

Tree Borrows

Neven Villani,¹ Johannes Hostert,² Derek Dreyer,³ Ralf Jung²

PLDI'25

2025-06-19

¹Univ. Grenoble Alpes, Verimag

²ETH Zurich

³MPI-SWS

Rust's type system provides powerful optimizations

```
fn write_both(x: &mut i32, y: &mut i32) -> i32 {  
  
    *x = 13;  
  
    *y = 20;  
  
    *x  
  
}
```

Rust's type system provides powerful optimizations

```
fn write_both(x: &mut i32, y: &mut i32) -> i32 {  
    *x = 13;  
    *y = 20;  
    *x  
}
```

mutable thus disjoint

*x is unchanged

Rust's type system provides powerful optimizations

```
fn write_both(x: &mut i32, y: &mut i32) -> i32 {  
    *x = 13;  
    *y = 20;  
    *x  
}
```

mutable thus disjoint

*x has known value

*x is unchanged

always returns 13

Rust's type system provides powerful optimizations

```
fn write_both(x: &mut i32, y: &mut i32) -> i32 {  
  
    *x = 13;  
  
    *y = 20;  
  
    13 // formerly *x: one fewer load from memory  
  
}
```

Type-level guarantees for references



`&mut` → mutation, no aliasing

`&` → aliasing, no mutation

Escape hatch: **unsafe**

Can **bypass typechecks** to implement **low-level manipulations**

```
unsafe {  
    // Code within this block has relaxed typechecking  
    . . .  
}
```

Within **unsafe** it is **the programmer's responsibility** to check

- that pointers are non-null
- that memory is initialized
- ...

What if **unsafe** code violates a necessary invariant ?

```
fn write_both(x: &mut i32, y: &mut i32) -> i32 {  
    *x = 13;  
    *y = 20;  
    *x  
}
```

```
fn main() {  
    let mut root = 42;  
    let ptr = &raw mut root;  
    let x = unsafe { &mut *ptr };  
    let y = unsafe { &mut *ptr };  
    println!("{}", write_both(x, y));  
}
```


What if **unsafe** code violates a necessary invariant?

```
fn write_both(x: &mut i32, y: &mut i32) -> i32 {  
    *x = 13;  
    *y = 20;  
    *x  
}
```

```
fn main() {  
    let mut root = 42;  
    let ptr = &raw mut root;  
    let x = unsafe { &mut *ptr };  
    let y = unsafe { &mut *ptr };  
    println!("{}", write_both(x, y));  
}
```

What if **unsafe** code violates a necessary invariant ?

```
fn write_both(x: &mut i32, y: &mut i32) -> i32 {  
    *x = 13;  
    *y = 20;  
    *x  
}
```

```
fn main() {  
    let mut root = 42;  
    let ptr = &raw mut root;  
    let x = unsafe { &mut *ptr };  
    let y = unsafe { &mut *ptr };  
    println!("{}", write_both(x, y));  
}
```

Not the compiler's responsibility

Within **unsafe** it is **the programmer's responsibility** to check

- that pointers are non-null
- that memory is initialized
- compliance with aliasing rules **NEW!**

Tree Borrows (TB): defines those aliasing rules

Not the compiler's responsibility

Within `unsafe` it is the programmer's responsibility to check

- that pointers are non-null
- that memory is initialized

- **Sounds familiar?**

Stacked Borrows has the same purpose,
Tree Borrows is its successor.

Stacked Borrows (SB)

In safe Rust, the Borrow Checker makes borrows well-bracketed. Stacked Borrows extends the well-bracketedness to **unsafe**.

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;
```

Desired outcome: UB

In safe Rust, the Borrow Checker makes borrows well-bracketed. Stacked Borrows extends the well-bracketedness to **unsafe**.

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;
```

A diagram showing a memory stack. It consists of a single rectangular box with the word "root" inside. The box has a double-line border, with the top line being thicker than the bottom line.

- new stack at root

Desired outcome: UB

In safe Rust, the Borrow Checker makes borrows well-bracketed. Stacked Borrows extends the well-bracketedness to **unsafe**.

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;
```

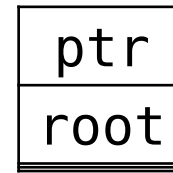
A diagram representing a memory stack. It consists of a single rectangular box with the word "root" inside. The box has a double-line border, with the top line being thicker than the bottom line, suggesting it is the top of the stack.

- ✓ root is at the top

Desired outcome: UB

In safe Rust, the Borrow Checker makes borrows well-bracketed. Stacked Borrows extends the well-bracketedness to **unsafe**.

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;
```

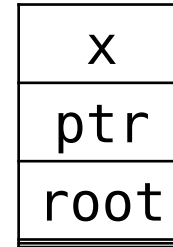


Desired outcome: UB

- ✓ root is at the top
- push ptr

In safe Rust, the Borrow Checker makes borrows well-bracketed. Stacked Borrows extends the well-bracketedness to **unsafe**.

