RustBelt: Securing the Foundations of the Rust Programming Language

Ralf Jung
Jacques-Henri Jourdan
Robbert Krebbers
Derek Dreyer

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

June 15, 2017
WG2.8, Edinburgh

Rust

Mozilla's replacement for C/C++

Systems programming language **focusing on safety**

- Control over memory allocation & layout
- **Sound type system** with guarantees:
  - Type and memory safety
  - Absence of data races
  - Idea: Prohibit aliased mutable state
    - Using *borrow types* with "lifetimes"
  - First-class functions, polymorphism/generics
  - *Traits* $\approx$ Type classes + associated types

Rust

Mozilla's replacement for C/C++

Systems programming language **focusing on safety**

- Control over memory allocation & layout
- **Sound?** **type system** with guarantees:
  - Type and memory safety
  - Absence of data races
  - Idea: Prohibit aliased mutable state
    - Using *borrow types* with "lifetimes"
  - First-class functions, polymorphism/generics
  - *Traits* $\approx$ Type classes + associated types

RustBelt: **prove the soundness** of Rust's type system (idealized)

The key challenge

Superficially, borrow types look like they mean one thing:

- Read-only (immutable) references to shared state

But they don't mean that thing!

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

So why is any of this sound?

The key challenge

Superficially, borrow types look like they mean one thing:

- Read-only (immutable) references to shared state

But they don't mean that thing!

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

So why is any of this sound?

Parametricity!

Introduction

Overview of Rust

A semantic model of Rust

Lifetime logic

```
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.
  });
```

```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);
    *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

```
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

```
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 under a shared borrow
- Written in Rust **using unsafe features**
- **Safely encapsulated**
  - The library interface restricts mutations

Mutex

An example of Interior mutability

```
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

A semantic model of Rust

Lifetime logic

Our semantic approach: a model in Iris

One can write **unsafe code** in a **safely encapsulated** manner

- Goal: prove that these library are safe
- $\Longrightarrow$ Syntactic approaches will not work

Rust type system: **Ownership**, complex **sharing protocols**, in a **concurrent setting**

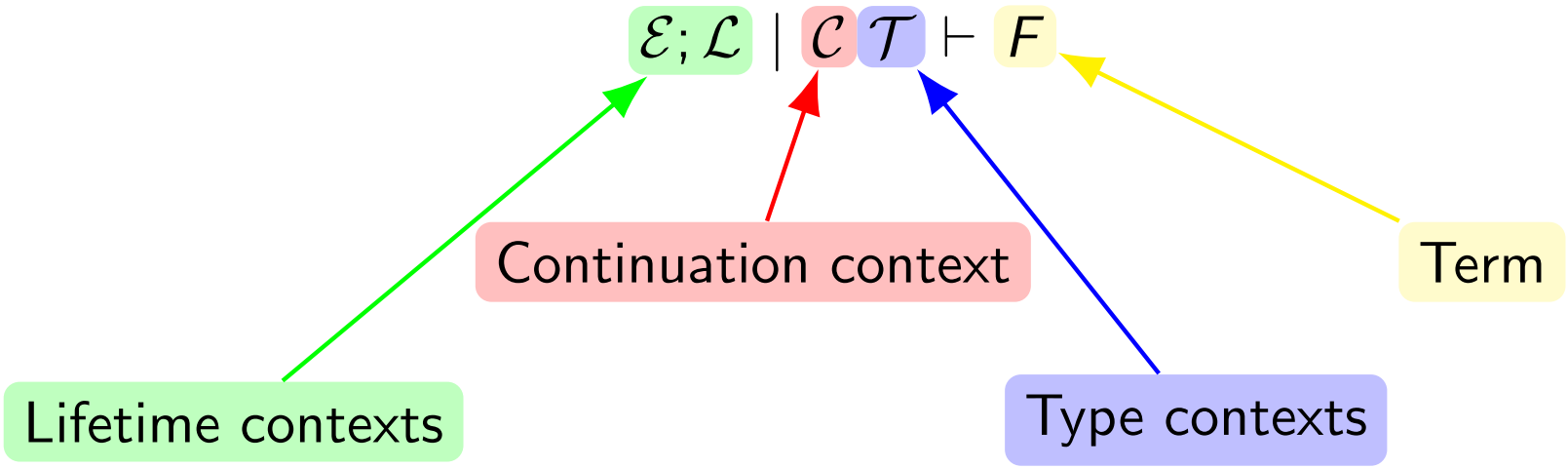
- **Iris** is a concurrent separation logic framework that we have been developing since 2014 [POPL'15, ICFP'16, ESOP'17, POPL'17, ECOOP'17]
- Iris has built-in support for these features and furthermore supports deriving new custom logics and mechanizing proofs in Coq
- $\Longrightarrow$ Iris is the right tool for modeling Rust!

