A Program Logic for Tree Borrows

Johannes Hostert[1], Ralf Jung[1]

Rust Verification Workshop 2026, Turin, Italy

14 April 2026

1: ETH Zurich, Switzerland

References Forbid Aliasing

```
fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
    bar();
    *a = x;
}
```

References Forbid Aliasing

We want to optimize this program:

```
fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
    bar();
    *a = x;
}
```

References Forbid Aliasing

We want to optimize this program:

```
fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
    bar();
    *a = x;
}
```

mutable references have no aliases.

References Forbid Aliasing

We want to optimize this program:

```
fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
    bar();
    *a = x;
}
```

```
fn foo(a: &mut i32) {
    let x = *a;
    // *a = 42;
    bar();
    *a = x;
}
```

mutable references have no aliases.

References Forbid Aliasing

We want to optimize this program:

```
fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
    bar();
    *a = x;
}
```

```
fn foo(a: &mut i32) {
    let x = *a;
    // *a = 42;
    bar();
    // *a = x;
}
```

mutable references have no aliases.

References Forbid Aliasing

We want to optimize this program:

```
fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
    bar();
    *a = x;
}
```

```
fn foo(a: &mut i32) {
    // let x = *a;
    // *a = 42;
    bar();
    // *a = x;
}
```

mutable references have no aliases.

References Forbid Aliasing

We want to optimize this program:

```
fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
    bar();
    *a = x;
}
```

```
fn foo(a: &mut i32) {
    // let x = *a;
    // *a = 42;
    bar();
    // *a = x;
}
```

Correctness: `bar()` can not access `a`,
since mutable references have no aliases.

References Forbid Aliasing

We want to optimize this program:

```
fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
    bar();
    *a = x;
}
```

```
fn foo(a: &mut i32) {
    // let x = *a;
    // *a = 42;
    bar();
    // *a = x;
}
```

Correctness: `bar()` can not access `a`,
since mutable references have no aliases... or do they?

Unsafe Code Can Create Aliasing References

```
fn foo(a: &mut i32) { .. }

static mut GLOBAL: i32 = 0;
fn bar() {
    unsafe { println!("{}", &GLOBAL); }
}
fn main() {
    let a = unsafe { &mut GLOBAL };
    foo(a);
}
```

Unsafe Code Can Create Aliasing References

```
fn foo(a: &mut i32) { .. }

static mut GLOBAL: i32 = 0;
fn bar() {
    unsafe { println!("{}", &GLOBAL); }
}
fn main() {
    let a = unsafe { &mut GLOBAL };
    foo(a);
}
```

Without optimizations:

$\leadsto 42$

Unsafe Code Can Create Aliasing References

```
fn foo(a: &mut i32) { .. }

static mut GLOBAL: i32 = 0;
fn bar() {
    unsafe { println!("{}", &GLOBAL); }
}
fn main() {
    let a = unsafe { &mut GLOBAL };
    foo(a);
}
```

Without optimizations:

$\rightsquigarrow 42$

With optimizations:

$\rightsquigarrow 0$

Why Is This Optimization Legal?

Idea: Declare that our `unsafe` code is wrong!

Why Is This Optimization Legal?

Idea: Declare that our `unsafe` code is wrong!

41

Stacked Borrows: An Aliasing Model for Rust

RALF JUNG, Mozilla, USA and MPI-SWS, Germany
HOANG-HAI DANG, MPI-SWS, Germany
JEEHOON KANG, KAIST, Korea
DEREK DREYER, MPI-SWS, Germany

Type systems are useful not just for the safety guarantees they provide, but also for helping compilers generate more efficient code by simplifying important program analyses. In Rust, the type system imposes a strict

Why Is This Optimization Legal?

Idea: Declare that our `unsafe` code is wrong!

Tree Borrows

NEVEN VILLANI*, Univ. Grenoble Alpes, CNRS, Grenoble INP (Institute of Engineering), France
JOHANNES HOSTERT*, ETH Zurich, Switzerland
DEREK DREYER, MPI-SWS, Germany
RALF JUNG†, ETH Zurich, Switzerland

The Rust programming language is well known for its ownership-based type system, which offers strong guarantees like memory safety and data race freedom. However, Rust also provides *unsafe* escape hatches, for which safety is not guaranteed automatically and must instead be manually upheld by the programmer.

What is Tree Borrows?

What is Tree Borrows?

Wrongly aliased references are defined to **have UB**!

change Operational Semantics

What is Tree Borrows?

Wrongly aliased references are defined to **have UB!**

change Operational Semantics

less UB

more UB

What is Tree Borrows?

Wrongly aliased references **are defined to have UB!**

change Operational Semantics

What is Tree Borrows?

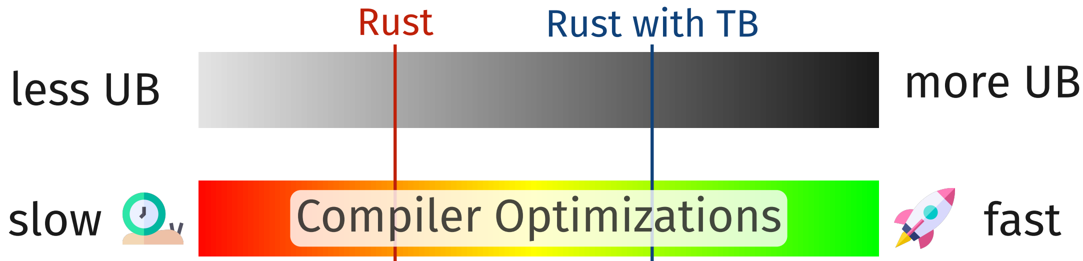
Wrongly aliased references **are defined to have UB!**

change Operational Semantics

What is Tree Borrows?

Wrongly aliased references **are defined to have UB!**

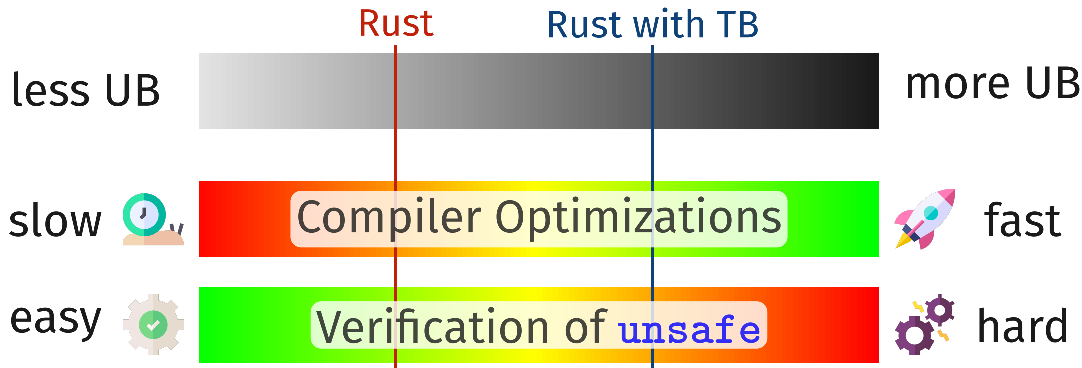
change Operational Semantics



What is Tree Borrows?