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;
```

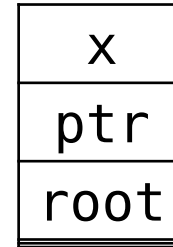


Desired outcome: UB

- ✓ ptr is at the top
- push x

In safe Rust, the Borrow Checker makes borrows well-bracketed. Stacked Borrows extends the well-bracketedness to **unsafe**.

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;
```

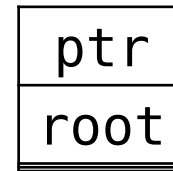


- pop until ptr is at the top

Desired outcome: UB

In safe Rust, the Borrow Checker makes borrows well-bracketed. Stacked Borrows extends the well-bracketedness to **unsafe**.

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;
```

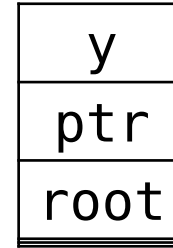


- pop until ptr is at the top

Desired outcome: UB

In safe Rust, the Borrow Checker makes borrows well-bracketed. Stacked Borrows extends the well-bracketedness to **unsafe**.

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;
```

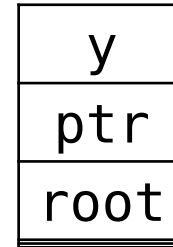


Desired outcome: UB

- pop until ptr is at the top
- push y

In safe Rust, the Borrow Checker makes borrows well-bracketed. Stacked Borrows extends the well-bracketedness to **unsafe**.

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;
```



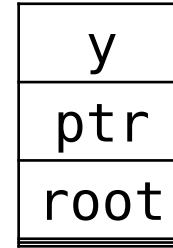
- search for x

Desired outcome: UB

In safe Rust, the Borrow Checker makes borrows well-bracketed. Stacked Borrows extends the well-bracketedness to **unsafe**.

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;
```

UB!



Can't use x if it is not in the stack

Desired outcome: UB

SB was **implemented** in Miri (official interpreter and UB detector)

→ included in many projects' CI

→ several bugs detected (e.g. in stdlib)

SB was **implemented** in Miri (official interpreter and UB detector)

→ included in many projects' CI

→ several bugs detected (e.g. in stdlib)

However Stacked Borrows is too strict

- analysis of 30 000 libraries
- 6000+ tests that should work are declared UB

Tree Borrows allows much more code

Stacked Borrows (SB)

Tree Borrows uses a **tree** instead of a stack to track borrows

Out of 30 000 most downloaded libraries,
> **50% fewer** tests with aliasing UB when using Tree Borrows

Tree Borrows allows much more code

Stacked Borrows (SB)

Tree Borrows uses a **tree** instead of a stack to track borrows

Out of 30 000 most downloaded libraries,
> **50% fewer** tests with aliasing UB when using Tree Borrows

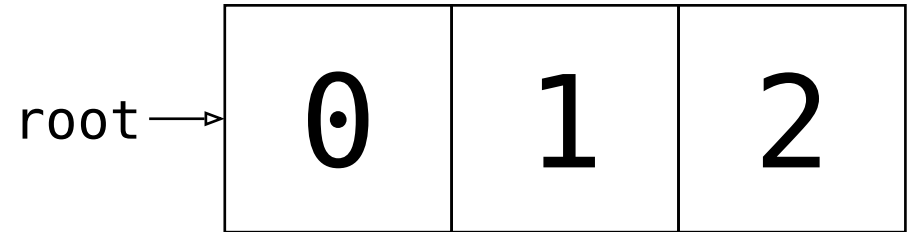
fixes known technical limitations of TB, incl. **handling of ranges**

From Stacks to Trees

```
let mut root = vec![0, 1, 2];  
let x0 = &raw mut root[0];  
let x2 = &raw mut root[2];
```

