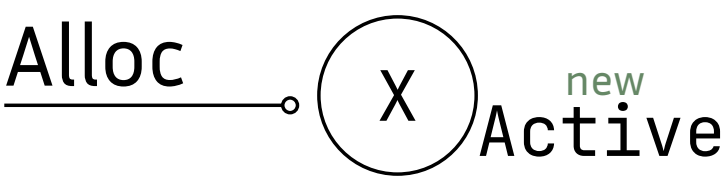

any read
Reserved
foreign write
child write
child r/w
Active
foreign read
shared access
any read
Frozen
other pointer gains exclusive access
no access
foreign r/w
Disabled

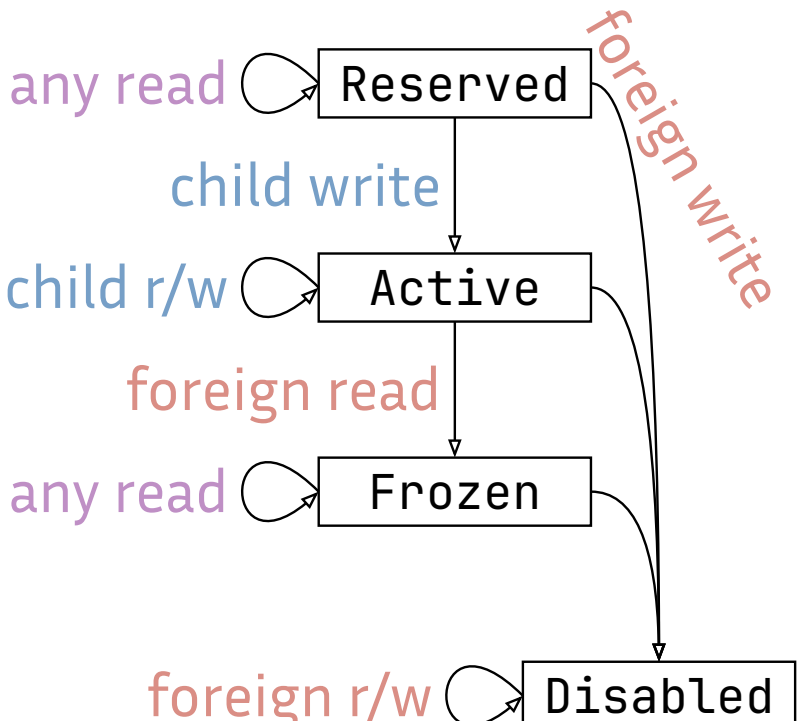
First example contains UB

```
static mut X = 0;
let y = &mut X;
let val = *y;
*y = 42;
print!(X); // read access violates uniqueness of y
*y = val;
```

Alloc x

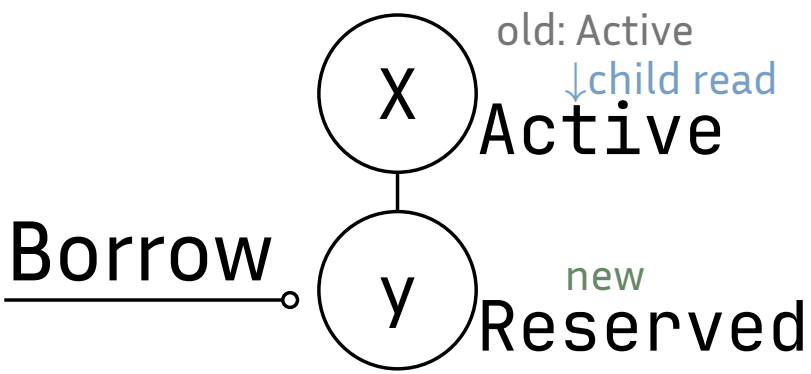
```
static mut X = 0;
let y = &mut X;
let val = *y;
*y = 42;
print!(X);
*y = val;
```

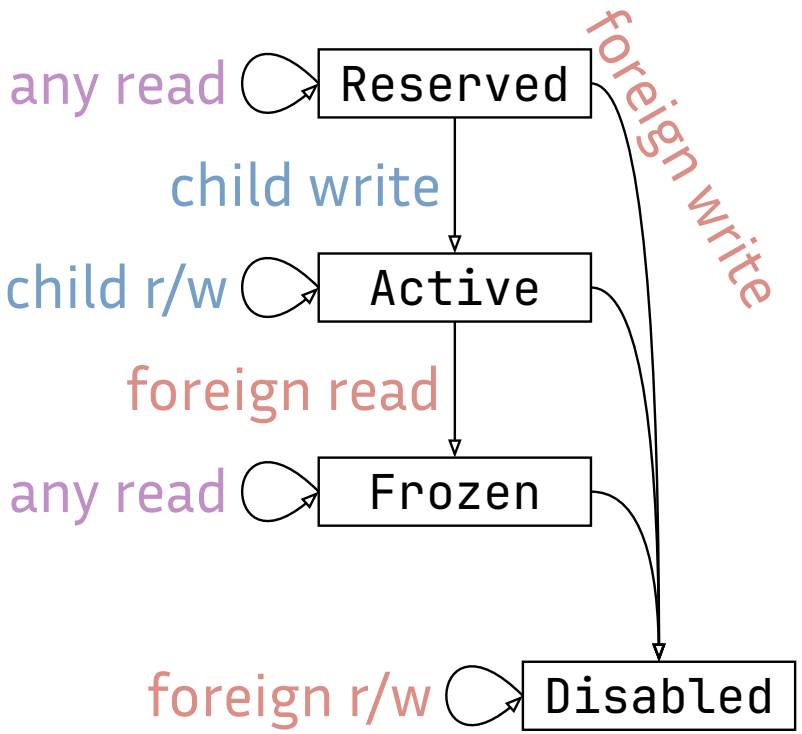




Borrow y

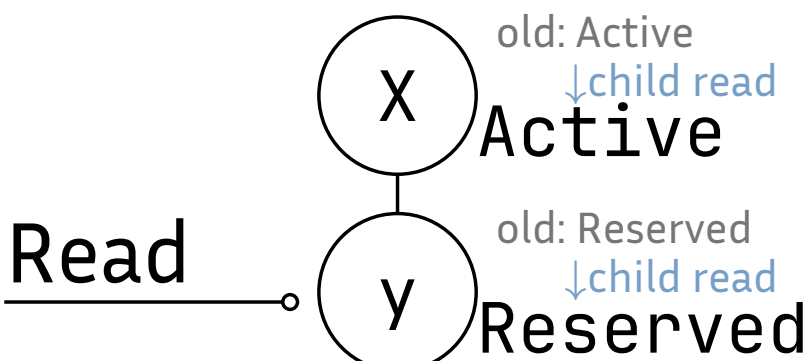
```
static mut X = 0;
let y = &mut X;
let val = *y;
*y = 42;
print!(X);
*y = val;
```

