Leveraging Rust's Lifetimes for Improved Performance and Correctness

Talk for the Aptitude Colloquium of

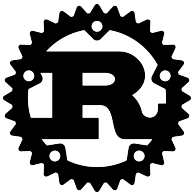
Johannes Hostert

17 September 2024

Rust: A Safe Systems Programming Language

Rust

A language empowering everyone
to build reliable and efficient software.



Rust: A Safe Systems Programming Language

Rust

A language empowering everyone
to build reliable and efficient software.

Rust: A Safe Systems Programming Language

Rust

A language empowering everyone
to build reliable and efficient software.

Rust is the most admired language, more than 80% of developers that use it want to use it again next year. Compare

2023
Developer
Survey

Rust: A Safe Systems Programming Language

Rust

A language empowering everyone
to build reliable and efficient software.

Rust is the most admired language, more than 80% of developers that use it want to use it again next year. Compare

2023 Developer Survey

Rust's addition to the Linux kernel seen as "enormous vote of confidence" in the language

Latest News Published: November 21st, 2022 - Jenna Barron

SD Times SOFTWARE DEVELOPMENT

Rust: A Safe Systems Programming Language

Rust

A language empowering everyone
to build reliable and efficient software.

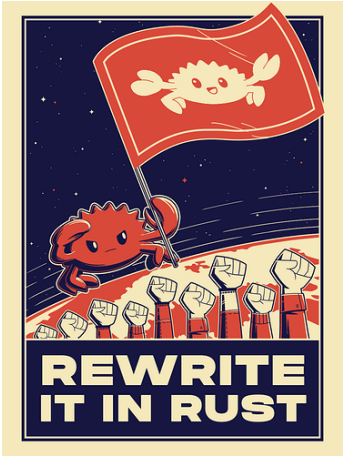
Rust is the most admired language, more than 80% of developers that use it want to use it again next year. Compare

2023 Developer Survey

Rust's addition to the Linux kernel seen as "enormous vote of confidence" in the language

Latest News Published: November 21st, 2022 - Jenna Barron

SD Times SOFTWARE DEVELOPMENT



Rust: References + Lifetimes = Memory + Thread Safety

Unlike C, Rust has several kinds of references (pointers):

Rust: References + Lifetimes = Memory + Thread Safety

Unlike C, Rust has several kinds of references (pointers):

```
& 'a mut T          & 'a T
```

Rust: References + Lifetimes = Memory + Thread Safety

Unlike C, Rust has several kinds of references (pointers):

```
& 'a mut T
```
Mutable

```
& 'a T
```
Immutable

Rust: References + Lifetimes = Memory + Thread Safety

Unlike C, Rust has several kinds of references (pointers):

`& 'a mut T`	`& 'a T`
Mutable	Immutable
Exclusive	Aliased/Shared

Rust: References + Lifetimes = Memory + Thread Safety

Unlike C, Rust has several kinds of references (pointers):

`&'a mut T`	`&'a T`
Mutable	Immutable
Exclusive	Aliased/Shared

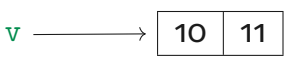
```
let mut v = vec![10, 11];
let vptr = &'a mut v[1];
Vec::push(&'b mut v, 12);
println!("v[1] = {}", *vptr);
```

Rust: References + Lifetimes = Memory + Thread Safety

Unlike C, Rust has several kinds of references (pointers):

`&'a mut T`	`&'a T`
Mutable	Immutable
Exclusive	Aliased/Shared

```
let mut v = vec![10, 11];
let vptr = &'a mut v[1];
Vec::push(&'b mut v, 12);
println!("v[1] = {}", *vptr);
```

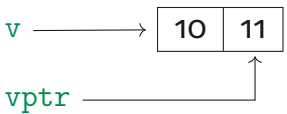


Rust: References + Lifetimes = Memory + Thread Safety

Unlike C, Rust has several kinds of references (pointers):

`&'a mut T`	`&'a T`
Mutable	Immutable
Exclusive	Aliased/Shared

```
let mut v = vec![10, 11];
let vptr = &'a mut v[1];
Vec::push(&'b mut v, 12);
println!("v[1] = {}", *vptr);
```