Desired outcome: not UB

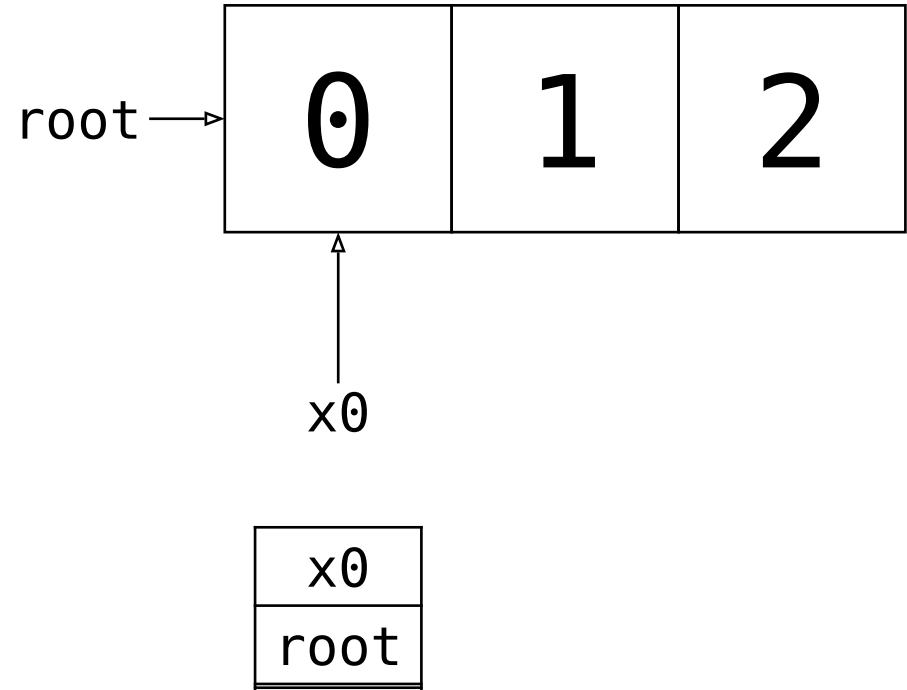
```
let mut root = vec![0, 1, 2];  
let x0 = &raw mut root[0];  
let x2 = &raw mut root[2];
```



Desired outcome: not UB

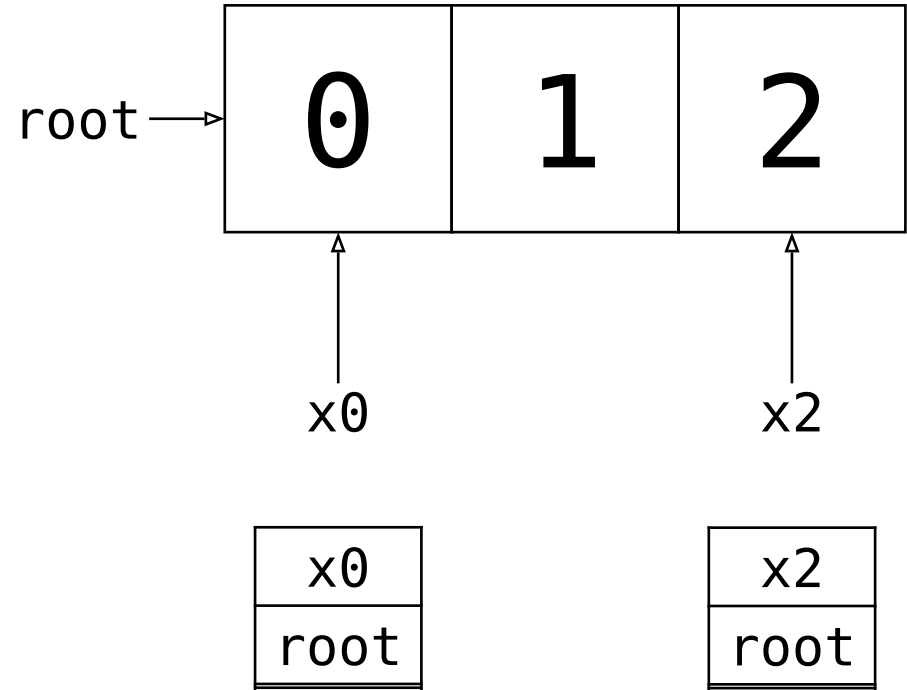
```
let mut root = vec![0, 1, 2];  
let x0 = &raw mut root[0];  
let x2 = &raw mut root[2];
```

Desired outcome: not UB

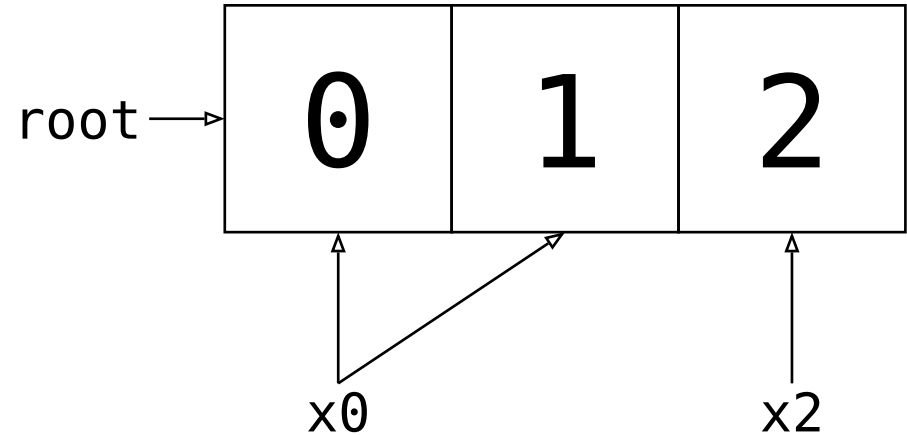


```
let mut root = vec![0, 1, 2];  
let x0 = &raw mut root[0];  
let x2 = &raw mut root[2];
```

