RustBelt: Securing the Foundations of the Rust Programming Language

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, Derek Dreyer

POPL 2018 in Los Angeles, USA

Max Planck Institute for Software Systems (MPI-SWS), TU Delft

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

A safe & modern systems PL

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

A safe & modern systems PL

- First-class functions
- Polymorphism/generics
- Traits $\approx$ Type classes incl. associated types

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

A safe & modern systems PL

- First-class functions
- Polymorphism/generics
- Traits ≈ Type classes incl. associated types
- Control over resource management (e.g., memory allocation and data layout)

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

A safe & modern systems PL

- First-class functions
- Polymorphism/generics
- Traits ≈ Type classes incl. associated types
- Control over resource management (e.g., memory allocation and data layout)
- Strong type system guarantees:
    - Type & memory safety; absence of data races

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

A

- Firs
- Poly
- Trai
  asso
- Con
  (e.g.

Goal of RustBelt project:
Prove safety of Rust and its standard library.

- Strong type system guarantees:
  - Type & memory safety; absence of data races

Contributions

- λ_{Rust}: Core calculus representing a fragment of Rust and its type system
- Semantic soundness proof using logical relation in Iris
- Safety proof of some important unsafe libraries

Rust 101

Rust 101

~~Aliasing~~

~~+~~

~~Mutation~~

Ownership

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

Ownership

```rust
// 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`:

```rust
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's 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);

// S
send
```

`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
let mut v : Vec<i32> = vec![1, 2, 3];
```

`&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:

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**?

Synchronization mechanisms:

- Locks, channels, ...

Memory management:

- Reference counting, ...

```
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
```

Interior mutability

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

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

Type of `lock`:

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

Interior mutability

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

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

Aliasing

+

Mutation

?

unsafe

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

unsafe

```
fn lock<'a>(&'a self) -> LockResult<MutexGuard<'a, T>>
{

    }
}
```

`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

$$\tau ::= \textbf{bool} \mid \textbf{int} \mid \textbf{own}_{n}\, \tau \mid \&^{\kappa}_{\textbf{mut}}\, \tau \mid \&^{\kappa}_{\textbf{shr}}\, \tau \mid \Pi \overline{\tau} \mid \Sigma \overline{\tau} \mid \ldots$$

The λ_{Rust} type system

$$\tau ::= \textbf{bool} \mid \textbf{int} \mid \textbf{own}_n\, \tau \mid \&^{\kappa}_{\textbf{mut}}\, \tau \mid \&^{\kappa}_{\textbf{shr}}\, \tau \mid \Pi \overline{\tau} \mid \Sigma \overline{\tau} \mid \ldots$$

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

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

The λ_{Rust} type system

$\tau ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{own}_n\,\tau \mid \&^{\kappa}_{\mathbf{mut}}\,\tau \mid \&^{\kappa}_{\mathbf{shr}}\,\tau \mid \Pi\overline{\tau} \mid \Sigma\overline{\tau} \mid \ldots$

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

Core substructural typing judgments:

$$\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{T}_1 \vdash I \dashv x.\, \mathbf{T}_2 \qquad\qquad \Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}, \mathbf{T} \vdash F$$

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

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

The λ_{Rust} type system

$\tau \mid \&^{\kappa}_{\textsf{shr}}\,\tau \mid \Pi\overline{\tau} \mid \Sigma\overline{\tau} \mid \ldots$

ments:

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

$$\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \textsf{static} \qquad \frac{\kappa \sqsubseteq_{\mathrm{l}} \overline{\kappa} \in \mathbf{L} \qquad \kappa' \in \overline{\kappa}}{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'} \qquad \frac{\kappa \sqsubseteq_{\mathrm{e}} \kappa' \in \mathbf{E}}{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'} \qquad \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa$$

