Understanding and Evolving the Rust Programming Language

Derek Dreyer
MPI-SWS, Germany

Collège de France
February 6, 2020

Rust – Mozilla’s replacement for C/C++

Rust is the only language to provide…

- Low-level control à la C/C++
- Strong safety guarantees
- Modern, functional paradigms
- Industrial development and backing

Core ingredients:

- Sophisticated ownership type system
- Safe encapsulation of unsafe code

Rust – Mozilla’s replacement for C/C++

Rus

- Lo
- St
- M
- In

Cor

- Sc
- Safe encapsulation of unsafe code

Goal of RustBelt project:
Build first formal foundations
for the Rust language!

Understanding Rust: λ_{Rust}

Building an extensible soundness proof of Rust that covers its core type system as well as standard libraries

Evolving Rust: Stacked Borrows

Defining the semantics of Rust in order to justify powerful intraprocedural type-based optimizations

Understanding Rust: λ_{Rust}

Building an extensible soundness proof of Rust that covers its core type system as well as standard libraries

Key challenge: Interaction of safe and unsafe code

Defining the semantics of Rust in order to justify powerful intraprocedural type-based optimizations

Understanding Rust: λ_{Rust}

Building an extensible soundness proof of Rust that covers its core type system as well as standard libraries

Evolving Rust: Stacked Borrows

Defining the semantics of Rust in order to justify powerful intraprocedural type-based optimizations

Rust 101

Rust 101

~~Aliasing~~

~~+~~

~~Mutation~~

Ownership

```
// Allocate v on the heap
let mut v: Vec<i32> = vec![1, 2, 3];
v.push(4);
```

Ownership

```
// Allocate v on the heap
let mut v: Vec<i32> = vec![1, 2, 3];
v.push(4);

// Send v to another thread
send(v);
```

Ownership transferred to `send`:

```
fn send(Vec<i32>)
```

Ownership

```
// Allocate v on the heap
let mut v: Vec<i32> = vec![1, 2, 3];
v.push(4);

// Send v to another thread
send(v);

// Let us try to use v again
v.push(5);
```

Error: `v` has been moved.
Prevents possible data race.

Ownership

```
// Allocate v on the heap
let mut v: Vec<i32> = vec![1, 2, 3];
v.push(4);

//
se
```

`x: T` expresses **ownership** of `x` at type `T`

- Mutation allowed, no aliasing
- We can deallocate `x`

Ownership

```
// Allocate v on the heap
let mut v: Vec<i32> = vec![1, 2, 3];
v.push(4);

// Send v to another thread
send(v);
```

Why is `v` not moved?

Borrowing and lifetimes

```
// Allocate v on the heap
let mut v: Vec<i32> = vec![1, 2, 3];
Vec::push(&mut v, 4);

// Send v to another thre
send(v);
```

Method call was just sugar.
`&mut v` creates a reference.

Borrowing and lifetimes

```
// Allocate v on the heap
let mut v: Vec<i32> = vec![1, 2, 3];
Vec::push(&mut v, 4);
```

Pass-by-reference: `Vec::push` borrows ownership temporarily

```
send(v);
```

Pass-by-value: Ownership moved to `send` permanently

Borrowing and lifetimes

```
// Allocate v on the heap
let mut v: Vec<i32> = vec![1, 2, 3];
Vec::push(&mut v, 4);
```

Pass-by-reference: `Vec::push` borrows ownership temporarily

```
send(v);
```

Borrowing and lifetimes

```
// Allocate v on the heap
let mut v: Vec<i32> = vec![1, 2, 3];
Vec::push(&mut v, 4);

// Send v to another thread
send(v);
```

Type of `push`:

```
fn Vec::push<'a>(&'a mut Vec<i32>, i32)
```

Borrowing and lifetimes

```
// Allocate v on the heap
let mut v: Vec<i32> = vec![1, 2, 3];
Vec::push(&mut v, 4);

// Send v to another thread
send(v);
```

Type of `push`:

```
fn Vec::push<’a>(&’a mut Vec<i32>, i32)
```

Lifetime `’a` is inferred by Rust.

Borrowing and lifetimes

```
// Allocate v on the heap
```

`&mut x` creates a mutable reference of type `&'a mut T`:

- Ownership temporarily borrowed
- Borrow lasts for inferred lifetime `'a`
- Mutation, no aliasing
    - Unique pointer

Shared Borrowing

```
let mut x = 1;
join(|| println!("Thread 1: {}", &x),
     || println!("Thread 2: {}", &x));
x = 2;
```

Shared Borrowing

```
let mut x = 1;
join(|| println!("Thread 1: {}", &x),
     || println!("Thread 2: {}", &x));
x = 2;
```

`&x` creates a shared reference of type `&'a T`

- Ownership borrowed for lifetime `'a`
- Can be aliased
- Does not allow mutation

Shared Borrowing

```
let mut x = 1;
join(|| println!("Thread 1: {}", &x),
     || println!("Thread 2: {}", &x));
x = 2;
```

After `’a` has ended, `x` is writeable again.

Rust's type system is based on ownership and borrowing:

1. Full ownership: `T`
2. Mutable (borrowed) reference: `&'a mut T`
3. Shared (borrowed) reference: `&'a T`

Lifetimes `'a` decide how long borrows last.

Aliasing

+

Mutation

But what if I need aliased mutable state?

Pointer-based data structures:

- Doubly-linked lists, …

Synchronization mechanisms:

- Locks, channels, semaphores, …

Memory management:

- Reference counting, …

```rust
let m = Mutex::new(1); // m : Mutex<i32>

// Concurrent increment:
// Acquire lock, mutate, release (implicit)
join(|| *(&m).lock().unwrap() += 1,
     || *(&m).lock().unwrap() += 1);

// Unique owner: no need to lock
println!("{}", m.get_mut().unwrap())
```

Type of `lock`:

```
fn lock<'a>(&'a Mutex<i32>)
            -> LockResult<MutexGuard<'a, i32>>
```

```
join(|| *(&m).lock().unwrap() += 1,
     || *(&m).lock().unwrap() += 1);

// Unique owner: no need to lock
println!("{}", m.get_mut().unwrap())
```

Type of `lock`:

```
fn lock<'a>(&'a Mutex<i32>)
          -> &'a mut i32
```

```
join(|| *(&m).lock().unwrap() += 1,
     || *(&m).lock().unwrap() += 1);

// Unique owner: no need to lock
println!("{}", m.get_mut().unwrap())
```

Type of `lock`:

```
fn lock<'a>(&'a Mutex<i32>)
         -> &'a mut i32
```

Shared mutable state:

Interior mutability

```
// Unique owner: no need to lock
println!("{}", m.get_mut().unwrap())
```

Type of `lock`:

```
fn lock<'a>(&'a Mutex<i32>)
         -> &'a mut i32
```

Shared mutable state:

Interior mutability

```
// Unique owner: no need to lock
println!("{}", m.get_mut().unwrap())
```

Aliasing

\+

Mutation

?

unsafe

```
fn lock<’a>(&’a self) -> LockResult<MutexGuard<’a, T>>
{
    unsafe {
        libc::pthread_mutex_lock(self.inner.get());
        MutexGuard::new(self)
    }
}
```

unsafe

`Mutex` has an unsafe implementation. But the interface (API) is safe:

```
fn lock<'a>(&'a Mutex<i32>) -> &'a mut T
```

unsafe

`Mutex` has an unsafe implementation. But the interface (API) is safe:

```
fn lock<'a>(&'a Mutex<i32>) -> &'a mut T
```

Similar for `join`: unsafely implemented user library, safe interface.