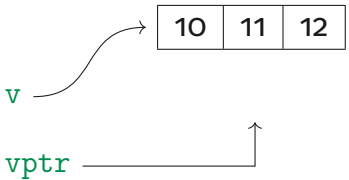


Rust: References + Lifetimes = Memory + Thread Safety

Unlike C, Rust has several kinds of references (pointers):

`&'a mut T`	`&'a T`
Mutable	Immutable
Exclusive	Aliased/Shared

```
let mut v = vec![10, 11];
let vptr = &'a mut v[1];
Vec::push(&'b mut v, 12);
println!("v[1] = {}", *vptr);
```

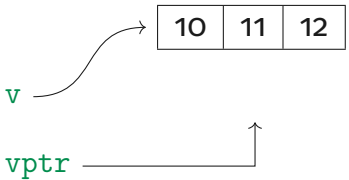


Rust: References + Lifetimes = Memory + Thread Safety

Unlike C, Rust has several kinds of references (pointers):

`&'a mut T`	`&'a T`
Mutable	Immutable
Exclusive	Aliased/Shared

```
let mut v = vec![10, 11];
let vptr = &'a mut v[1];
Vec::push(&'b mut v, 12);
println!("v[1] = {}", *vptr);
```



Rust: References + Lifetimes = Memory + Thread Safety

Unlike C, Rust has several kinds of references (pointers):

`&'a mut T`	`&'a T`
Mutable	Immutable
Exclusive	Aliased/Shared

```
let mut v = vec![10, 11];
let vptr = &'a mut v[1];
Vec::push(&'b mut v, 12);
println!("v[1] = {}", *vptr);
```

Lifetime `'b`

Lifetime `'a`

Rust: References + Lifetimes = Memory + Thread Safety

Unlike C, Rust has several kinds of references (pointers):

`&'a mut T`	`&'a T`
Mutable	Immutable
Exclusive	Aliased/Shared

```
let mut v = vec![10, 11];
let vptr = &'a mut v[1];
Vec::push(&'b mut v, 12);
println!("v[1] = {}", *vptr);
```

Lifetime `'b`

Lifetime `'a`

Rust: References + Lifetimes = Memory + Thread Safety

Unlike C, Rust has several kinds of references (pointers):

`&'a mut T`	`&'a T`	`*mut T`
Mutable	Immutable	Mutable
Exclusive	Aliased/Shared	Aliased
		⚠️ `unsafe` ⚠️

```
let mut v = vec![10, 11];
let vptr = &'a mut v[1];
Vec::push(&'b mut v, 12);
println!("v[1] = {}", *vptr);
```

Lifetime `'b`

Lifetime `'a`

Leveraging Rust’s Lifetimes

Leveraging Rust's Lifetimes

for

and

Leveraging Rust's Lifetimes

for

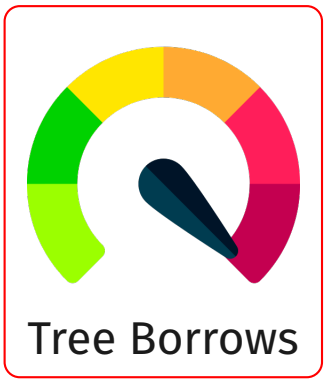
and

Tree Borrows

Spec Testing

Leveraging Rust's Lifetimes

for



Tree Borrows

and

Spec Testing

References Forbid Aliasing

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

References Forbid Aliasing

Consider an optimizing compiler:

```
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;
}
```

References Forbid Aliasing

Consider an optimizing compiler:

```
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;
}
```

References Forbid Aliasing

Consider an optimizing compiler:

```
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;
}
```

References Forbid Aliasing

Consider an optimizing compiler:

```
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;
}
```

References Forbid Aliasing

Consider an optimizing compiler:

```
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`.

References Forbid Aliasing

Consider an optimizing compiler:

```
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

Consider an optimizing compiler:

```
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:

$\rightsquigarrow 0$

How To Recover This Optimization?

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

How To Recover This Optimization?

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

Type system

How To Recover This Optimization?

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

Type system

Borrow Checker

How To Recover This Optimization?

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

Type system

Borrow Checker

`unsafe` code opts out

How To Recover This Optimization?

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

Type system

Borrow Checker

`unsafe` code opts out

$\Rightarrow$ Not useful for optimizations

How To Recover This Optimization?

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

Type system

Borrow Checker

`unsafe` code opts out

$\Rightarrow$ Not useful for optimizations

Operational Semantics

How To Recover This Optimization?

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

Type system	Operational Semantics
Borrow Checker	
`unsafe` code opts out	No opt-out from *UB*
$\Rightarrow$ Not useful for optimizations	

How To Recover This Optimization?

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

Type system	Operational Semantics
Borrow Checker	
`unsafe` code opts out	No opt-out from *UB*
$\Rightarrow$ Not useful for optimizations	$\Rightarrow$ Useful for optimizations

How To Recover This Optimization?

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

Type system	Operational Semantics
Borrow Checker	Aliasing Model
`unsafe` code opts out	No opt-out from *UB*
$\Rightarrow$ Not useful for optimizations	$\Rightarrow$ Useful for optimizations

How To Recover This Optimization?

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

Type system	Operational Semantics
Borrow Checker	Aliasing Model
`unsafe` code opts out	No opt-out from *UB*
$\Rightarrow$ Not useful for optimizations	$\Rightarrow$ Useful for optimizations

Aliasing model tracks aliasing, declares our `unsafe` code is *UB*.

How To Recover This Optimization?

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

Type system	Operational Semantics
Borrow Checker	Aliasing Model
`unsafe` code opts out	No opt-out from *UB*
$\Rightarrow$ Not useful for optimizations	$\Rightarrow$ Useful for optimizations

Aliasing model tracks aliasing, declares our `unsafe` code is *UB*.

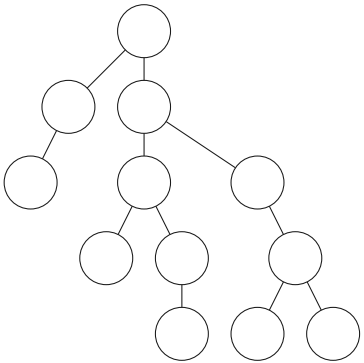
Aliasing model is purely *ghost*, not present in compiled binary.

Putting The Borrows Into Trees

Aliasing model tracks references in a tree data structure:

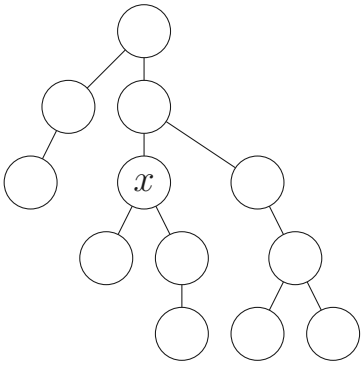
Putting The Borrows Into Trees

Aliasing model tracks references in a tree data structure:



Putting The Borrows Into Trees

Aliasing model tracks references in a tree data structure:

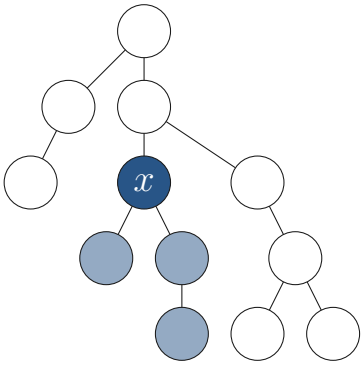


Nodes precisely represent references.

Each node x splits tree in two:

Putting The Borrows Into Trees

Aliasing model tracks references in a tree data structure:



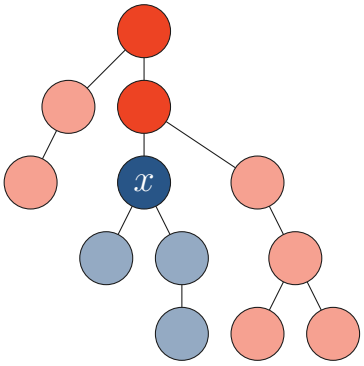
Nodes precisely represent references.

Each node x splits tree in two:

- Local: references x “knows about”

Putting The Borrows Into Trees

Aliasing model tracks references in a tree data structure:



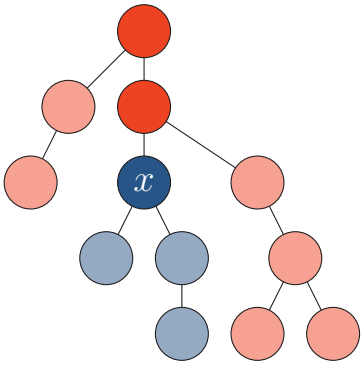
Nodes precisely represent references.

Each node x splits tree in two:

- Local: references x “knows about”
- Foreign: reference x does not know about

Putting The Borrows Into Trees

Aliasing model tracks references in a tree data structure:



Nodes precisely represent references.

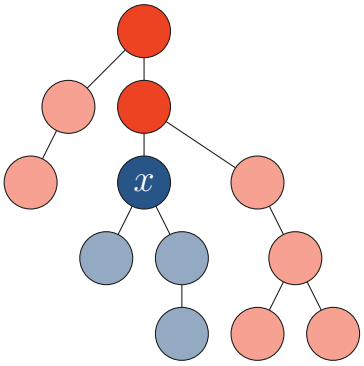
Each node x splits tree in two:

- Local: references x "knows about"
- Foreign: reference x does not know about

Access to foreign reference: foreign access

Putting The Borrows Into Trees

Aliasing model tracks references in a tree data structure:



Nodes precisely represent references.

Each node x splits tree in two:

- Local: references x "knows about"
- Foreign: reference x does not know about

Access to	foreign reference:	foreign access
	local	local

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