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

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

$$\mathbf{E}; \mathbf{L} \vdash \textsf{static alive} \qquad \frac{\kappa \sqsubseteq_{\mathrm{l}} \overline{\kappa} \in \mathbf{L} \qquad \forall i.\, \mathbf{E}; \mathbf{L} \vdash \overline{\kappa}_i \ \textsf{alive}}{\mathbf{E}; \mathbf{L} \vdash \kappa \ \textsf{alive}} \qquad \frac{\mathbf{E}; \mathbf{L} \vdash \kappa \ \textsf{alive} \qquad \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\mathbf{E}; \mathbf{L} \vdash \kappa' \ \textsf{alive}}$$

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

$$\mathbf{E}_1; \mathbf{L}_1 \vdash \emptyset \qquad \frac{\mathbf{E}_1; \mathbf{L}_1 \vdash \kappa \sqsubseteq \kappa' \qquad \mathbf{E}_1; \mathbf{L} \ldots}{\mathbf{E}_1; \mathbf{L}_1 \vdash \mathbf{E}_2, \kappa \sqsubseteq_{\mathrm{e}} \ldots}$$

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
$$\frac{\mathbf{E}; \mathbf{L} \vdash \tau \Rightarrow \tau' \qquad \mathbf{E}; \mathbf{L} \vdash \tau' \Rightarrow \tau''}{\mathbf{E}; \mathbf{L} \vdash \tau \Rightarrow \tau''}$$

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
$$\mathbf{E}; \mathbf{L} \vdash \lightning_{\Sigma\overline{n}} \Leftrightarrow \Pi\overline{\lightning_{n}}$$

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

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

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

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
$$\frac{\begin{array}{c}\Gamma, \overline{\alpha}', \digamma : \textsf{lft} \mid \mathbf{E}', \mathbf{E}_0; \mathbf{L}_0 \vdash \mathbf{E}[\overline{\kappa}/\overline{\alpha}] \\ \forall i.\, \Gamma, \overline{\alpha}', \digamma : \textsf{lft} \mid \mathbf{E}', \mathbf{E}_0; \mathbf{L}_0 \vdash \overline{\tau}_i' \Rightarrow \overline{\tau}_i \qquad \Gamma, \overline{\alpha}', \digamma : \textsf{lft} \mid \mathbf{E}', \mathbf{E}_0; \mathbf{L}_0 \vdash \tau \Rightarrow \tau'\end{array}}{\Gamma \mid \mathbf{E}_0; \mathbf{L}_0 \vdash \forall \overline{\alpha}.\, \textsf{fn}(\digamma : \mathbf{E}; \overline{\tau}) \rightarrow \tau \Rightarrow \forall \overline{\alpha}'.\, \textsf{fn}(\digamma : \mathbf{E}'; \overline{\tau}') \rightarrow \tau'}$$

The λ_{Rust} type syst

Lifetime inclusion

F-LETCONT
$$\frac{\Gamma, k, \overline{x} : \mathsf{val} \mid \mathbf{E}; \mathbf{L}_1 \mid \mathbf{K}, k \lhd \mathsf{cont}(\mathbf{L}_1; \overline{x}.\,\mathbf{T}'); \mathbf{T}' \vdash F_1 \qquad \Gamma, k : \mathsf{val} \mid \mathbf{E}; \mathbf{L}_2 \mid \mathbf{K}, k \lhd \mathsf{cont}(\mathbf{L}_1; \overline{x}.\,\mathbf{T}'); \mathbf{T} \vdash F_2}{\Gamma \mid \mathbf{E}; \mathbf{L}_2 \mid \mathbf{K}; \mathbf{T} \vdash \mathtt{letcont}\ k(\overline{x}) := F_1\ \mathtt{in}\ F_2}$$

F-IF
$$\frac{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F_1 \qquad \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F_2}{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p \lhd \mathsf{bool} \vdash \mathtt{if}\ p\ \mathtt{then}\ F_1\ \mathtt{else}\ F_2}$$

F-JUMP
$$\frac{\mathbf{E}; \mathbf{L} \vdash \mathbf{T} \Rightarrow \mathbf{T}'[\overline{y}/\overline{x}]}{\mathbf{E}; \mathbf{L} \mid k \lhd \mathsf{cont}(\mathbf{L}; \overline{x}.\,\mathbf{T}'); \mathbf{T} \vdash \mathtt{jump}\ k(\overline{y})}$$

