Tree Borrows

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

PLDI’25

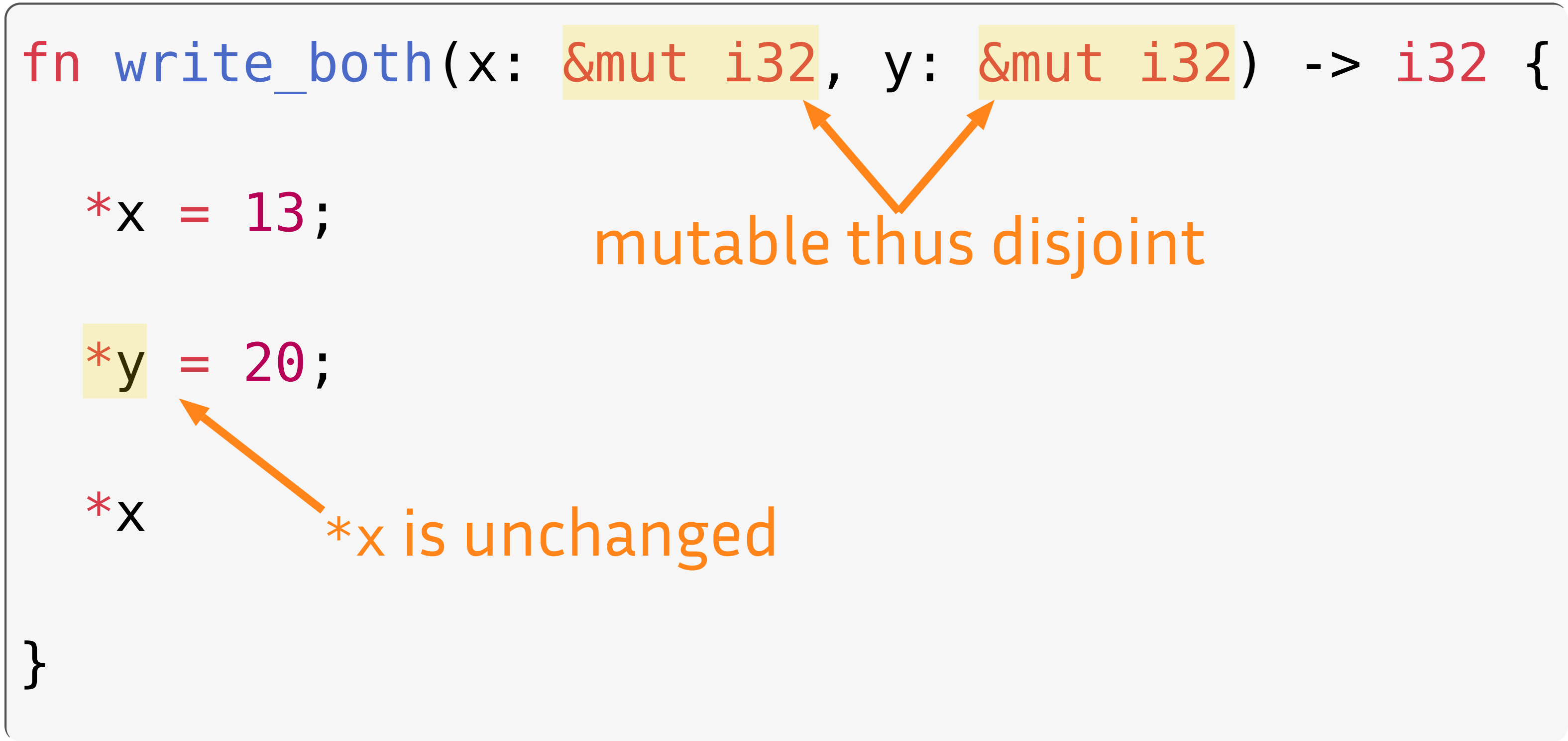
2025-06-19

[1] Univ. Grenoble Alpes, Verimag
[2] ETH Zurich
[3] MPI-SWS

Rust’s type system enables powerful optimizations

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

Rust’s type system enables powerful optimizations

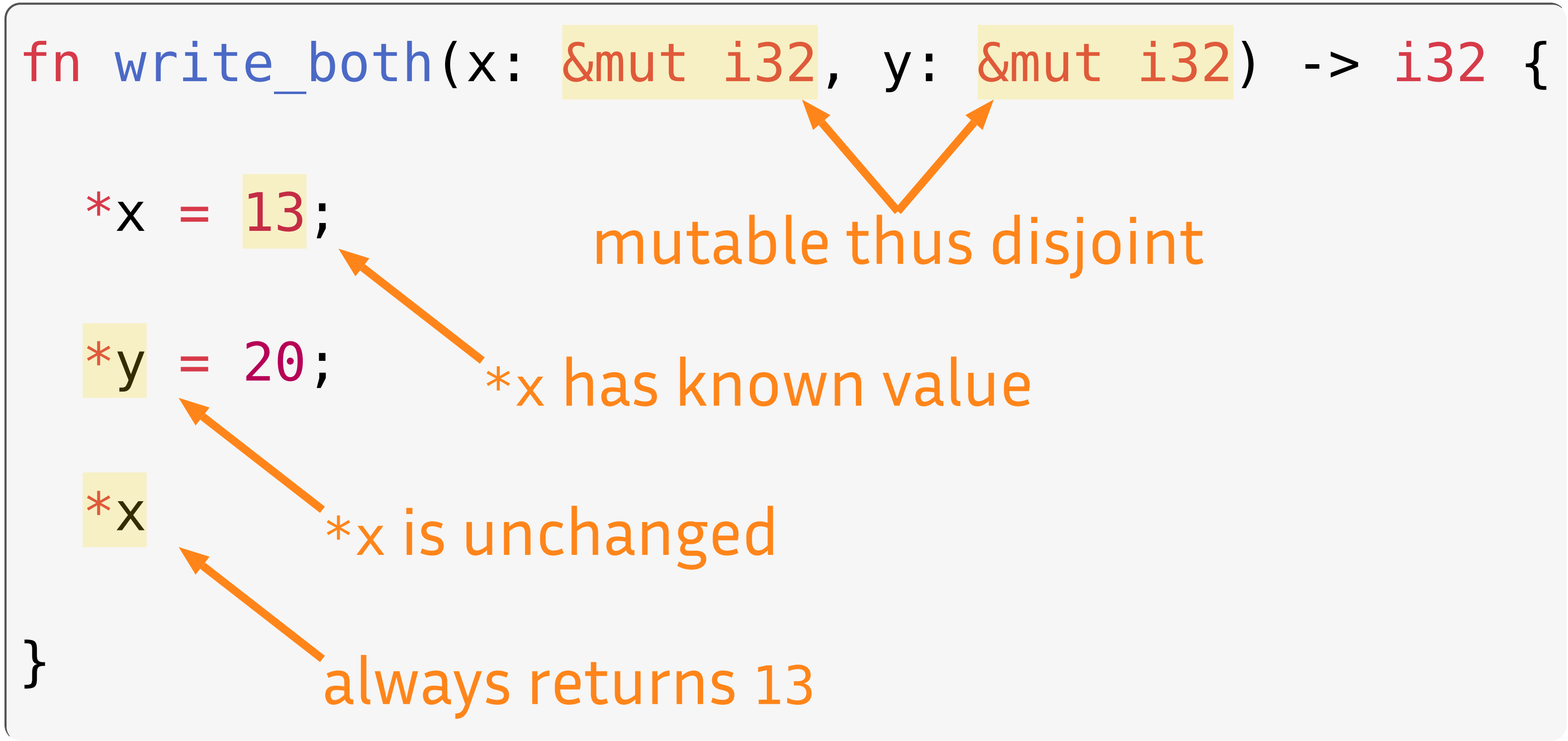


```
fn write_both(x: &mut i32, y: &mut i32) -> i32 {
  *x = 13;
  *y = 20;
  *x
}
```

mutable thus disjoint

*x is unchanged

Rust’s type system enables powerful optimizations



```rust
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 enables 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

aliasing

&

mutability

`&mut` → mutation, no aliasing

`&` → aliasing, no mutation

Escape hatch: `unsafe`

Can use **unchecked operations** to do **low-level manipulations**

```
unsafe {
  // Code within this block can effectively
  // bypass some parts of the typechecker.
  ...
}
```

Within `unsafe` it is **the programmer’s responsibility** to check

- that pointers are non-null
- that memory is initialized
- absence of data races
- …

violations trigger UB (Undefined Behavior)

What if unsafe code is misused ?

```
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)); // prints 20
}
```

What if unsafe code is misused ?

```rust
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)); // prints 20
}
```

What if unsafe code is misused ?

```rust
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)); // prints 20
}
```

What if unsafe code is misused ?

