A Program Logic for Tree Borrows

Johannes Hostert[1], Ralf Jung[1]

Iris Workshop 2026, Vienna, Austria

14 April 2026

1: ETH Zurich, Switzerland

Rust type system

1-slide summary

Rust type system

1-slide summary

Aliasing: two pointers pointing to the same object

Rust's type system is based on three flavors of ownership:

1. Full ownership: `T`

Rust's type system is based on three flavors of ownership:

1. Full ownership: `T`

2. Mutable reference: `&mut T`
   - not aliased
   - temporarily borrowed

3. Shared reference: `&T`
   - immutable
   - temporarily borrowed

Rust's type system is based on three flavors of ownership:

1. Full ownership: `T`

2. Mutable reference: `&mut T`
   - not aliased
   - temporarily borrowed

3. Shared reference: `&T`
   - immutable
   - temporarily borrowed

Lifetimes `'a` decide how long borrows last (not relevant today).

Rust’s type system is based on three flavors of ownership:

1. Full ownership: `T`

2. Mutable reference: `&mut T`

- r
- t

3. Shared reference: `&T`
   - immutable
   - temporarily borrowed

This system ensures safety!

Can it do more?

Lifetimes `'a` decide how long borrows last (not relevant today).

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:

$\rightsquigarrow 42$

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?

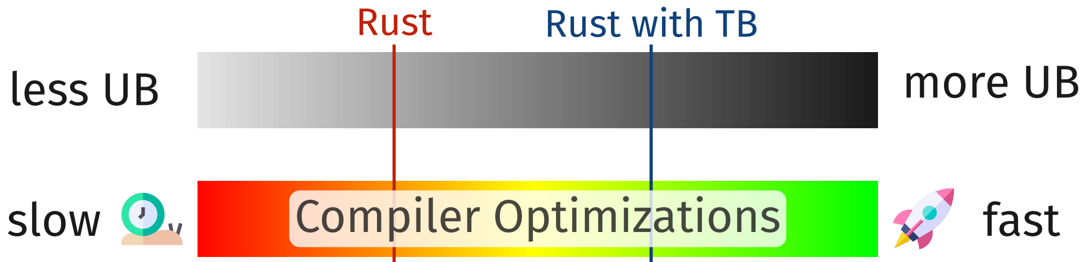
Wrongly aliased references **are defined to have UB!**

change Operational Semantics

What is Tree Borrows?

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

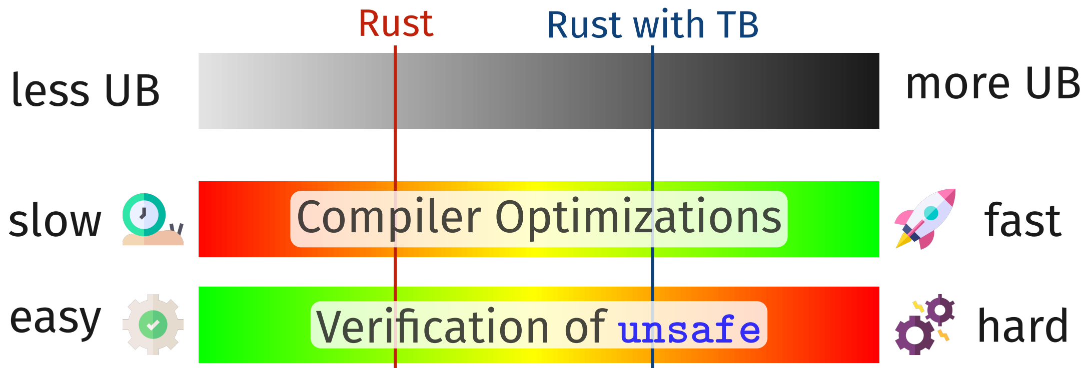
change Operational Semantics



What is Tree Borrows?

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

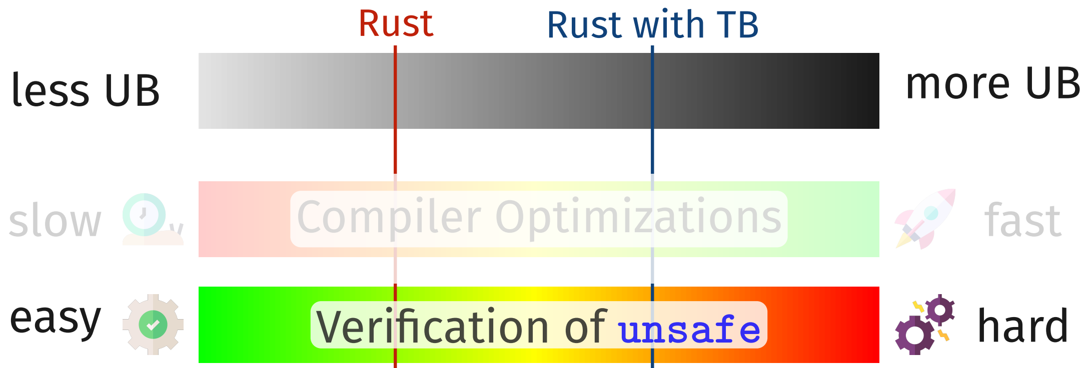
change Operational Semantics



What is Tree Borrows?

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

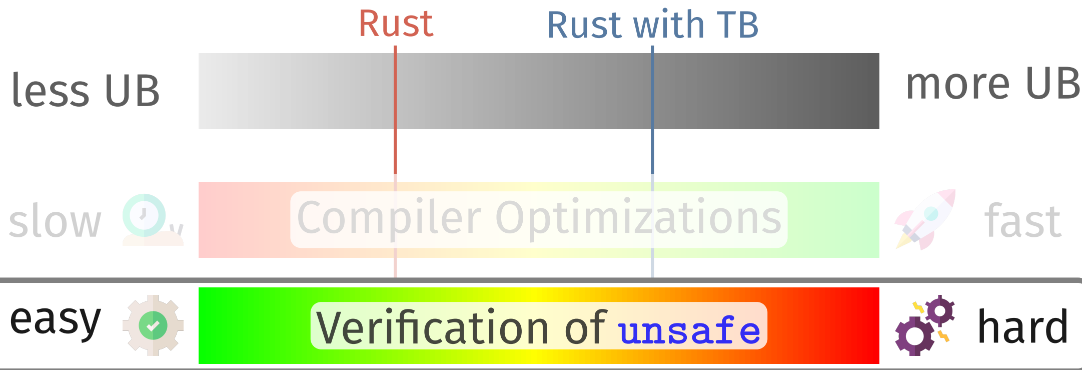
change Operational Semantics



What is Tree Borrows?