F-ENDLFT
$$\frac{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}' \vdash F \qquad \mathbf{T} \Rightarrow^{\dagger\kappa} \mathbf{T}'}{\mathbf{E}; \mathbf{L}, \kappa \sqsubseteq_1 \overline{\kappa} \mid \mathbf{K}; \mathbf{T} \vdash \mathtt{endlft}; F}$$

$\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\tau} \tau_2}$

TWRITE-BOR
$$\frac{\mathbf{E}; \mathbf{L} \vdash \kappa\ \mathsf{alive}}{\mathbf{E}; \mathbf{L} \vdash \&^{\kappa}_{\mathsf{mut}}\,\tau \multimap^{\tau} \&^{\kappa}_{\mathsf{mut}}\,\tau}$$

$\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \circ\!\!-^{\tau} \tau_2}$

TREAD-OWN-MOVE
$$\frac{n = \mathsf{size}(\tau)}{\mathbf{E}; \mathbf{L} \vdash \mathsf{own}_m\,\tau \circ\!\!-^{\tau} \mathsf{own}_m\,\lightning_n}$$

TREAD-BOR
$$\frac{\tau\ \mathsf{copy} \qquad \mathbf{E}; \mathbf{L} \vdash \kappa\ \mathsf{alive}}{\mathbf{E}; \mathbf{L} \vdash \&^{\kappa}_{\mu}\,\tau \circ\!\!-^{\tau} \&^{\kappa}_{\mu}\,\tau}$$

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

S-FALSE
$$\mathbf{E}; \mathbf{L} \mid \emptyset \vdash \mathtt{false} \dashv x.\,x \lhd \mathsf{bool}$$

S-NUM
$$\mathbf{E}; \mathbf{L} \mid \emptyset \vdash z \dashv x.\,x \lhd \mathsf{int}$$

S-FN
$$\frac{\overline{\tau}'\ \mathsf{copy} \qquad \overline{\tau}'\ \mathsf{send} \qquad \Gamma, \overline{\alpha}, \digamma : \mathsf{lft}, f, \overline{x}, k : \mathsf{val} \mid \mathbf{E}, \mathbf{E}'; \digamma \sqsubseteq_1 [\,] \mid k \lhd \mathsf{cont}(\digamma \sqsubseteq_1 [\,]; y.\,y \lhd \mathsf{own}\,\tau); \ \overline{p} \mathrel{\overline{\lhd}} \overline{\tau}', \overline{x} \mathrel{\overline{\lhd}} \mathsf{own}\,\overline{\tau}, f \lhd \forall\overline{\alpha}.\,\mathsf{fn}(\digamma : \mathbf{E}; \overline{\tau}) \to \tau \vdash F}{\Gamma \mid \mathbf{E}'; \mathbf{L}' \mid \overline{p} \mathrel{\overline{\lhd}} \overline{\tau}' \vdash \mathtt{funrec}\ f(\overline{x})\ \mathtt{ret}\ k := F \dashv f.\,f \lhd \forall\overline{\alpha}.\,\mathsf{fn}(\digamma : \mathbf{E}; \overline{\tau}) \to \tau}$$

S-PATH
$$\mathbf{E}; \mathbf{L} \mid p \lhd \tau \vdash p \dashv x.\,x \lhd \tau$$

S-NAT-OP
$$\mathbf{E}; \mathbf{L} \mid p_1 \lhd \mathsf{int}, p_2 \lhd \mathsf{int} \vdash p_1 \{+,-\}\, p_2 \dashv x.\,x \lhd \mathsf{int}$$

S-NAT-LEQ
$$\mathbf{E}; \mathbf{L} \mid p_1 \lhd \mathsf{int}, p_2 \lhd \mathsf{int} \vdash p_1 \le p_2 \dashv x.\,x \lhd \mathsf{bool}$$

S-NEW
$$\mathbf{E}; \mathbf{L} \mid \emptyset \vdash \mathtt{new}(n) \dashv x.\,x \lhd \mathsf{own}_n\,\lightning_n$$