Goal: Prove safety of Rust and its standard library.

Safety proof needs to be extensible.

The λ_{Rust} type system

$$\tau ::= \textbf{bool} \mid \textbf{int} \mid \textbf{own}_{n}\,\tau \mid \&^{\kappa}_{\textbf{mut}}\,\tau \mid \&^{\kappa}_{\textbf{shr}}\,\tau \mid \mu\,\alpha.\,\tau \mid \ldots$$

The λ_{Rust} type system

$$\tau ::= \textbf{bool} \mid \textbf{int} \mid \textbf{own}_n\,\tau \mid \&^{\kappa}_{\textbf{mut}}\,\tau \mid \&^{\kappa}_{\textbf{shr}}\,\tau \mid \mu\,\alpha.\,\tau \mid \ldots$$

The λ_{Rust} type system

$$\tau ::= \textbf{bool} \mid \textbf{int} \mid \textbf{own}_{n}\, \tau \mid \&^{\kappa}_{\textbf{mut}}\, \tau \mid \&^{\kappa}_{\textbf{shr}}\, \tau \mid \mu\, \alpha.\, \tau \mid \ldots$$

The λ_{Rust} type system

$$\tau ::= \textbf{bool} \mid \textbf{int} \mid \textbf{own}_n\,\tau \mid \&^{\kappa}_{\textbf{mut}}\,\tau \mid \&^{\kappa}_{\textbf{shr}}\,\tau \mid \mu\,\alpha.\,\tau \mid \ldots$$

The λ_{Rust} type system

$$\tau ::= \textbf{bool} \mid \textbf{int} \mid \textbf{own}_n\,\tau \mid \&^{\kappa}_{\textbf{mut}}\,\tau \mid \&^{\kappa}_{\textbf{shr}}\,\tau \mid \mu\,\alpha.\,\tau \mid \ldots$$

$$\mathbf{T} ::= \emptyset \mid \mathbf{T}, p \lhd \tau \mid \ldots$$

Typing context assigns types to paths p (denoting fields of structures)

The λ_{Rust} type system

$$\tau ::= \textbf{bool} \mid \textbf{int} \mid \textbf{own}_n\,\tau \mid \&^{\kappa}_{\textbf{mut}}\,\tau \mid \&^{\kappa}_{\textbf{shr}}\,\tau \mid \mu\,\alpha.\,\tau \mid \dots$$

$$\mathbf{T} ::= \emptyset \mid \mathbf{T}, p \lhd \tau \mid \dots$$

Core substructural typing judgments:

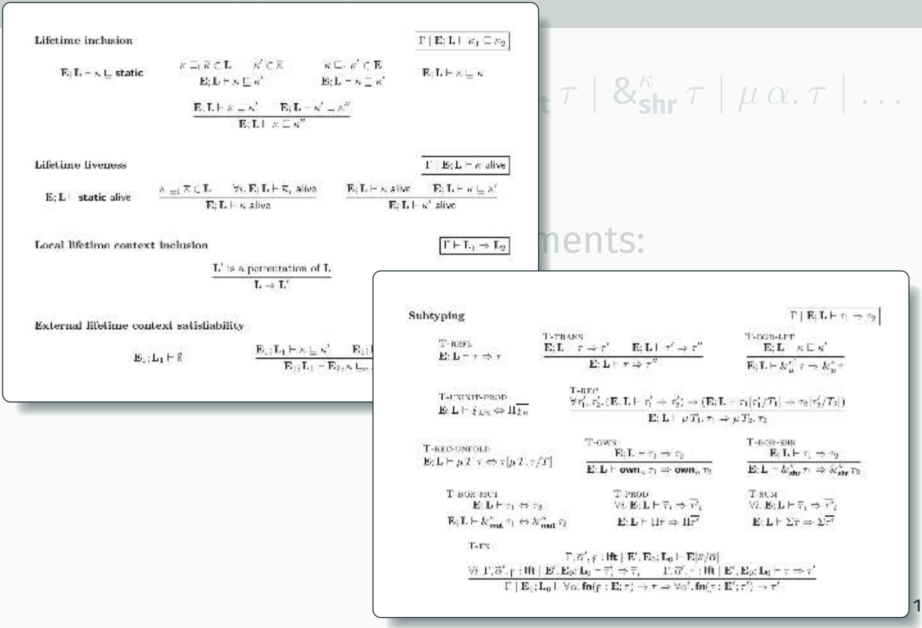
$$\mathbf{E}, \mathbf{L};\ \mathbf{T}_1 \vdash I \dashv x.\,\mathbf{T}_2$$

Typing individual instructions *I*
(**E** and **L** track lifetimes)

$$\mathbf{E}, \mathbf{L};\ \mathbf{K}, \mathbf{T} \vdash F$$

Typing whole functions *F*
(**K** tracks continuations)

The λ_{Rust} type system



Lifetime inclusion $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa_1 \sqsubseteq \kappa_2}$

$$\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \textsf{static} \qquad \frac{\kappa \sqsubseteq_l \overline{\kappa} \in \mathbf{L} \quad \kappa' \in \overline{\kappa}}{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'} \qquad \frac{\kappa \sqsubseteq_e \kappa' \in \mathbf{E}}{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'} \qquad \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa$$

$$\frac{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa' \quad \mathbf{E}; \mathbf{L} \vdash \kappa' \sqsubseteq \kappa''}{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa''}$$

Lifetime liveness $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \text{ alive}}$

$$\mathbf{E}; \mathbf{L} \vdash \textsf{static alive} \qquad \frac{\kappa \sqsubseteq_l \overline{\kappa} \in \mathbf{L} \quad \forall i.\ \mathbf{E}; \mathbf{L} \vdash \overline{\kappa}_i \text{ alive}}{\mathbf{E}; \mathbf{L} \vdash \kappa \text{ alive}} \qquad \frac{\mathbf{E}; \mathbf{L} \vdash \kappa \text{ alive} \quad \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\mathbf{E}; \mathbf{L} \vdash \kappa' \text{ alive}}$$

Local lifetime context inclusion $\boxed{\Gamma \vdash \mathbf{L}_1 \Rightarrow \mathbf{L}_2}$

$$\frac{\mathbf{L}' \text{ is a permutation of } \mathbf{L}}{\mathbf{L} \Rightarrow \mathbf{L}'}$$