Wrongly aliased references are defined to **have UB!**

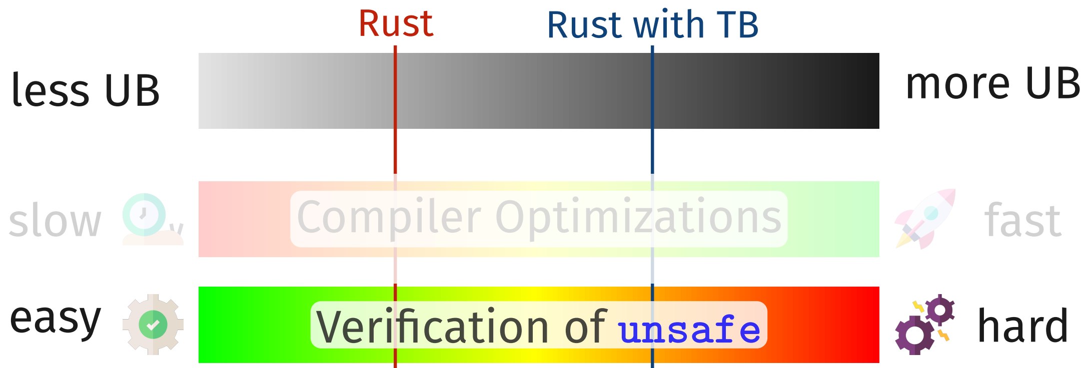
change Operational Semantics



What is Tree Borrows?

Wrongly aliased references are defined to **have UB!**

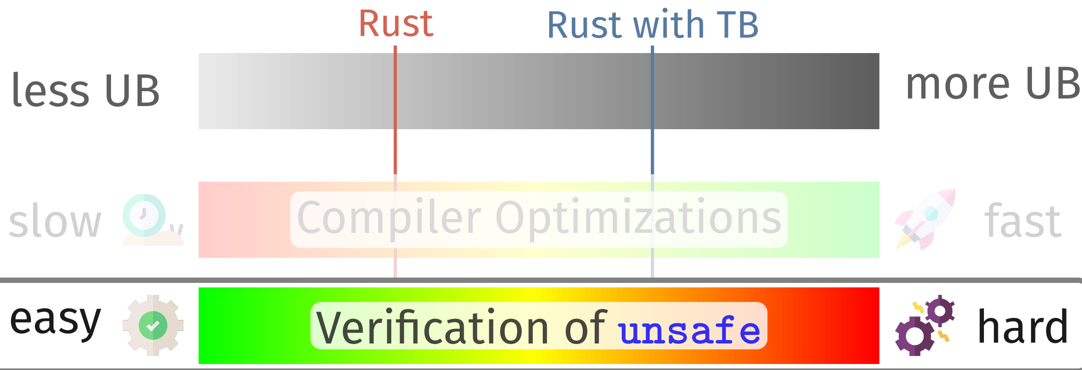
change Operational Semantics



What is Tree Borrows?

Wrongly aliased references are defined to **have UB!**

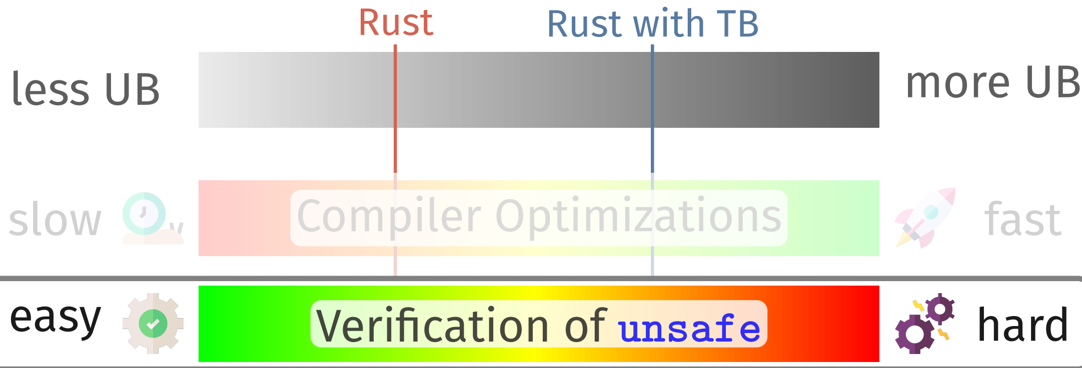
change Operational Semantics



What is Tree Borrows?

Wrongly aliased references **are defined to have UB!**

change Operational Semantics

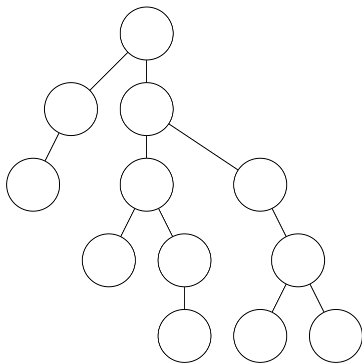


Putting The Borrows Into Trees

Tree Borrows tracks references in a tree data structure:

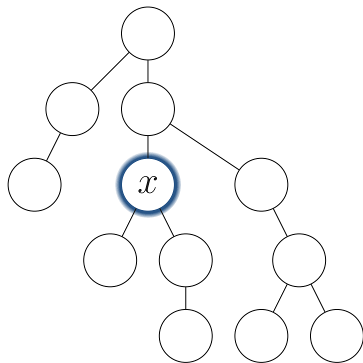
Putting The Borrows Into Trees

Tree Borrows tracks references in a tree data structure:



Putting The Borrows Into Trees

Tree Borrows tracks references in a tree data structure:



Nodes precisely represent references.

One tree for each memory location.

Each access to x splits tree in two:

Putting The Borrows Into Trees

Tree Borrows tracks references in a tree data structure:

Nodes precisely represent references.

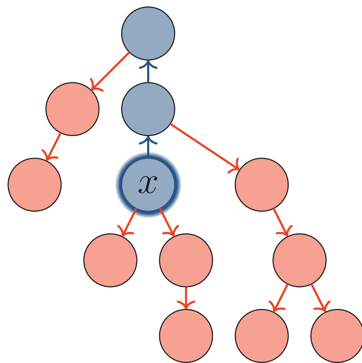
One tree for each memory location.

Each access to x splits tree in two:

- Local: references x is reborrowed from

Putting The Borrows Into Trees

Tree Borrows tracks references in a tree data structure:



Nodes precisely represent references.

One tree for each memory location.

Each access to x splits tree in two:

- Local: references x is reborrowed from
- Foreign: references unrelated to x

Putting The Borrows Into Trees

Tree Borrows tracks references in a tree data structure:

Nodes precisely represent references.

One tree for each memory location.

Each access to x splits tree in two:

- Local: references x is reborrowed from
- Foreign: references unrelated to x

Access to x is foreign access in foreign nodes

Putting The Borrows Into Trees

Tree Borrows tracks references in a tree data structure:

x

Nodes precisely represent references.

One tree for each memory location.