S-DELETE
$$\frac{n = \mathsf{size}(\tau)}{\mathbf{E}; \mathbf{L} \mid p \lhd \mathsf{own}_n\,\tau \vdash \mathtt{delete}(n, p) \dashv \emptyset}$$

S-DEREF
$$\frac{\mathbf{E}; \mathbf{L} \vdash \tau_1 \circ\!\!-^{\tau} \tau_1' \qquad \mathsf{size}(\tau) = 1}{\mathbf{E}; \mathbf{L} \mid p \lhd \tau_1 \vdash {}^*p \dashv x.\,p \lhd \tau_1', x \lhd \tau}$$

S-DEREF-BOR-OWN
$$\frac{\mathbf{E}; \mathbf{L} \vdash \kappa\ \mathsf{alive}}{\mathbf{E}; \mathbf{L} \mid p \lhd \&^{\kappa}_{\mu}\,\mathsf{own}_n\,\tau \vdash {}^*p \dashv x.\,x \lhd \&^{\kappa}_{\mu}\,\tau}$$

S-DEREF-BOR-BOR
$$\frac{\mathbf{E}; \mathbf{L} \vdash \kappa\ \mathsf{alive} \qquad \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\mathbf{E}; \mathbf{L} \mid p \lhd \&^{\kappa}_{\mu}\,\&^{\kappa'}_{\mathsf{mut}}\,\tau \vdash {}^*p \dashv x.\,x \lhd \&^{\kappa}_{\mu}\,\tau}$$

S-ASSGN
$$\frac{\mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\tau} \tau_1'}{\mathbf{E}; \mathbf{L} \mid p_1 \lhd \tau_1, p_2 \lhd \tau \vdash p_1 := p_2 \dashv p_1 \lhd \tau_1'}$$

S-SUM-ASSGN-UNIT
$$\frac{\overline{\tau}_i = \Pi[\,] \qquad \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\Sigma\overline{\tau}} \tau_1'}{\mathbf{E}; \mathbf{L} \mid p \lhd \tau_1 \vdash p \overset{\mathsf{inj}\ i}{:=} () \dashv p \lhd \tau_1'}$$

S-SUM-ASSGN
$$\frac{\overline{\tau}_i = \tau \qquad \tau_1 \multimap^{\Sigma\overline{\tau}} \tau_1'}{\mathbf{E}; \mathbf{L} \mid p_1 \lhd \tau_1, p_2 \lhd \tau \vdash p_1 \overset{\mathsf{inj}\ i}{:=} p_2 \dashv p_1 \lhd \tau_1'}$$

S-MEMCPY
$$\frac{\mathsf{size}(\tau) = n \qquad \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\tau} \tau_1' \qquad \mathbf{E}; \mathbf{L} \vdash \tau_2 \circ\!\!-^{\tau} \tau_2'}{\mathbf{E}; \mathbf{L} \mid p_1 \lhd \tau_1, p_2 \lhd \tau_2 \vdash p_1 :=_n {}^*p_2 \dashv p_1 \lhd \tau_1', p_2 \lhd \tau_2'}$$

S-SUM-MEMCPY
$$\frac{\mathsf{size}(\tau) = n \qquad \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\Sigma\overline{\tau}} \tau_1' \qquad \mathbf{E}; \mathbf{L} \vdash \tau_2 \circ\!\!-^{\tau} \tau_2' \qquad \overline{\tau}_i = \tau}{\mathbf{E}; \mathbf{L} \mid p_1 \lhd \tau_1, p_2 \lhd \tau_2 \vdash p_1 \overset{\mathsf{inj}\ i}{:=}_n {}^*p_2 \dashv p_1 \lhd \tau_1', p_2 \lhd \tau_2'}$$

Syntactic type safety

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

Usually proven by progress and preservation.

But what about unsafe code?

Syntactic type safety

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

Usually proven by progress and preservation.

But what about unsafe code?

Use a semantic approach based on logical relations.