External lifetime context satisfiability

$$\mathbf{E}_1; \mathbf{L}_1 \vdash \emptyset \qquad \frac{\mathbf{E}_1; \mathbf{L}_1 \vdash \kappa \sqsubseteq \kappa' \quad \mathbf{E}_1; }{\mathbf{E}_1; \mathbf{L}_1 \vdash \mathbf{E}_2, \kappa \sqsubseteq_e}$$

Subtyping $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \Rightarrow \tau_2}$

T-REFL
$$\mathbf{E}; \mathbf{L} \vdash \tau \Rightarrow \tau$$

T-TRANS
$$\frac{\mathbf{E}; \mathbf{L} \vdash \tau \Rightarrow \tau' \quad \mathbf{E}; \mathbf{L} \vdash \tau' \Rightarrow \tau''}{\mathbf{E}; \mathbf{L} \vdash \tau \Rightarrow \tau''}$$

T-BOR-LFT
$$\frac{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\mathbf{E}; \mathbf{L} \vdash \&^{\kappa'}_{\mu} \tau \Rightarrow \&^{\kappa}_{\mu} \tau}$$

T-UNINIT-PROD
$$\mathbf{E}; \mathbf{L} \vdash \not{\imath}_{\Sigma n} \Leftrightarrow \Pi \overline{\not{\imath}_n}$$

T-REC
$$\frac{\forall \tau_1', \tau_2'.\ (\mathbf{E}; \mathbf{L} \vdash \tau_1' \Rightarrow \tau_2') \Rightarrow (\mathbf{E}; \mathbf{L} \vdash \tau_1[\tau_1'/T_1] \Rightarrow \tau_2[\tau_2'/T_2])}{\mathbf{E}; \mathbf{L} \vdash \mu T_1.\, \tau_1 \Rightarrow \mu T_2.\, \tau_2}$$

T-REC-UNFOLD
$$\mathbf{E}; \mathbf{L} \vdash \mu T.\, \tau \Leftrightarrow \tau[\mu T.\, \tau / T]$$

T-OWN
$$\frac{\mathbf{E}; \mathbf{L} \vdash \tau_1 \Rightarrow \tau_2}{\mathbf{E}; \mathbf{L} \vdash \mathbf{own}_n\, \tau_1 \Rightarrow \mathbf{own}_n\, \tau_2}$$

T-BOR-SHR
$$\frac{\mathbf{E}; \mathbf{L} \vdash \tau_1 \Rightarrow \tau_2}{\mathbf{E}; \mathbf{L} \vdash \&^{\kappa}_{\mathbf{shr}} \tau_1 \Rightarrow \&^{\kappa}_{\mathbf{shr}} \tau_2}$$

T-BOR-MUT
$$\frac{\mathbf{E}; \mathbf{L} \vdash \tau_1 \Leftrightarrow \tau_2}{\mathbf{E}; \mathbf{L} \vdash \&^{\kappa}_{\mathbf{mut}} \tau_1 \Leftrightarrow \&^{\kappa}_{\mathbf{mut}} \tau_2}$$

T-PROD
$$\frac{\forall i.\ \mathbf{E}; \mathbf{L} \vdash \overline{\tau}_i \Rightarrow \overline{\tau'}_i}{\mathbf{E}; \mathbf{L} \vdash \Pi \overline{\tau} \Rightarrow \Pi \overline{\tau'}}$$

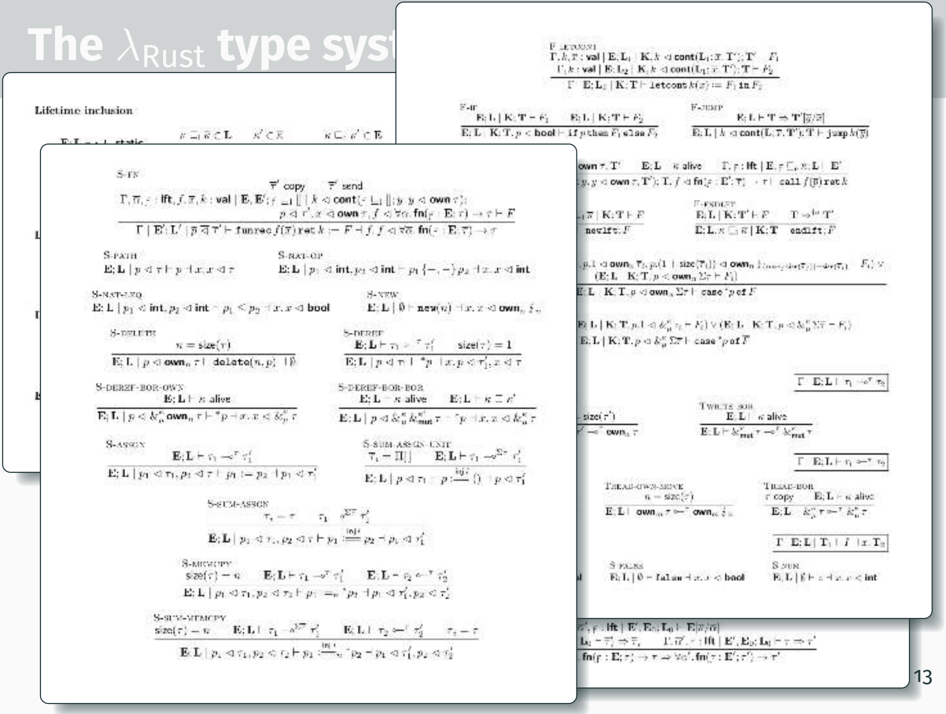
T-SUM
$$\frac{\forall i.\ \mathbf{E}; \mathbf{L} \vdash \overline{\tau}_i \Rightarrow \overline{\tau'}_i}{\mathbf{E}; \mathbf{L} \vdash \Sigma \overline{\tau} \Rightarrow \Sigma \overline{\tau'}}$$

T-FN
$$\frac{\begin{array}{c}\Gamma, \overline{\alpha}', \varrho : \mathbf{lft} \mid \mathbf{E}', \mathbf{E}_0; \mathbf{L}_0 \vdash \mathbf{E}[\overline{\kappa}/\overline{\alpha}] \\ \forall i.\ \Gamma, \overline{\alpha}', \varrho : \mathbf{lft} \mid \mathbf{E}', \mathbf{E}_0; \mathbf{L}_0 \vdash \overline{\tau}_i' \Rightarrow \overline{\tau}_i \qquad \Gamma, \overline{\alpha}', \varrho : \mathbf{lft} \mid \mathbf{E}', \mathbf{E}_0; \mathbf{L}_0 \vdash \tau \Rightarrow \tau'\end{array}}{\Gamma \mid \mathbf{E}_0; \mathbf{L}_0 \vdash \forall \alpha.\ \mathbf{fn}(\varrho : \mathbf{E}; \overline{\tau}) \rightarrow \tau \Rightarrow \forall \alpha'.\ \mathbf{fn}(\varrho : \mathbf{E}'; \overline{\tau}') \rightarrow \tau'}$$

$\mathbf{t}\ \tau \mid \&^{\kappa}_{\mathbf{shr}}\ \tau \mid \mu\, \alpha.\, \tau \mid \ldots$

ments:

The λ_{Rust} type syst



Syntactic type soundness

$$\mathbf{E}, \mathbf{L};\ \mathbf{K}, \mathbf{T} \vdash F \quad \Longrightarrow \quad F \text{ is safe}$$

Usually proven by progress and preservation.