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

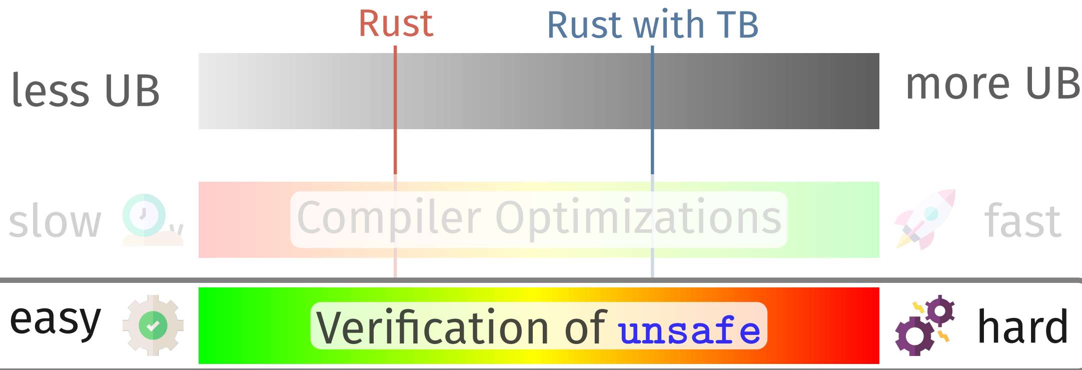
change Operational Semantics



What is Tree Borrows?

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

change Operational Semantics

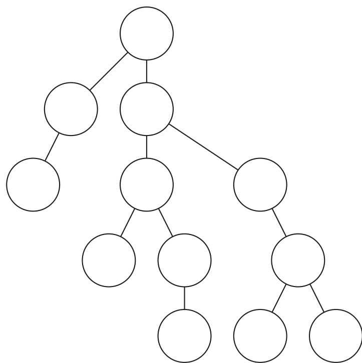


Putting The Borrows Into Trees

Tree Borrows tracks references in a tree data structure:

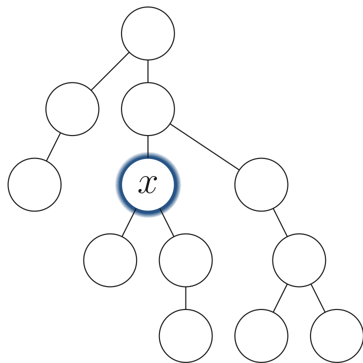
Putting The Borrows Into Trees

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.

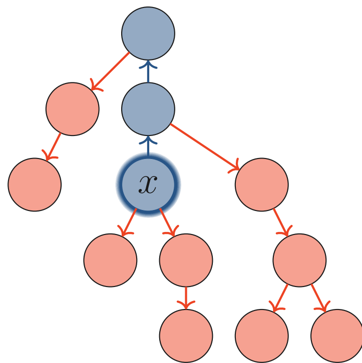
One tree for each memory location.

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:

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 access in foreign nodes

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 | 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 (↑) vs. Foreign (↓) 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

```
→ 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

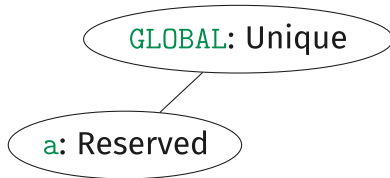
Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

→ foo(&mut GLOBAL);

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

GLOBAL: Unique

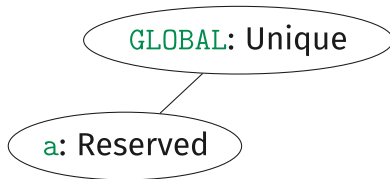
Tree Borrows By Example

```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;
}
```

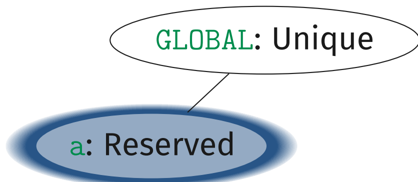


Tree Borrows By Example

```
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

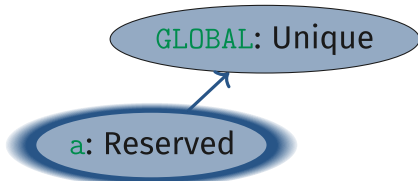
```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

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



Reserved + ↑R → Reserved

Tree Borrows By Example

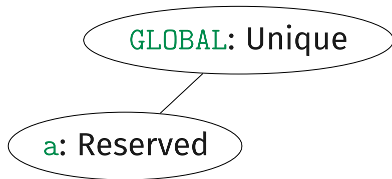
```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

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



Unique + ↑R → Unique

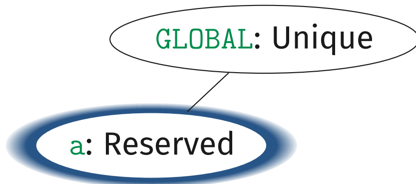
Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

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



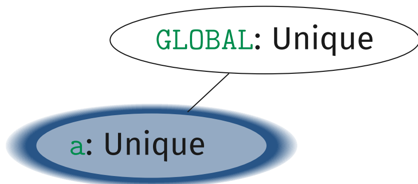
Tree Borrows By Example

```
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

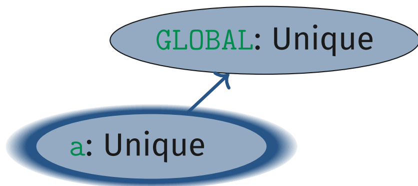
```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

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



Reserved + ↑W → Unique

Tree Borrows By Example

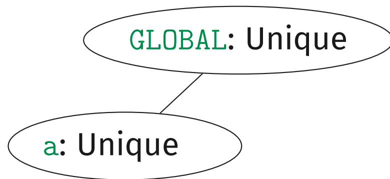
```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

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



Unique + ↑W → Unique

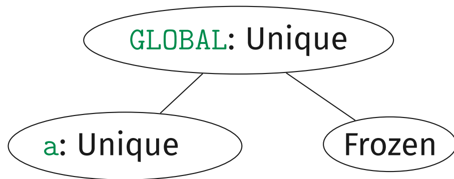
Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

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



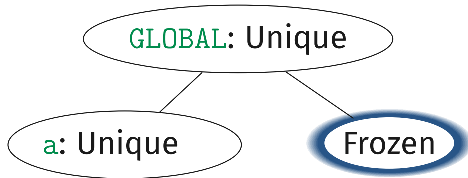
Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

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



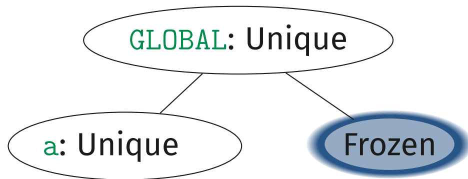
Tree Borrows By Example