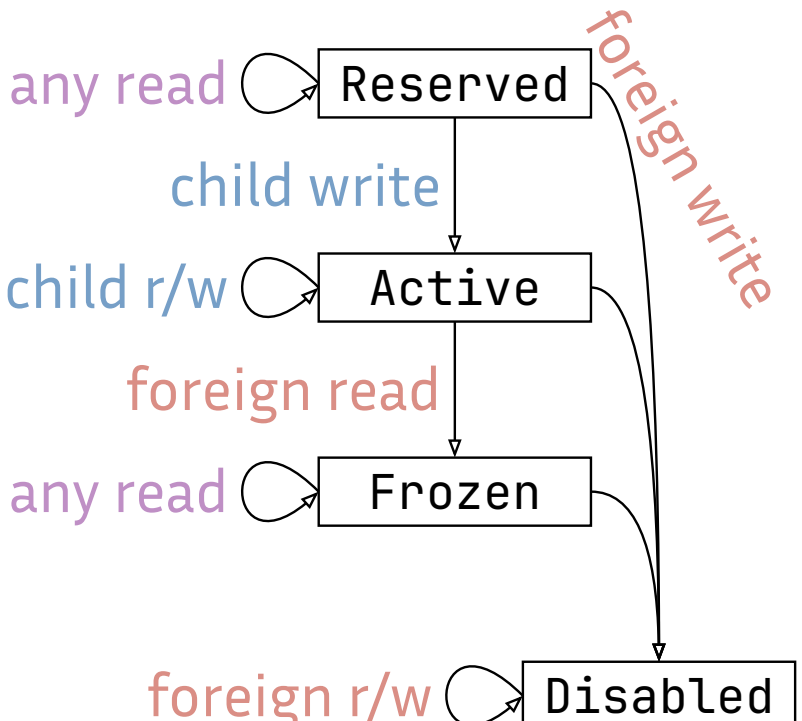




```
static mut X = 0;
let y = &mut X;
let val = *y;
*y = 42;
print!(X);
*y = val;
```

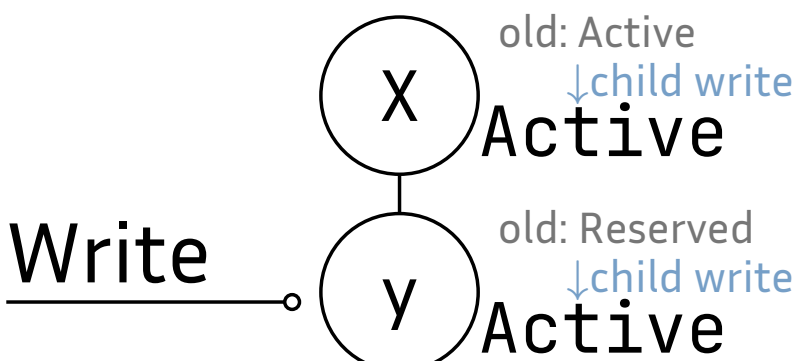
Read y

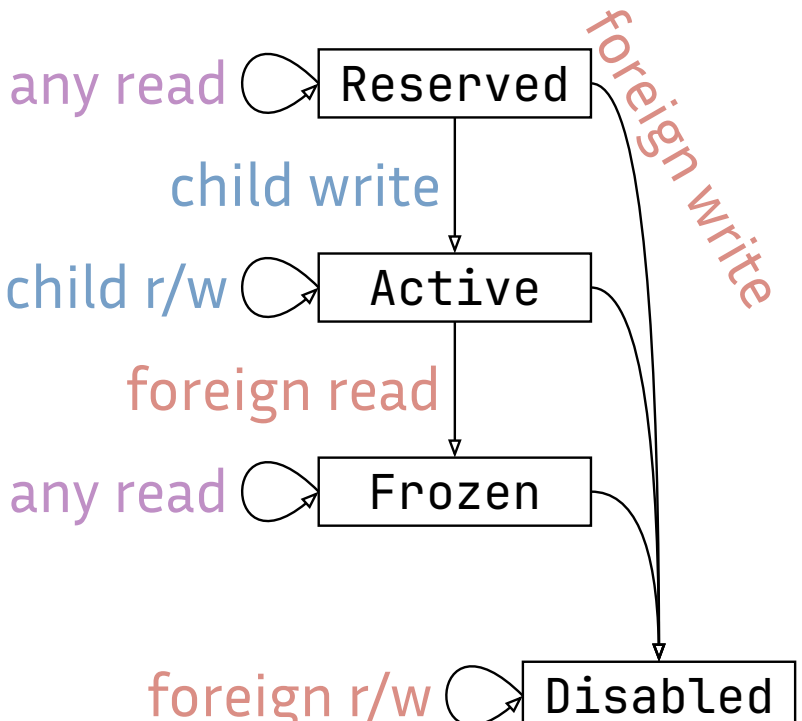




```
static mut X = 0;
let y = &mut X;
let val = *y;
*y = 42;
print!(X);
*y = val;
```