Syntactic type soundness

$$\mathbf{E}, \mathbf{L};\ \mathbf{K}, \mathbf{T} \vdash F \quad \Longrightarrow \quad F \text{ is safe}$$

Usually proven by progress and preservation.

But says nothing about unsafe code!

Syntactic type soundness

$$\mathbf{E}, \mathbf{L};\ \mathbf{K}, \mathbf{T} \vdash F \implies F \text{ is safe}$$

Usually proven by progress and preservation.

But says nothing about unsafe code!

Instead, we prove semantic type soundness using the method of logical relations.

Syntactic type soundness

Logical relations in four “easy” steps:

1. Semantic interpretation of types ($[\![\tau]\!]$)
2. Lift that to all judgments ($\vDash$)
3. Prove “compatibility lemmas”
4. Profit!

Instead, we prove semantic type soundness using the method of logical relations.

1. Semantic interpretation of types

Define ownership predicate for every type τ:

$$[\![\tau]\!].\mathrm{own}(t, \overline{v})$$

1. Semantic interpretation of types

Define ownership predicate for every type τ:

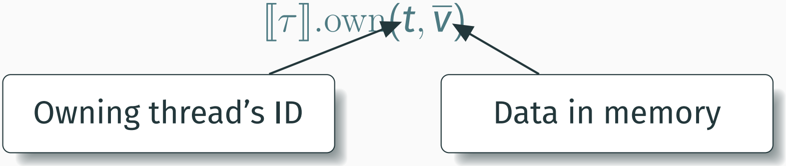
$$[\![\tau]\!].\mathrm{own}(t, \overline{v})$$

Owning thread's ID

Data in memory

1. Semantic interpretation of types

Define ownership predicate for every type τ:



$$[\![\tau]\!].\mathrm{own}(t, \overline{v})$$

Owning thread's ID

Data in memory

What logic should we use to assert ownership?

Separation Logic to the Rescue!

Separation Logic to the Rescue!

Extension of Hoare logic (O'Hearn-Reynolds-…, 1999)

- For reasoning about pointer-manipulating programs

Major influence on many verification & analysis tools

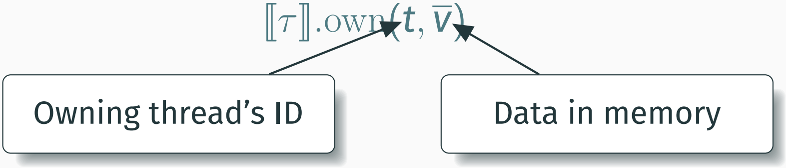
- e.g. Infer, VeriFast, Viper, Bedrock, jStar, …

Separation logic = Ownership logic

- Perfect fit for modeling Rust's ownership types!

1. Semantic interpretation of types

Define ownership predicate for every type τ:



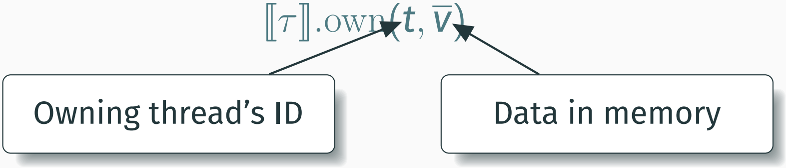
$$[\![\tau]\!].\mathrm{own}(t, \overline{v})$$

Owning thread's ID

Data in memory

1. Semantic interpretation of types

Define ownership predicate for every type τ:



$$[\![\tau]\!].\mathrm{own}(t, \overline{v})$$

Owning thread’s ID

Data in memory

We use a modern, higher-order, concurrent separation logic framework called Iris:

- Implemented in the Coq proof assistant
- Designed to derive new reasoning principles inside the logic

2. Lift to all judgments

Define ownership predicate for every type τ:

$$[\![\tau]\!].\mathrm{own}(t, \overline{v})$$

Lift to semantic contexts $[\![\mathbf{T}]\!](t)$:

$$[\![p_1 \lhd \tau_1, p_2 \lhd \tau_2]\!](t) \quad := \quad [\![\tau_1]\!].\mathrm{own}(t, [p_1]) * [\![\tau_2]\!].\mathrm{own}(t, [p_2])$$

2. Lift to all judgments

Define ownership predicate for every type τ:

$$[\![\tau]\!].\mathrm{own}(t, \overline{v})$$

Lift to semantic contexts $[\![\mathbf{T}]\!](t)$:

$$[\![p_1 \lhd \tau_1, p_2 \lhd \tau_2]\!](t) \quad := \quad [\![\tau_1]\!].\mathrm{own}(t, [p_1]) * [\![\tau_2]\!].\mathrm{own}(t, [p_2])$$

Separating conjunction

2. Lift to all judgments

Define ownership predicate for every type τ:

$$[\![\tau]\!].\mathrm{own}(t, \overline{v})$$

Lift to semantic typing judgments:

$$\mathbf{E}, \mathbf{L};\ \mathbf{T}_1 \models I \mathrel{=\!\!|} \mathbf{T}_2 \quad := $$

$$\forall t.\ \{[\![\mathbf{E}]\!] * [\![\mathbf{L}]\!] * [\![\mathbf{T}_1]\!](t)\}\ I\ \{[\![\mathbf{E}]\!] * [\![\mathbf{L}]\!] * [\![\mathbf{T}_2]\!](t)\}$$

Crucially, semantic typing implies safety.