```
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

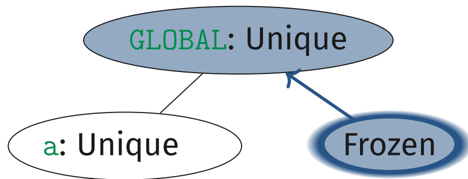
```rust
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

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



Frozen + ↑R → Frozen

Tree Borrows By Example

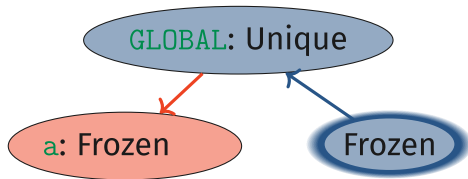
```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

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



Unique + ↑R → Unique

Tree Borrows By Example

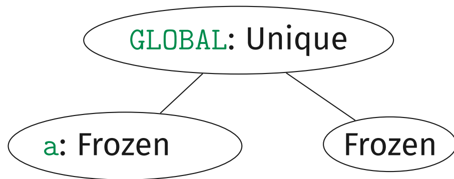
```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

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



Unique + ↓R → Frozen

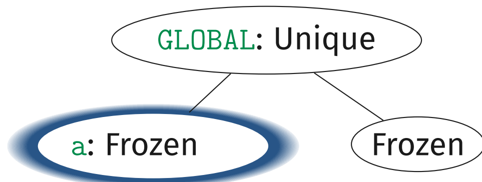
Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

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



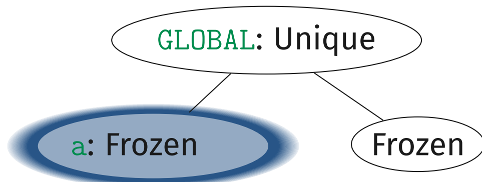
Tree Borrows By Example

```
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

```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

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



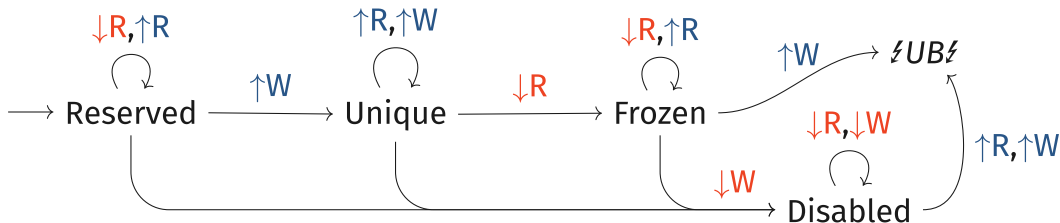
Frozen + ↑W → ⚡UB⚡

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

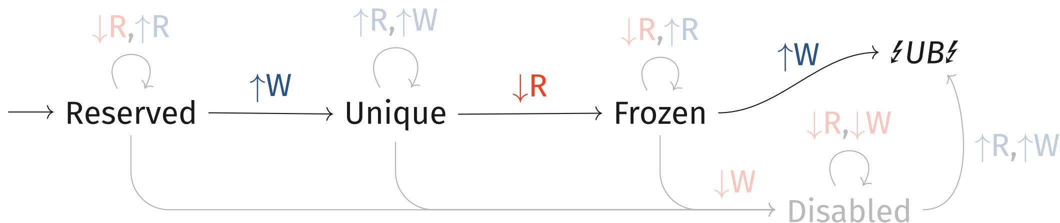


Each Node is a State Machine

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

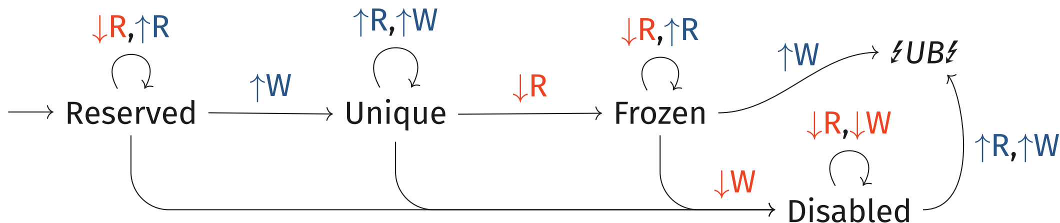


Each Node is a State Machine

States represent permissions to read or write.

Every access causes transitions at every node, based on

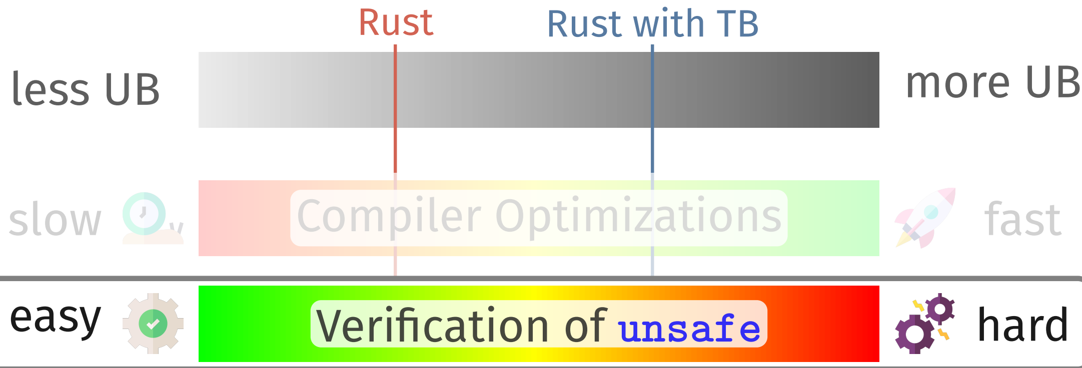
- Local (↑) vs. Foreign (↓) access
- Read (R) vs. Write (W) access



What is Tree Borrows?

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

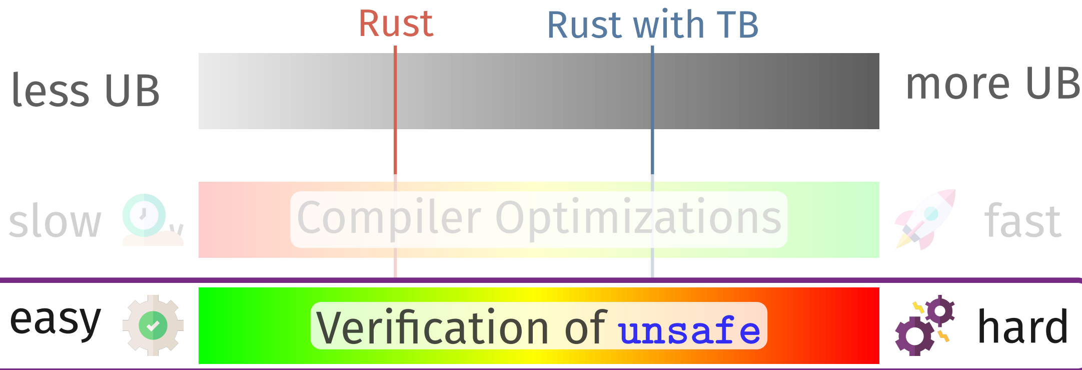
change Operational Semantics



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

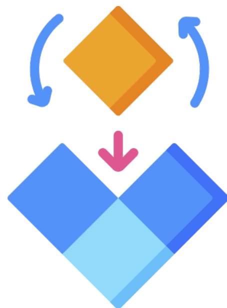
Separation Logic

Full Tree Details

The Solution: A Program Logic for Tree Borrows

Separation Logic

Full Tree Details



Modular Reasoning

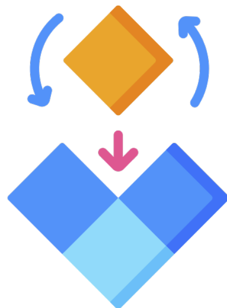
The Solution: A Program Logic for Tree Borrows

Theory at the Foundation of `unsafe` Verification Tools

Prusti, GillianRust, Verifast, Verus, ...

Separation Logic

Full Tree Details

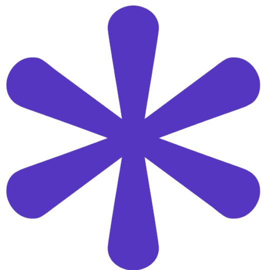


Modular Reasoning

The Solution: A Program Logic for Tree Borrows

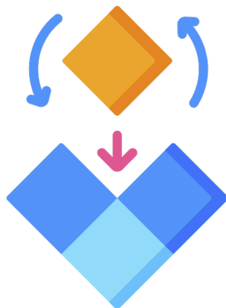
Theory at the Foundation of `unsafe` Verification Tools

Prusti, GillianRust, Verifast, Verus, ...



Separation Logic

Full Tree Details

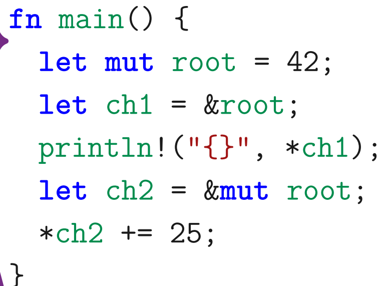


Modular Reasoning

Resources in Action