Desired outcome: not UB




```
let mut root = vec![0, 1, 2];  
let x0 = &raw mut root[0];  
let x2 = &raw mut root[2];  
  
// Scenario 1  
let v1 = *x0.add(1);
```



root

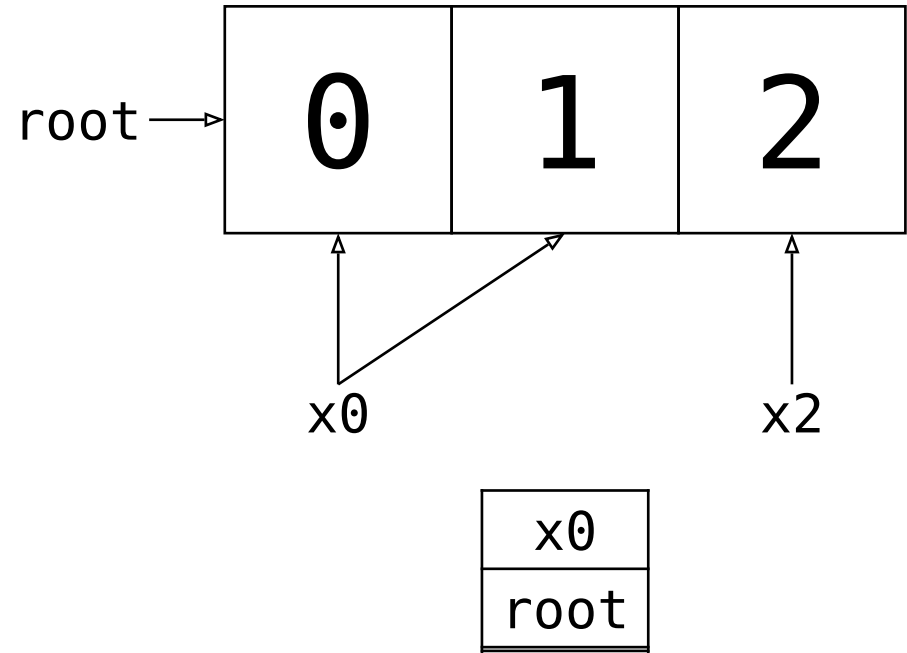
Desired outcome: not UB

```
let mut root = vec![0, 1, 2];  
let x0 = &raw mut root[0];  
let x2 = &raw mut root[2];
```

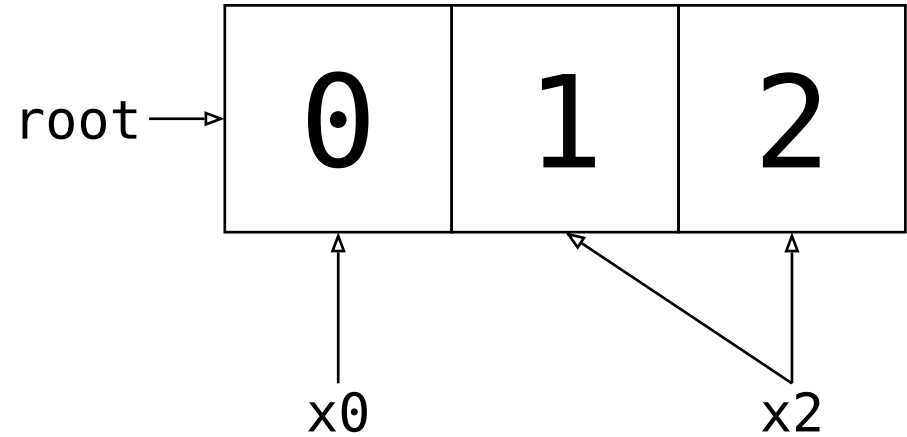
```
// Scenario 1
```

```
let v1 = *x0.add(1);
```

Desired outcome: not UB



```
let mut root = vec![0, 1, 2];  
let x0 = &raw mut root[0];  
let x2 = &raw mut root[2];  
  
// Scenario 2  
let v1 = *x2.sub(1);
```