3. Compatibility lemmas

Connect logical relation to type system:
Semantic versions of all syntactic typing rules.

$$\frac{\mathbf{E}, \mathbf{L} \vdash \kappa \text{ alive}}{\mathbf{E}, \mathbf{L};\ p_1 \lhd \&^{\kappa}_{\mathbf{mut}}\, \tau, p_2 \lhd \tau \vdash p_1 := p_2 \dashv p_1 \lhd \&^{\kappa}_{\mathbf{mut}}\, \tau}$$

$$\frac{\mathbf{E}, \mathbf{L};\ \mathbf{T}_1 \vdash I \dashv x.\, \mathbf{T}_2 \qquad \mathbf{E}, \mathbf{L};\ \mathbf{K}; \mathbf{T}_2, \mathbf{T} \vdash F}{\mathbf{E}, \mathbf{L};\ \mathbf{K}; \mathbf{T}_1, \mathbf{T} \vdash \mathtt{let}\, x = I\, \mathtt{in}\, F}$$

3. Compatibility lemmas

Connect logical relation to type system:
Semantic versions of all syntactic typing rules.

$$\frac{\mathbf{E}, \mathbf{L} \models \kappa\ \mathsf{alive}}{\mathbf{E}, \mathbf{L};\ p_1 \lhd \&^{\kappa}_{\mathbf{mut}}\, \tau, p_2 \lhd \tau \models p_1 := p_2 \Dashv p_1 \lhd \&^{\kappa}_{\mathbf{mut}}\, \tau}$$

$$\frac{\mathbf{E}, \mathbf{L};\ \mathbf{T}_1 \models I \Dashv x.\, \mathbf{T}_2 \qquad \mathbf{E}, \mathbf{L};\ \mathbf{K}; \mathbf{T}_2, \mathbf{T} \models F}{\mathbf{E}, \mathbf{L};\ \mathbf{K}; \mathbf{T}_1, \mathbf{T} \models \mathtt{let}\, x = I\, \mathtt{in}\, F}$$

3. Compatibility lemmas

Connect logical relation to type system:
Semantic versions of all syntactic typing rules.

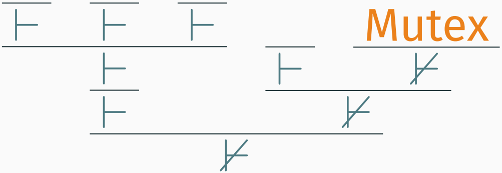
Well-typed programs can't go wrong

- No data race
- No invalid memory access

$\mathbf{E}, \mathbf{L};$ ut $\mathcal{T}$

$$\frac{\mathbf{E}, \mathbf{L};\ \mathbf{T}_1 \models I \mathrel{=\!\!|} x.\, \mathbf{T}_2 \qquad \mathbf{E}, \mathbf{L};\ \mathbf{K}; \mathbf{T}_2, \mathbf{T} \models F}{\mathbf{E}, \mathbf{L};\ \mathbf{K}; \mathbf{T}_1, \mathbf{T} \models \mathtt{let}\, x = I \,\mathtt{in}\, F}$$

4. Profit! – Linking with unsafe code



4. Profit! – Linking with **unsafe** code

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash}\quad\dfrac{}{\vdash}\quad\dfrac{}{\vdash}}{\vdash}}{\vdash}\qquad\dfrac{\dfrac{}{\vdash}\qquad\dfrac{\llbracket\mathrm{Mutex}\rrbracket}{\vDash}}{\vDash}}{\vDash}$$

4. Profit! – Linking with unsafe code

$$\frac{\vdash \quad \vdash \quad \vdash}{\quad} \qquad \frac{\quad}{\quad} \qquad \frac{[\![\text{Mutex}]\!]}{\quad}$$

The whole program is safe if the unsafe pieces are safe!

How do we define $[\![\tau]\!].\mathrm{own}(t, \overline{v})$?

$$\llbracket \mathbf{own}_n\, \tau \rrbracket.\mathrm{own}(t, \overline{v}) :=$$

$$\exists \ell.\ \overline{v} = [\ell] * \triangleright\big(\exists \overline{w}.\ \ell \mapsto \overline{w} * \llbracket \tau \rrbracket.\mathrm{own}(t, \overline{w})\big) * \ldots$$

$$\llbracket \mathbf{own}_n\, \tau \rrbracket.\mathrm{own}(t, \overline{v}) :=$$

$$\exists \ell.\ \overline{v} = [\ell] * \triangleright\big(\exists \overline{w}.\ \ell \mapsto \overline{w} * \llbracket \tau \rrbracket.\mathrm{own}(t, \overline{w})\big) * \ldots$$

$$\llbracket \mathbf{own}_n\, \tau \rrbracket.\mathrm{own}(t, \overline{v}) :=$$
$$\exists \ell.\; \overline{v} = [\ell] * \triangleright\big(\exists \overline{w}.\; \ell \mapsto \overline{w} * \llbracket \tau \rrbracket.\mathrm{own}(t, \overline{w})\big) * \ldots$$

$$\llbracket \&^{\kappa}_{\mathbf{mut}}\, \tau \rrbracket.\mathrm{own}(t, \overline{v}) :=$$
$$\exists \ell.\; \overline{v} = [\ell] * \&^{\kappa}_{\mathbf{full}}\big(\exists \overline{w}.\; \ell \mapsto \overline{w} * \llbracket \tau \rrbracket.\mathrm{own}(t, \overline{w})\big)$$

$$[\![\textbf{own}_n\,\tau]\!].\mathrm{own}(t,\overline{v}) :=$$

$$\exists \ell.\ \overline{v} = [\ell] * \triangleright\big(\exists \overline{w}.\ \ell \mapsto \overline{w} * [\![\tau]\!].\mathrm{own}(t,\overline{w})\big) * \dots$$

$$[\![\&^{\kappa}_{\textbf{mut}}\,\tau]\!].\mathrm{own}(t,\overline{v}) :=$$

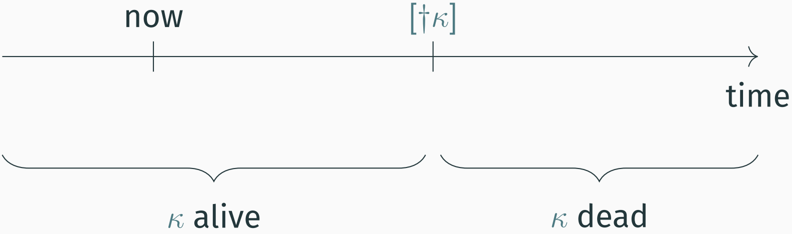
$$\exists \ell.\ \overline{v} = [\ell] * \&^{\kappa}_{\textbf{full}}\big(\exists \overline{w}.\ \ell \mapsto \overline{w} * [\![\tau]\!].\mathrm{own}(t,\overline{w})\big)$$

Lifetime logic connective

Traditionally, $P * Q$ splits ownership in space.

