RustBelt: Securing the Foundations of the Rust Programming Language

Ralf Jung
Jacques-Henri Jourdan
Robbert Krebbers
Derek Dreyer

MPI-SWS & TU Delft

October 13, 2017
ETH Zürich

Rust

Mozilla's replacement for C/C++

A safe & flexible systems programming language

- Modern strongly-typed PL:
  - First-class functions, polymorphism/generics
  - *Traits* ≈ Type classes + associated types
- But with control over resource management (e.g., memory allocation and data layout)
- **Sound type system** with strong guarantees:
  - Type & memory safety; absence of data races

Rust

Mozilla's replacement for C/C++

A safe & flexible systems programming language

- Modern strongly-typed PL:
  - First-class functions, polymorphism/generics
  - *Traits* $\approx$ Type classes + associated types
- But with control over resource management (e.g., memory allocation and data layout)
- **Sound?** **type system** with strong guarantees:
  - Type & memory safety; absence of data races

Goal of ERC RustBelt project:

- **Prove the soundness** of Rust's type system in Coq!

The key challenge

Superficially, Rust's approach to ensuring safety is sold as:

"No mutation through aliased pointers"

The key challenge

Superficially, Rust's approach to ensuring safety is sold as:

"No mutation through aliased pointers"

But this is not always true!

- **Many Rust libraries permit mutation through aliased pointers**
- The safety of this is highly non-obvious because these libraries make use of **unsafe features**!

The key challenge

Superficially, Rust's approach to ensuring safety is sold as:

"No mutation through aliased pointers"

But this is not always true!

- **Many Rust libraries permit mutation through aliased pointers**
- The safety of this is highly non-obvious because these libraries make use of **unsafe features**!

So why is any of this sound?

Introduction

Overview of Rust

RustBelt

Conclusion

```rust
let (snd, rcv) = channel();
join(
  move || { // First thread
    // Allocating [b] as Box<i32> (pointer to heap)
    let mut b = Box::new(0);
    *b = 1;

    // Transferring the ownership to the other thread...
    snd.send(b);



  },
  move || { // Second thread
    let b = rcv.recv().unwrap(); // ... that receives it
    println!("{}", *b);          // ... and uses it.
  });
```

```
let (snd, rcv) = channel();
join(
  move || { // First thread
    // Allocating [b] as Box<i32> (pointer to heap)
    let mut b = Box::new(0);
    *b = 1;

    // Transferring the ownership to the other thread...
    snd.send(b);
    *b = 2;     // Error: lost ownership of [b]
                // ==> Prevents data race
  },
  move || { // Second thread
    let b = rcv.recv().unwrap(); // ... that receives it
    println!("{}", *b);          // ... and uses it.
  });
```

Borrowing and lifetimes

```rust
let mut v = vec![1, 2, 3];

v[1] = 4;




v.push(6);
println!("{:?}", v);
```

Borrowing and lifetimes

```
let mut v = vec![1, 2, 3];

{ let mut inner_ptr = Vec::index_mut(&mut v, 1);


  *inner_ptr = 4; }

v.push(6);
println!("{:?}", v);
```

Borrowing and lifetimes

```rust
let mut v = vec![1, 2, 3];

{ let mut inner_ptr = Vec::index_mut(&mut v, 1);
  // Error: can invalidate [inner_ptr]
  v.push(1);
  *inner_ptr = 4; }

v.push(6);
println!("{:?}", v);
```

Borrowing and lifetimes

```
let mut v = vec![1, 2, 3];

{ let mut inner_ptr = Vec::index_mut(&mut v, 1);
  // Error: can invalidate [inner_ptr]
  v.push(1);
  *inner_ptr = 4; }
```

```
v.push(6);
println!("{:?}", v);
```

We **temporarily** lost ownership of vector `v`

We get back the full ownership of vector `v`

Borrowing and lifetimes

```
let mu

{ let

  *inn

v.push
printl
```

Type of `index_mut`:

```
fn<'a> index_mut(&'a mut Vec<i32>, usize)
        -> &'a mut i32
```