```rust
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)); // prints 20
}
```

What if unsafe code is misused ?

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

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)); // prints 13
}
```

What if unsafe code is misused ?

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

```
  let ptr = &raw mut root;
  let x = unsafe { &mut *ptr };
  let y = unsafe { &mut *ptr };
  println!("{}", write_both(x, y)); // prints 20
}
```

unsafe code can break the assumptions that optimizations need!

Expanding our notion of UB

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

- that pointers are non-null
- that memory is initialized
- absence of data races
- compliance with aliasing rules NEW!

violations trigger UB

Expanding our notion of UB

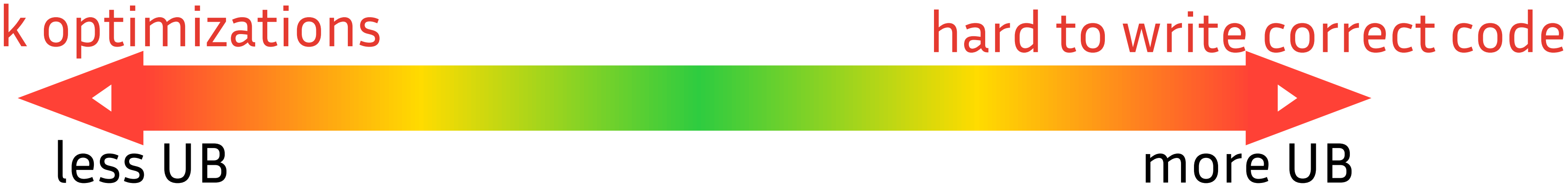
Within `unsafe` it is **the programmer’s responsibility** to check

- that pointers are non-null
- that memory is initialized
- absence of data races
- compliance with aliasing rules NEW!

violations trigger UB

Tree Borrows (TB): defines those aliasing rules

Compiler **assumes absence of UB**, exploits this for optimizations



Expanding our notion of UB

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

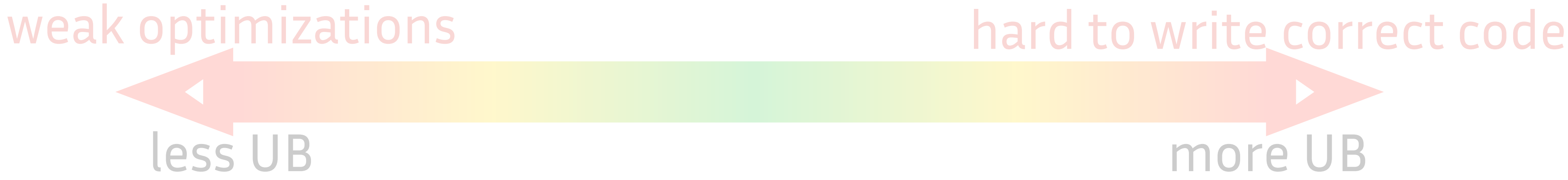
- that pointers are non-null
- that memory is initialized

Tr

Co

Sounds familiar?

TB is the successor of **Stacked Borrows**, which has the same purpose.



Stacked Borrows (SB)

[Jung et al., POPL’20]

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 };
// inline write_both(x, y):
*x = 13;
```

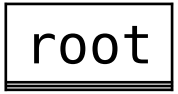
Desired outcome: UB

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 };
// inline write_both(x, y):
*x = 13;
```



root

- new stack at root

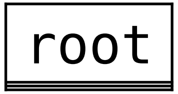
Desired outcome: UB

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 };
// inline write_both(x, y):
*x = 13;
```



- ✓ root is at the top

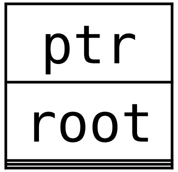
Desired outcome: UB

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 };
// inline write_both(x, y):
*x = 13;
```



ptr

root

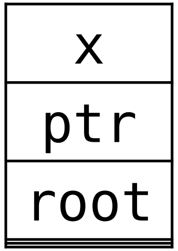
- ✓ root is at the top
- push ptr

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 };
// inline write_both(x, y):
*x = 13;
```



- ✓ `ptr` is at the top
- push `x`

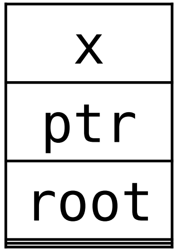
Desired outcome: UB

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 };
// inline write_both(x, y):
*x = 13;
```



x

ptr

root

- pop until `ptr` is at the top

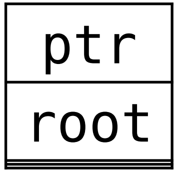
Desired outcome: UB

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 };
// inline write_both(x, y):
*x = 13;
```



- pop until ptr is at the top

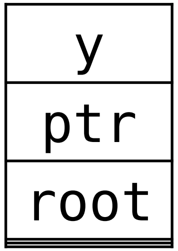
Desired outcome: UB

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 };
// inline write_both(x, y):
*x = 13;
```


y
ptr
root

- pop until ptr is at the top
- push y

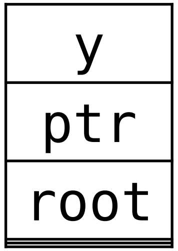
Desired outcome: UB

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 };
// inline write_both(x, y):
*x = 13;
```



y

ptr

root

- search for x

Desired outcome: UB

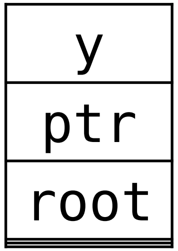
Stacked Borrows (SB)

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 };
// inline write_both(x, y):
*x = 13;
```

UB!



y

ptr

root

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

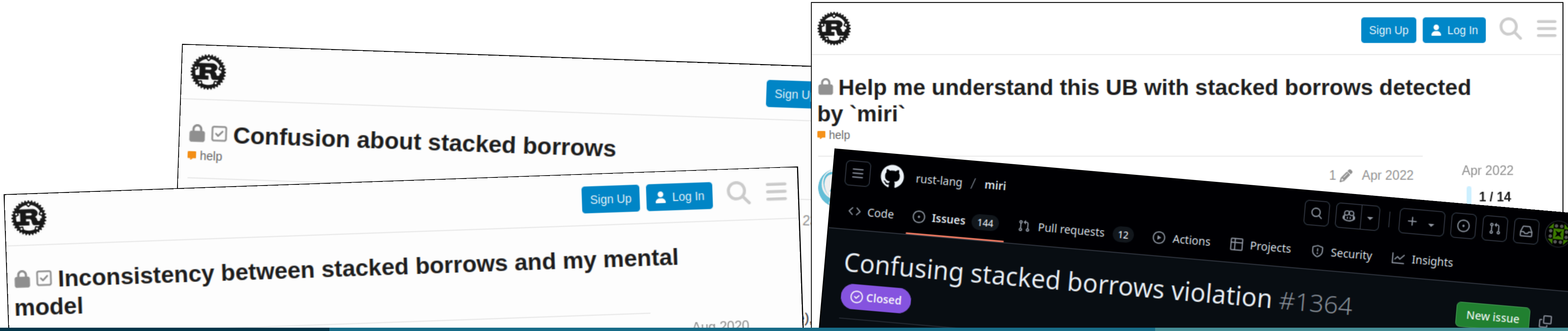
→ many bugs detected (e.g. in stdlib)

Stacked Borrows (SB)

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

→ included in many projects' CI

→ many bugs detected (e.g. in stdlib)



Confusion about stacked borrows

Inconsistency between stacked borrows and my mental model

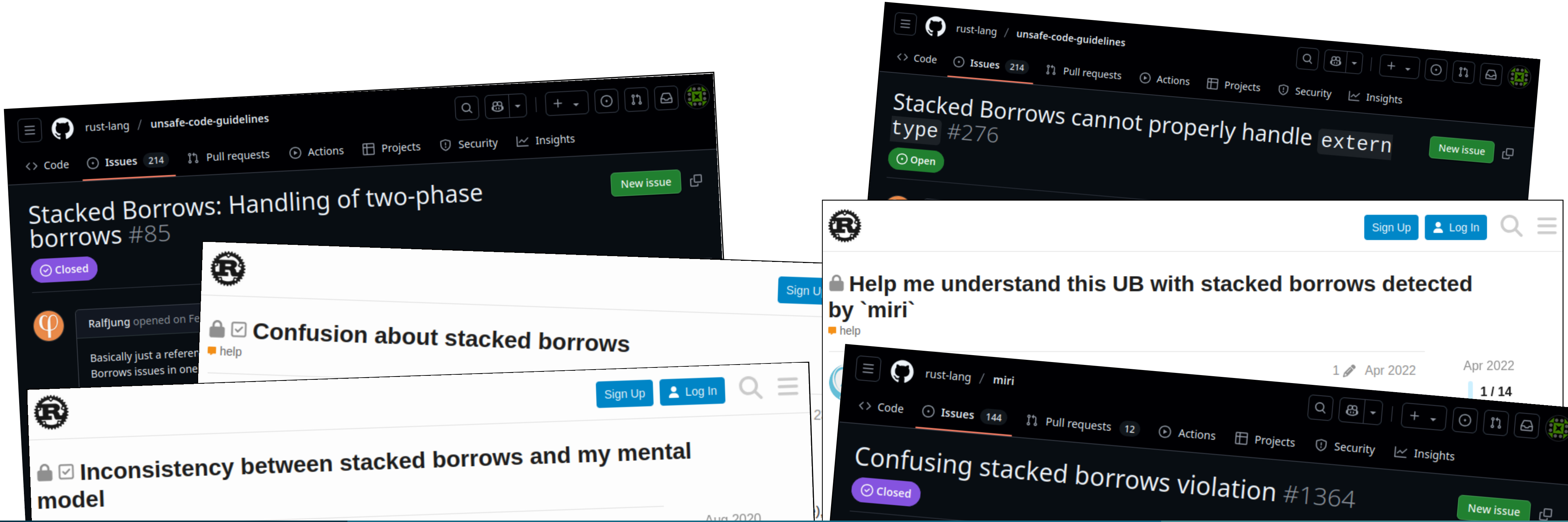
Help me understand this UB with stacked borrows detected by `miri`