Each access to x splits tree in two:

- Local: references x is reborrowed from
- Foreign: references unrelated to x

Access to x is foreign | local access in foreign | local nodes

Each Node is a State Machine

States represent permissions to read or write.

Each Node is a State Machine

States represent permissions to read or write.

Every access causes transitions at every node, based on

Each Node is a State Machine

States represent permissions to read or write.

Every access causes transitions at every node, based on

- Local ($\uparrow$) vs. Foreign ($\downarrow$) access
- Read (R) vs. Write (W) access

Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
    println!("{}", &GLOBAL);
    *a = x;
}
```

Tree Borrows By Example

```rust
→ static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
    println!("{}", &GLOBAL);
    *a = x;
}
```

Tree Borrows By Example

```
→ static mut GLOBAL: i32 = 0;

  foo(&mut GLOBAL);

  fn foo(a: &mut i32) {
      let x = *a;
      *a = 42;
      println!("{}", &GLOBAL);
      *a = x;
  }
```

GLOBAL: Unique

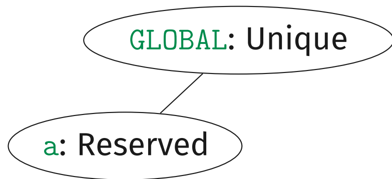
Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

→foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
    println!("{}", &GLOBAL);
    *a = x;
}
```

GLOBAL: Unique

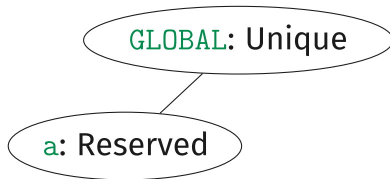
Tree Borrows By Example

```rust
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
    println!("{}", &GLOBAL);
    *a = x;
}
```



Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
→   let x = *a;
    *a = 42;
    println!("{}", &GLOBAL);
    *a = x;
}
```

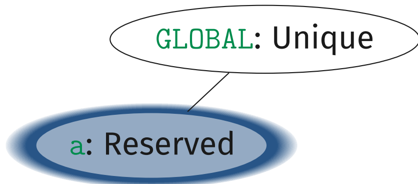


Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
→   let x = *a;
    *a = 42;
    println!("{}", &GLOBAL);
    *a = x;
}
```

Tree Borrows By Example

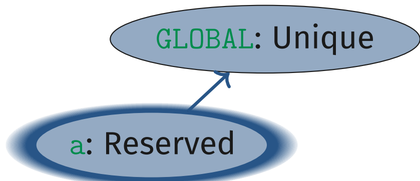
```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
→   let x = *a;
    *a = 42;
    println!("{}", &GLOBAL);
    *a = x;
}
```



Reserved + ↑R → Reserved

Tree Borrows By Example

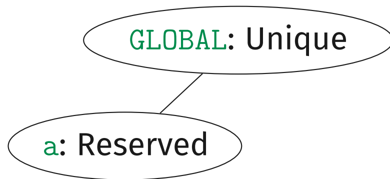
```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
→   let x = *a;
    *a = 42;
    println!("{}", &GLOBAL);
    *a = x;
}
```



Unique + ↑R → Unique

Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
→   *a = 42;
    println!("{}", &GLOBAL);
    *a = x;
}
```

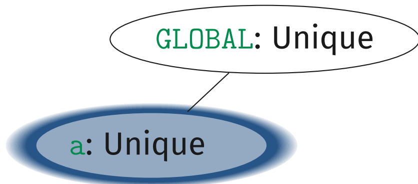


Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
→   *a = 42;
    println!("{}", &GLOBAL);
    *a = x;
}
```

Tree Borrows By Example

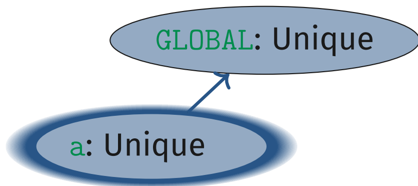
```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
→   *a = 42;
    println!("{}", &GLOBAL);
    *a = x;
}
```



Reserved + ↑W → Unique

Tree Borrows By Example

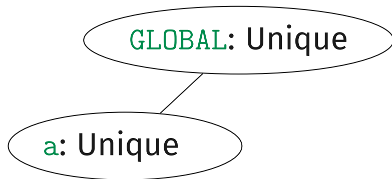
```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
→   *a = 42;
    println!("{}", &GLOBAL);
    *a = x;
}
```



Unique + ↑W → Unique

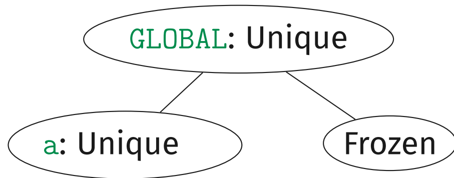
Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
→   println!("{}", &GLOBAL);
    *a = x;
}
```



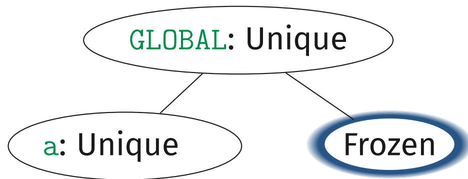
Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
    println!("{}", &GLOBAL);
    *a = x;
}
```



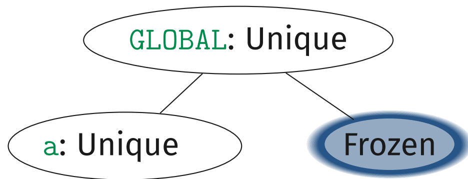
Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
→   println!("{}", &GLOBAL);
    *a = x;
}
```



Tree Borrows By Example

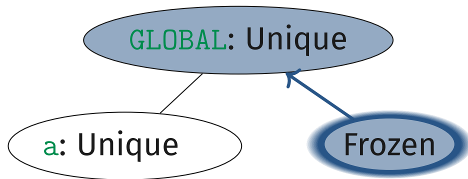
```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
→   println!("{}", &GLOBAL);
    *a = x;
}
```



Frozen + ↑R → Frozen

Tree Borrows By Example

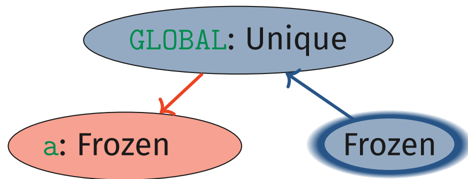
```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
→   println!("{}", &GLOBAL);
    *a = x;
}
```



Unique + ↑R → Unique

Tree Borrows By Example

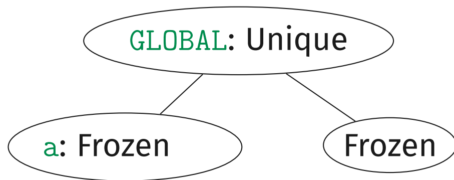
```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
→   println!("{}", &GLOBAL);
    *a = x;
}
```



Unique + ↓R → Frozen

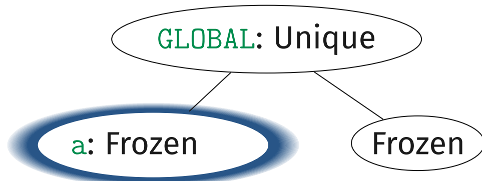
Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
    println!("{}", &GLOBAL);
→   *a = x;
}
```