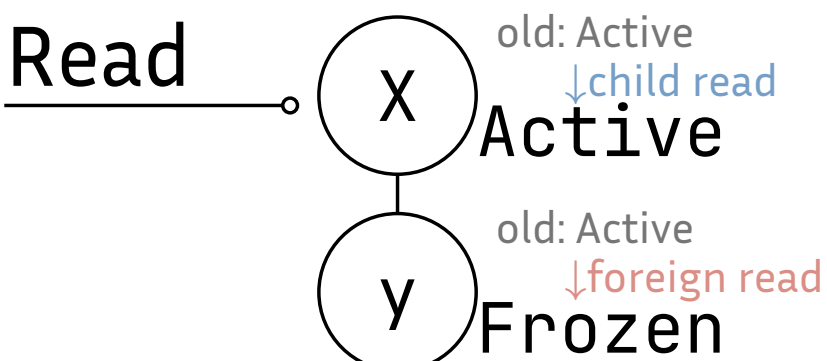
Write y

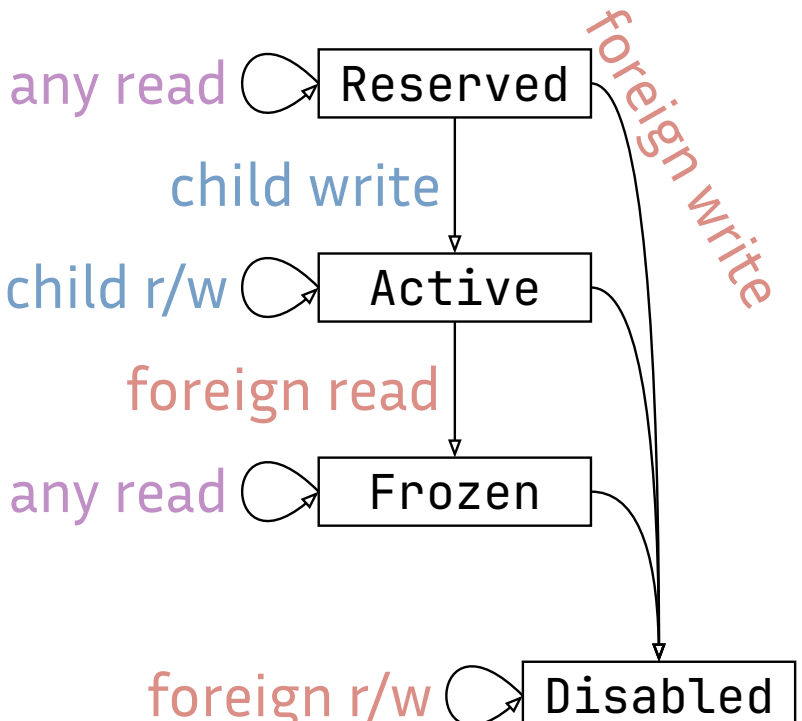




```
static mut X = 0;
let y = &mut X;
let val = *y;
*y = 42;
print!(X);
*y = val;
```