Confusing stacked borrows violation #1364

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

→ included in many projects' CI

→ many bugs detected (e.g. in stdlib)



rust-lang / unsafe-code-guidelines

Stacked Borrows: Handling of two-phase borrows #85

Closed

RalfJung opened on Fe

Basically just a referen
Borrows issues in one

Confusion about stacked borrows

help

Inconsistency between stacked borrows and my mental model

Aug 2020

rust-lang / unsafe-code-guidelines

Stacked Borrows cannot properly handle `extern type` #276

Open

Help me understand this UB with stacked borrows detected by `miri`

help

1 Apr 2022

Apr 2022

1 / 14

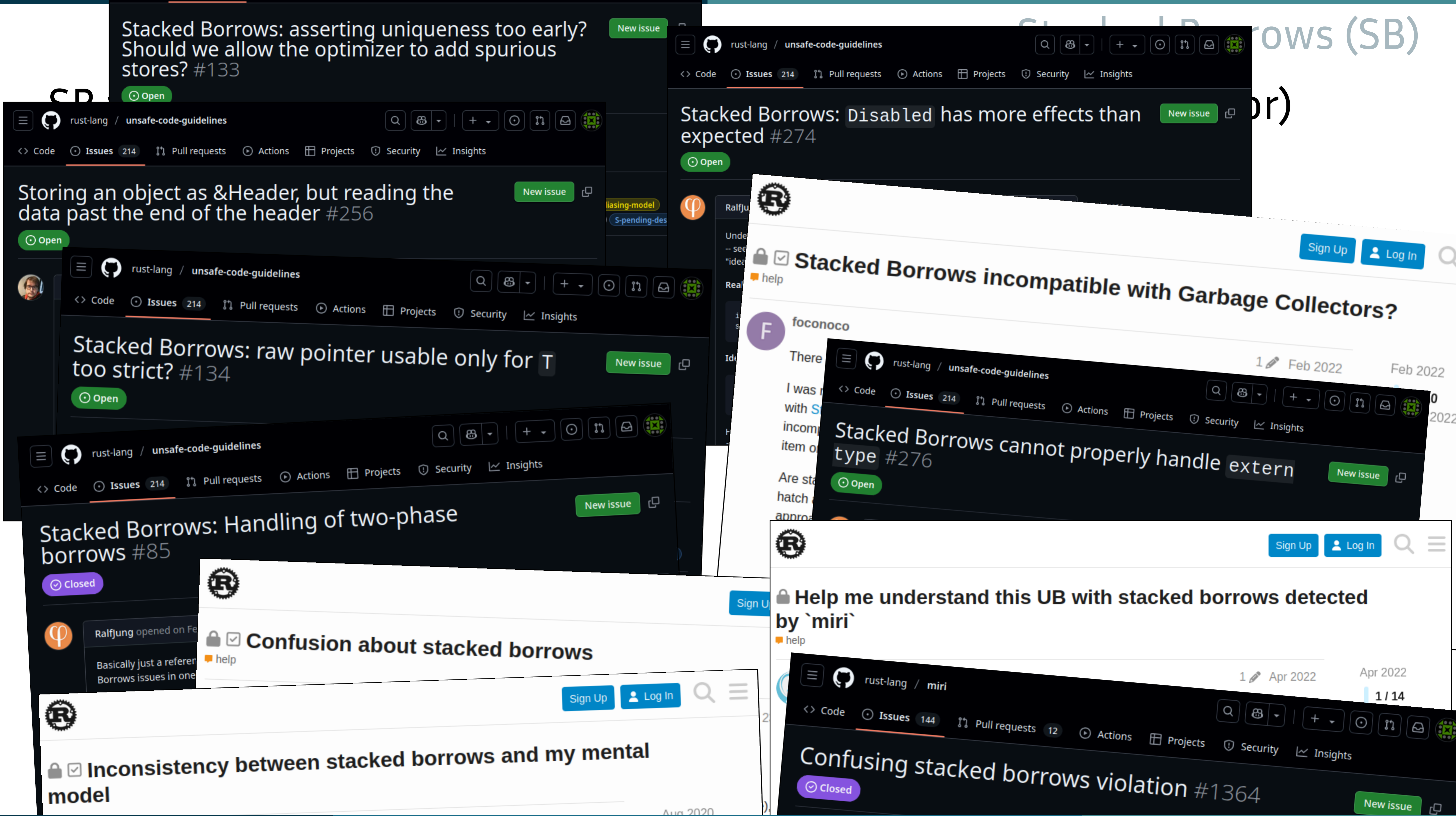
rust-lang / miri

Confusing stacked borrows violation #1364

Closed

Stacked Borrows (SB)

SB … r)



Stacked Borrows: asserting uniqueness too early? Should we allow the optimizer to add spurious stores? #133

New issue

Open

rust-lang / unsafe-code-guidelines

Code · Issues 214 · Pull requests · Actions · Projects · Security · Insights

Storing an object as &Header, but reading the data past the end of the header #256

New issue

Open

liasing-model

S-pending-des

Stacked Borrows: raw pointer usable only for `T` too strict? #134

New issue

Open

Stacked Borrows: Handling of two-phase borrows #85

New issue

Closed

RalfJung opened on Fe

Basically just a referen
Borrows issues in one

Stacked Borrows: `Disabled` has more effects than expected #274

New issue

Open

RalfJu

Unde
-- see
"idea

Real

Ide

Stacked Borrows incompatible with Garbage Collectors?