New pointer type: `&'a mut T`:

- **mutable borrow**ed reference
- valid only for **lifetime** `'a`

Borrowing and lifetimes

```
let mut v = vec![1, 2, 3];

{ let mut inner_ptr = Vec::index_mut(&mut v, 1);


  *inner_ptr = 4; }

v.push(6);
println!("{:?}", v);
```

Lifetime `'a` inferred by Rust

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 borrow** of `x`

- Type: `&'a i32`
- Can be copied/shared
- Does not allow mutation

Summing up

- Rust's type system is based on **ownership**
- Three kinds of ownership:
  1. Full ownership: `Vec<T>` (vector), `Box<T>` (pointer to heap)
  2. **Mutable borrow**ed reference: `&'a mut T`
  3. **Shared borrow**ed reference: `&'a T`
- **Lifetimes** decide when borrows are valid

Interior mutability

What if we want **shared mutable data structures**?

Rust standard library provides types with **interior mutability**

- Allows mutation using only a shared reference `&'a T`
- Implemented in Rust **using unsafe features**
- Unsafety is claimed to be **safely encapsulated**
  - The library interface restricts what mutations are possible

Mutex

An example of Interior mutability

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

// We can mutate the integer
// *with a shared borrow* only
join (|| *(&m).lock().unwrap() += 1,
      || *(&m).lock().unwrap() += 1);

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

Mutex

An example of Interior mutability

```
let m
// We
// *w
join (
// Un
printl
```

A shared borrow **establishes a sharing protocol**:

- `&'a i32`
  - $\Longrightarrow$ **Read-only**
  - Safety: trivial
- `&'a Mutex<i32>`
  - $\Longrightarrow$ Read-write **by taking the lock**
  - Safety: ensured by proper synchronization

How do we know this all works?

Introduction

Overview of Rust

RustBelt

Conclusion

The λ_{Rust} type system

- Syntactic (built-in types)

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

- Typing context $\mathbf{T}$ assigns types τ to paths p
- Typing individual instructions:

(Γ binds variables, **E** and **L** track lifetimes)

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

- Typing whole functions: (**K** tracks continuations)

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

Some typing rules

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

Some typing rules

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

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

Syntactic type safety

The standard “syntactic” approach to language safety is to prove a theorem like the following, via good old “progress and preservation”:

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

- **Problem**: This theorem does not help when unsafe code is used!
- **Solution**: A more semantic approach based on **logical relations**

The logical relation

- Define, for every type τ, an **ownership predicate**, where t is the owning thread's id and $\overline{v}$ is the representation of τ:

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

The logical relation

- Define, for every type τ, an **ownership predicate**, where t is the owning thread's id and $\overline{v}$ is the representation of τ:

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

- Lift to semantic contexts $[\![\mathbf{T}]\!](t)$ using **separating conjunction**:

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

The logical relation

- Define, for every type τ, an **ownership predicate**, where t is the owning thread's id and $\overline{v}$ is the representation of τ:

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

- Lift to semantic contexts $[\![\mathbf{T}]\!](t)$ using **separating conjunction**:

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

- Lift to semantic typing judgments:

$$\begin{aligned}\mathbf{E}; \mathbf{L} \mid \mathbf{T}_1 \models S \dashv \mathbf{T}_2 \quad &:= \\ &\forall t.\, \{[\![\mathbf{E}]\!] * [\![\mathbf{L}]\!] * [\![\mathbf{T}_1]\!](t)\}\ S\ \{[\![\mathbf{E}]\!] * [\![\mathbf{L}]\!] * [\![\mathbf{T}_2]\!](t)\}\end{aligned}$$

Compatibility lemmas

To connect logical relation to type system,
we show **semantic versions** of all **syntactic typing rules**.

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

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

Compatibility lemmas

To connect logical relation to type system,
we show **semantic versions** of all **syntactic typing rules**.