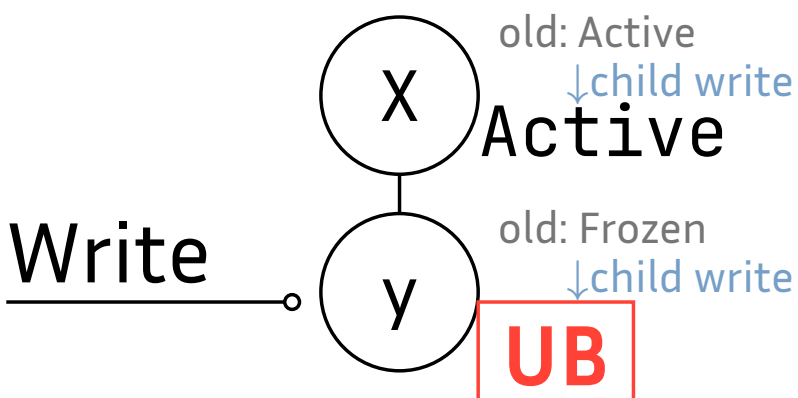
Read X

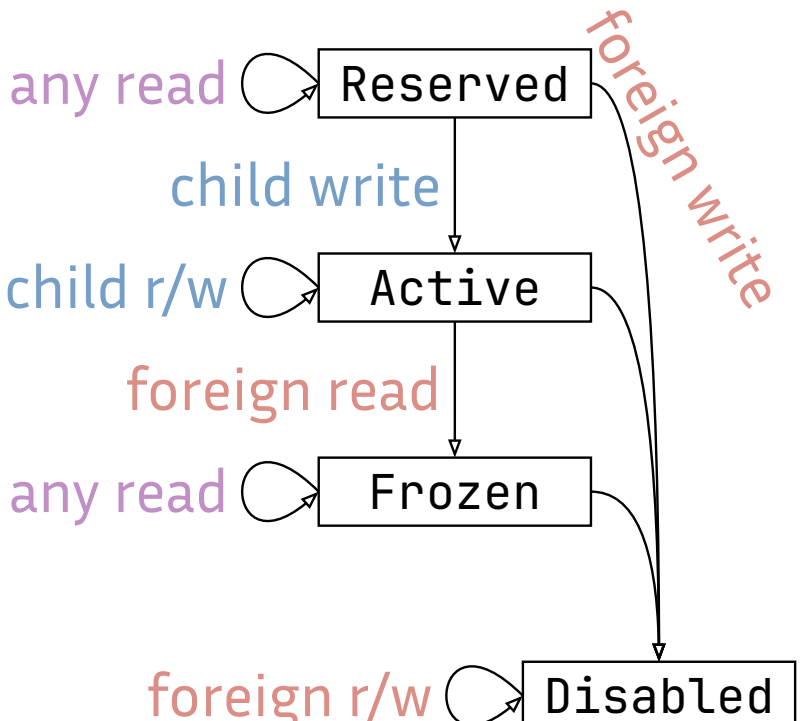




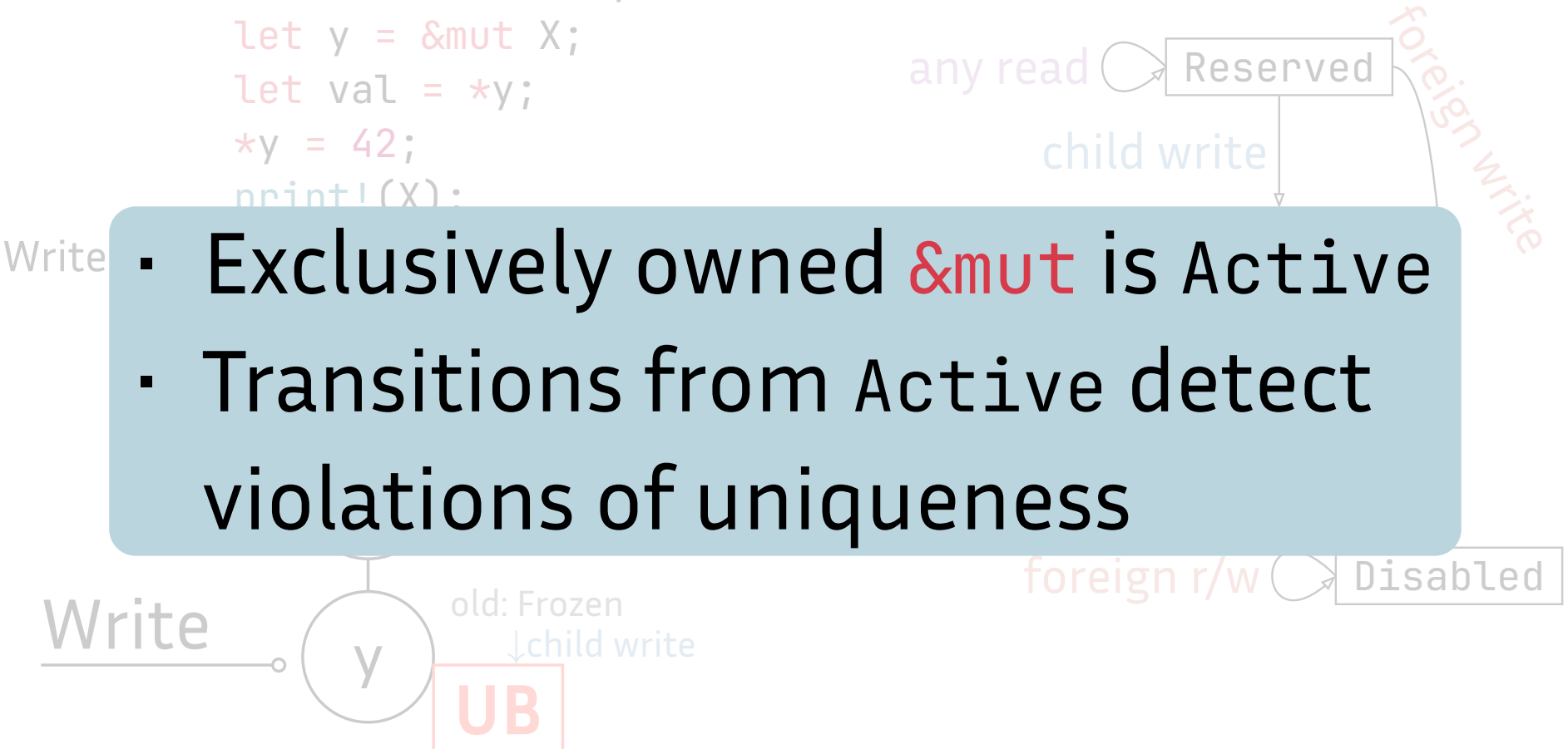
```
static mut X = 0;
let y = &mut X;
let val = *y;
*y = 42;
print!(X);
*y = val;
```

Write y





```
static mut X = 0;
let y = &mut X;
let val = *y;
*y = 42;
print!(X);
```



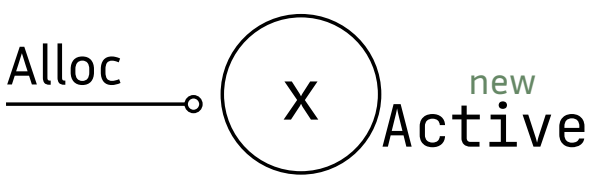
- Exclusively owned `&mut` is `Active`
- Transitions from `Active` detect violations of uniqueness

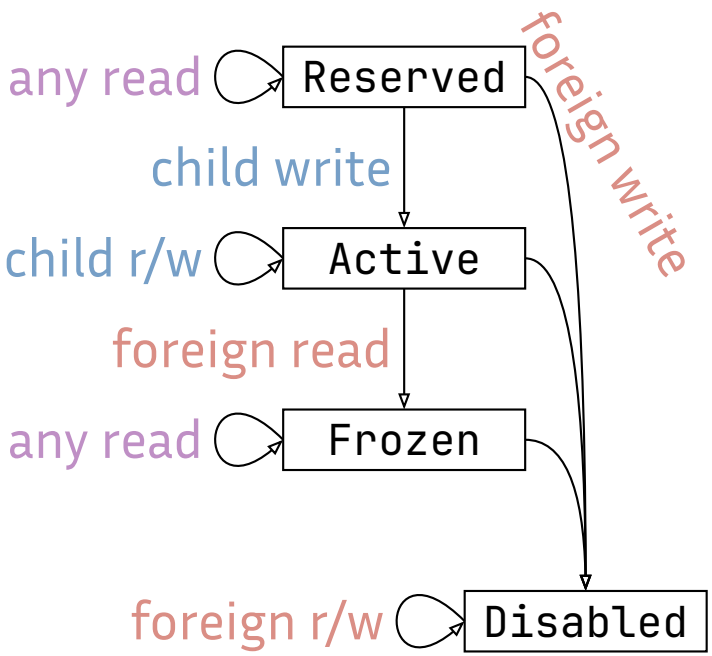
Raw pointers

```
let mut x = 0u64;
let r = addr_of_mut!(x);
x = 42; // x and r should be interchangeable
r.write(50);
```

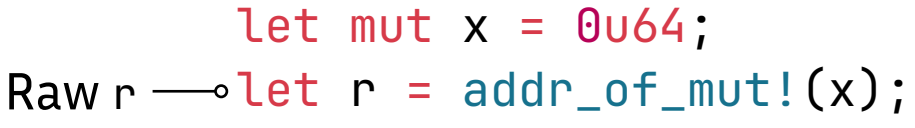
Alloc x

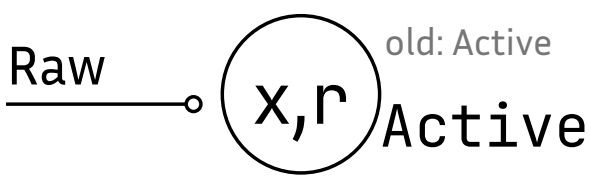
```
let mut x = 0u64;
let r = addr_of_mut!(x);
x = 42;
r.write(50);
```