help

Sign Up · Log In

foconoco

There

I was r
with S
incomp
item o

Are sta
hatch
approa

1 Feb 2022

Feb 2022

Stacked Borrows cannot properly handle `extern type` #276

New issue

Open

Confusion about stacked borrows

help

Sign U

Inconsistency between stacked borrows and my mental model

Sign Up · Log In

Aug 2020

Help me understand this UB with stacked borrows detected by `miri`

help

Sign Up · Log In

1 Apr 2022

Apr 2022

1 / 14

rust-lang / miri

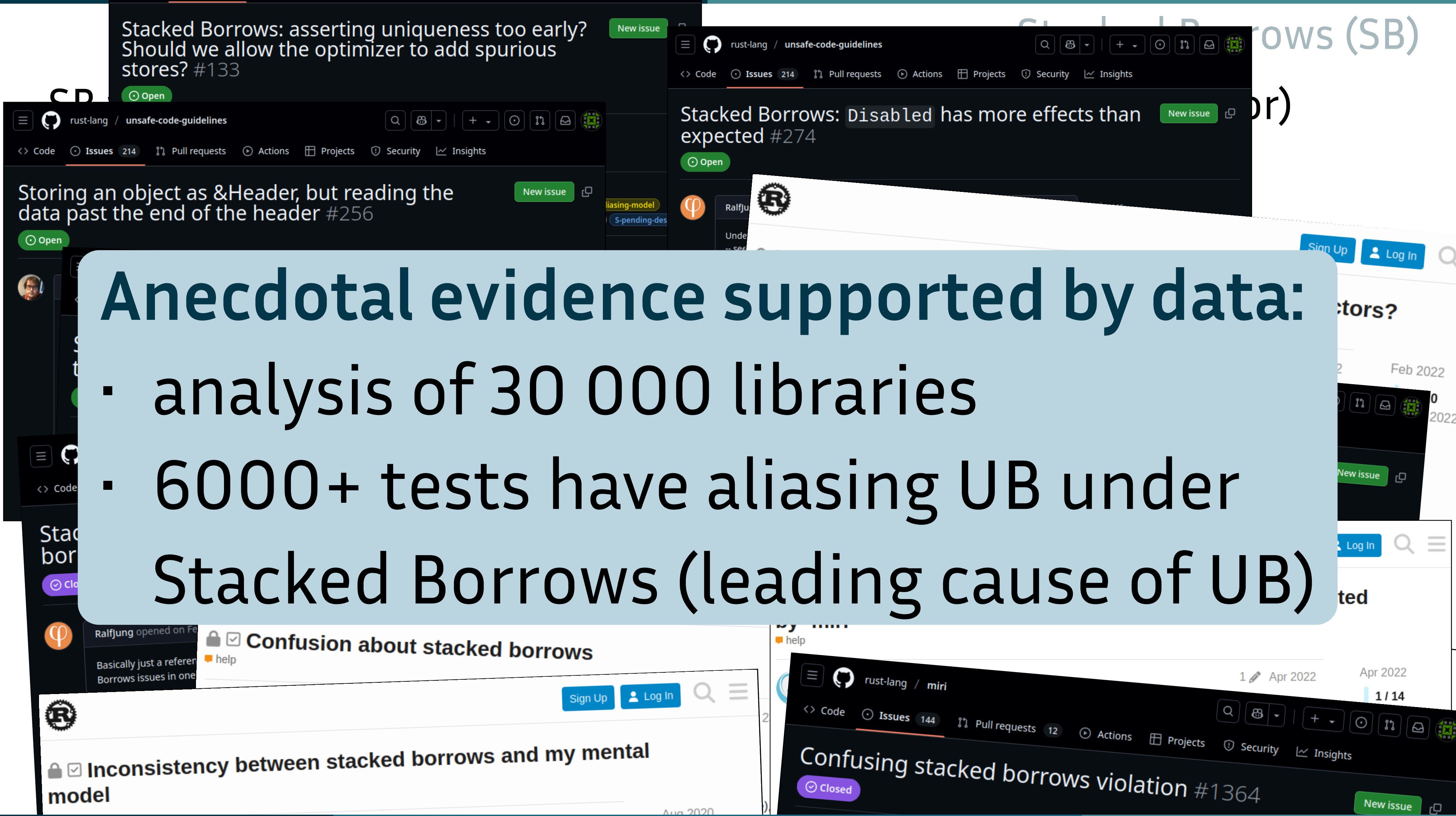
Code · Issues 144 · Pull requests 12 · Actions · Projects · Security · Insights

Confusing stacked borrows violation #1364

New issue

Closed

Stacked Borrows (SB)



Stacked Borrows: asserting uniqueness too early? Should we allow the optimizer to add spurious stores? #133

Open

rust-lang / unsafe-code-guidelines

Storing an object as &Header, but reading the data past the end of the header #256

Open

rust-lang / unsafe-code-guidelines

Stacked Borrows: `Disabled` has more effects than expected #274

Open

Confusion about stacked borrows

help

Inconsistency between stacked borrows and my mental model

Aug 2020

rust-lang / miri

Confusing stacked borrows violation #1364

Closed

Anecdotal evidence supported by data:

- analysis of 30 000 libraries
- 6000+ tests have aliasing UB under Stacked Borrows (leading cause of UB)

Tree Borrows allows much more code

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

Out of 30 000 most downloaded libraries,
54% **fewer tests** with aliasing UB when using Tree Borrows

Tree Borrows allows much more code

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

Out of 30 000 most downloaded libraries,
54% **fewer tests** with aliasing UB when using Tree Borrows

Fixes known technical limitations of SB,
incl. 2-phase borrows, extern types, **pointer offsets**

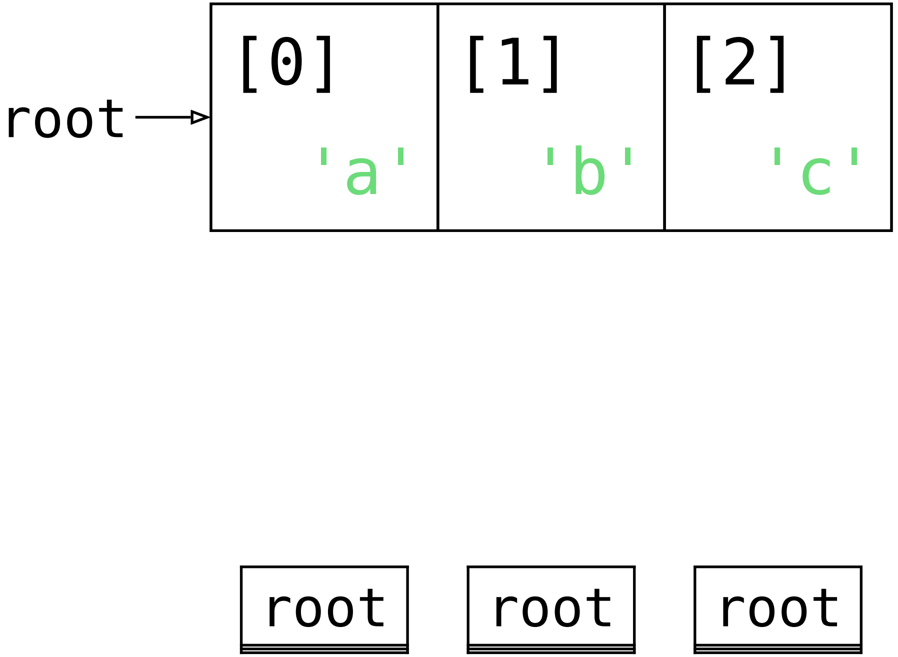
From Stacks to Trees

From Stacks to Trees

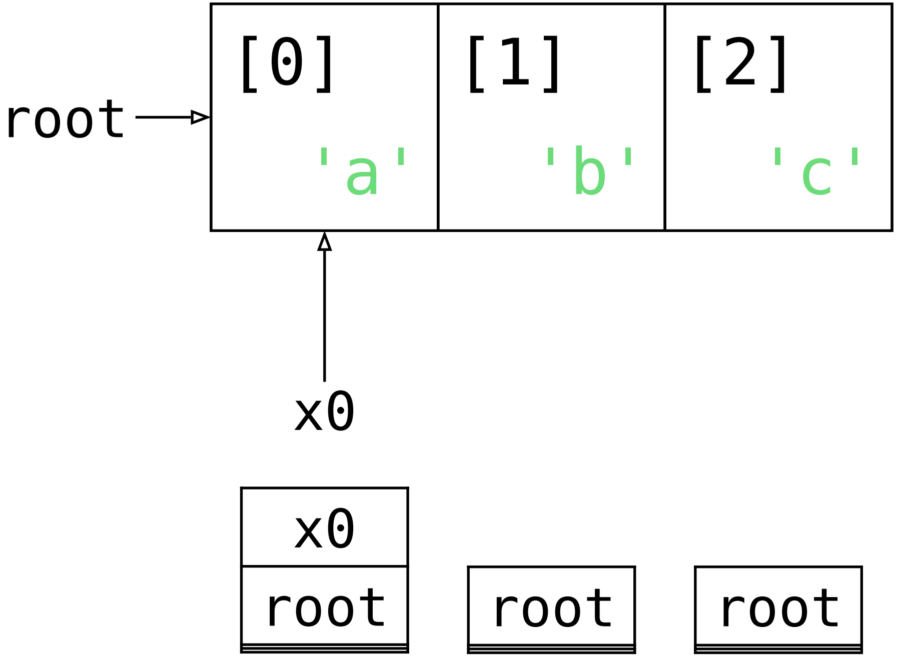
```rust
let mut root = vec!['a','b','c'];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];
```

From Stacks to Trees

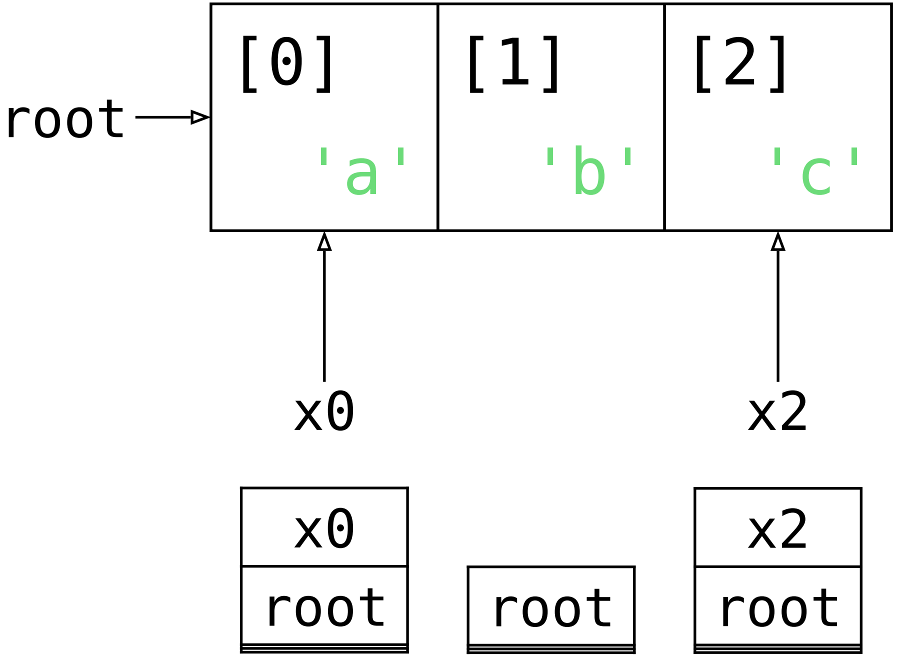
```
let mut root = vec!['a','b','c'];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];
```



```
let mut root = vec!['a','b','c'];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];
```



```
let mut root = vec!['a','b','c'];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];
```