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

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

Type safety (revisited)

- From compatibility:

$$\mathbf{E}; \mathbf{L} \mid \mathbf{K}, \mathbf{T} \vdash F \dashv \mathbf{T}_2 \quad \Longrightarrow \quad \mathbf{E}; \mathbf{L} \mid \mathbf{K}, \mathbf{T} \models F \mathrel{=\!\!|} \mathbf{T}_2$$

- Finally, we show that the relation is **adequate**:

$$\mathbf{E}; \mathbf{L} \mid \mathbf{T}_1 \models F \mathrel{=\!\!|} \mathbf{T}_2 \quad \Longrightarrow \quad F \text{ is safe}$$

- Conclusion: **well-typed programs can't go wrong**
  - No data race, no memory error, ...

Type safety (semantic version)

The semantic approach provides a much stronger safety theorem than syntactic type safety:

- For well-typed code, $\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F_{\mathsf{safe}} \Longrightarrow \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \models F_{\mathsf{safe}}$
- If unsafe features are used, manually prove $\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \models F_{\mathsf{unsafe}}$
- By compatibility, we can compose these proofs and obtain safety of the entire program!

Type safety (semantic version)

The semantic approach provides a much stronger safety theorem than syntactic type safety:

- For well-typed code, $\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F_{\mathsf{safe}} \Longrightarrow \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \models F_{\mathsf{safe}}$
- If unsafe features are used, manually prove $\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \models F_{\mathsf{unsafe}}$
- By compatibility, we can compose these proofs and obtain safety of the entire program!

The whole program is safe if the "unsafe" pieces are safe.

How do we define the logical interpretation of types?

Choosing the right logic

Rust type system has **ownership** + complex **sharing protocols** in a **higher-order concurrent** setting

Choosing the right logic

Rust type system has **ownership** + complex **sharing protocols** in a **higher-order concurrent** setting

"Obvious" choice of a logic for interpreting Rust types:

Higher-order concurrent separation logic

Choosing the right logic

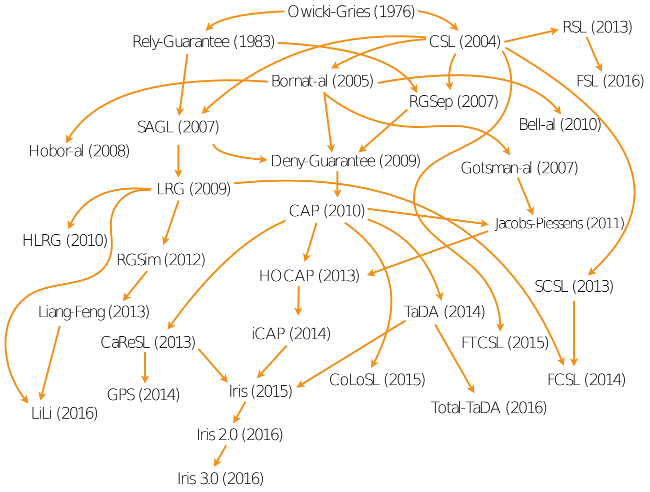
Rust type system has **ownership** + complex **sharing protocols** in a **higher-order concurrent** setting

"Obvious" choice of a logic for interpreting Rust types:

Higher-order concurrent separation logic

But which one?