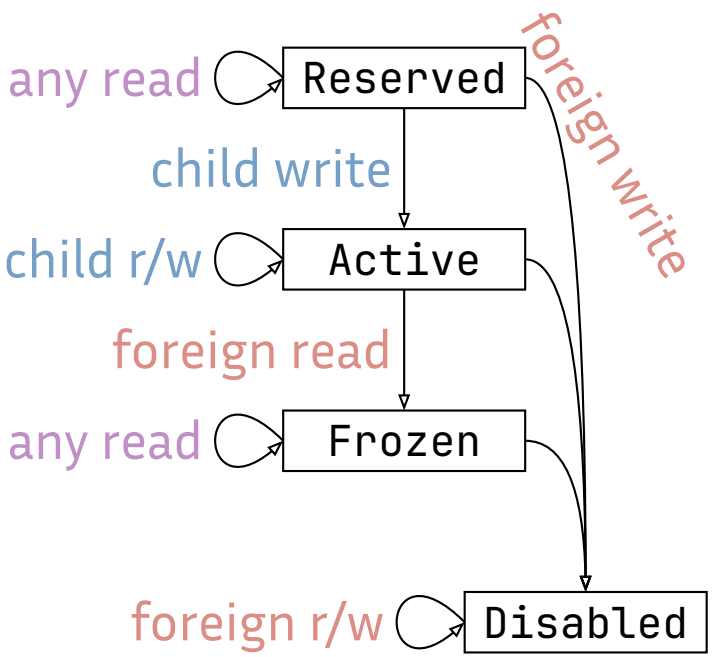




```
let mut x = 0u64;
let r = addr_of_mut!(x);
x = 42;
r.write(50);
```

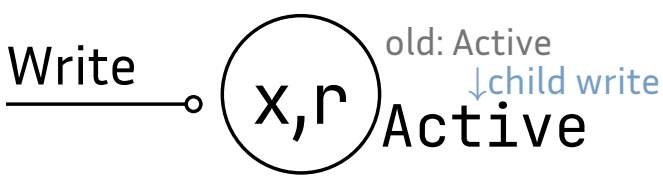


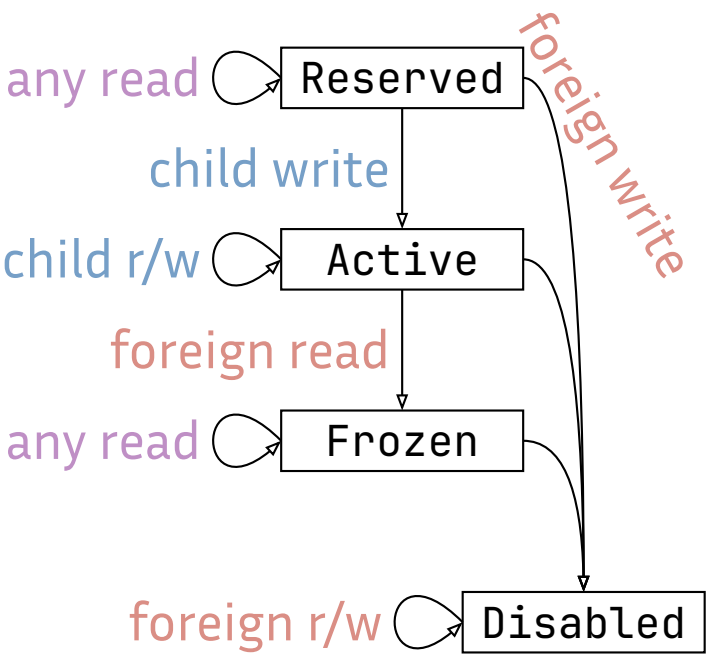




```
let mut x = 0u64;
let r = addr_of_mut!(x);
x = 42;
r.write(50);
```

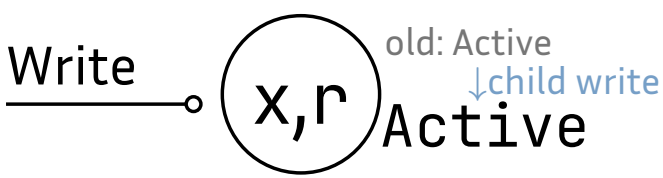
Write x

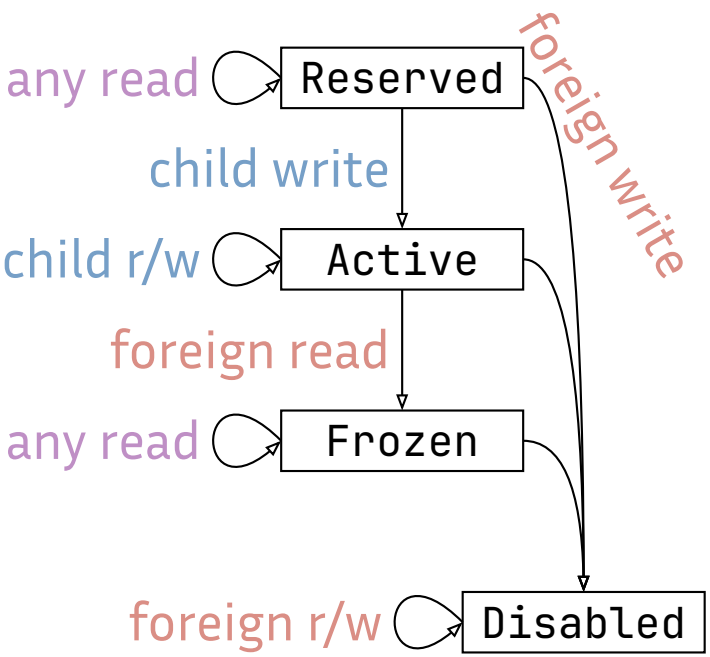




```
let mut x = 0u64;
let r = addr_of_mut!(x);
x = 42;
r.write(50);
```

Write r





```
let mut x = 0u64;
let r = addr_of_mut!(x);
x = 42;
r.write(50);
```

Write r

- Raw pointers inherit tag (and permissions with it)
- Same approach for interior mutability

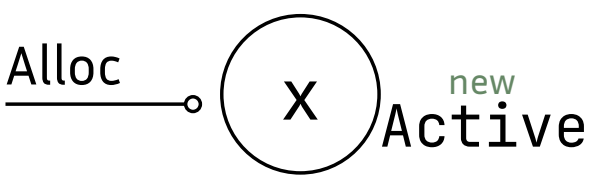
All mutable references are two-phase borrows