```
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;
}
```

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

Resources in Action

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

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

ch1: Frozen

Resources in Action

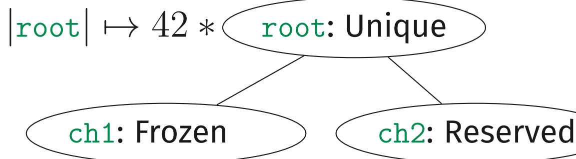
```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  *ch2 += 25;
}
```

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

ch1: Frozen

Resources in Action

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



Resources in Action

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

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

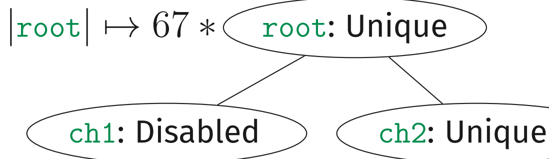
ch1: Disabled

ch2: Unique

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;
}
```



Abstraction

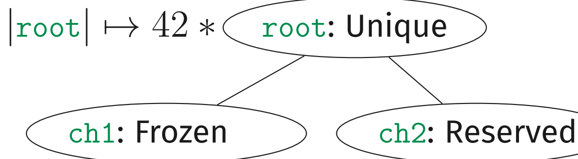
```
fn main() {
  let mut root = 42;
  let ch1 = &root;
  println!("{}", *ch1);
  let ch2 = &mut root;
  incr(ch2);
}
```

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

Abstraction

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

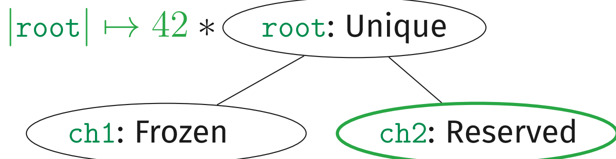
```
fn incr(r : &mut i32) {
  *r += 25;
}
```



Abstraction

```
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

```
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 $\}$

```
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);
}
```

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

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

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

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

- ch1: Frozen
- ch2: Unique

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: Frozen

ch2: ——

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

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

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

Abstraction

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

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

ch1: Frozen

ch2:

↑W ↓-

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

↑W | ↓-

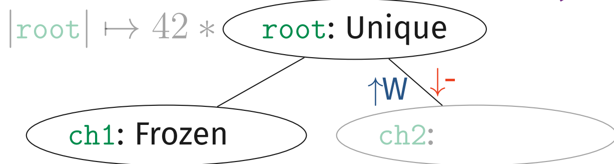
```
fn incr(r : &mut i32) {
  *r += 25;
}
```

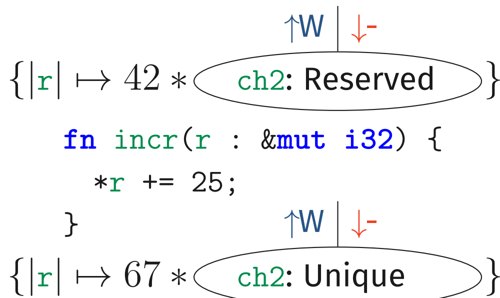
↑W | ↓-

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

Abstraction

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



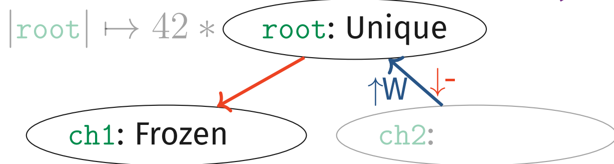


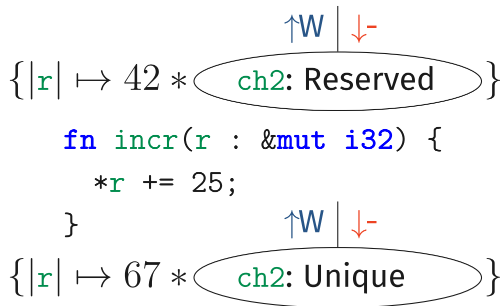
Protocol ↑W↓- imposes:

- 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);
}
```



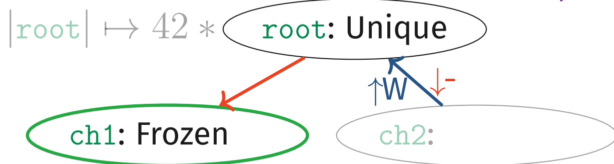


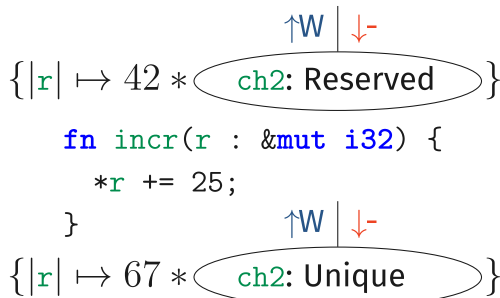
Protocol ↑W↓- imposes:

- ch2 can write
- 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);
}
```

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

ch1: Disabled

ch2:

↑W ↓-

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

↑W | ↓-

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

↑W | ↓-

$\{|\texttt{r}| \mapsto 67 *$ 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);
}
```

$|\mathtt{root}| \mapsto 42 \, *$ root: Unique

ch1: Disabled

ch2:

↑W ↓-

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

↑W | ↓-

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

↑W | ↓-

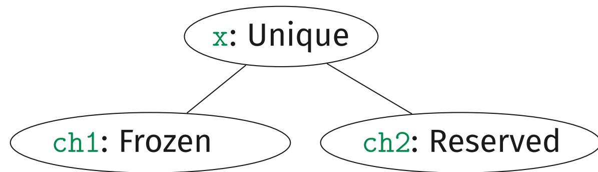
$\{|\mathtt{r}| \mapsto 67 \, *$ ch2: Unique $\}$

Protocol ↑W↓- imposes:

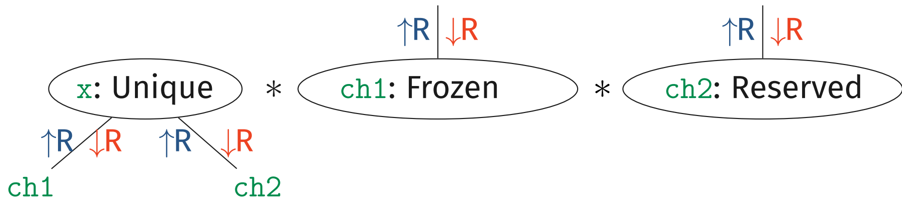
- ch2 can write (Invariance)
- Others may not access

Protocols + Invariance = Local Reasoning

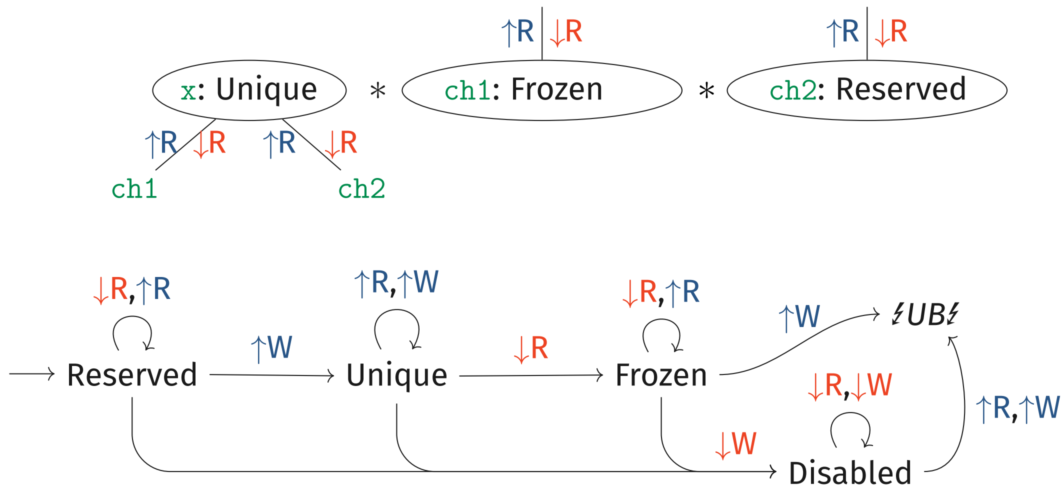
Protocols + Invariance = Local Reasoning



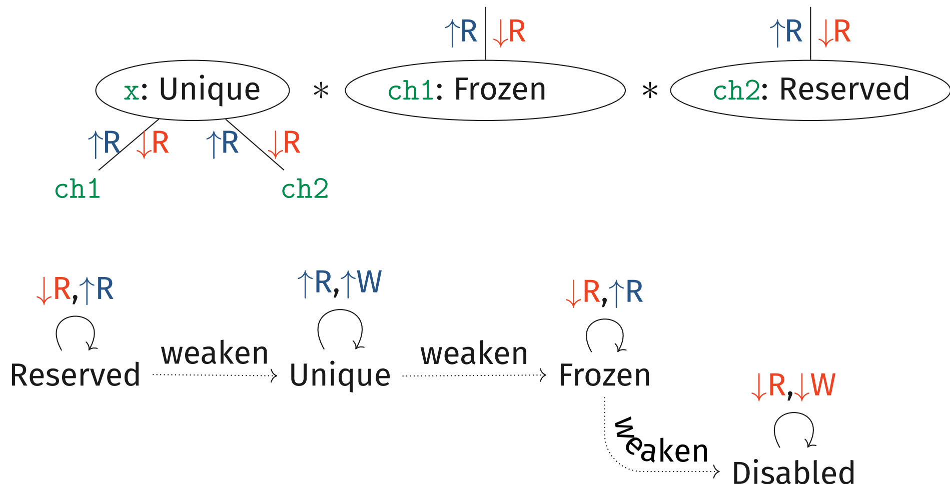
Protocols + Invariance = Local Reasoning



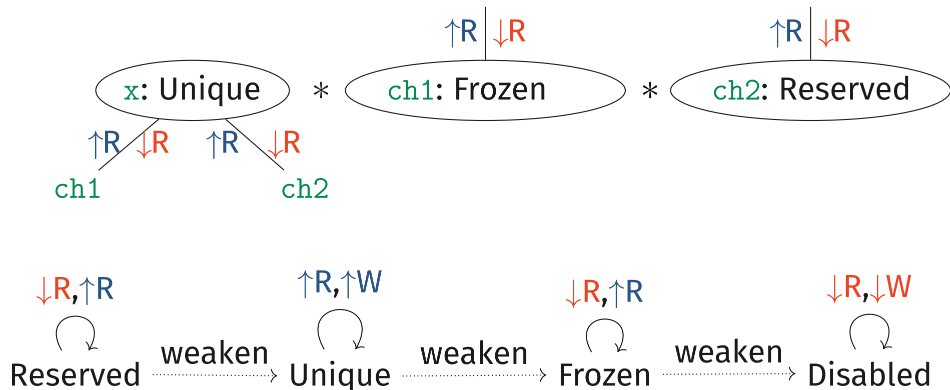