A brief history of concurrent separation logic



Picture by Ilya Sergey

A brief history of concurrent separation logic

$$\dfrac{\begin{array}{c}\Gamma, \Delta \mid \Phi \vdash \mathsf{stable}(\mathsf{P}) \qquad \Gamma, \Delta \mid \Phi \vdash \forall y.\ \mathsf{stable}(\mathsf{Q}(y)) \\ \Gamma, \Delta \mid \Phi \vdash n \in C \qquad \Gamma, \Delta \mid \Phi \vdash \forall x \in X.\ (x, f(x)) \in \overline{T(A)} \vee f(x) = x \\ \Gamma \mid \Phi \vdash \forall x \in X.\ (\Delta).\langle \mathsf{P} * \circledast_{\alpha \in A} [\alpha]^{\mathsf{n}}_{g(\alpha)} * \triangleright I(x) \rangle\ c\ \langle \mathsf{Q}(x) * \triangleright I(f(x)) \rangle^{C \setminus \{n\}}\end{array}}{\begin{array}{c}\Gamma \mid \Phi \vdash (\Delta).\ \langle \mathsf{P} * \circledast_{\alpha \in A} [\alpha]^{\mathsf{n}}_{g(\alpha)} * \mathsf{region}(X, T, I, n) \rangle \\ c \\ \langle \exists x.\ \mathsf{Q}(x) * \mathsf{region}(\{f(x)\}, T, I, n) \rangle^{C}\end{array}}\ \textsc{Atomic}$$

$$\dfrac{\mathcal{C} \vdash \forall b \overset{\text{rely}}{\sqsupseteq_\pi} b_0.\ (\!| \pi[\![b]\!] * P |\!)\ i \Rrightarrow_{\mathrm{I}} a\ (\!| x.\ \exists b' \overset{\text{guar}}{\sqsupseteq_\pi} b.\ \pi[\![b']\!] * Q |\!)}{\mathcal{C} \vdash \left\{ \boxed{b_0}^{n}_{\pi} * \triangleright P \right\}\ i \Rrightarrow a\ \left\{ x.\ \exists b'.\ \boxed{b'}^{n}_{\pi} * \right.}\ \textsc{UpdIsl}$$

Use atomic rule

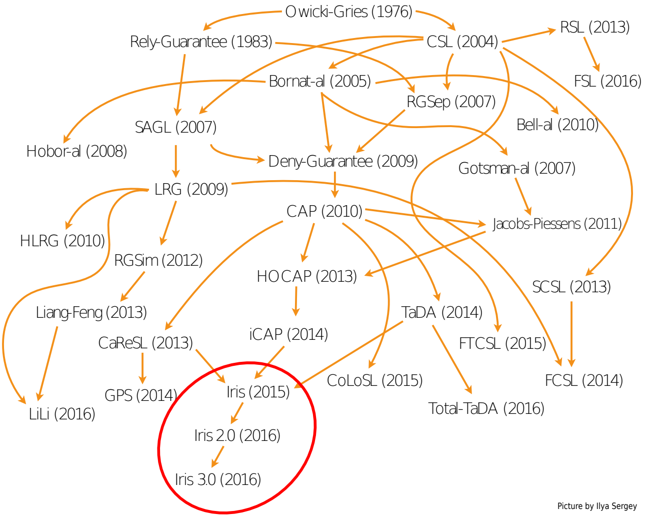
$$\dfrac{\begin{array}{c} a \notin \mathcal{A} \quad \forall x \in X.\ (x, f(x)) \in \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \\ \lambda; \mathcal{A} \vdash \forall x \in X.\ \langle p_p \mid I(\mathbf{t}^\lambda_a(x)) * p(x) * [\mathrm{G}]_a \rangle\ \mathbb{C}\ \ \exists y \in Y.\ \langle q_p(x, y) \mid I(\mathbf{t}^\lambda_a(f(x))) * q(x, y) \rangle \end{array}}{\lambda + 1; \mathcal{A} \vdash \forall x \in X.\ \langle p_p \mid \mathbf{t}^\lambda_a(x) * p(x) * [\mathrm{G}]_a \rangle\ \mathbb{C}\ \ \exists y \in Y.\ \langle q_p(x, y) \mid \mathbf{t}^\lambda_a(f(x)) * q(x, y) \rangle}$$