```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;
}
```

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;
}
```

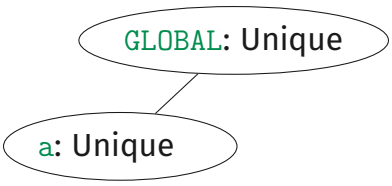
Reserved + $\uparrow$R $\rightarrow$ Reserved

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;
}
```

Tree Borrows By Example

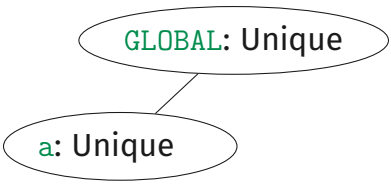
```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

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



Reserved + ↑W → Unique

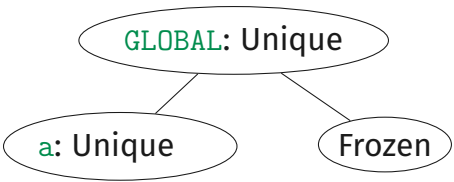
Tree Borrows By Example

```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

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



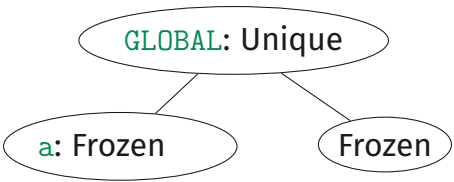
Tree Borrows By Example

```
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

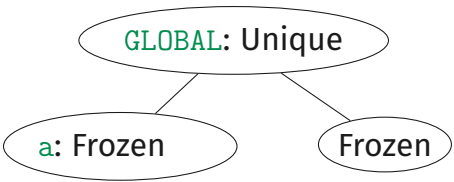
```
static mut GLOBAL: i32 = 0;

foo(&mut GLOBAL);

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



Unique + $\downarrow R \rightarrow$ Frozen

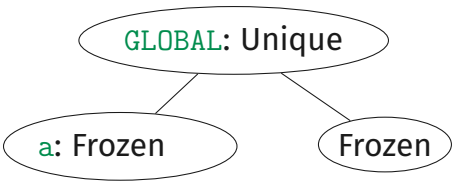
Tree Borrows By Example