The logical relation

Ownership predicate for every type τ:

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

The logical relation

Ownership predicate for every type τ:

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

Owning thread's ID

Data in memory

The logical relation

Ownership predicate for every type τ:

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

Owning thread's ID

Data in memory

We use Iris to define ownership:

- Concurrent separation logic
- Designed to derive new reasoning principles inside the logic

The logical relation

Ownership predicate for every type τ:

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

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

$$[\![p_1 \lhd \tau_1, p_2 \lhd \tau_2]\!](t) \quad := $$

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

The logical relation

Ownership predicate for every type τ:

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

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

$$[\![p_1 \lhd \tau_1, p_2 \lhd \tau_2]\!](t) \quad := $$

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

Separating conjunction

The logical relation

Ownership predicate for every type τ:

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

Lift to semantic typing judgments:

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

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

Compatibility lemmas

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

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

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

Compatibility lemmas

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

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

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

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

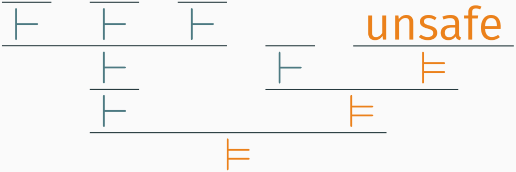
$\Gamma \mid \mathbf{E}; \mathbf{L}$ … **mut** τ

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

Linking with unsafe code

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash} \quad \dfrac{}{\vdash} \quad \dfrac{}{\vdash}}{\vdash}}{\vdash} \qquad \dfrac{\dfrac{}{\vdash} \quad \dfrac{\text{unsafe}}{\nvdash}}{\nvdash}}{\nvdash}$$

Linking with unsafe code



Linking with unsafe code

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

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

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

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

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

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

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

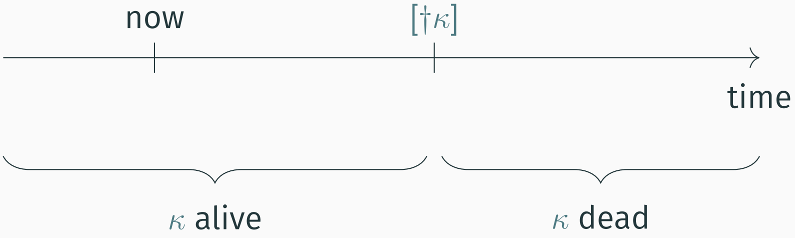
$$\llbracket \&^{\kappa}_{\mathbf{mut}}\, \tau \rrbracket.\mathrm{own}(t, \overline{v}) :=$$
$$\exists \ell.\ \overline{v} = [\ell] * \&^{\kappa}_{\mathbf{full}}\big(\exists \overline{w}.\ \ell \mapsto \overline{w} * \llbracket \tau \rrbracket.\mathrm{own}(t, \overline{w})\big)$$

Lifetime logic connective

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

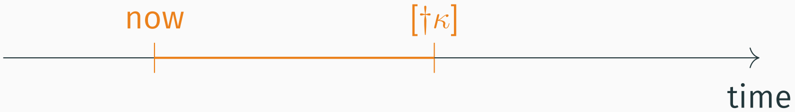
Lifetime logic allows splitting ownership in time!

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



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

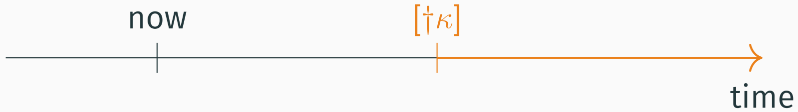
Access to P while κ lasts



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

Access to P while κ lasts

Access to P when κ has ended



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

Acc

The lifetime logic has been fully derived inside Iris.

time

What else is in the paper?

- 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<T>`, `Cell<T>`

What else is in the paper?

- 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<T>`, `Cell<T>`, `RefCell<T>`, `Rc<T>`, `Arc<T>`, `RwLock<T>` (found a bug), …

What else is in the paper?

- 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<T>`, `Cell<T>`, `RefCell<T>`, `Rc<T>`, `Arc<T>`, `RwLock<T>` (found a bug), ...

Still missing from RustBelt:

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

Logical relations can be used to prove safety of languages with unsafe operations.

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

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