$$\dfrac{\begin{array}{c}\Gamma \mid \Phi \vdash x \in X \qquad \Gamma \mid \Phi \vdash \forall \alpha \in \mathsf{Action}.\ \forall x \in \mathsf{SId} \times \mathsf{SId}.\ up(T(\alpha)(x)) \\ \Gamma \mid \Phi \vdash A \text{ and } B \text{ are finite} \qquad \Gamma \mid \Phi \vdash C \text{ is infinite} \\ \Gamma \mid \Phi \vdash \forall n \in C.\ \mathsf{P} * \circledast_{\alpha \in A} [\alpha]^n_1 \Rightarrow \triangleright I(n)(x) \\ \Gamma \mid \Phi \vdash \forall n \in C.\ \forall s.\ \mathsf{stable}(I(n)(s)) \qquad \Gamma \mid \Phi \vdash A \cap B = \emptyset\end{array}}{\Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq^{C} \exists n \in C.\ \mathsf{region}(X, T, I(n), n) * \circledast_{\alpha \in B} [\alpha]^n_1}\ \textsc{VAlloc}$$

Update region rule

$$\dfrac{\lambda; \mathcal{A} \vdash \forall x \in X.\ \left\langle p_p \,\middle|\, I(\mathbf{t}^\lambda_a(x)) * p(x) \right\rangle\ \mathbb{C}\ \ \exists y \in Y.\ \left\langle q_p(x, y) \,\middle|\, \begin{array}{c} I(\mathbf{t}^\lambda_a(Q(x))) * q_1(x, y) \\ \vee\ I(\mathbf{t}^\lambda_a(x)) * q_2(x, y) \end{array} \right\rangle}{\lambda{+}1; a : x \in X \rightsquigarrow Q(x), \mathcal{A} \vdash \begin{array}{c} \forall x \in X.\ \langle p_p \mid \mathbf{t}^\lambda_a(x) * p(x) * a \Rrightarrow \blacklozenge \rangle \\ \mathbb{C} \\ \exists y \in Y.\ \left\langle q_p(x, y) \,\middle|\, \begin{array}{c} \exists z \in Q(x).\ \mathbf{t}^\lambda_a(z) * q_1(x, y) * a \Rrightarrow (x, z) \\ \vee\ \mathbf{t}^\lambda_a(x) * q_2(x, y) * a \Rrightarrow \blacklozenge \end{array} \right\rangle \end{array}}$$

A brief history of concurrent separation logic



Picture by Ilya Sergey

Iris

Iris is a higher-order concurrent separation logic framework that we have been developing since 2014 [POPL'15, ICFP'16, POPL'17, ESOP'17, ECOOP'17]

Distinguishing features of Iris:

- **Simple** foundation: Higher-order BI + a handful of modalities
- Rules for complex "sharing protocols" (which were built in as primitive in prior logics) are **derivable** in Iris
- Supports **impredicative invariants**, which arise when modeling recursive & generic types in Rust
- Excellent tactical support for **mechanization in Coq**

Iris

Iris is a higher-order concurrent separation logic framework that we have been developing since 2014 [POPL'15, ICFP'16, POPL'17, ESOP'17, ECOOP'17]

Distinguishing features of Iris:

- **Simple** foundation: Higher-order BI + a handful of modalities
- Rules for complex "sharing protocols" (which were built in as primitive in prior logics) are **derivable** in Iris
- Supports **impredicative invariants**, which arise when modeling recursive & generic types in Rust
- Excellent tactical support for **mechanization in Coq**

Iris is ideal for modeling Rust!

Ownership interpretations of simple types

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

$$:=$$

$$\overline{v} = [\texttt{true}] \ \vee \ \overline{v} = [\texttt{false}]$$

$$[\![\tau_1 \times \tau_2]\!].\mathrm{own}(t, \overline{v})$$

$$:=$$

$$\exists \overline{v}_1, \overline{v}_2.\ \overline{v} = \overline{v}_1 +\!\!+ \overline{v}_2 * [\![\tau_1]\!].\mathrm{own}(t, \overline{v}_1) * [\![\tau_2]\!].\mathrm{own}(t, \overline{v}_2)$$

Ownership interpretations of pointer types

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

$$:=$$

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

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

$$:=$$

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

Ownership interpretations of pointer types

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

$$:=$$

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

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

$$:=$$

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

What is this? Not your grandma's separation logic!