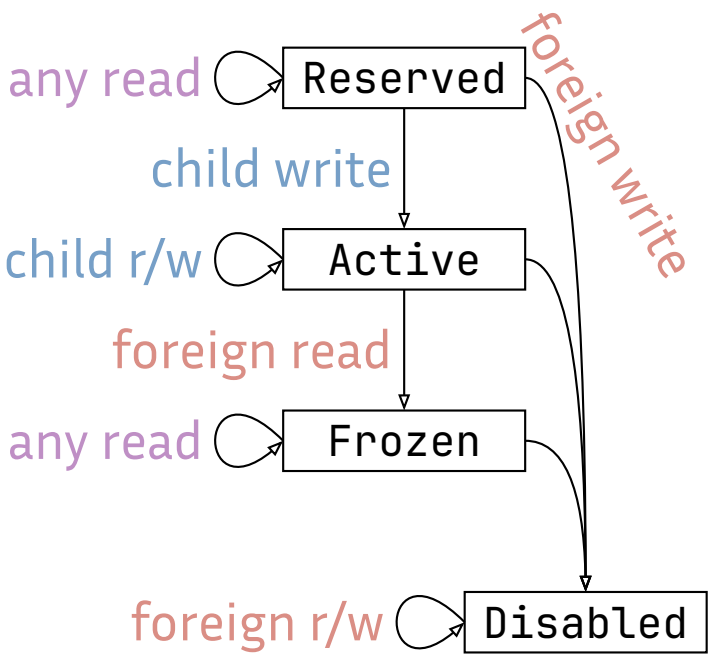
```rust
let mut x = 0u64;
let y = &*addr_of!(x);
let z = &mut x;  // Create mutable reference
let v = read(y);
write(z, 42);    // Use it mutably
```

```
let mut x = 0u64;
let y = &*addr_of!(x);
let z = &mut x;
let v = read(y); // Read accesses still allowed
write(z, 42);
```

Alloc x

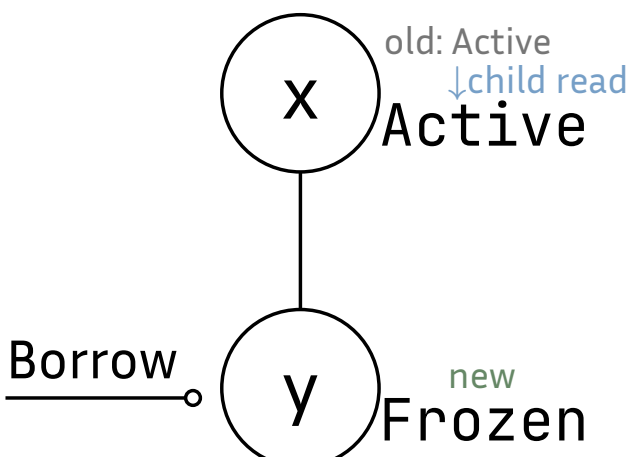
```
let mut x = 0u64;
let y = &*addr_of!(x);
let z = &mut x;
let v = read(y);
write(z, 42);
```

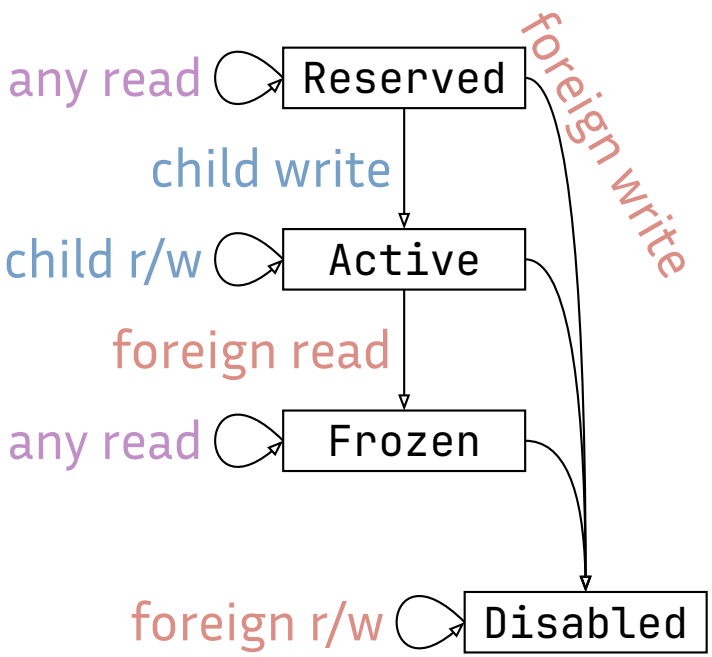




```
let mut x = 0u64;
let y = &*addr_of!(x);
let z = &mut x;
let v = read(y);
write(z, 42);
```