```
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;
}
```



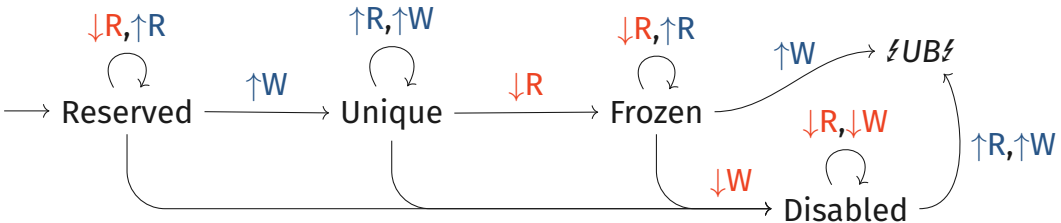
Frozen + ↑W ↛

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

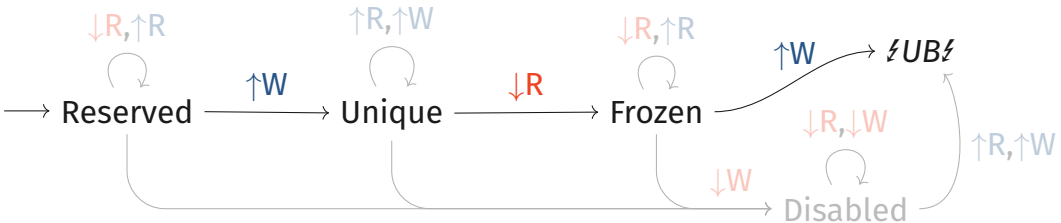


Each Node is a State Machine

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



Tree Borrows: What Else is There?

Tree Borrows: What Else is There?

1. Protectors: More *UB*, more optimizations

Tree Borrows: What Else is There?

1. Protectors: More *UB*, more optimizations
2. Concurrency: Supported

Tree Borrows: What Else is There?

1. Protectors: More *UB*, more optimizations
2. Concurrency: Supported
3. Lazy initialization: Use `&mut v[0]` at offset `1`

Tree Borrows: What Else is There?

1. Protectors: More *UB*, more optimizations
2. Concurrency: Supported
3. Lazy initialization: Use `&mut v[0]` at offset `1`
4. Two-phased Borrows: Supported

Tree Borrows: What Else is There?

1. Protectors: More *UB*, more optimizations
2. Concurrency: Supported
3. Lazy initialization: Use `&mut v[0]` at offset `1`
4. Two-phased Borrows: Supported
5. Interior Mutability: Supported

Tree Borrows: What Else is There?

1. Protectors: More *UB*, more optimizations
2. Concurrency: Supported
3. Lazy initialization: Use `&mut v[0]` at offset `1`
4. Two-phased Borrows: Supported
5. Interior Mutability: Supported
6. Dirty Hacks: Not Required

Tree Borrows: What Else is There?

1. Protectors: More *UB*, more optimizations
2. Concurrency: Supported
3. Lazy initialization: Use `&mut v[0]` at offset `1`
4. Two-phased Borrows: Supported
5. Interior Mutability: Supported
6. Dirty Hacks: Not Required

Our predecessor, *Stacked Borrows*, lacks support for (3)-(6).

Tree Borrows: A Collaboration

Neven

Ralf

Johannes

Derek

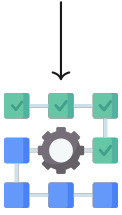
Tree Borrows: A Collaboration

Neven

Ralf

Johannes

Derek



Miri

Tree Borrows: A Collaboration

Neven

Ralf

Johannes

Derek

Miri

Simuliris

Tree Borrows: A Collaboration

Neven

Ralf

Johannes

Derek

Miri

Simuliris

Tree Borrows: A Collaboration

Neven

Ralf

Johannes

Derek

Miri

Simuliris

Tree Borrows: A Collaboration

Neven

Ralf

Johannes

Derek

Miri

Simuliris

Program Logic

Tree Borrows: A Collaboration

Neven

Ralf

Johannes

Derek

Miri

Simuliris (WPo)

Program Logic

Tree Borrows: A Collaboration

Neven

Ralf

Johannes

Derek

Miri

Simuliris (WP0)

Program Logic (WP1)

Leveraging Rust's Lifetimes

for

Tree Borrows

and

Spec Testing

Leveraging Rust's Lifetimes

for

Tree Borrows

and

Spec Testing

Specify All The Things

Two of my future work packages mention specification(s):

Specify All The Things

Two of my future work packages mention specification(s):

Work Package 2:

Work Package 3:

Specify All The Things