root

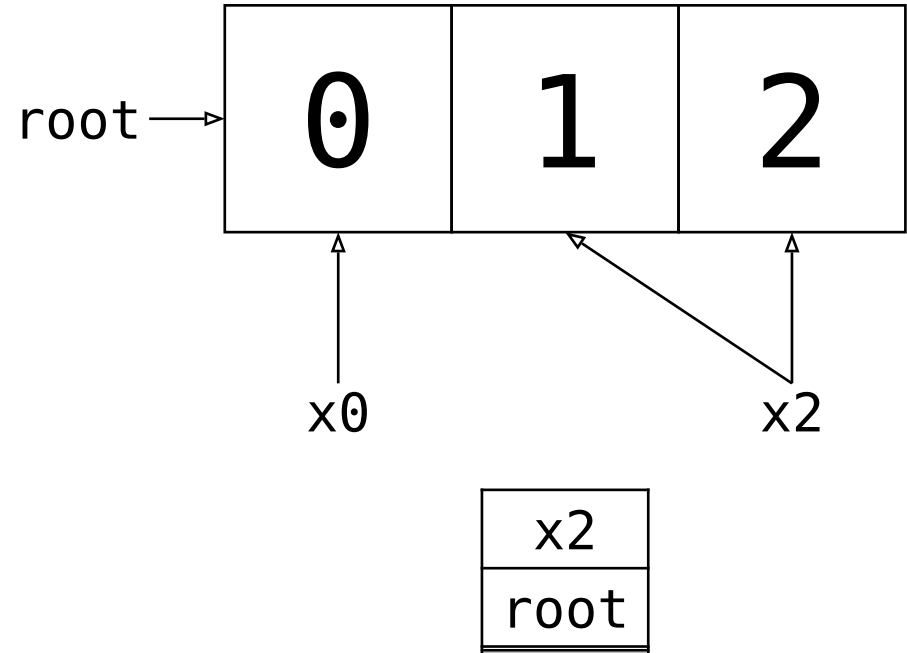
Desired outcome: not UB

```
let mut root = vec![0, 1, 2];  
let x0 = &raw mut root[0];  
let x2 = &raw mut root[2];
```

// Scenario 2

```
let v1 = *x2.sub(1);
```

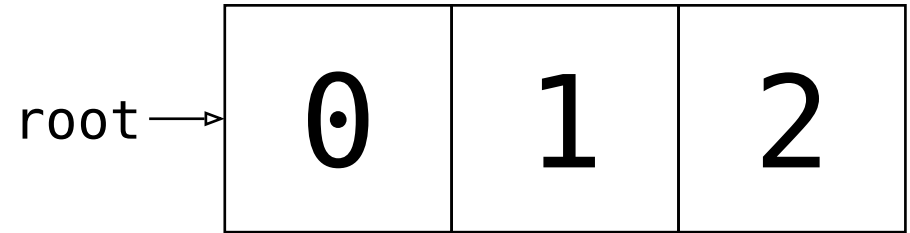
Desired outcome: not UB



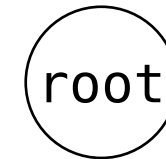
```
let mut root = vec![0, 1, 2];  
let x0 = &raw mut root[0];  
let x2 = &raw mut root[2];
```

Desired outcome: not UB

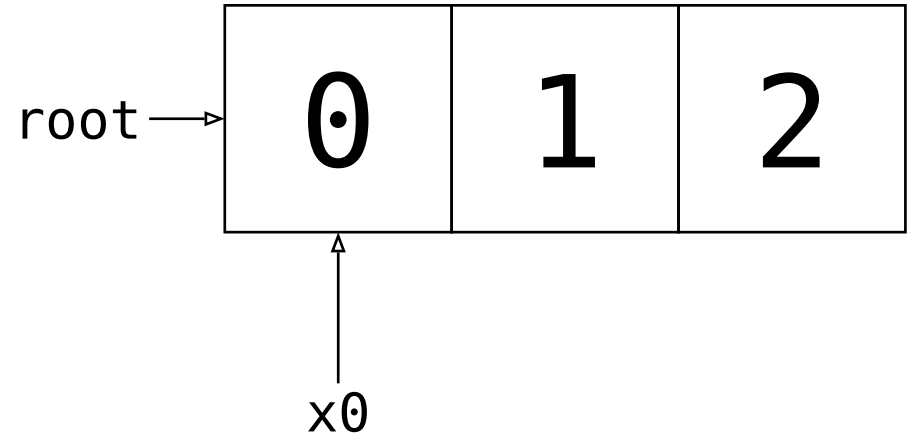
```
let mut root = vec![0, 1, 2];  
let x0 = &raw mut root[0];  
let x2 = &raw mut root[2];
```



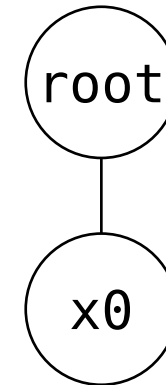
Desired outcome: not UB



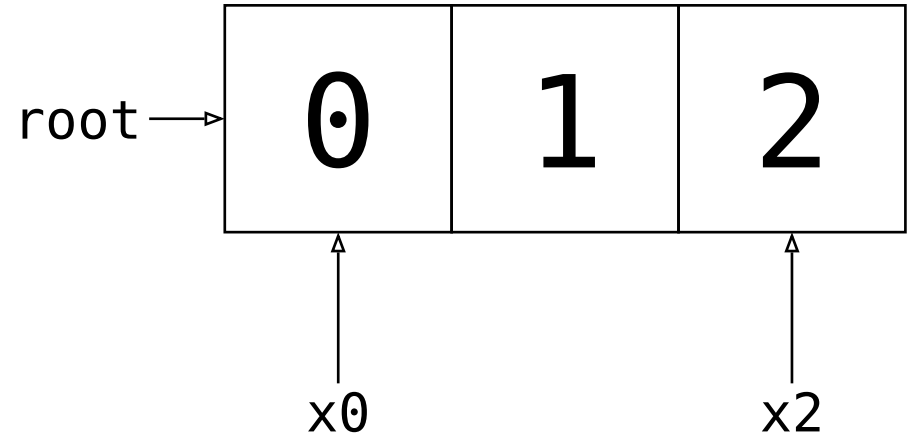
```
let mut root = vec![0, 1, 2];  
let x0 = &raw mut root[0];  
let x2 = &raw mut root[2];
```