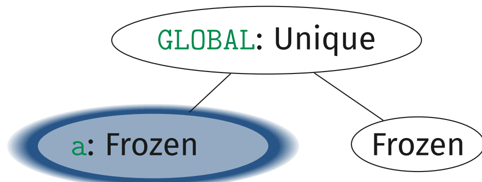
Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
    println!("{}", &GLOBAL);
→   *a = x;
}
```



Tree Borrows By Example

```rust
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

fn foo(a: &mut i32) {
    let x = *a;
    *a = 42;
    println!("{}", &GLOBAL);
→   *a = x;
}
```



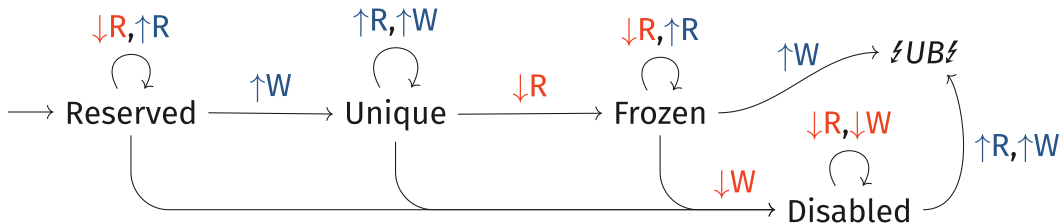
Frozen + ↑W → ↯UB↯

Each Node is a State Machine

States represent permissions to read or write.

Every access causes transitions at every node, based on

- Local (↑) vs. Foreign (↓) access
- Read (R) vs. Write (W) access

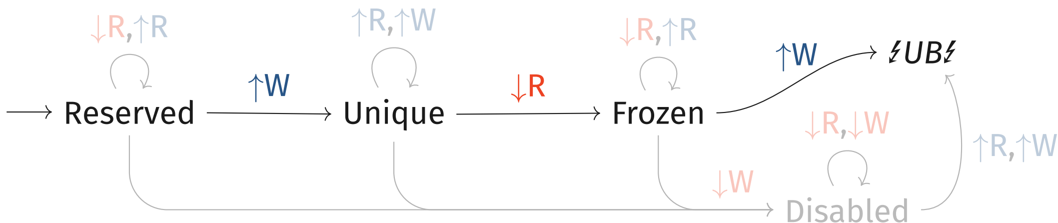


Each Node is a State Machine

States represent permissions to read or write.

Every access causes transitions at every node, based on

- Local (↑) vs. Foreign (↓) access
- Read (R) vs. Write (W) access

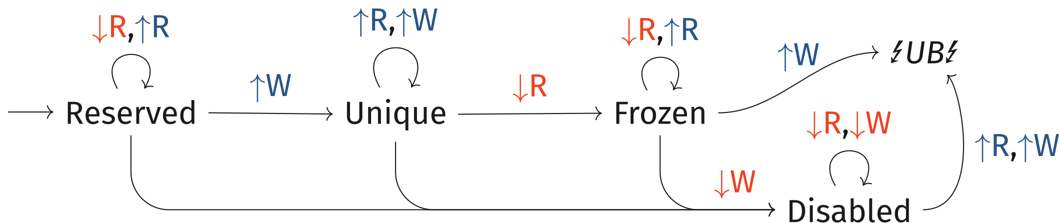


Each Node is a State Machine

States represent permissions to read or write.

Every access causes transitions at every node, based on

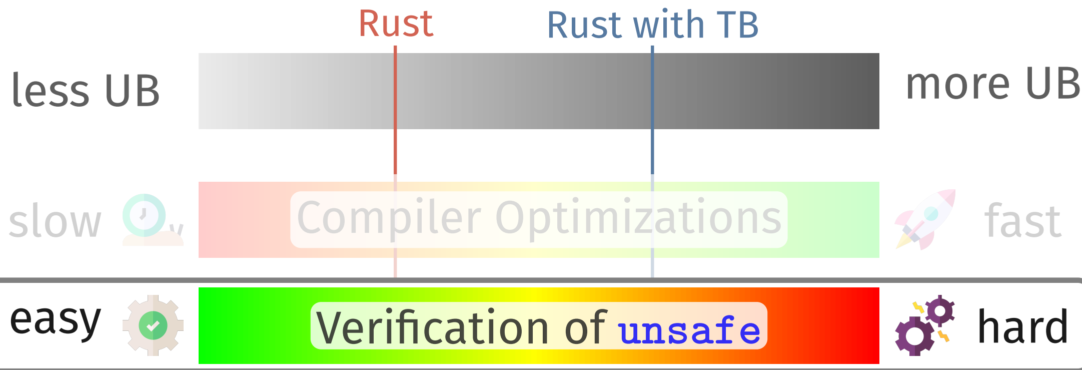
- Local (↑) vs. Foreign (↓) access
- Read (R) vs. Write (W) access



What is Tree Borrows?

Wrongly aliased references are defined to **have UB!**

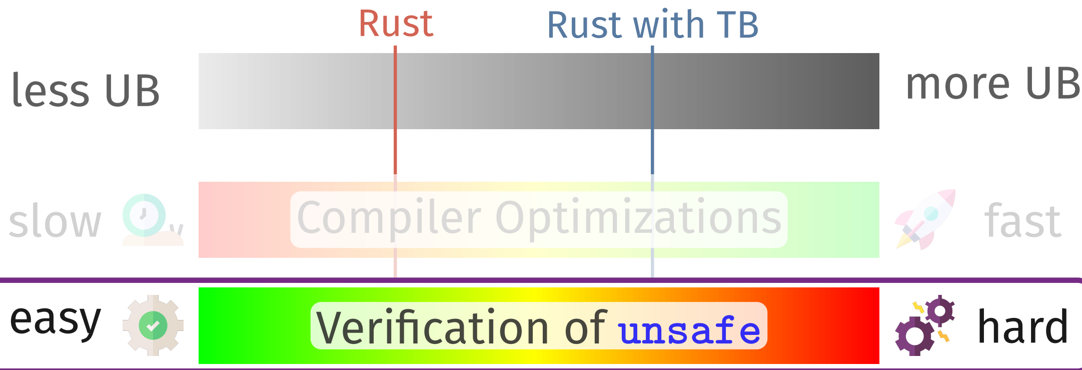
change Operational Semantics



What is Tree Borrows?

Wrongly aliased references are defined to **have UB!**

change Operational Semantics



The Solution: A Program Logic for Tree Borrows

The Solution: A Program Logic for Tree Borrows

Separation Logic

The Solution: A Program Logic for Tree Borrows

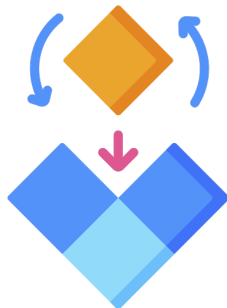
Separation Logic

Full Tree Details

The Solution: A Program Logic for Tree Borrows

Separation Logic

Full Tree Details



Modular Reasoning

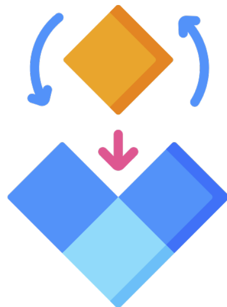
The Solution: A Program Logic for Tree Borrows

Theory at the Foundation of `unsafe` Verification Tools

Prusti, GillianRust, Verifast, Verus, ...

Separation Logic

Full Tree Details



Modular Reasoning

A (Classical) Separation Logic Refresher

$\ell \mapsto v$ Location ℓ stores value v, *exclusive ownership*

A (Classical) Separation Logic Refresher

$\ell \mapsto v$ Location ℓ stores value v, *exclusive ownership*

$P * Q$

A (Classical) Separation Logic Refresher

$\ell \mapsto v$	Location ℓ stores value v, *exclusive ownership*
$P * Q$	P and Q hold separately

A (Classical) Separation Logic Refresher

$\ell \mapsto v$	Location ℓ stores value v, *exclusive ownership*
$P * Q$	P and Q hold separately