Protocols + Invariance = Local Reasoning



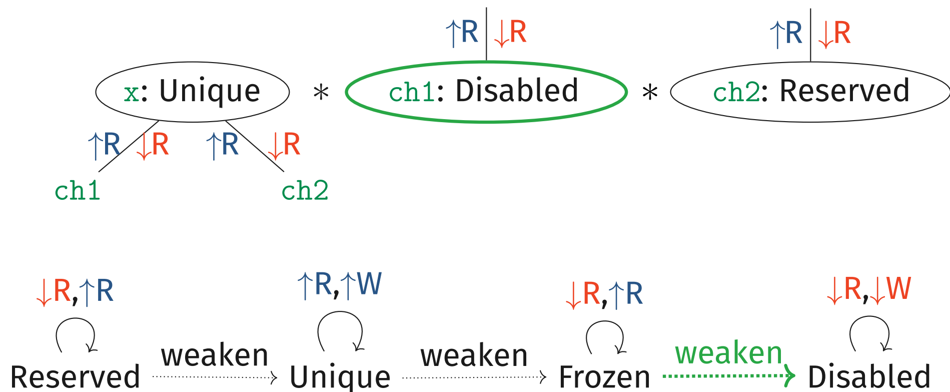
Protocols + Invariance = Local Reasoning



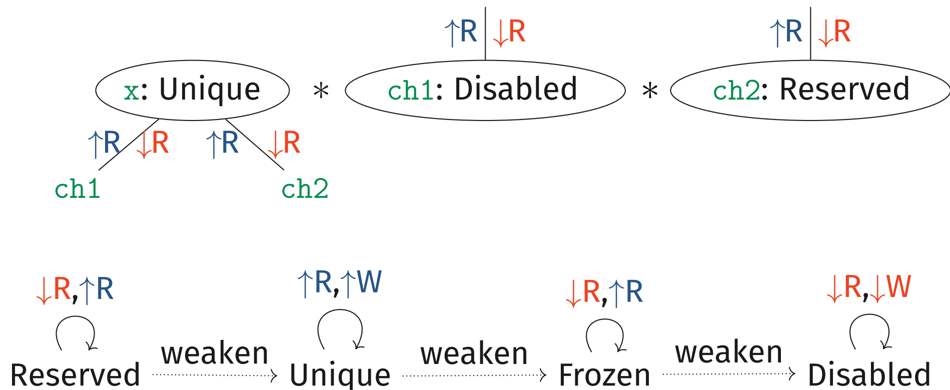
Protocols + Invariance = Local Reasoning



Protocols + Invariance = Local Reasoning

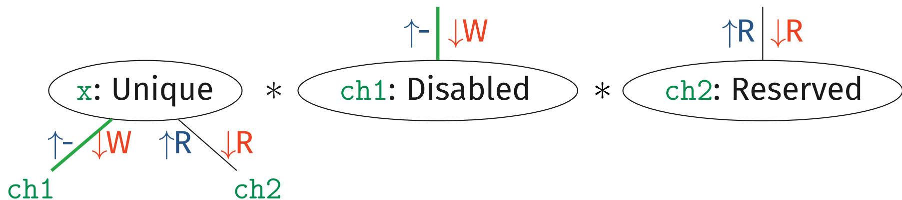


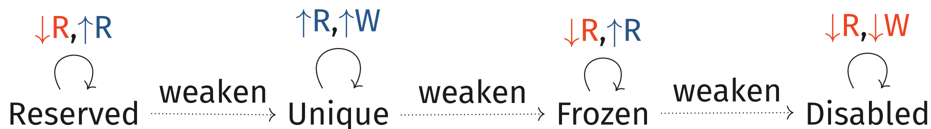
Protocols + Invariance = Local Reasoning



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

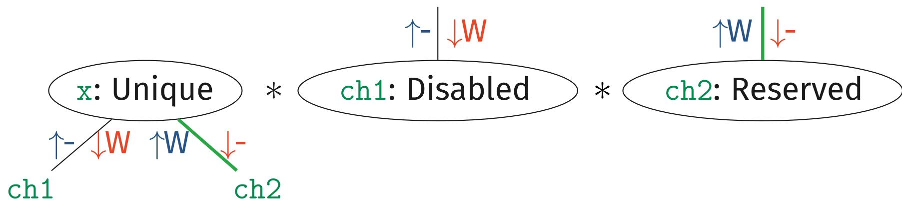
Protocols + Invariance = Local Reasoning

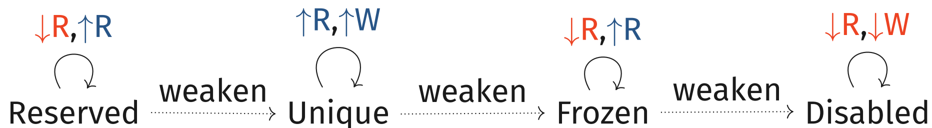




Protocol Compatibility: Nothing (-) $\leq$ Read $\leq$ Write

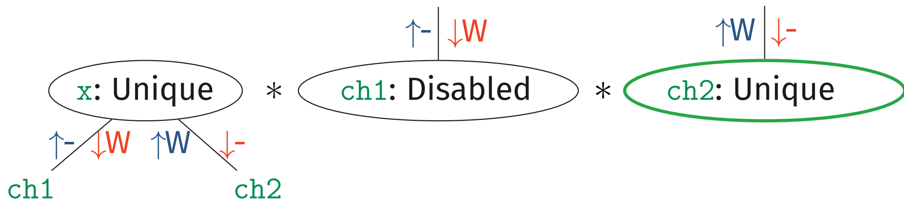
Protocols + Invariance = Local Reasoning

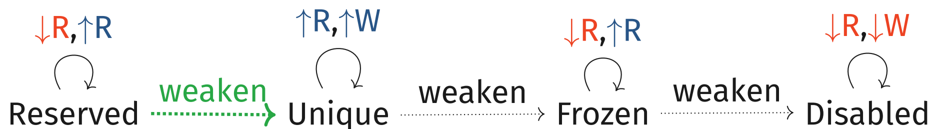




Protocol Compatibility: Nothing (-) $\leq$ Read $\leq$ Write

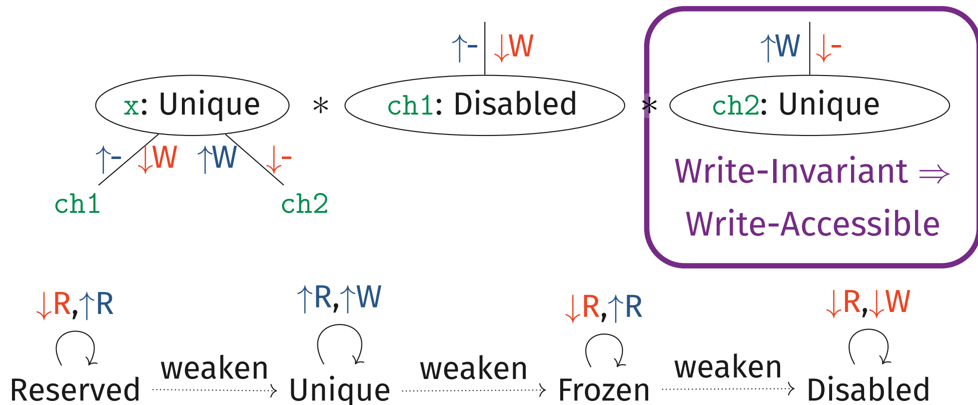
Protocols + Invariance = Local Reasoning





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

Protocols + Invariance = Local Reasoning



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

How Is This Implemented?

I have world's largest view relation

How Is This Implemented?

I have world's largest view relation (I think)

How Is This Implemented?

I have world's largest view relation (I think)