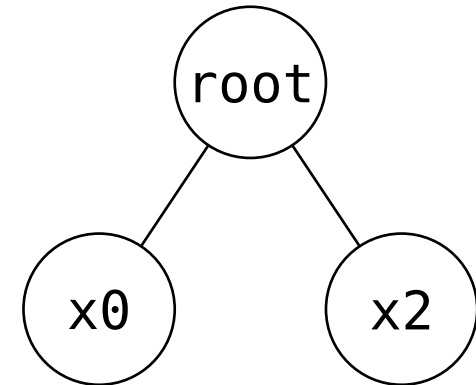
Desired outcome: not UB



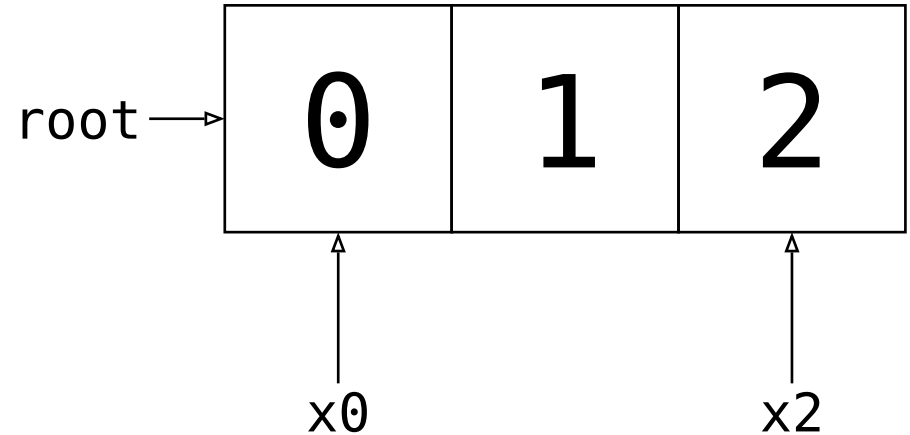
```
let mut root = vec![0, 1, 2];  
let x0 = &raw mut root[0];  
let x2 = &raw mut root[2];
```



Desired outcome: not UB

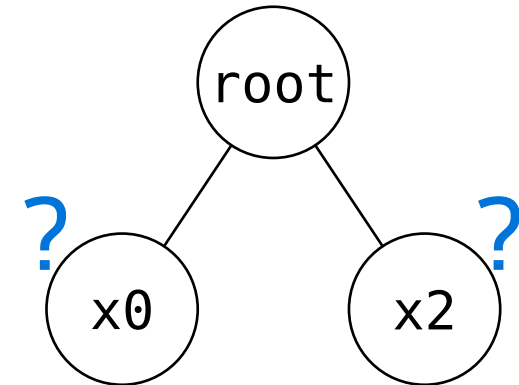



```
let mut root = vec![0, 1, 2];  
let x0 = &raw mut root[0];  
let x2 = &raw mut root[2];
```



Desired outcome: not UB

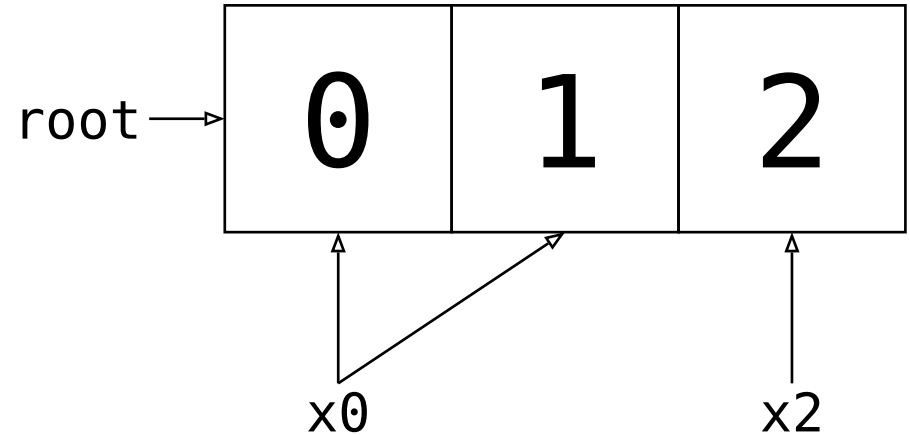
? “Reserved”



```
let mut root = vec![0, 1, 2];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];
```