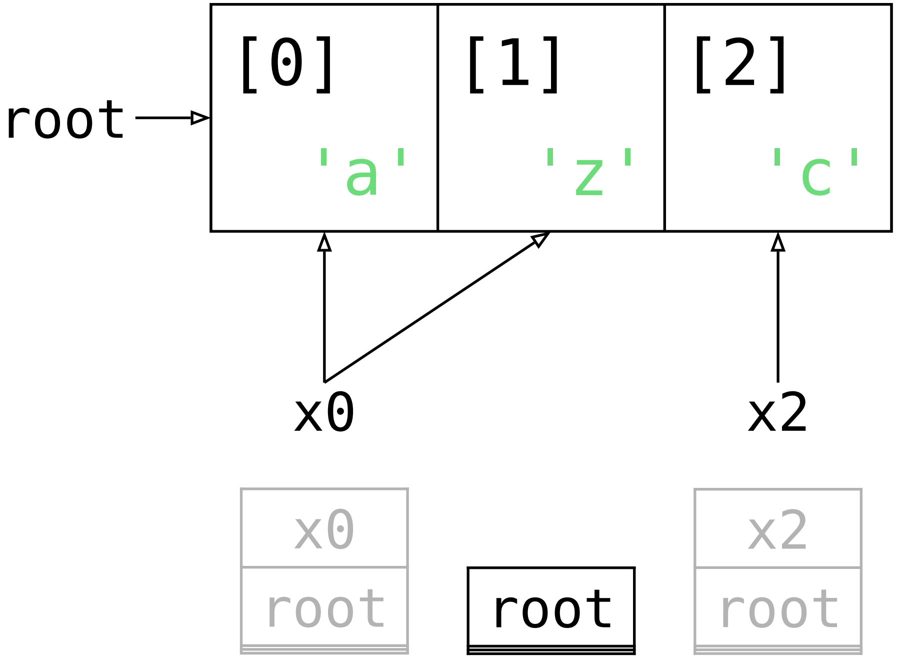


From Stacks to Trees

```
let mut root = vec!['a','b','c'];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];

// Scenario 1
unsafe { *x0.add(1) = 'z'; }
```

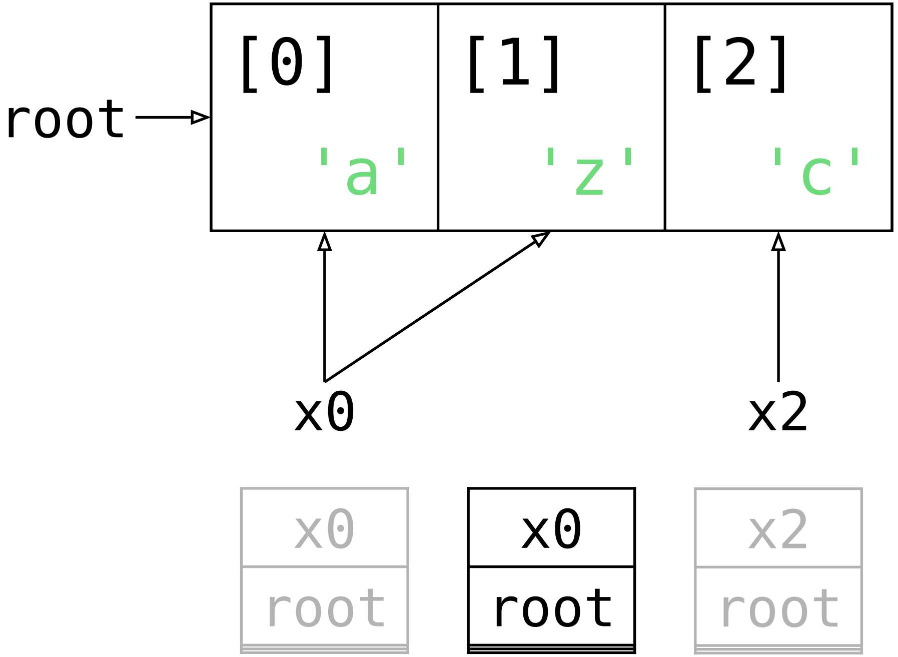
Desired outcome: not UB



```
let mut root = vec!['a','b','c'];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];

// Scenario 1
unsafe { *x0.add(1) = 'z'; }
```

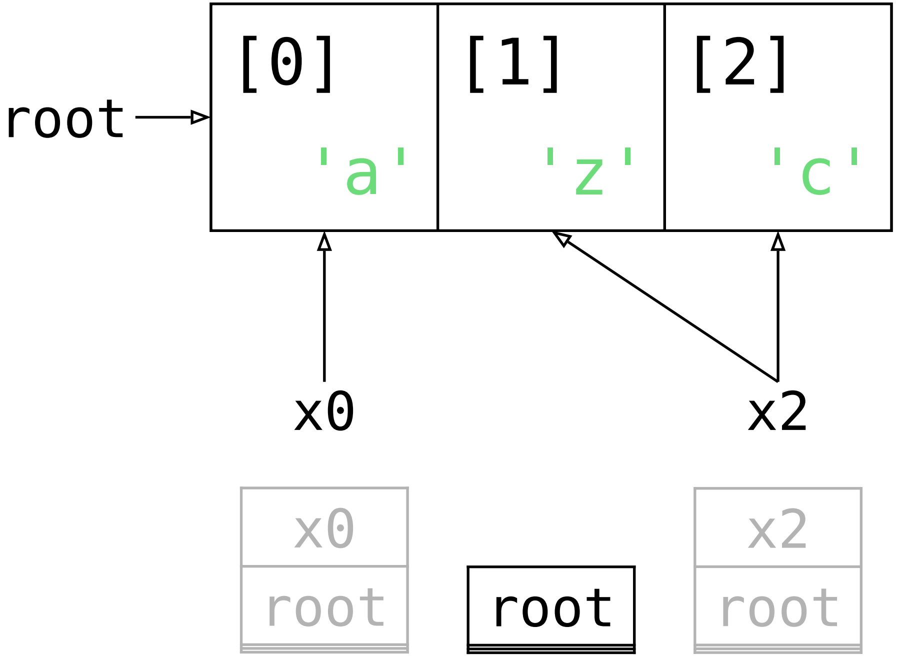
Desired outcome: not UB



```
let mut root = vec!['a','b','c'];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];

// Scenario 2
unsafe { *x2.sub(1) = 'z'; }
```

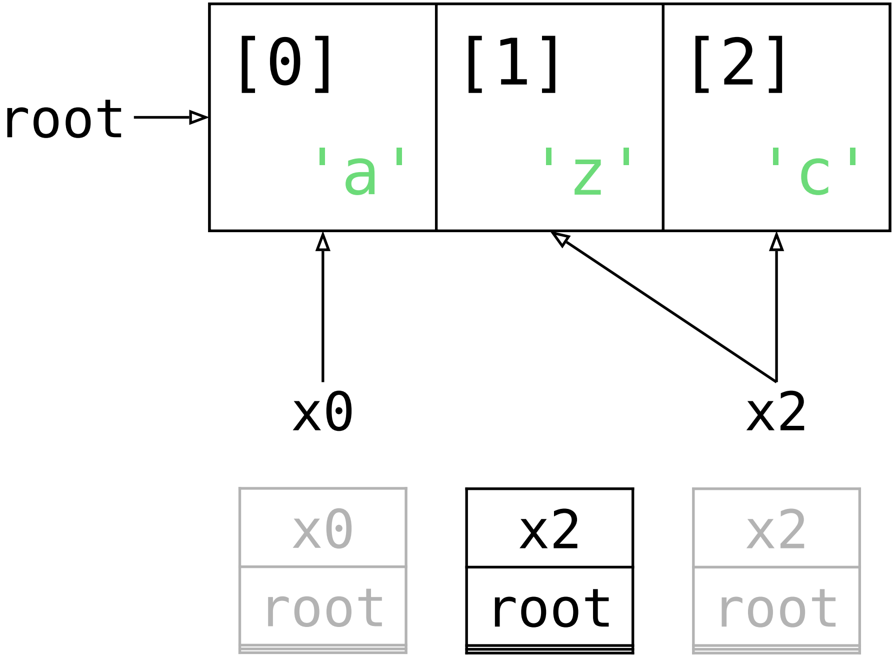
Desired outcome: not UB



```
let mut root = vec!['a','b','c'];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];

// Scenario 2
unsafe { *x2.sub(1) = 'z'; }
```