```
fn swap(x: &mut i32, y: &mut i32) {..}
```

A (Classical) Separation Logic Refresher

$\ell \mapsto v$ Location ℓ stores value v, *exclusive ownership*

$P * Q$ P and Q hold separately

$$\forall z_1, z_2. \{x \mapsto z_1 * y \mapsto z_2\}$$

```
fn swap(x: &mut i32, y: &mut i32) {..}
```

$$\{x \mapsto z_2 * y \mapsto z_1\}$$

A (Classical) Separation Logic Refresher

$\ell \mapsto v$ Location ℓ stores value v, *exclusive ownership*

$P * Q$ P and Q hold separately

$$\forall z_1, z_2. \{x \mapsto z_1 * y \mapsto z_2\} \qquad \Longrightarrow x \neq y$$

```
fn swap(x: &mut i32, y: &mut i32) {..}
```

$$\{x \mapsto z_2 * y \mapsto z_1\}$$

A (Classical) Separation Logic Refresher

$\ell \mapsto v$ Location ℓ stores value v, *exclusive ownership*

$P * Q$ P and Q hold separately

$\forall z_1, z_2. \{x \mapsto z_1 * y \mapsto z_2\}$ $\implies x \neq y$

```
fn swap(x: &mut i32, y: &mut i32) {..}
```

$\{x \mapsto z_2 * y \mapsto z_1\}$ $\implies$ all other state untouched

A (Classical) Separation Logic Refresher

??	Describes the Tree Borrows state
$\ell \mapsto v$	Location ℓ stores value v, *exclusive ownership*
$P * Q$	P and Q hold separately

$$\forall z_1, z_2. \{x \mapsto z_1 * y \mapsto z_2\} \qquad \Longrightarrow \quad x \neq y$$

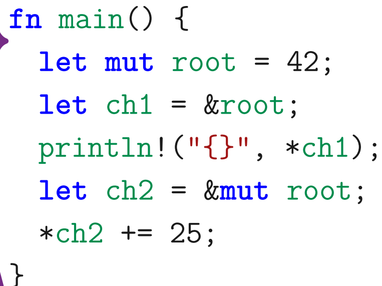
```
fn swap(x: &mut i32, y: &mut i32) {..}
```

$$\{x \mapsto z_2 * y \mapsto z_1\} \qquad \Longrightarrow$$ all other state untouched

Resources in Action

```rust
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  *ch2 += 25;
}
```

Resources in Action



```rust
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  *ch2 += 25;
}
```

Resources in Action

```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  *ch2 += 25;
}
```

$|\texttt{root}| \mapsto 42 *$ root: Unique

Resources in Action

```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  *ch2 += 25;
}
```

$|\texttt{root}| \mapsto 42 *$ root: Unique

ch1: Frozen

Resources in Action

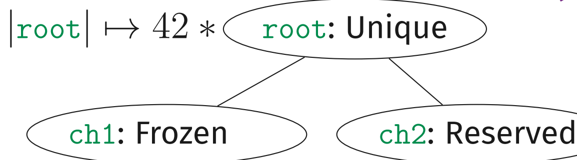
```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  *ch2 += 25;
}
```

$|\texttt{root}| \mapsto 42 \ast$ root: Unique

ch1: Frozen

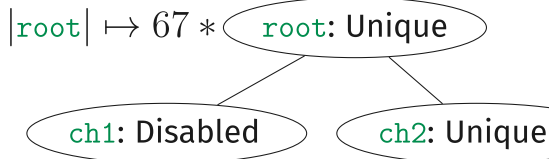
Resources in Action

```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  *ch2 += 25;
}
```



Resources in Action

```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  *ch2 += 25;
}
```



Resources in Action

```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  *ch2 += 25;
}
```

```
fn incr(r : &mut i32) {
  *r += 25;
}
```

$|\texttt{root}| \mapsto 67 \ast$ root: Unique

ch1: Disabled

ch2: Unique

Abstraction

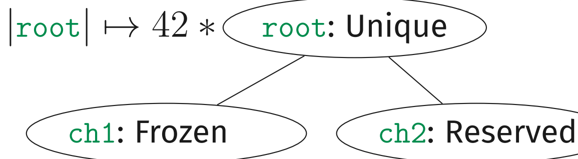
```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  incr(ch2);
}
```

```
fn incr(r : &mut i32) {
  *r += 25;
}
```

Abstraction

```rust
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  incr(ch2);
}
```

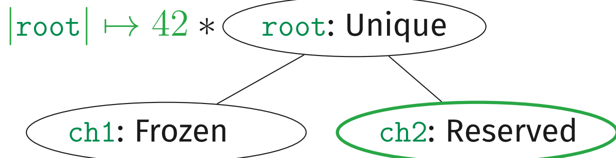
```rust
fn incr(r : &mut i32) {
  *r += 25;
}
```



Abstraction

```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  incr(ch2);
}
```

```
fn incr(r : &mut i32) {
  *r += 25;
}
```



Abstraction

```rust
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  incr(ch2);
}
```

$|\texttt{root}| \mapsto 42 *$ root: Unique

ch1: Frozen

ch2: ———

$\{|\texttt{r}| \mapsto 42 *$ ch2: Reserved $\}$

```rust
fn incr(r : &mut i32) {
  *r += 25;
}
```

$\{|\texttt{r}| \mapsto 67 *$ ch2: Unique $\}$

Abstraction

```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  incr(ch2);
}
```

$\{|\texttt{r}| \mapsto 42 * (\texttt{ch2}\text{: Reserved})\}$

```
fn incr(r : &mut i32) {
  *r += 25;
}
```

$\{|\texttt{r}| \mapsto 67 * (\texttt{ch2}\text{: Unique})\}$

$|\texttt{root}| \mapsto 67 * (\texttt{root}\text{: Unique})$

ch1: Frozen

ch2: Unique

Abstraction

```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  incr(ch2);
}
```

$\{|r| \mapsto 42 *$ (ch2: Reserved) $\}$

```
fn incr(r : &mut i32) {
  *r += 25;
}
```

$\{|r| \mapsto 67 *$ (ch2: Unique) $\}$

$|root| \mapsto 42 *$ (root: Unique)

- ch1: Frozen
- ch2: ——

Abstraction