// Scenario 1

```
let v1 = *x0.add(1);
```

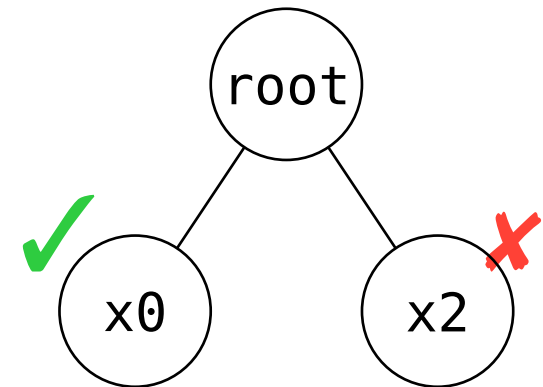


Desired outcome: not UB

? “Reserved”

✓ “Unique”

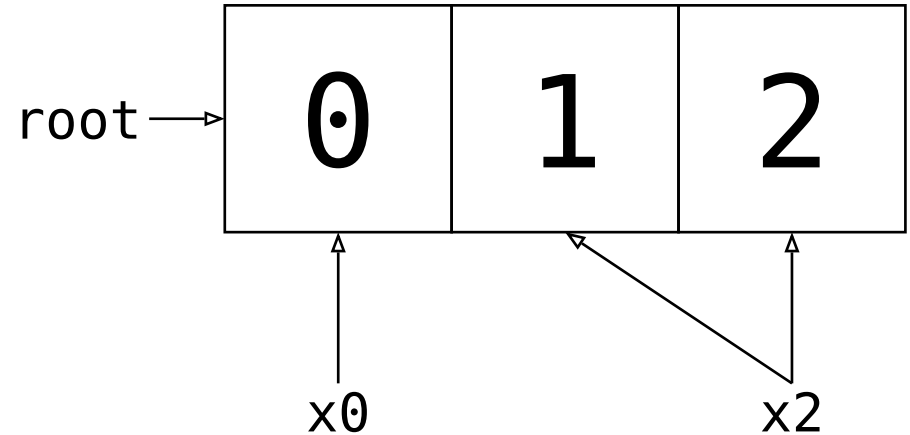
✗ “Disabled”



```
let mut root = vec![0, 1, 2];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];
```

// Scenario 2

```
let v1 = *x2.sub(1);
```

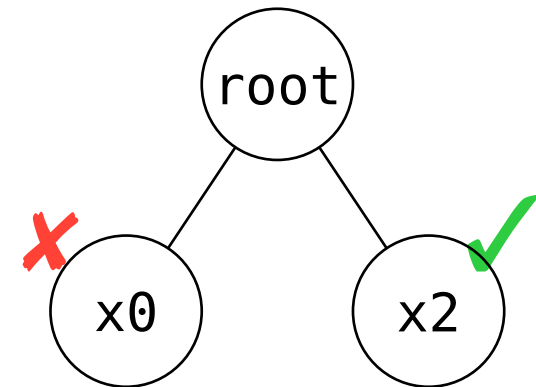


Desired outcome: not UB

? "Reserved"

✓ "Unique"

✗ "Disabled"



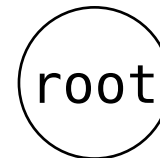
A second look at the motivating example

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;  
*y = 20;
```

Desired outcome: UB

A second look at the motivating example

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;  
*y = 20;
```

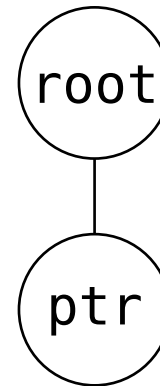


Desired outcome: UB

A second look at the motivating example

From Stacks to Trees

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;  
*y = 20;
```



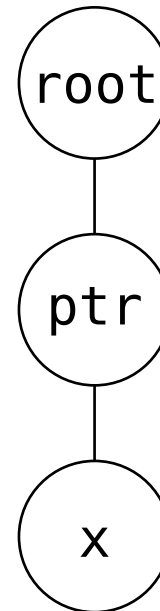
Desired outcome: UB

A second look at the motivating example

From Stacks to Trees

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;  
*y = 20;
```