Desired outcome: not UB

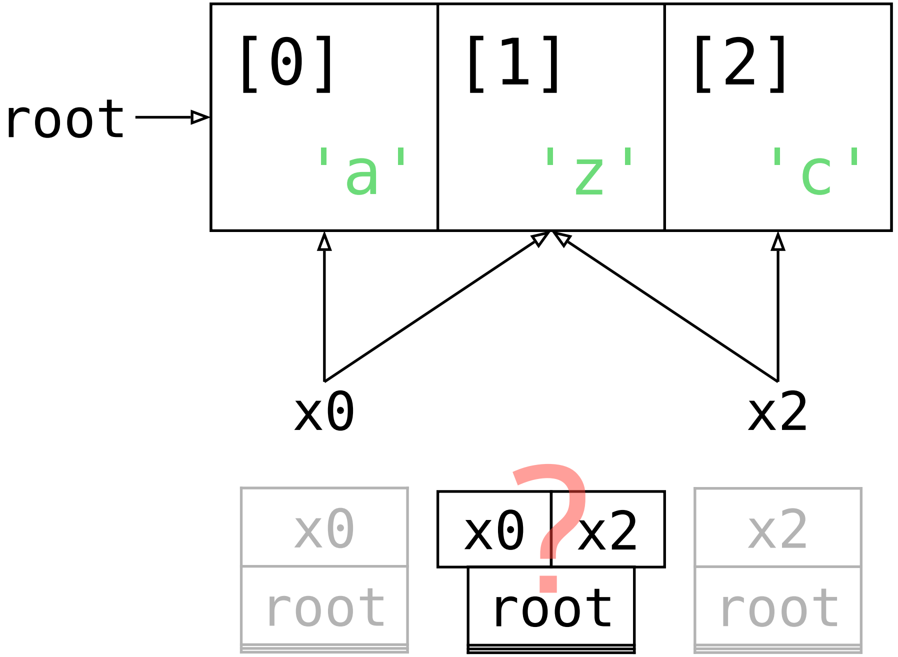


From Stacks to Trees

```
let mut root = vec!['a','b','c'];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];

// Scenario 1 or 2
unsafe { *??? = 'z'; }
```

Desired outcome: not UB



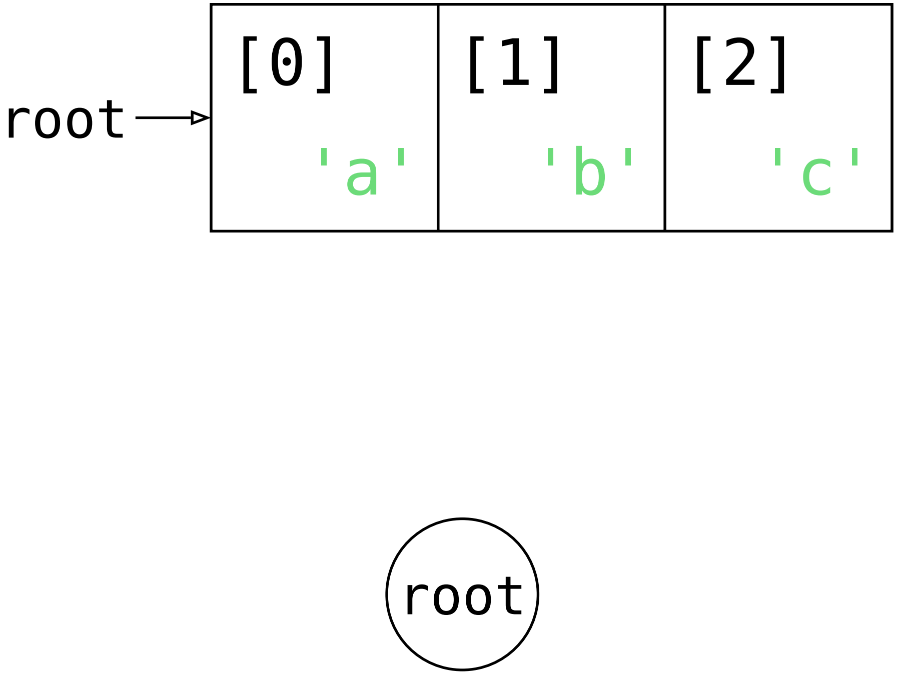
From Stacks to Trees

```
let mut root = vec!['a','b','c'];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];
```

Desired outcome: not UB

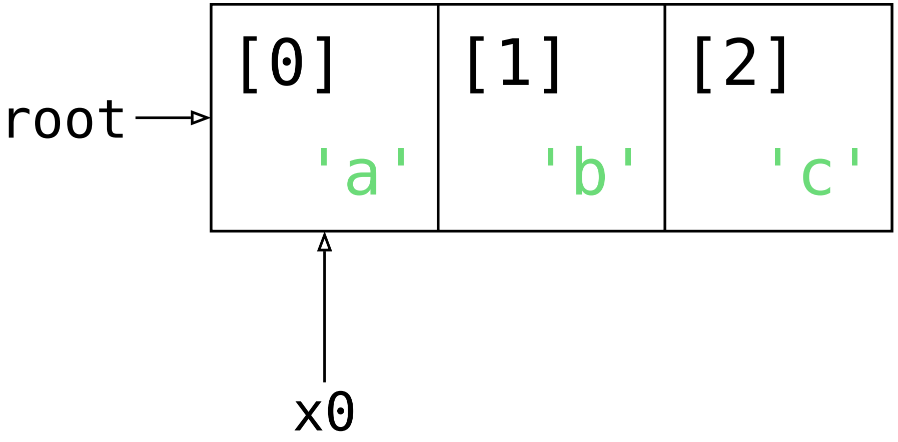
From Stacks to Trees

```
let mut root = vec!['a','b','c'];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];
```

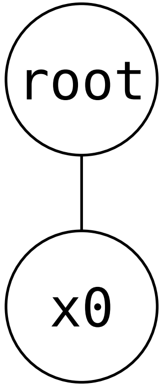


Desired outcome: not UB

```
let mut root = vec!['a','b','c'];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];
```

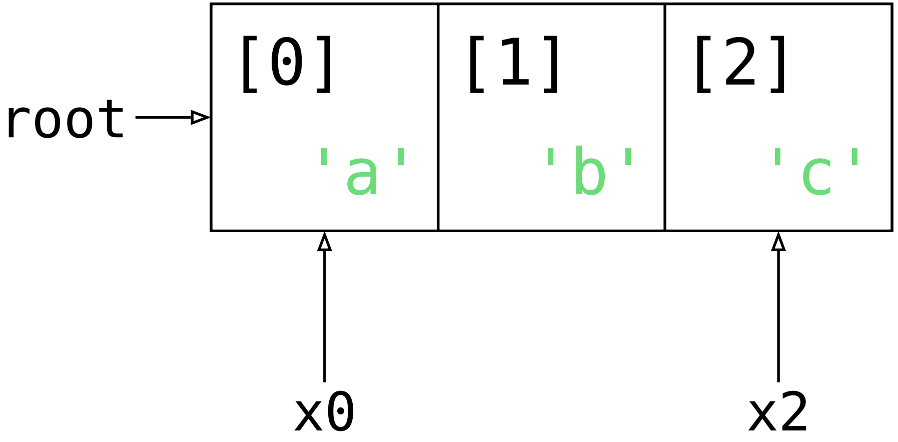


Desired outcome: not UB

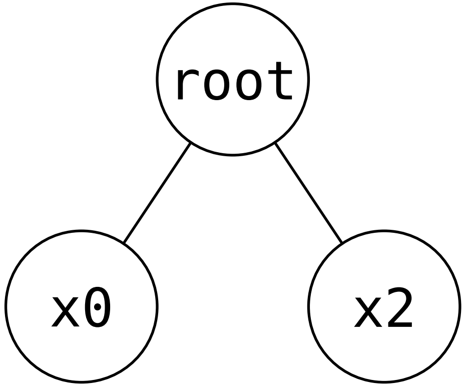


From Stacks to Trees

```rust
let mut root = vec!['a','b','c'];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];
```