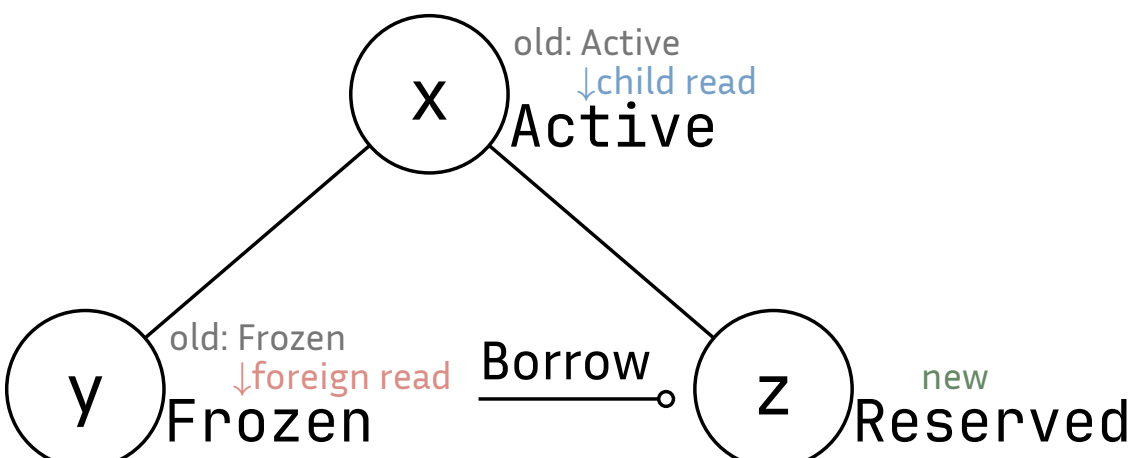
Borrow y

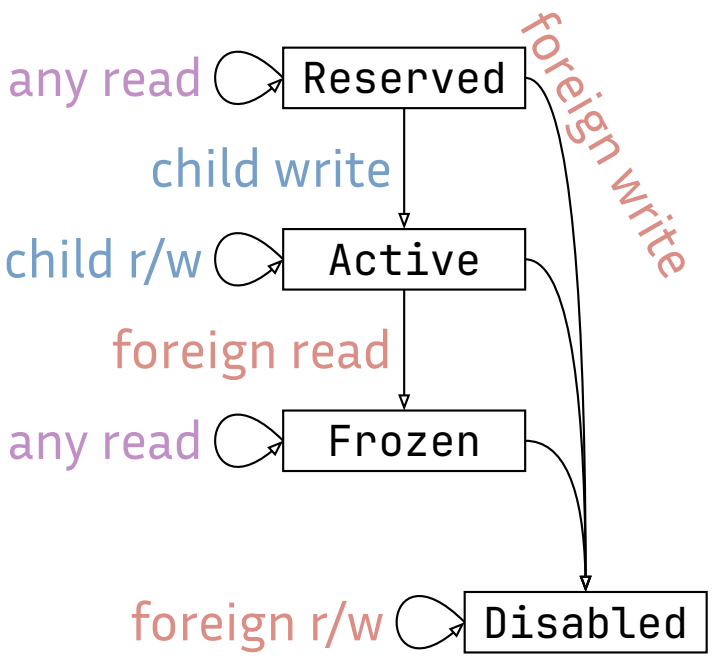




```
let mut x = 0u64;
let y = &*addr_of!(x);
let z = &mut x;
let v = read(y);
write(z, 42);
```

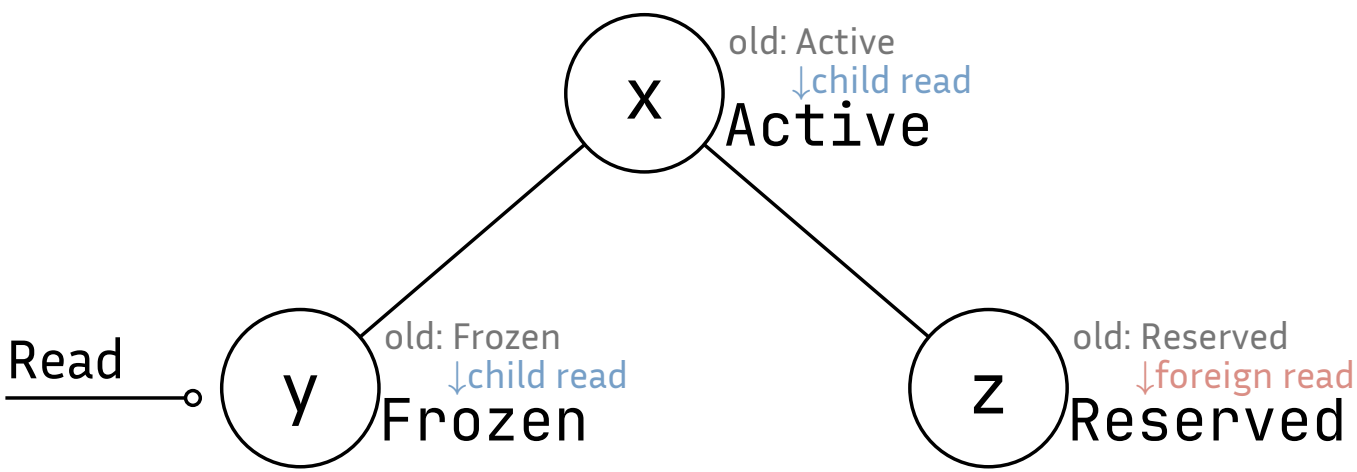
Borrow z

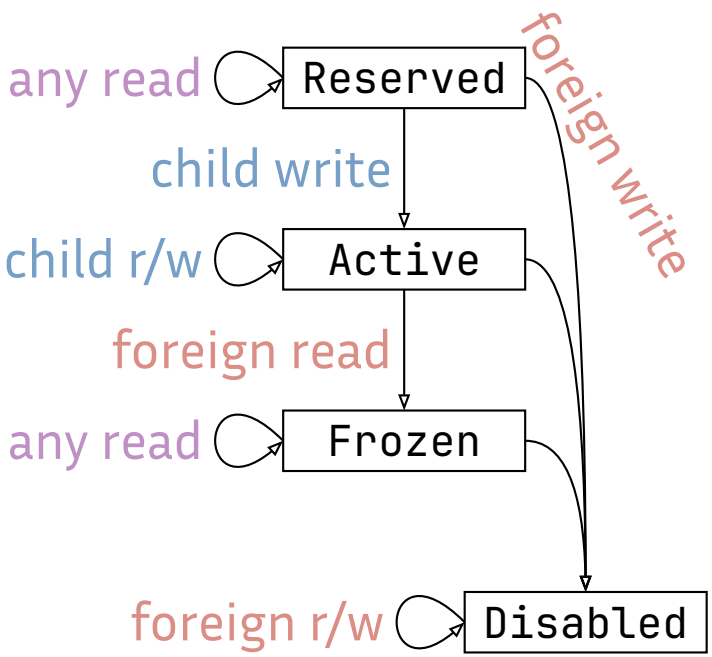




```
let mut x = 0u64;
let y = &*addr_of!(x);
let z = &mut x;
let v = read(y);
write(z, 42);
```