Two of my future work packages mention specification(s):

Work Package 2:

Specification
of Rust's Operational Semantics

Work Package 3:

Specification
of functions written in Rust

Specify All The Things

Two of my future work packages mention specification(s):

Work Package 2:

Specification

of Rust's Operational Semantics

What does `*x = 42;` do?

Work Package 3:

Specification

of functions written in Rust

Specify All The Things

Two of my future work packages mention specification(s):

Work Package 2:

Specification

of Rust's Operational Semantics

What does `*x = 42;` do?

Work Package 3:

Specification

of functions written in Rust

Pre- and Post-conditions

Specify All The Things

Two of my future work packages mention specification(s):

Work Package 2:

Specification

of Rust's Operational Semantics

What does `*x = 42;` do?

Goal: Give Formal Definition

Work Package 3:

Specification

of functions written in Rust

Pre- and Post-conditions

Goal: Verification Tool Interoperability

Specify All The Things

Two of my future work packages mention specification(s):

Work Package 2:

Specification

of Rust's Operational Semantics

What does `*x = 42;` do?

Goal: Give Formal Definition

Work Package 3:

Specification

of functions written in Rust

Pre- and Post-conditions

Goal: Verification Tool Interoperability

Rust Has Many Verification Tools

Rust Has Many Verification Tools

$$\mathrm{P} * \mathrm{rust} \mathrel{-\!\!*} \mathrm{i}$$

Rust Has Many Verification Tools

$\mathrm{P}{*}\mathrm{rust}{-\!\!*}\mathrm{i}$

Rust Has Many Verification Tools

$\mathrm{P}{*}\mathrm{rust}{-\!\!*}\mathrm{i}$

Verus

Rust Has Many Verification Tools

RefinedRust

$P * \text{rust} \mathrel{-\!\!*} i$

Verus

Rust Has Many Verification Tools

RefinedRust

P∗rust−∗i

Verus

Rust Has Many Verification Tools

RefinedRust

P∗rust−∗i

Verus

Rust Has Many Verification Tools

RefinedRust

P∗rust−∗i

Verus

Gillian-Rust

Rust Has Many Verification Tools

RefinedRust

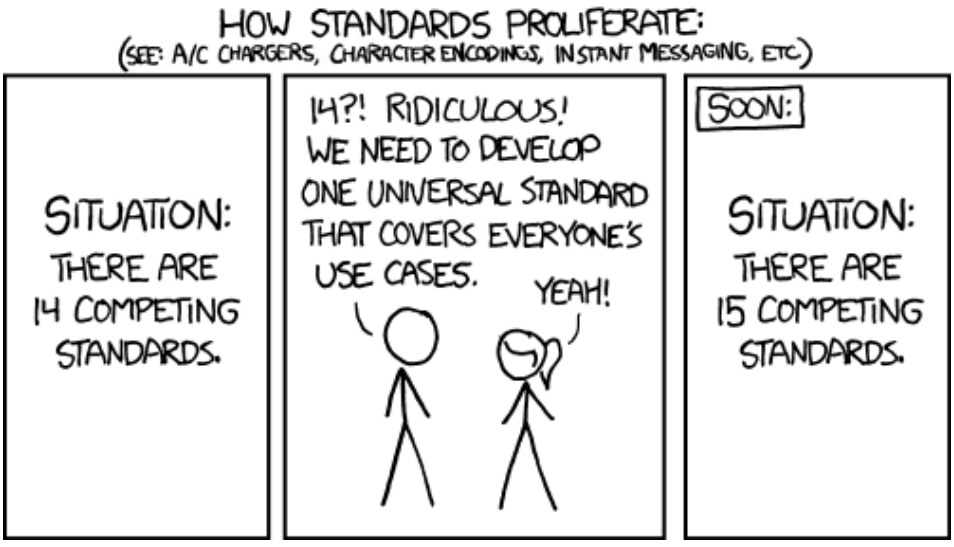
P∗rust−∗i

kani

Verus

Gillian-Rust

Can we improve this?



Can we improve this?

Miri spot-checks for absence of *UB*.

Can we improve this?

Miri spot-checks for absence of *UB*.

Miri spot-checks that code obeys specification?

Can we improve this?

Miri spot-checks for absence of *UB*.

Miri spot-checks that code obeys specification?

But what about?

-  Spec languages mutually incompatible
-  Spec languages not made for spot-checking

Can we improve this?

Miri spot-checks for absence of *UB*.

Miri spot-checks that code obeys specification?

But what about?

Spec languages mutually incompatible

Spec languages not made for spot-checking

Miri is widely used, testing is more approachable

Different Styles Of Verification Languages