```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  incr(ch2);
}
```

$|\texttt{root}| \mapsto 42 *$ root: Unique

ch1: Frozen

ch2:

↑W ↓-

↑W | ↓-

$\{|\texttt{r}| \mapsto 42 *$ ch2: Reserved $\}$

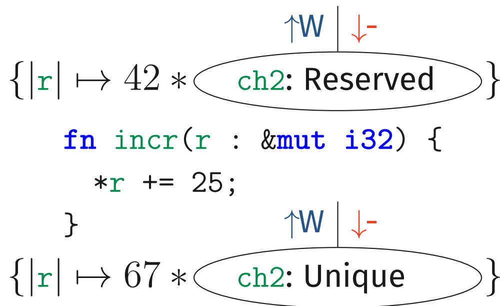
```
fn incr(r : &mut i32) {
  *r += 25;
}
```

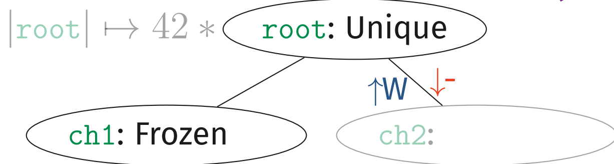
↑W | ↓-

$\{|\texttt{r}| \mapsto 67 *$ ch2: Unique $\}$

Abstraction

```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  incr(ch2);
}
```



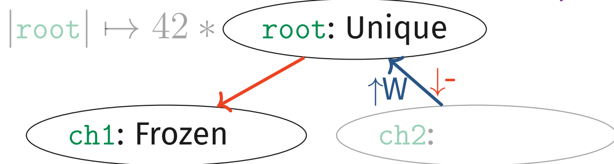


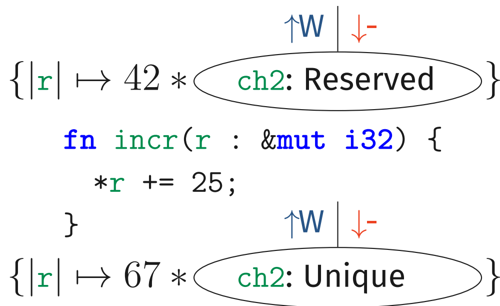
Protocol ↑W↓- imposes:

- ch2 can write
- Others may not access

Abstraction

```rust
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  incr(ch2);
}
```



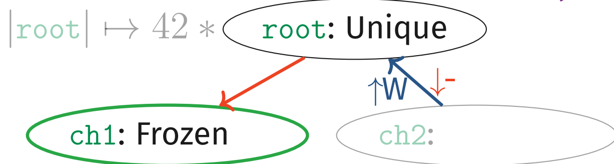


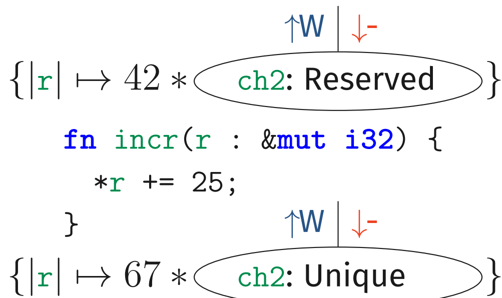
Protocol ↑W↓- imposes:

- ch2 can write
- Others may not access

Abstraction

```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  incr(ch2);
}
```





Protocol ↑W↓- imposes:

- ch2 can write (Invariance)
- Others may not access

Abstraction

```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  incr(ch2);
}
```

$|root| \mapsto 42 *$ root: Unique

ch1: Disabled

ch2:

↑W ↓-

↑W | ↓-

$\{|r| \mapsto 42 *$ ch2: Reserved $\}$

```
fn incr(r : &mut i32) {
  *r += 25;
}
```

↑W | ↓-

$\{|r| \mapsto 67 *$ ch2: Unique $\}$

Protocol ↑W↓- imposes:

- ch2 can write (Invariance)
- Others may not access

Abstraction

```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  incr(ch2);
}
```

$|\text{root}| \mapsto 42 *$ root: Unique

ch1: Disabled

ch2:

↑W ↓-

↑W | ↓-

$\{|\text{r}| \mapsto 42 *$ ch2: Unique $\}$