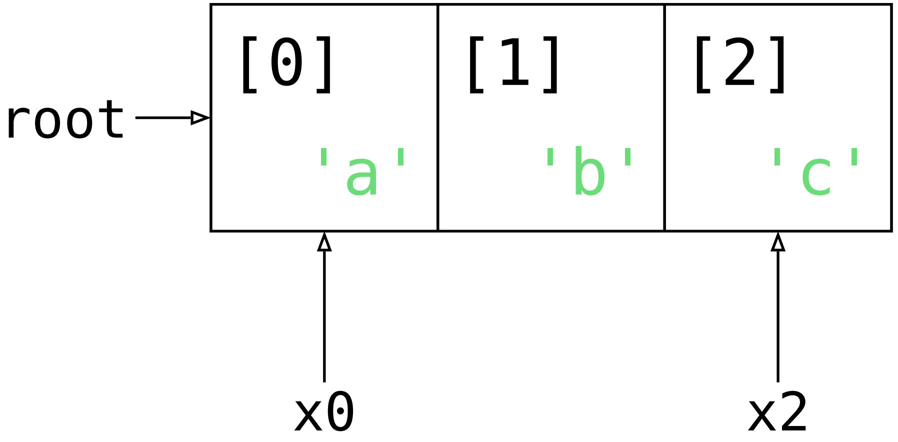


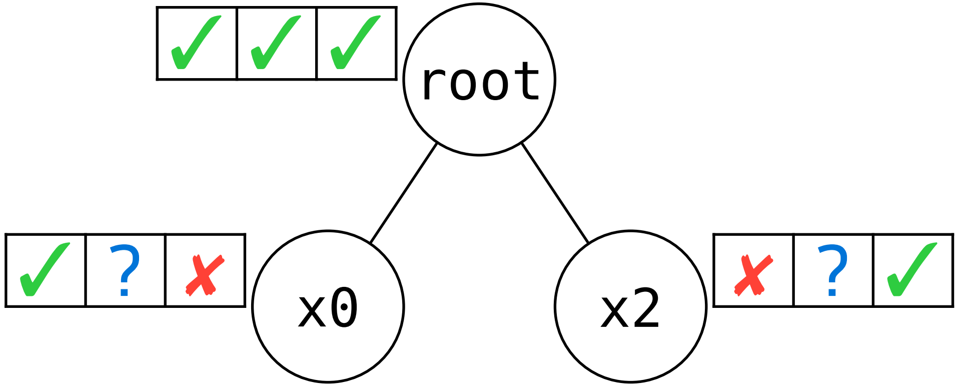
Desired outcome: not UB



From Stacks to Trees

```
let mut root = vec!['a','b','c'];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];
```



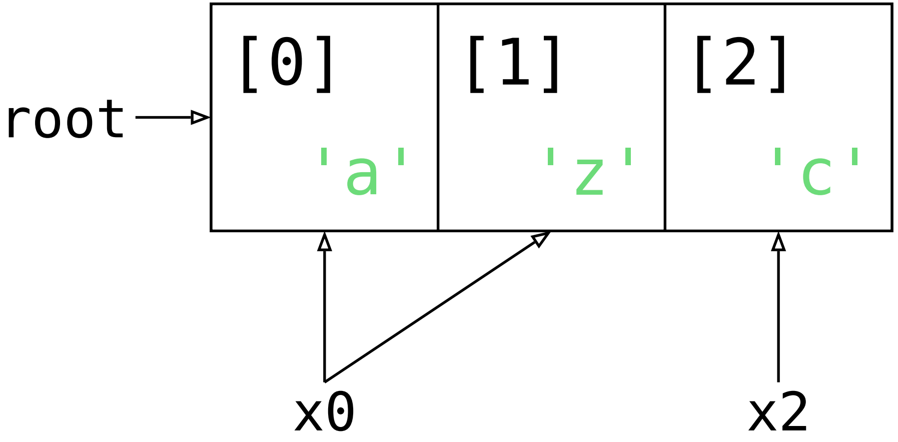


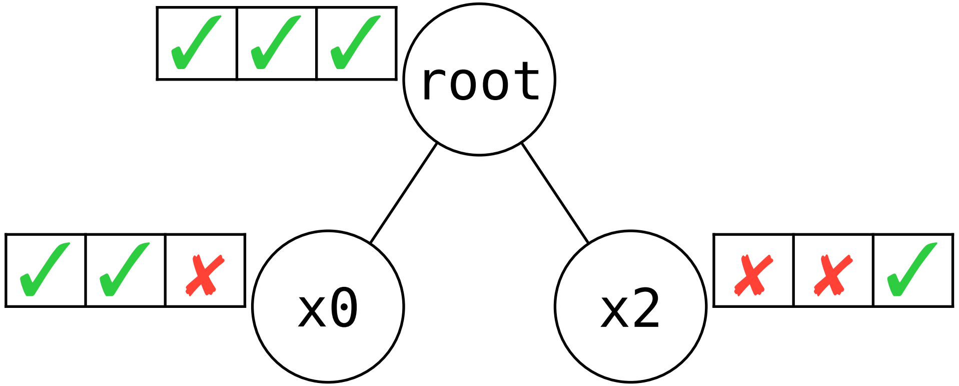
Desired outcome: not UB

From Stacks to Trees

```
let mut root = vec!['a','b','c'];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];

// Scenario 1
unsafe { *x0.add(1) = 'z' };
```





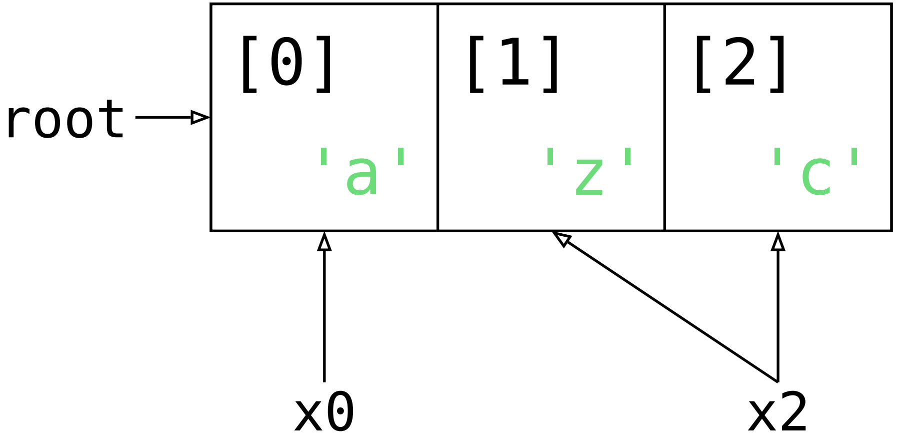
Desired outcome: not UB

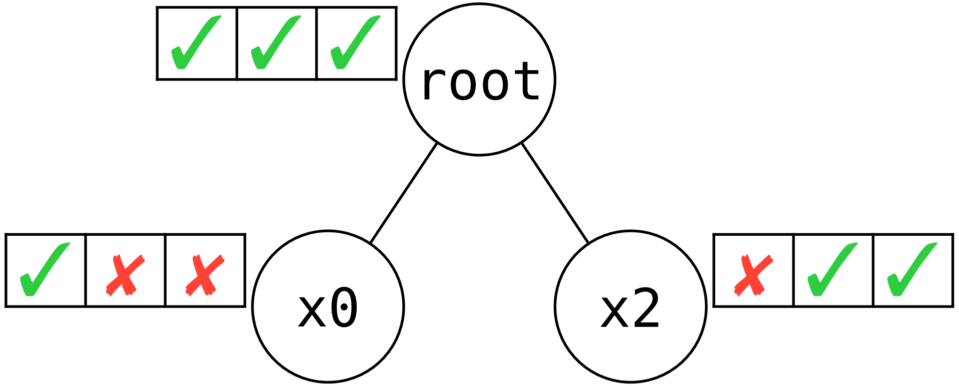
From Stacks to Trees

```
let mut root = vec!['a','b','c'];
let x0 = &raw mut root[0];
let x2 = &raw mut root[2];

// Scenario 2
unsafe { *x2.sub(1) = 'z'; }
```

Desired outcome: not UB





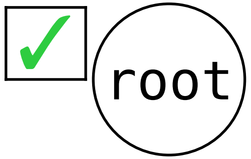
A second look at the motivating example

```
let mut root = 42;
let ptr = &raw mut root;
let x = unsafe { &mut *ptr };
let y = unsafe { &mut *ptr };
// inline write_both(x, y):
*x = 13;
*y = 20;
```

Desired outcome: UB

A second look at the motivating example

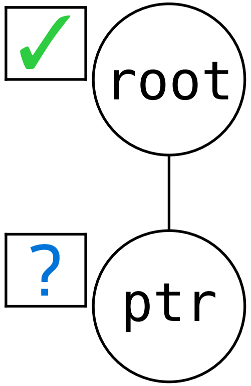
```
let mut root = 42;
let ptr = &raw mut root;
let x = unsafe { &mut *ptr };
let y = unsafe { &mut *ptr };
// inline write_both(x, y):
*x = 13;
*y = 20;
```