Desired outcome: UB

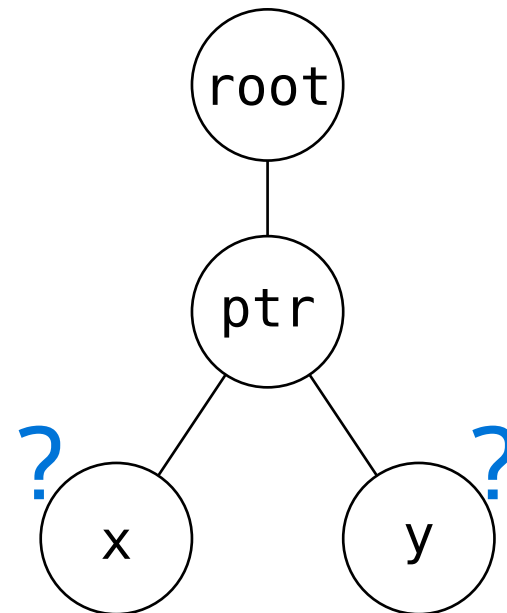


A second look at the motivating example

From Stacks to Trees

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;  
*y = 20;
```

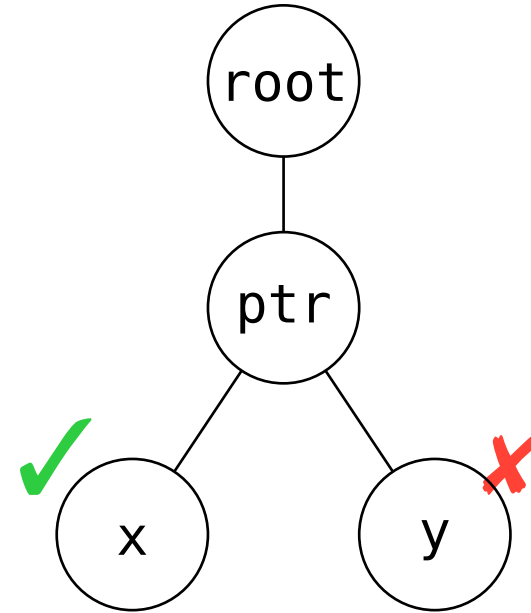
Desired outcome: UB



A second look at the motivating example

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;  
*y = 20;
```

Desired outcome: UB

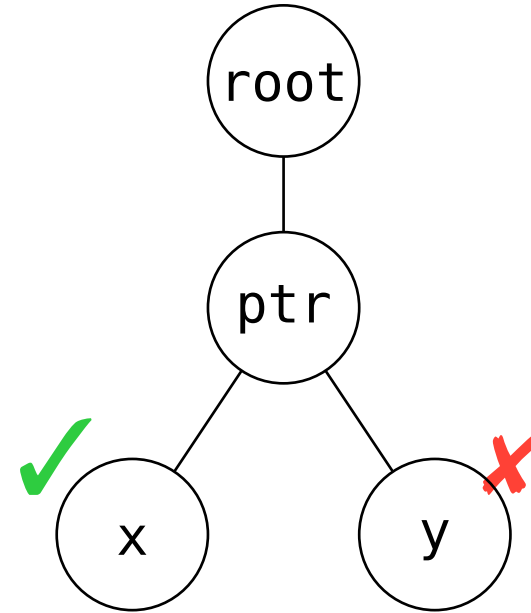


A second look at the motivating example

```
let mut root = 42;  
let ptr = &raw mut root;  
let x = unsafe { &mut *ptr };  
let y = unsafe { &mut *ptr };  
*x = 13;  
*y = 20;
```

UB!

Desired outcome: UB




A second look at the motivating example

Forbids reordering reads

In SB:

```
let mut root = 0;  
let x = &mut root;  
let v1 = *x;  
let v2 = root;
```



root, root, x, x, root is well-bracketed
root, root, x, root, x is not

In TB: a read never prevents another read.

Evaluation

TB should enable desired optimizations

i.e. have enough UB to rule out problematic patterns

- formalized in Rocq (+Simuliris)
- a selection of optimizations proven
 - ✓ delete read through `&mut` or `&`
 - ✓ insert read through `&` in function
 - ✓ move read down for `&mut` or `&` in function
- ...

+ read-read reordering!

It should be possible to write **unsafe** code free of UB Evaluation

i.e. UB should be predictable and not too common

- implemented in Miri
- tested against 30 000 most downloaded libraries on `crates.io`
 - 400 000+ working tests
 - measure how many have UB from Stacked / Tree Borrows

It should be possible to write **unsafe** code free of UB Evaluation

i.e. UB should be predictable and not too common

- implemented in Miri
- tested against 30 000 most downloaded libraries on `crates.io`
 - 400 000+ working tests
 - measure how many have UB from Stacked / Tree Borrows

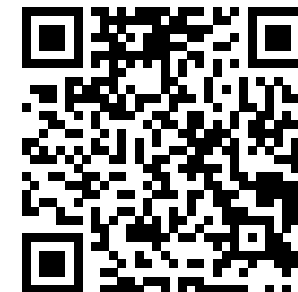
Tree Borrows reduces aliasing-related UB by over 50%

Only 31 ($< 0.5\%$) tests are regressions, all easily fixable.

Conclusion

Try it out:

Rust Playground supports TB
play.rust-lang.org



Learn more:

[plf.inf.ethz.ch/research/
pldi25-tree-borrows.html](https://plf.inf.ethz.ch/research/pldi25-tree-borrows.html)

- detailed state machine
- raw pointers
- interior mutability