```
fn incr(r : &mut i32) {
  *r += 25;
}
```

↑W | ↓-

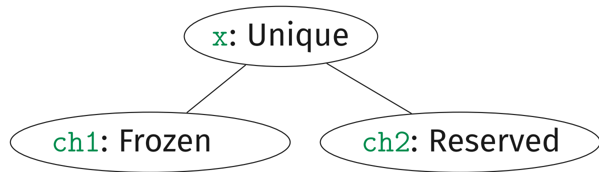
$\{|\text{r}| \mapsto 67 *$ ch2: Unique $\}$

Protocol ↑W↓- imposes:

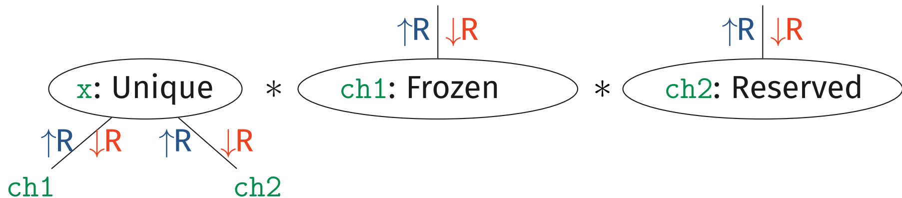
- ch2 can write (Invariance)
- Others may not access

Protocols + Invariance = Local Reasoning

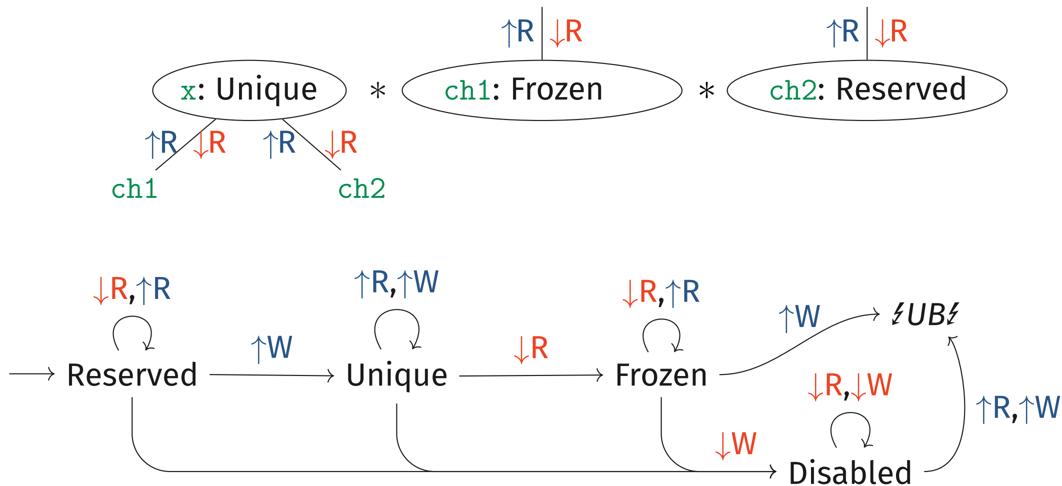
Protocols + Invariance = Local Reasoning



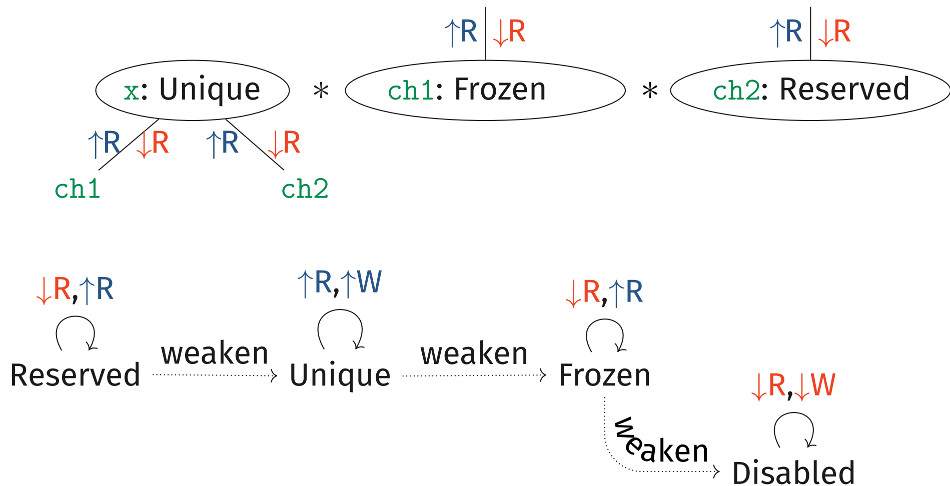
Protocols + Invariance = Local Reasoning



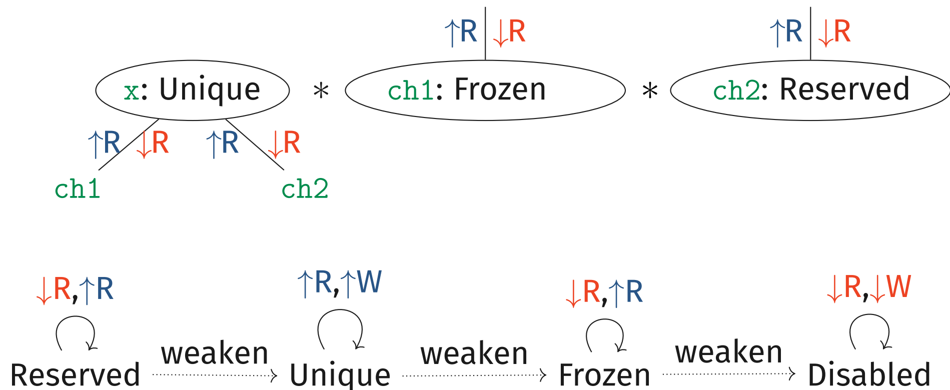
Protocols + Invariance = Local Reasoning



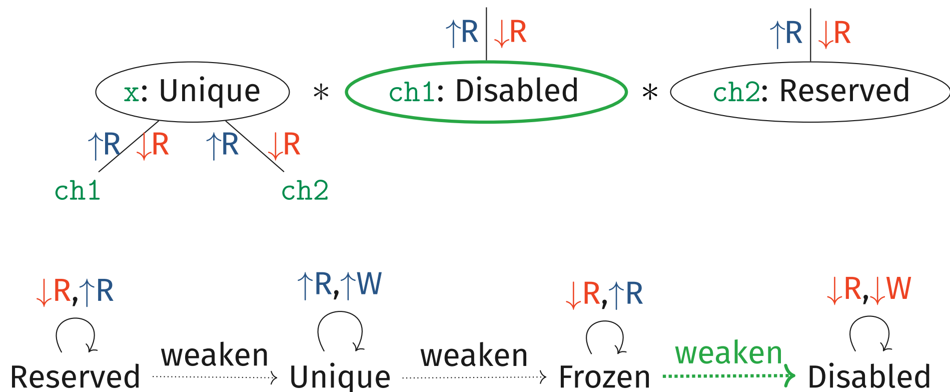
Protocols + Invariance = Local Reasoning



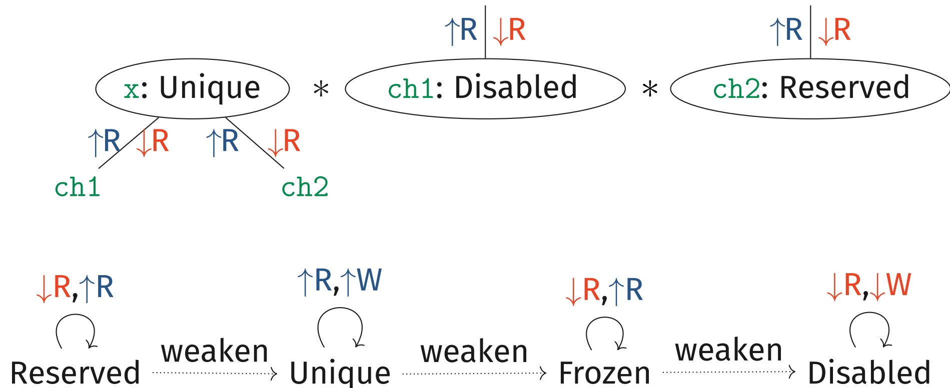
Protocols + Invariance = Local Reasoning



Protocols + Invariance = Local Reasoning

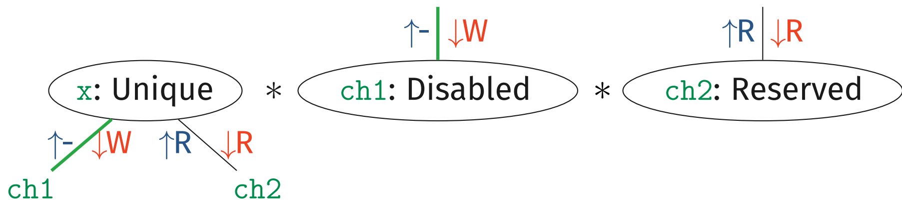


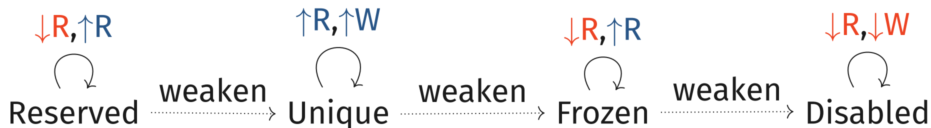
Protocols + Invariance = Local Reasoning



Protocol Compatibility: Nothing (-) $\leq$ Read $\leq$ Write

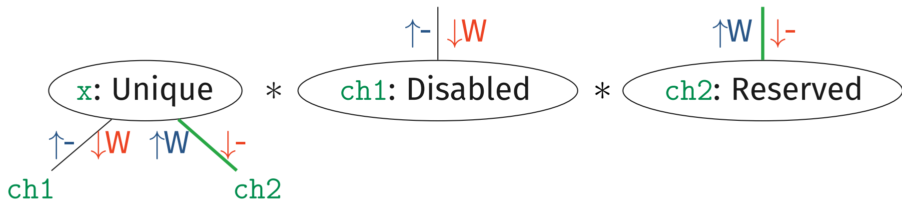
Protocols + Invariance = Local Reasoning

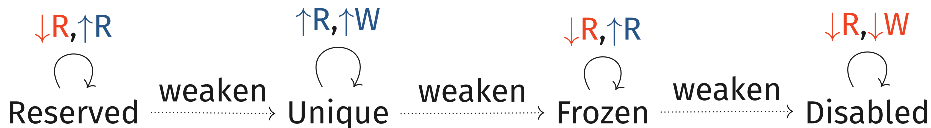




Protocol Compatibility: Nothing (-) $\leq$ Read $\leq$ Write

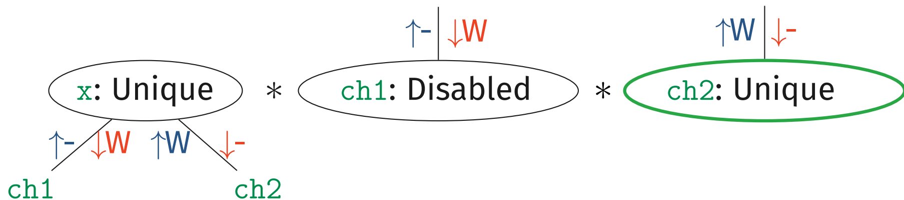
Protocols + Invariance = Local Reasoning

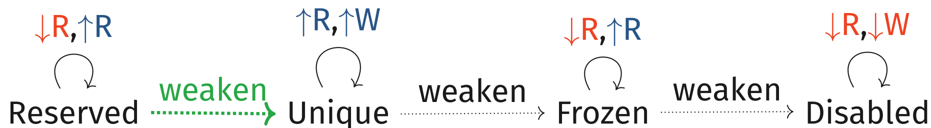




Protocol Compatibility: Nothing (-) $\leq$ Read $\leq$ Write

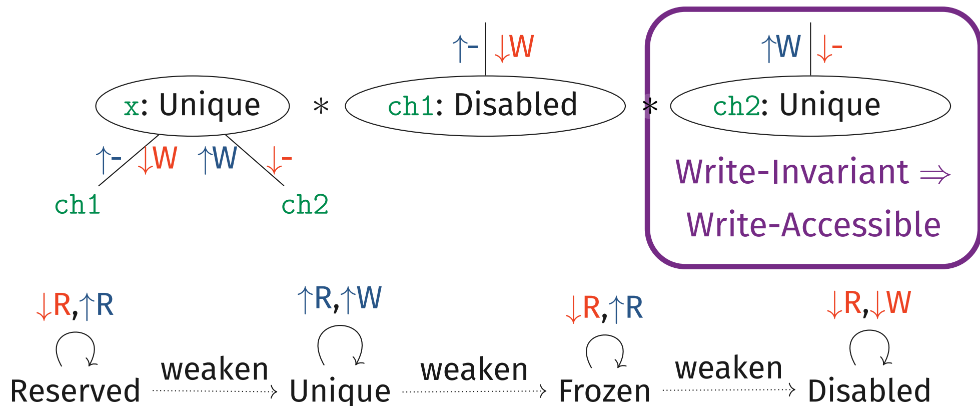
Protocols + Invariance = Local Reasoning





Protocol Compatibility: Nothing (-) $\leq$ Read $\leq$ Write

Protocols + Invariance = Local Reasoning



Protocol Compatibility: Nothing (-) $\leq$ Read $\leq$ Write

This is Just the Beginning

Our logic can do much more:

This is Just the Beginning

Our logic can do much more:

- Node Deletion

This is Just the Beginning

Our logic can do much more:

- Node Deletion, Child Separation (Fractional Retags), ...

This is Just the Beginning

Our logic can do much more:

- Node Deletion, Child Separation (Fractional Retags), ...
- Protectors, thanks to Lazy Protector Ends

This is Just the Beginning

Our logic can do much more:

- Node Deletion, Child Separation (Fractional Retags), ...
- Protectors, thanks to Lazy Protector Ends
- Pointer Arithmetic (“Dynamic Ranges”)

This is Just the Beginning

Our logic can do much more:

- Node Deletion, Child Separation (Fractional Retags), ...
- Protectors, thanks to Lazy Protector Ends