Desired outcome: UB

A second look at the motivating example

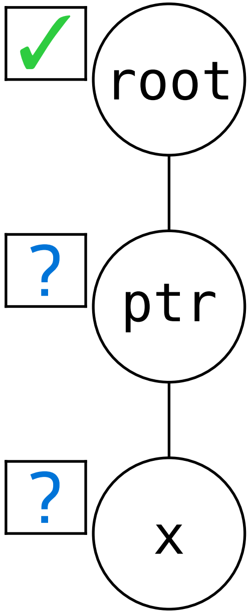
```
let mut root = 42;
let ptr = &raw mut root;
let x = unsafe { &mut *ptr };
let y = unsafe { &mut *ptr };
// inline write_both(x, y):
*x = 13;
*y = 20;
```



Desired outcome: UB

A second look at the motivating example

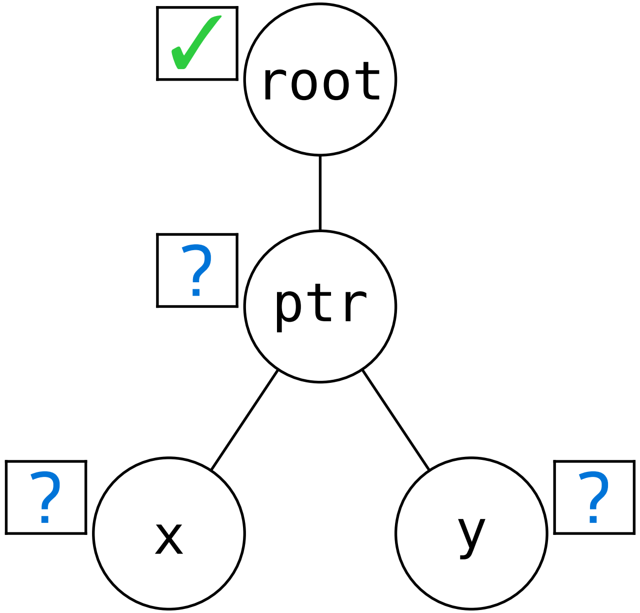
```
let mut root = 42;
let ptr = &raw mut root;
let x = unsafe { &mut *ptr };
let y = unsafe { &mut *ptr };
// inline write_both(x, y):
*x = 13;
*y = 20;
```



Desired outcome: UB

A second look at the motivating example

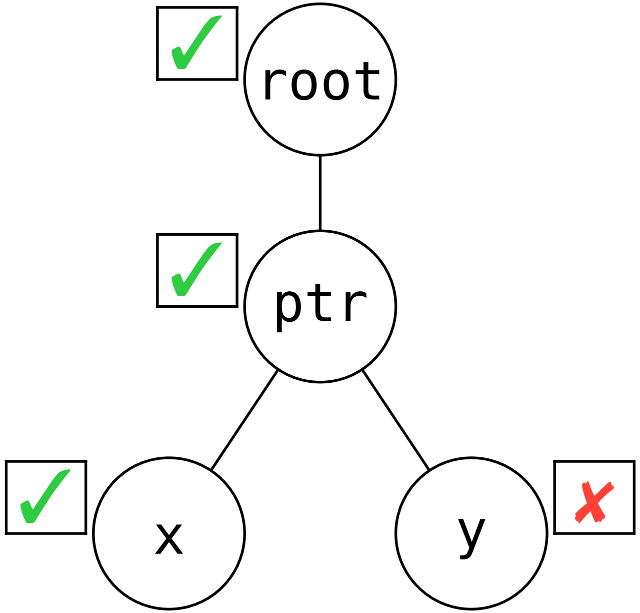
```
let mut root = 42;
let ptr = &raw mut root;
let x = unsafe { &mut *ptr };
let y = unsafe { &mut *ptr };
// inline write_both(x, y):
*x = 13;
*y = 20;
```



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 };
// inline write_both(x, y):
*x = 13;
*y = 20;
```

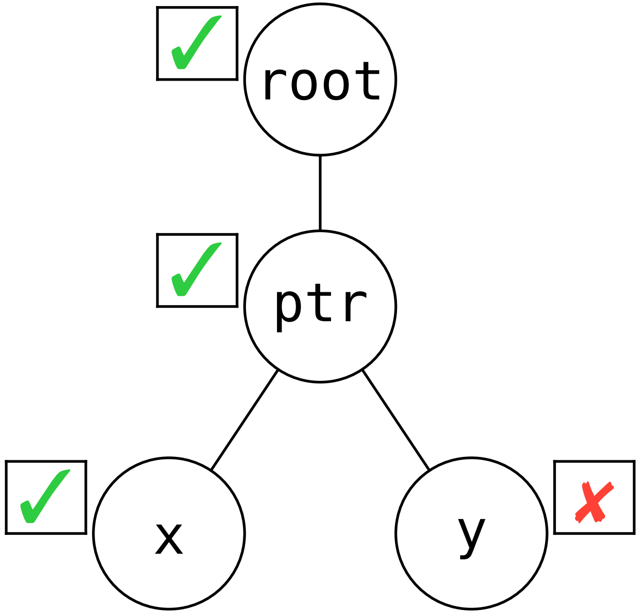


Desired outcome: UB

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 };
// inline write_both(x, y):
*x = 13;
*y = 20;
```

UB!



Desired outcome: UB

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

  …

It should be possible to write `unsafe` code free of UB

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

54% fewer tests have aliasing UB according to Tree Borrows

Only 31 ($< 0.01\%$) tests are regressions, all easily fixable.

(Out of 30 000 libraries, 400 000+ working tests)

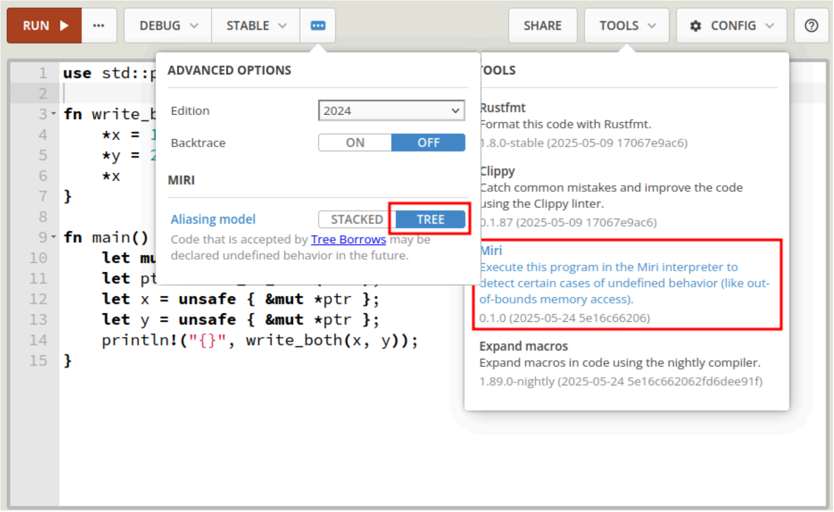
“Tree Borrows accepts more real-world programs that call foreign functions than Stacked Borrows due to differences in handling pointer arithmetic.”

A Study of Undefined Behavior Across Foreign Function Boundaries in Rust Libraries, by I. McCormack, J. Sunshine, J. Aldrich @ ICSE’25

Conclusion

Try it out: supported by Miri

Also on the Rust Playground (`play.rust-lang.org`)



```
use std::p
|
fn write_b
    *x = 1
    *y = 2
    *x
}

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

ADVANCED OPTIONS

Edition: 2024

Backtrace: ON [✓ OFF]

MIRI

Aliasing model: STACKED [✓ TREE]

Code that is accepted by Tree Borrows may be declared undefined behavior in the future.

TOOLS

Rustfmt
Format this code with Rustfmt.
1.8.0-stable (2025-05-09 17067e9ac6)

Clippy
Catch common mistakes and improve the code using the Clippy linter.
0.1.87 (2025-05-09 17067e9ac6)

Miri
Execute this program in the Miri interpreter to detect certain cases of undefined behavior (like out-of-bounds memory access).
0.1.0 (2025-05-24 5e16c66206)

Expand macros
Expand macros in code using the nightly compiler.
1.89.0-nightly (2025-05-24 5e16c662062fd6dee91f)

Postdoc positions available

@ ETH Zurich	`ralf.jung@inf.ethz.ch`
@ MPI-SWS	`dreyer@mpi-sws.org`

Learn more:

`plf.inf.ethz.ch/research/pldi25-tree-borrows.html`

Includes e.g. handling of raw pointers and interior mutability.