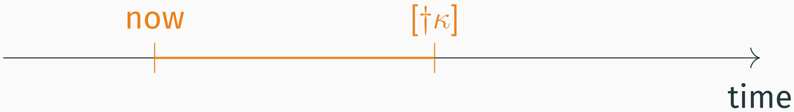
Lifetime logic allows splitting ownership in time!

$$P \Rrightarrow \&^{\kappa}_{\mathbf{full}} P * \left([\dagger\kappa] \Rrightarrow P\right)$$



$$P \Rrightarrow \&^{\kappa}_{\mathbf{full}} P * ([\dagger\kappa] \Rrightarrow P)$$

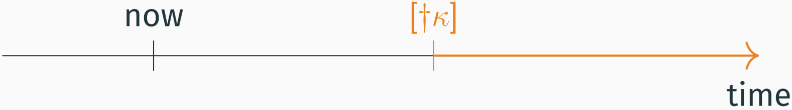
Access to P while κ lasts



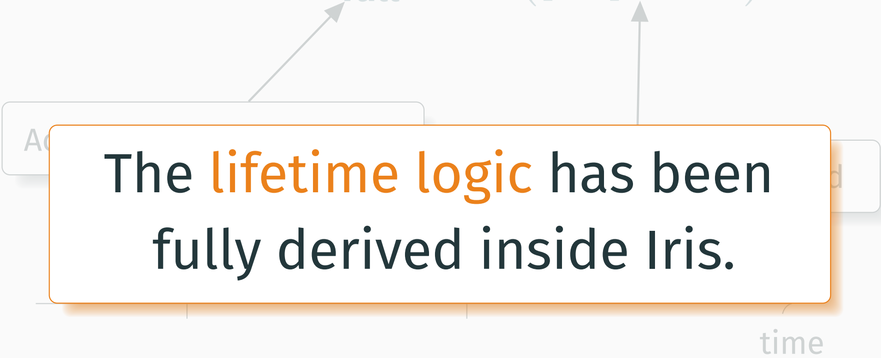
$$P \Rrightarrow \&^{\kappa}_{\mathbf{full}}\, P * \left([\dagger\kappa] \Rrightarrow P\right)$$

Access to P while κ lasts

Access to P when κ has ended



$$P \Rrightarrow \&_{\mathbf{full}}^{\kappa} P \ast ([\dagger\kappa] \Rrightarrow P)$$



The lifetime logic has been fully derived inside Iris.

What else? [POPL'18 and POPL'20 papers]

- More details about λ_{Rust}, the type system, and the lifetime logic
- How to handle interior mutability that is safe for subtle reasons (e.g., mutual exclusion)
  - `Mutex`, `Cell`, `RefCell`, `Rc`, `Arc`, `RwLock`
  - Found bugs in `Mutex`, `Arc`, ...

- Scaling from sequentially consistent concurrency model to a more realistic relaxed memory model

Still missing from λ_{Rust}:

- Trait objects (existential types), `drop`, ...

Logical relations are the tool of choice for proving safety of languages with unsafe operations.

Advances in separation logic (as embodied in Iris) make this possible for a language as sophisticated as Rust!

Understanding Rust: λ_{Rust}

Building an extensible soundness proof of Rust that covers its core type system as well as standard libraries

Evolving Rust: Stacked Borrows

Defining the semantics of Rust in order to justify powerful intraprocedural type-based optimizations

Understanding Rust: λ_{Rust}

Building an extensible soundness proof of Rust that covers its core type system as well as standard libraries

Evolving Rust: Stacked Borrows

Defining the semantics of Rust in order to justify powerful intraprocedural type-based optimizations

Rust’s type system is based on ownership and borrowing:

1. Full ownership: `T`
2. Mutable (borrowed) reference: `&’a mut T`
3. Shared (borrowed) reference: `&’a T`

Lifetimes `’a` decide how long borrows last.

Aliasing

+

Mutation

Rust’s type system is based on ownership and borrowing:

1.

2.

3.

Lifetimes ’a decide how long borrows last.

Rust’s reference types provide strong aliasing information.

The Rust compiler should exploit them for optimization!

Aliasing guarantees: `&mut T` Examples

```
fn test_noalias(x: &mut i32, y: &mut i32) -> i32 {
  // x, y cannot alias: they are unique pointers
  *x = 42;
  *y = 37;
  return *x; // must return 42
}
```

Aliasing guarantees: `&mut T` Examples

```
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  // unknown_function cannot have an alias to x
  unknown_function();
  return *x; // must return 42
}
```

Aliasing guarantees: `&mut T` Examples

escaped pointer

```
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  // unknown_function cannot have an alias to x
  unknown_function();
  return *x; // must return 42
}
```

unknown code

Aliasing guarantees: `&T` Examples

```
fn test_noalias_shared(x: &i32, y: &mut i32) -> i32 {
  let val = *x;
  // cannot mutate x: x points to immutable data
  *y = 37;
  return *x == val; // must return true
}
```

Aliasing guarantees: `&T` Examples

```
fn test_shared(x: &i32) -> bool {
  let val = *x;
  // unknown_function_shared cannot mutate x
  unknown_function_shared(x);
  return *x == val; // must return true
}
```

Aliasing guarantees: `&T` Examples

escaped pointer

```
fn test_shared(x: &i32) -> bool {
  let val = *x;
  // unknown_function_shared cannot mutate x
  unknown_function_shared(x);
  return *x == val; // must return true
}
```

unknown code with access to `x`

Aliasing guarantees: `&T` Examples

These optimizations go beyond the wildest dreams of C compiler developers!

Aliasing guarantees: `&T` Examples

These optimizations go beyond the wildest dreams of C compiler developers!

But there is a problem:

Aliasing guarantees: `&T` Examples

These optimizations go beyond the wildest dreams of C compiler developers!

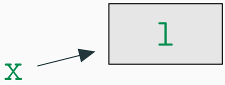
But there is a problem:

UNSAFE CODE!

```
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  unknown_function();
  return *x; // must return 42
}
```

```
fn main() {
  let mut l = 13;

  let answer = test_unique(&mut l);
  println!("The answer is {}", answer);
}
```

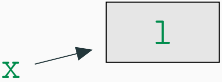


```
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  unknown_function();
  return *x; // must return 42
}
```

```
static mut ALIAS: *mut i32 = std::ptr::null_mut();
fn main() {
  let mut l = 13;

  let answer = test_unique(&mut l);
  println!("The answer is {}", answer);
}
```

ALIAS is a raw pointer (*mut T)



```
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  unknown_function();
  return *x; // should return 42
}
```

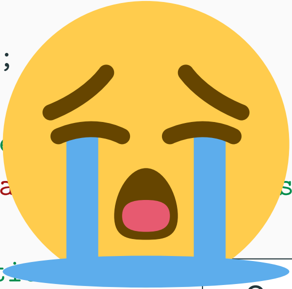
```rust
static mut ALIAS: *mut i32 = std::ptr::null_mut();
fn main() {
  let mut l = 13;
  unsafe { ALIAS = &mut l as *mut i32; }
  let answer = test_unique(&mut l);
  println!("The answer is {}", answer);
}
```

```rust
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  unknown_function();
  return *x; // should return 42
}
```

```rust
static mut ALIAS: *mut i32 = std::ptr::null_mut();
fn main() {
  let mut l = 13;
  unsafe { ALIAS = &mut l as *mut i32; }
  let answer = test_unique(&mut l);
  println!("The answer is {}", answer); // prints 7
}
fn unknown_function() {
  unsafe { *ALIAS = 7; }
}
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  unknown_function();
  return *x; // should return 42, but returns 7
}
```

Overwrites `*x` with 7

```
static mut ALIAS: *mut i32 = std::ptr::null_mut();
fn main() {
  let mut l = 13;
  unsafe { ALIAS                                  i32; }
  let answer = te                                 ;
  println!("The a                               swer); // prints 7
}
fn unknown_functi
  unsafe { *ALIAS = 7; }
}
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  unknown_function();
  return *x; // should return 42, but returns 7
}
```



Overwrites `*x` with 7

```rust
static mut ALIAS: *mut i32 = std::ptr::null_mut();
fn main() {
  let mut l = 13;
  unsafe { ALIAS = &mut l as *mut i32; }
  let answer = test_unique(&mut l);
  println!("The answer is {}", answer); // prints 7
}
```

```rust
}

fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  unknown_function();
  return *x; // should return 42, but returns 7
}
```

Goal: rule out misbehaving programs

Review: Undefined Behavior

Use of unsafe code imposes
proof obligations on the programmer:

No use of dangling/NULL pointers, no data races, …

Review: Undefined Behavior

Use of unsafe code imposes proof obligations on the programmer:

No use of dangling/NULL pointers, no data races, ...