- Pointer Arithmetic (“Dynamic Ranges”)
- Interior Mutability

This is Just the Beginning

Our logic can do much more:

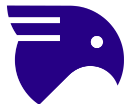
- Node Deletion, Child Separation (Fractional Retags), ...
- Protectors, thanks to Lazy Protector Ends
- Pointer Arithmetic (“Dynamic Ranges”)
- Interior Mutability, ReservedIM, ...

This is Just the Beginning

Our logic can do much more:

- Node Deletion, Child Separation (Fractional Retags), ...
- Protectors, thanks to Lazy Protector Ends
- Pointer Arithmetic (“Dynamic Ranges”)
- *all dark corners* of Tree Borrows

This is Just the Beginning

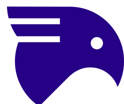


Our logic can do much more:

- Node Deletion, Child Separation (Fractional Retags), ...
- Protectors, thanks to Lazy Protector Ends
- Pointer Arithmetic (“Dynamic Ranges”)
- *all dark corners* of Tree Borrows

Current Status: Iris implementation, small case-studies

This is Just the Beginning



Our logic can do much more:

- Node Deletion, Child Separation (Fractional Retags), ...
- Protectors, thanks to Lazy Protector Ends
- Pointer Arithmetic (“Dynamic Ranges”)
- *all dark corners* of Tree Borrows

Current Status: Iris implementation, small case-studies

Further plans: More examples, simplification, completeness?, verifier integration?? . . .

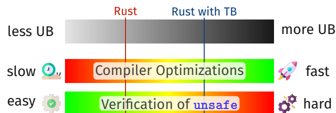
The End

Thanks for your attention!

What is Tree Borrows?

Wrongly aliased references are defined to **have UB**!

change Operational Semantics

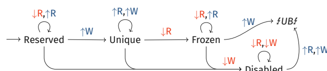


Each Node is a State Machine

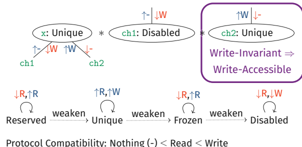
States represent permissions to read or write.

Every access causes transitions at every node, based on

- Local (↑) vs. Foreign (↓) access
- Read (R) vs. Write (W) access



Protocols + Invariance = Local Reasoning



Protocol Compatibility: Nothing (-) ≤ Read ≤ Write

This is Just the Beginning

Our logic can do much more:

- Node Deletion, Child Separation (Fractional Retags), ...
- Protectors, thanks to Lazy Protector Ends
- Pointer Arithmetic ("Dynamic Ranges")
- *all dark corners* of Tree Borrows

Current Status: Iris implementation, small case-studies

Further plans: More examples, simplification, completeness?, verifier integration?? ...

More info:

All icons by https://www.flaticon.com/authors/freepik