```
Definition tree_cloud_internally_consistent nm :=
  (* constraints on fragments caused by our parent protocols *)
  global_iface_consistency_parents nm
  (* constraints on fragments caused by our children protocols *)
∧ global_iface_consistency_children nm
  (* if a child exists, we must reference it in our list of children (i.e. no dangling children) *)
∧ global_parent_child_consistency nm
  (* if a direct child exists, it's parent protocol must be _identical_ with the one we recorded *)
∧ local_parent_child_consistency nm
  (* all nodes are safe for the protector end action *)
∧ cloud_safe_for_protector_end nm.
```

How Is This Implemented?

I have world's largest view relation (I think)

```
Definition tree_cloud_internally_consistent nm :=
  (* constraints on fragments caused by our parent protocols *)
  global_iface_consistency_parents nm
  (* constraints on fragments caused by our children protocols *)
∧ global_iface_consistency_children nm
  (* if a child exists, we must reference it in our list of children (i.e. no dangling children) *)
∧ global_parent_child_consistency nm
  (* if a direct child exists, it's parent protocol must be _identical_ with the one we recorded *)
∧ local_parent_child_consistency nm
  (* all nodes are safe for the protector end action *)
∧ cloud_safe_for_protector_end nm.
```

about 100 lines just for protocol-protocol interactions

How Is This Implemented?

I have world's largest view relation (I think)