```
fn foo<'a>(x: &'a mut Vec<i32>, i: usize)
    -> &'a mut i32 {
    &mut x[i]
}
```

Different Styles Of Verification Languages

Kani-style:

Creusot-style:

```
fn foo<'a>(x: &'a mut Vec<i32>, i: usize)
    -> &'a mut i32 {
    &mut x[i]
}
```

Different Styles Of Verification Languages

Kani-style:

$\{i < x.\texttt{len}()\}$

Creusot-style:

$\{i < x.\texttt{len}()\}$

```
fn foo<'a>(x: &'a mut Vec<i32>, i: usize)
    -> &'a mut i32 {
    &mut x[i]
}
```

Different Styles Of Verification Languages

Kani-style:

$\{i < x.\texttt{len}()\}$

Creusot-style:

$\{i < x.\texttt{len}()\}$

```
fn foo<'a>(x: &'a mut Vec<i32>, i: usize)
    -> &'a mut i32 {
    &mut x[i]
}
```

$\{result \equiv_{\text{ptr}} x[i]\}$

Different Styles Of Verification Languages

Kani-style:

$\{i < x.\texttt{len}()\}$

Creusot-style:

$\{i < x.\texttt{len}()\}$

```
fn foo<'a>(x: &'a mut Vec<i32>, i: usize)
    -> &'a mut i32 {
    &mut x[i]
}
```

$\{result \equiv_{\text{ptr}} x[i]\}$

$$\left\{ \begin{array}{ll} & *result = (*x)[i] \\ \wedge & ◎result = (◎x)[i] \\ \wedge & \forall n \neq i : (*x)[n] = (◎x)[n] \end{array} \right\}$$

Verification Languages Are (Not) Executable

$\forall \exists \infty$

Quantifiers

Verification Languages Are (Not) Executable

$\forall \exists \infty$

Quantifiers

Not executable!

Verification Languages Are (Not) Executable

$\forall \exists \infty$

Quantifiers

$* \quad \longrightarrow\!*$

Ownership

Not executable!

Verification Languages Are (Not) Executable

$\forall \exists \infty$

Quantifiers

Not executable!

$* \quad \longrightarrow\!*$

Ownership

Seems executable...

Verification Languages Are (Not) Executable

$\forall \exists \infty$

Quantifiers

Not executable!

$* \quad \longrightarrow\!*$

Ownership

Seems executable...

Prophecies

Verification Languages Are (Not) Executable

$\forall \exists \infty$

Quantifiers

Not executable!

$* \quad \longrightarrow\!*$

Ownership

Seems executable...

Prophecies

Maybe executable?

Verification Languages Are (Not) Executable

$\forall \exists \infty$

Quantifiers

Not executable!

$* \quad -\!\!* $

Ownership

Seems executable...

Prophecies

Maybe executable?

Goal for WP3: Design a testing-based spec checker...

Verification Languages Are (Not) Executable

$\forall \exists \infty$

Quantifiers

Not executable!

$* \quad \longrightarrow *$

Ownership

Seems executable...

Prophecies

Maybe executable?

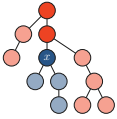
Goal for WP3: Design a testing-based spec checker...
...for an ownership-based, prophetic specification language!

The End

Thanks for your attention!

Putting The Borrows Into Trees

Aliasing model tracks references in a tree data structure:



Nodes precisely represent references.

Each node x splits tree in two:

- Local: references x "knows about"
- Foreign: reference x does not know about

Access to	foreign reference:	foreign access
	local	local

Tree Borrows: A Collaboration

Can we improve this?

- Miri spot-checks for absence of *UB*.
- Miri spot-checks that code obeys specification?
- But what about?
  - Spec languages mutually incompatible
  - Spec languages not made for spot-checking
- Miri is widely used, testing is more approachable

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