Violation of proof obligation leads to Undefined Behavior.

Image: dbeast32

Review: Undefined Behavior

Use of unsafe code imposes

Compilers can rely on these proof obligations when justifying optimizations

Violation of proof obligation leads to Undefined Behavior.

```
static mut ALIAS: *mut i32 = std::ptr::null_mut();
fn main() {
  let mut l = 13;
  unsafe { ALIAS = &mut l as *mut i32; }
  let answer = test_unique(&mut l);
  println!("The answer is {}", answer); // prints 7
}
fn unknown_function() {
  unsafe { *ALIAS = 7; }
}
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  unknown_function();
  return *x; // should return 42, but returns 7
}
```

Plan: make this
Undefined Behavior

Stacked Borrows

Stacked Borrows

Aliasing model defining which pointers may be used to access memory, ensuring

- uniqueness of mutable references, and
- immutability of shared references.

Stacked Borrows

- Stacked Borrows is restrictive enough to enable useful optimizations
  - ✓ formal proof

Stacked Borrows

- Stacked Borrows is restrictive enough to enable useful optimizations
  - ✓ formal proof
- Stacked Borrows is permissive enough to enable programming
  - ✓ checked standard library test suite by instrumenting the Rust interpreter Miri

Stacked Borrows: Key Idea

Model proof obligations after existing static “borrow” check

Borrow Checker	Stacked Borrows
static	dynamic
only safe code	safe & unsafe code

```rust
let mut l = 13;
let a = &mut l; // a *borrows* from l
```

```rust
let mut l = 13;
let a = &mut l; // a *borrows* from l
let b = &mut *a; // b *reborrows* from a
```

```rust
let mut l = 13;
let a = &mut l; // a *borrows* from l
let b = &mut *a; // b *reborrows* from a
*b = 3;
```

```
let mut l = 13;
let a = &mut l; // a *borrows* from l
let b = &mut *a; // b *reborrows* from a
*b = 3;
*a = 4;
```

```
let mut l = 13;
let a = &mut l; // a *borrows* from l
let b = &mut *a; // b *reborrows* from a
*b = 3;
*a = 4;
*b = 4; // ERROR: lifetime of ‘b‘ has ended
```

```
let mut l = 13;
let a = &mut l; // a *borrows* from l
let b = &mut *a; // b *reborrows* from a
*b = 3;
*a = 4;
*b = 4; // ERROR: lifetime of ‘b‘ has ended
```

Conflicting use of `a`

1. The lender `a` does not get used until the lifetime of the loan has expired.

```
let mut l = 13;
let a = &mut l; // a *borrows* from l
let b = &mut *a; // b *reborrows* from a
*b = 3;
*a = 4;
*b = 4; // ERROR: lifetime of ‘b‘ has ended
```

Conflicting use of `a`

1. The lender `a` does not get used until the lifetime of the loan has expired.
2. The recipient of the borrow `b` may only be used while its lifetime is ongoing.

```
let mut l = 13;
let a = &mut l; // a *borrows* from l
let b = &mut *a; // b *reborrows* from a
*b = 3;
*a = 4;
*b = 4; // ERROR: lifetime of ‘b‘ has ended
```

Conflicting use of `a`

- Chain of borrows:
  `l` borrowed to `a` reborrowed to `b`
- Well-bracketed: no ABAB

```
let mut l = 13;
let a = &mut l; // a *borrows* from l
let b = &mut *a; // b *reborrows* from a
*b = 3;
*a = 4;
```

(Re)borrows are organized
in a stack.

- Chain of borrows:
  `l` borrowed to `a` reborrowed to `b`
- Well-bracketed: no ABAB

Stacked Borrows ingredients

Pointer values carry a tag $(\mathit{PtrVal} := \mathit{Loc} \times \mathbb{N})$

Example: (`0x40`, 1)

references (`&mut T`) are identified by a tag

Stacked Borrows ingredients

Pointer values carry a tag ($\mathit{PtrVal} := \mathit{Loc} \times \mathbb{N}$)
Example: (`0x40`, `1`)

Every location in memory comes with an associated stack ($\mathit{Mem} := \mathit{Loc} \xrightharpoonup{\text{fin}} \mathit{Byte} \times \mathit{Stack}$)

```
⋮
0x40: 0xFE, [0: Unique, 1: Unique]
⋮
```

Stacked Borrows ingredients

Reference tagged 1 borrows from reference tagged 0

Every location in memory comes with an associated stack ($Mem := Loc \xrightarrow{\text{fin}} Byte \times Stack$)

$\vdots$

```
0x40: 0xFE, [0: Unique, 1: Unique]
```

$\vdots$

Stacked Borrows ingredients

Pointer values carry a tag $(PtrVal := Loc \times \mathbb{N})$

For every use of a reference or raw pointer:

- Extra proof obligation:
  $\Rightarrow$ the tag must be in the stack
- Extra operational effect:
  $\Rightarrow$ pop elements further up off the stack

```rust
static mut ALIAS: *mut i32 = std::ptr::null_mut();
fn main() {
  let mut l = 13;
  unsafe { ALIAS = &mut l as *mut i32; }
  let answer = test_unique(&mut l);
  println!("The answer is {}", answer); // prints 7
}
```

“Lifetime” of `ALIAS` begins here.