Lifetime logic: A custom logic derived within Iris

Traditionally, $P * Q$ splits ownership w.r.t. space

Let's allow **splitting ownership w.r.t. time**!

$$\triangleright P \quad \Rrightarrow \quad \&^{\kappa} P \;\; * \; \left([\dagger\kappa] \Rrightarrow \triangleright P\right)$$

Lifetime logic: A custom logic derived within Iris

Traditionally, $P * Q$ splits ownership w.r.t. space

Let's allow **splitting ownership w.r.t. time**!

$$\triangleright P \quad \Rrightarrow \qquad \&^{\kappa} P \;\; * \;\; \big([\dagger\kappa] \Rrightarrow \triangleright P\big)$$

$\triangleright P$ can be transformed into...

Lifetime logic: A custom logic derived within Iris

Traditionally, $P * Q$ splits ownership w.r.t. space

Let's allow **splitting ownership w.r.t. time**!

$$\triangleright P \quad \Rrightarrow \quad \boxed{\&^{\kappa} P} \;*\; \left([\dagger\kappa] \Rrightarrow \triangleright P\right)$$

A *borrowed* part:

- access of P when κ is ongoing
- P must be *preserved* when κ ends

Lifetime logic: A custom logic derived within Iris

Traditionally, $P * Q$ splits ownership w.r.t. space

Let's allow **splitting ownership w.r.t. time**!

$$\triangleright P \quad \Rrightarrow \quad \&^{\kappa} P \;\; * \;\; \left([\dagger\kappa] \Rrightarrow \triangleright P\right)$$

An *inheritance* part, that gives back P when κ is finished.

Lifetime tokens

How to witness that κ is alive?

We use a **lifetime token** $[\kappa]$

- **Left in deposit** when opening a borrow:

$$\&^{\kappa} P * [\kappa] \quad \Rrightarrow \quad \triangleright P \ * \ \big(\triangleright P \Rrightarrow \&^{\kappa} P * [\kappa]\big)$$

- Needed to **terminate** κ:

$$[\kappa] \Rrightarrow [\dagger\kappa]$$

Modeling shared references

As we've seen, each type `T` may have a different "sharing protocol" defining the semantics of `&'a T`.

- E.g., `&'a i32` is read-only, whereas `&'a Mutex<i32>` grants mutable access to its contents once a lock is acquired

We model this by defining for each τ a "sharing predicate" $[\![\tau]\!].\mathrm{shr}$:

$$[\![\&^{\kappa}_{\mathbf{shr}}\,\tau]\!].\mathrm{own}(t, \overline{v})$$

$$:=$$

$$\exists \ell.\ \ \overline{v} = [\ell] * [\![\tau]\!].\mathrm{shr}([\![\kappa]\!], t, \ell)$$

The sharing predicate is required to be **persistent**:

- I.e., freely duplicable, since in Rust `&'a T` is a `Copy` type

Modeling “thread-safety” of types

Some interior-mutable types are not thread-safe

- They support shared mutable access without atomics
- Examples: reference-counted pointer (`Rc<T>`), ...

Still, Rust **guarantees absence of data races**

- Ownership transfer between threads only allowed for some types
- `T : Send` $\Longleftrightarrow$ `T` is thread-safe

In our model:

- Interpretations of types **may depend on the thread ID**
- $[\![\texttt{T : Send}]\!] \Longleftrightarrow [\![\texttt{T}]\!]$ does not depend on TID

Introduction

Overview of Rust

RustBelt

Conclusion

What else is in the paper [POPL'18]

More details about the λ_{Rust} type system and "lifetime logic"

How to model essential Rust types featuring interior mutability

- `Cell<T>`, `RefCell<T>`, `Rc<T>`, `Arc<T>`, `Mutex<T>`, `RwLock<T>`

How to handle lifetime inclusion and subtyping

Still missing from RustBelt:

- Trait objects (existential types), weak memory, panics, ...

Conclusion

Logical relations are a great way to prove safety of a real language in an "extensible" way.

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

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