```
Definition tree_cloud_internally_consistent nm :=
  (* constraints on fragments caused by our parent protocols *)
  global_iface_consistency_parents nm
  (* constraints on fragments caused by our children protocols *)
∧ global_iface_consistency_children nm
  (* if a child exists, we must reference it in our list of children (i.e. no dangling children) *)
∧ global_parent_child_consistency nm
  (* if a direct child exists, it's parent protocol must be _identical_ with the one we recorded *)
∧ local_parent_child_consistency nm
  (* all nodes are safe for the protector end action *)
∧ cloud_safe_for_protector_end nm.
```

about 100 lines just for protocol-protocol interactions

10 000 lines of Rocq in total

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:

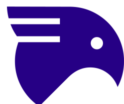
- Node Deletion, Child Separation (Fractional Retags), ...
- Protectors, thanks to Lazy Protector Ends
- Pointer Arithmetic (“Dynamic Ranges”)
- Interior Mutability, ReservedIM, ...

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

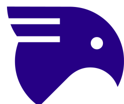


Our logic can do much more:

- 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



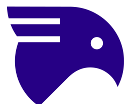
Our logic can do much more:

- 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??, RustBelt 2.0???, ...

This is Just the Beginning



ROCQ
APPROVED

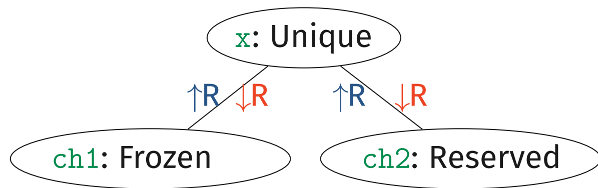
Our logic can do much more:

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

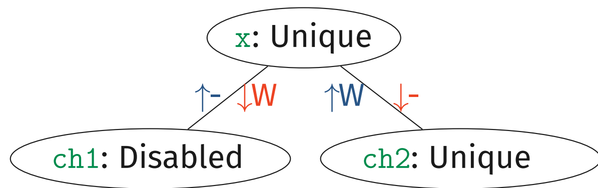
Current Status: Iris implementation, small case-studies

Further plans: More examples, simplification, completeness?, verifier integration??, RustBelt 2.0???, . . .

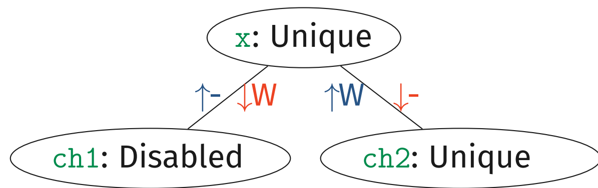
Are Protocols Too Klunky?



Are Protocols Too Klunky?

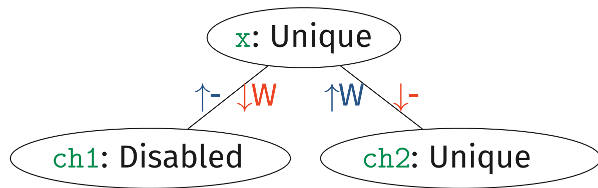


Are Protocols Too Klunky?



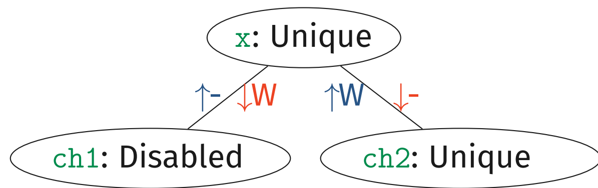
+ Proven Sound

Are Protocols Too Klunky?



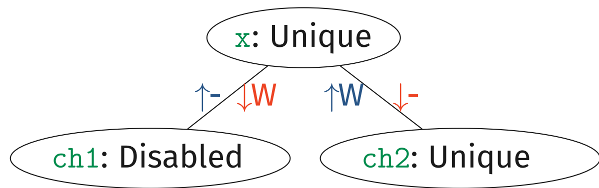
- \+ Proven Sound
- ± Low-Level
  - ⇒ Completeness Obvious

Are Protocols Too Klunky?

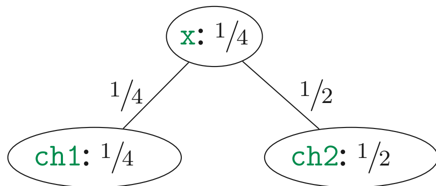


- \+ Proven Sound
- ± Low-Level
  - ⇒ Completeness Obvious
- \- Idiosyncratic Protocols
- \- Exhausting to Use

Are Protocols Too Klunky?



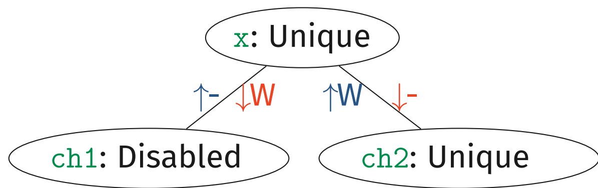
Wannes Tas' idea: use fractions!

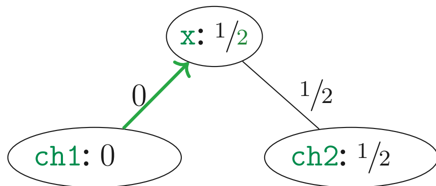


- + Proven Sound
- ± Low-Level
  - ⇒ Completeness Obvious
- - Idiosyncratic Protocols
- - Exhausting to Use

Are Protocols Too Klunky?

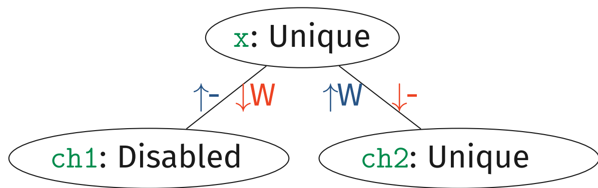
Wannes Tas' idea: use fractions!



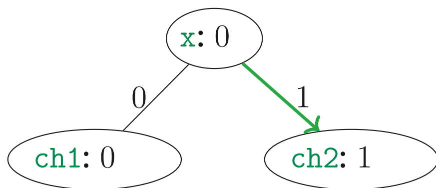


- + Proven Sound
- ± Low-Level
  - ⇒ Completeness Obvious
- \- Idiosyncratic Protocols
- \- Exhausting to Use

Are Protocols Too Klunky?

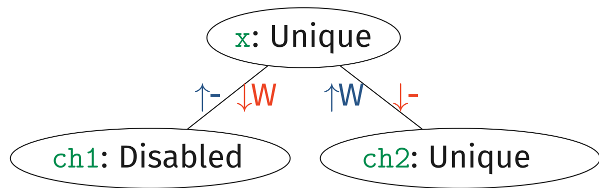


Wannes Tas' idea: use fractions!



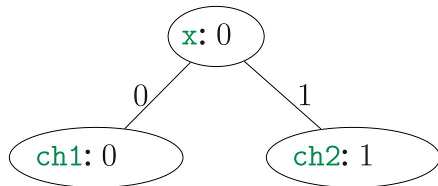
- + Proven Sound
- ± Low-Level
  - ⇒ Completeness Obvious
- - Idiosyncratic Protocols
- - Exhausting to Use

Are Protocols Too Klunky?



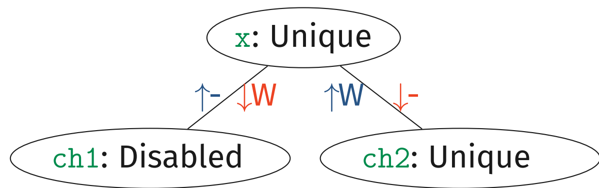
- + Proven Sound
- ± Low-Level
  - ⇒ Completeness Obvious
- - Idiosyncratic Protocols
- - Exhausting to Use

Wannes Tas' idea: use fractions!

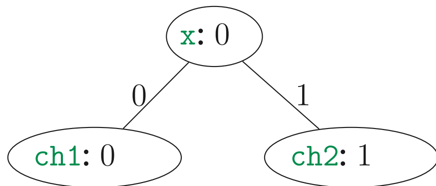


- - Just an idea, no proofs

Are Protocols Too Klunky?



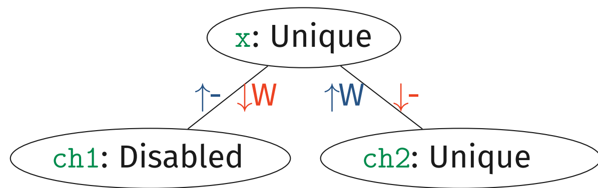
Wannes Tas' idea: use fractions!



- \+ Proven Sound
- ± Low-Level
  - ⇒ Completeness Obvious
- \- Idiosyncratic Protocols
- \- Exhausting to Use

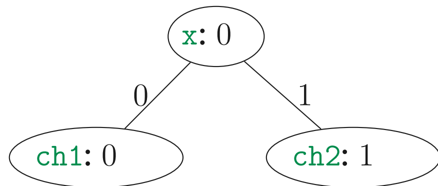
- \- Just an idea, no proofs
- ± Higher-Level
  - ⇒ Completeness Doubtful

Are Protocols Too Klunky?



- + Proven Sound
- ± Low-Level
  - ⇒ Completeness Obvious
- - Idiosyncratic Protocols
- - Exhausting to Use

Wannes Tas' idea: use fractions!



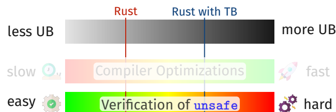
- - Just an idea, no proofs
- ± Higher-Level
  - ⇒ Completeness Doubtful
- + "Natural" Fractions
- + Deeper Intuitions

The End

Thanks for your attention!

What is Tree Borrows?

Wrongly aliased references are defined to **have UB**!
change Operational Semantics

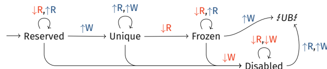


Each Node is a State Machine

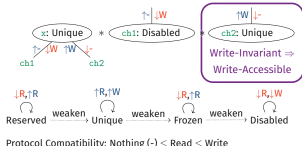
States represent permissions to read or write.

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??, RustBelt 2.0???, …

More info:

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