Stack: `[l, ALIAS]`

```rust
static mut ALIAS: *mut i32 = std::ptr::null_mut();
fn main() {
  let mut l = 13;
  unsafe { ALIAS = &mut l as *mut i32; }
  let answer = test_unique(&mut l);
  println!("The answer is {}", answer); // prints 7
}
```

“Lifetime” of `ALIAS` begins here.
Stack: `[l, ALIAS]`

“Lifetime” ends here: lender `l` is used again, removing `ALIAS`.
Stack: `[l, x]`

```
static mut ALIAS: *mut i32 = std::ptr::null_mut();
fn main() {
  let mut l = 13;
  unsafe { ALIAS = &mut l as *mut i32; }
  let answer = test_unique(&mut l);
  println!("The answer is {}", answer); // prints 7
}
fn unknown_function() {
  unsafe { *ALIAS = 7; }
}
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  unknown_function();
  return *x; // should return 42, but returns 7
}
```

Stack: `[l, x]`

`ALIAS` is not on the stack 💥

Stacked Borrows

- Stacked Borrows is restrictive enough to enable useful optimizations
  - ✓ formal proof
- Stacked Borrows is permissive enough to enable programming
  - ✓ checked standard library test suite by instrumenting the Rust interpreter Miri

Stacked Borrows

- Stacked Borrows is restrictive enough to enable useful optimizations
  - ✓ formal proof

Incomplete proof sketch

```
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  unknown_function();
  return *x; // must return 42
}
```

Incomplete proof sketch

x’s tag is at the top of the stack

```
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  unknown_function();
  return *x; // must return 42
}
```

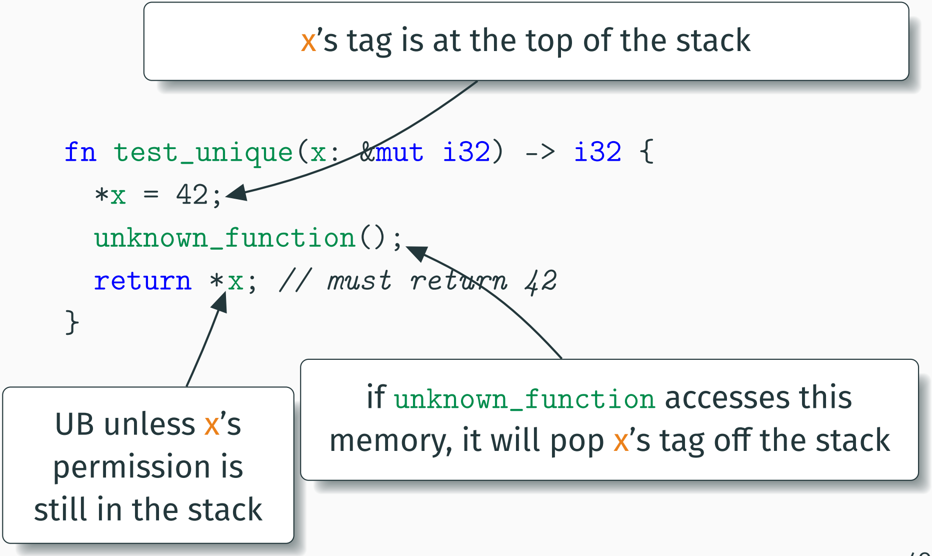
Incomplete proof sketch

x's tag is at the top of the stack

```
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  unknown_function();
  return *x; // must return 42
}
```

if `unknown_function` accesses this memory, it will pop x's tag off the stack

Incomplete proof sketch



x's tag is at the top of the stack

```
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  unknown_function();
  return *x; // must return 42
}
```

UB unless x's permission is still in the stack

if `unknown_function` accesses this memory, it will pop x's tag off the stack

Stacked Borrows

- Stacked Borrows is restrictive enough to enable useful optimizations
  - ✓ formal proof

Stacked Borrows

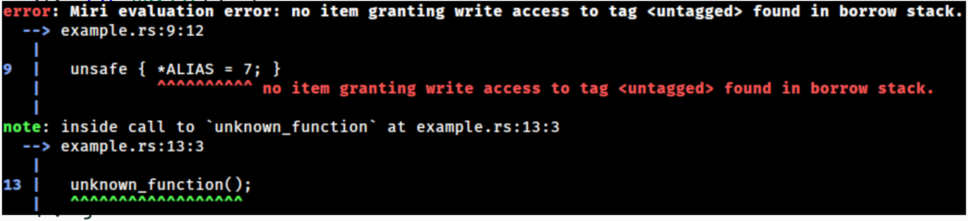
- Stacked Borrows is permissive enough to enable programming
  - ✓ checked standard library test suite by instrumenting the Rust interpreter Miri

```
static mut ALIAS: *mut i32 = std::ptr::null_mut();
fn main() {
  let mut l = 13;
  unsafe { ALIAS = &mut l as *mut i32; }
  let answer = test_unique(&mut l);
  println!("The answer is {}", answer); // prints 7
}
fn unknown_function() {
  unsafe { *ALIAS = 7; }
}
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  unknown_function();
  return *x; // should return 42, but returns 7
}
```

Stack: `[l, x]`

`ALIAS` is not on the stack 💥

```
 1: static mut ALIAS: *mut i32 = std::ptr::null_mut();
 2: fn main() {
```



```
error: Miri evaluation error: no item granting write access to tag <untagged> found in borrow stack.
  --> example.rs:9:12
   |
9  |   unsafe { *ALIAS = 7; }
   |            ^^^^^^^^^^ no item granting write access to tag <untagged> found in borrow stack.
   |
note: inside call to `unknown_function` at example.rs:13:3
  --> example.rs:13:3
   |
13 |   unknown_function();
   |   ^^^^^^^^^^^^^^^^^^
```

```
 8: fn unknown_function() {
 9:   unsafe { *ALIAS = 7; }
10: }
11: fn test_unique(x: &mut i32) -> i32 {
12:   *x = 42;
13:   unknown_function();
14:   return *x; // should return 42, but returns 7
15: }
```

`ALIAS` is not on the stack 💥

```
static mut ALIAS: *mut i32 = std::ptr::null_mut();
fn main() {
  let mut l = 13;
  unsafe { ALIAS = &mut l as *mut i32; }
  let answer = test_unique(&mut l);
```

We are regularly running the Rust standard library test suite in Miri to catch regressions.

Found and fixed 6 aliasing violations.

```
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  unknown_function();
  return *x; // should return 42, but returns 7
}
```

What else? [POPL'20 paper #2]

What I didn't talk about:

- Shared references & interior mutability
- Protectors (enable writes to be moved across unknown code)

Future work:

- Concurrency
- Integrating Stacked Borrows into RustBelt

A dynamic model of Rust's reference checker ensures soundness of type-based optimizations, even in the presence of unsafe code.

Try Miri out yourself!

- Web version: `https://play.rust-lang.org/` (“Tools”)
- Installation: `rustup component add miri`
- Miri website: `https://github.com/rust-lang/miri/`

Also check out our project website:

`https://plv.mpi-sws.org/rustbelt`

The **RustBelt** Team

@MPI-SWS


Ralf Jung


Jan-Oliver Kaiser


David Swasey


Hai Dang


Azalea Raad


Michael Sammler


Joshua Yanovski


Viktor Vafeiadis


Derek Dreyer

@CNRS @Tel Aviv


Jacques-Henri Jourdan


Ori Lahav

@Delft @Aarhus


Robbert Krebbers


Lars Birkedal

@KAIST @SNU


Jeehoon Kang


Chung-Kil Hur