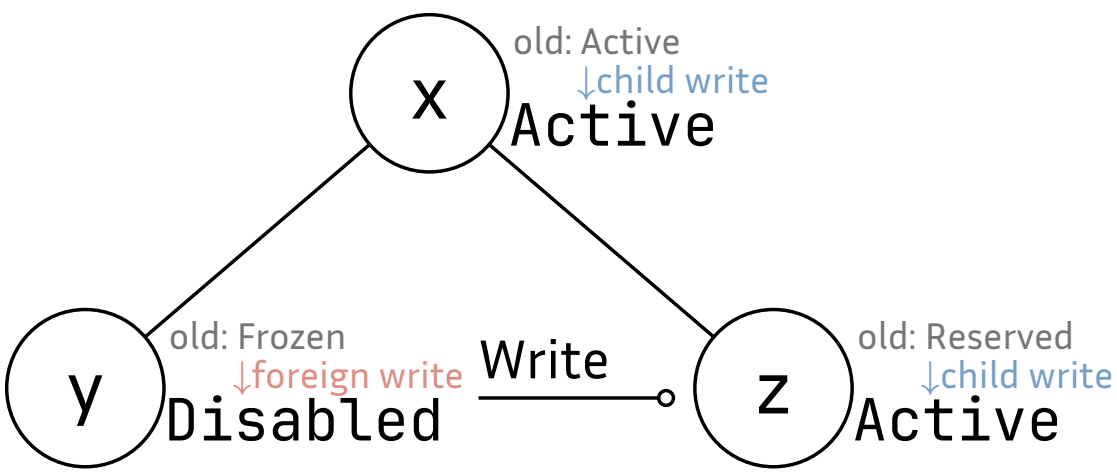
Read z

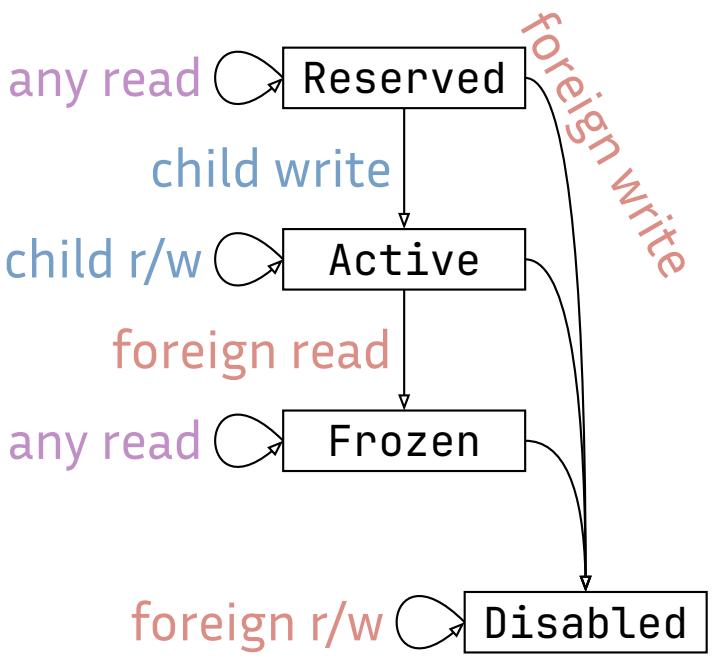




```
let mut x = 0u64;
let y = &*addr_of!(x);
let z = &mut x;
let v = read(y);
write(z, 42);
```

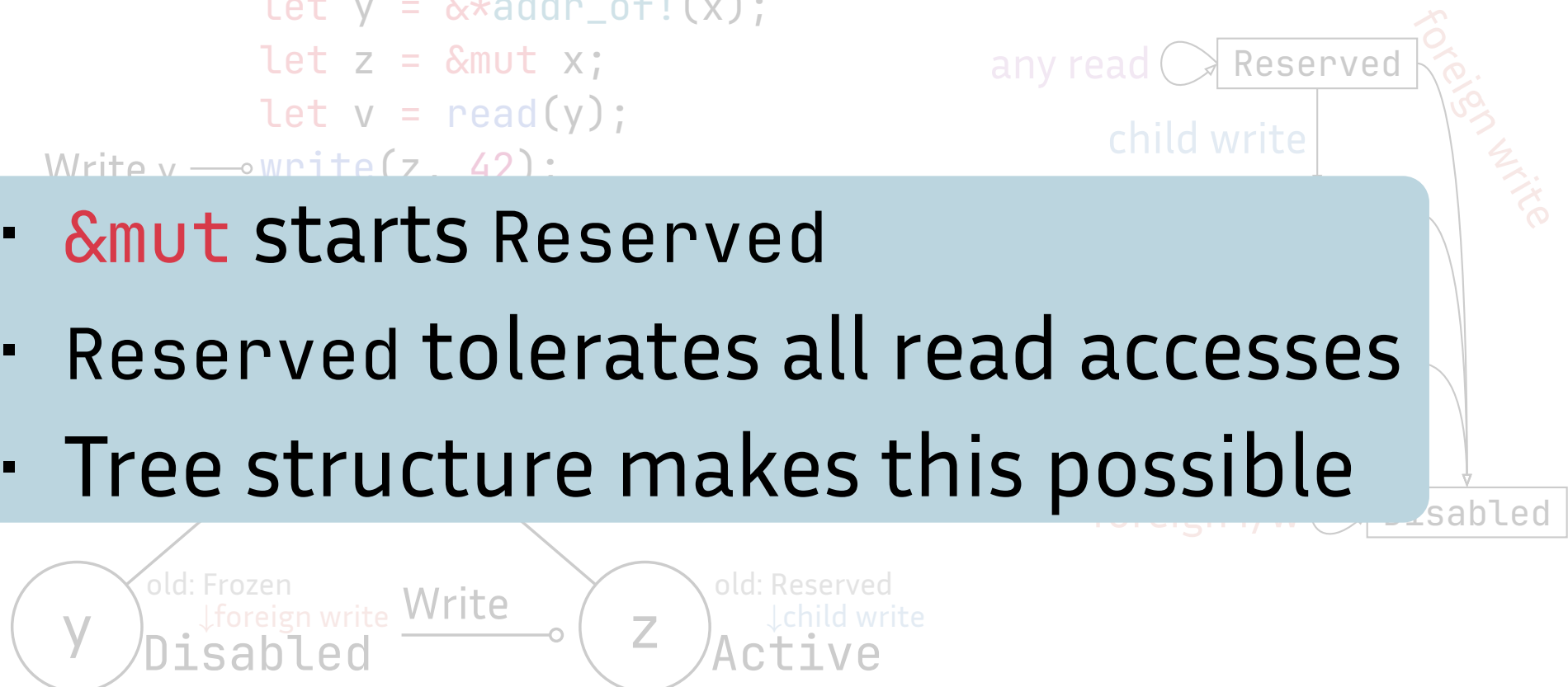
Write y





```
let mut x = 0u64;
let y = &*addr_of!(x);
let z = &mut x;
let v = read(y);
write(z, 42);
```

- `&mut` starts `Reserved`
- `Reserved` tolerates all read accesses
- Tree structure makes this possible



Conclusion

Learn more:

`https://perso.crans.org/vanille/treebor/`

- protectors on function arguments
- no range restriction on reborrow

Try it out:

`https://github.com/rust-lang/miri`

- use the flag `-Zmiri-tree-borrows`
- **test** your unsafe code, **report** any surprises!