Setup

Our language λ_{Rust}:

- Idealized version of Rust
- Lambda-calculus with heap, in CPS style

Type system: formalized & idealized version of Rust's:



Proof of soundness outline

- We define a **logical relation** for λ_{Rust} **in Iris**
  - Interpretation $[\![\cdot]\!]$ for types, typing judgments, ...
- The relation is **compatible** with type-checking rules:

$$\mathcal{E}; \mathcal{L} \mid \mathcal{C}; \mathcal{T} \vdash F \implies [\![\mathcal{E}; \mathcal{L} \mid \mathcal{C}; \mathcal{T} \vdash F]\!]$$

- The relation is **adequate**:

$$[\![\mathcal{E}; \mathcal{L} \mid \mathcal{C}; \mathcal{T} \vdash F]\!] \implies F \text{ is safe}$$

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

Interpretation of types

Types represent ownership

A type `T` $\Longrightarrow$ an Iris predicate:

$$[\![\texttt{T}]\!].\text{own} \ : \ \texttt{list val} \to \texttt{iProp}$$

Interpretation of types

Types represent ownership

A type `T` $\Longrightarrow$ an Iris predicate:

$$[\![\texttt{T}]\!].\mathrm{own} \ : \ \texttt{list val} \to \texttt{iProp}$$

Example:

$$[\![\texttt{Box<T>}]\!].\mathrm{own}(\bar{v}) \quad \triangleq \quad \begin{cases} \exists \bar{w}. \quad l \mapsto \bar{w} \ * \ \triangleright \, [\![\texttt{T}]\!].\mathrm{own}(\bar{w}) & \text{if } \bar{v} = [l] \\ \bot & \text{otherwise} \end{cases}$$

Interpretation of mutable borrows

$$[\![\texttt{\&'a mut T}]\!].\text{own} \triangleq ?$$

Need a way to express **“temporary ownership”**

Interpretation of mutable borrows

$$[\![\texttt{\&'a mut T}]\!].\mathrm{own} \triangleq ?$$

Need a way to express **"temporary ownership"**

Lifetime logic: library developed in Iris

- New **logical** connective: $\&^{\alpha} P$
  - Represents ownership of P **while the lifetime α is ongoing**
- We can then define:

$$[\![\texttt{\&'a mut T}]\!].\mathrm{own}([l]) \triangleq \&^{[\![\texttt{'a}]\!]}(\exists \bar{v}.\ l \mapsto \bar{v} * [\![\texttt{T}]\!].\mathrm{own}(\bar{v}))$$

Interpretation of shared borrows

$$[\![\texttt{\&'a T}]\!].\mathrm{own} \triangleq ?$$

How can one use `&'a T`?

Interpretation of shared borrows

$$[\![\texttt{\&'a T}]\!].\mathrm{own} \triangleq ?$$

How can one use `&'a T`? Depends on the **protocol** chosen by `T`.

Interpretation of shared borrows

$$\llbracket \texttt{\&'a T} \rrbracket.\mathrm{own} \triangleq ?$$

How can one use `&'a T`? Depends on the **protocol** chosen by `T`.

$\Longrightarrow$ We use a **parametric model** for shared borrows:

$$\llbracket \texttt{\&'a T} \rrbracket.\mathrm{own}([\mathit{l}]) \triangleq \llbracket \texttt{T} \rrbracket.\mathrm{shr}(\llbracket \texttt{'a} \rrbracket, \mathit{l}) \quad \textbf{is specific to } \texttt{T}$$

$\llbracket \texttt{T} \rrbracket.\mathrm{shr} : \texttt{lifetime} \times \texttt{loc} \to \texttt{iProp}$ is an **Iris predicate** with:

- $\llbracket \texttt{T} \rrbracket.\mathrm{shr}(\alpha, \mathit{l})$ is **persistent**
- $\llbracket \texttt{\&'a mut T} \rrbracket.\mathrm{own}([\mathit{l}]) \Rrightarrow \llbracket \texttt{T} \rrbracket.\mathrm{shr}(\llbracket \texttt{'a} \rrbracket, \mathit{l})$

Introduction

Overview of Rust

A semantic model of Rust

Lifetime logic

Borrows and inheritance

Usually: we split ownership with respect to space

Let's allow **splitting ownership over time**:

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

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

Borrows and inheritance

Usually: we split ownership with respect to space

Let's allow **splitting ownership over time**:

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

A *borrowed* part:

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

Borrows and inheritance

Usually: we split ownership with respect to space

Let's allow **splitting ownership over time**:

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

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

Lifetime tokens

How to witness that α is alive?

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

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

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

- Needed to **terminate** α:

$$[\alpha] \Rrightarrow [\dagger\alpha]$$

```
fn f<'a>(pair : &'a mut (i32, i32)) {
    let fst : &'a mut i32 = &mut pair.0;
    let snd : &'a mut i32 = &mut pair.1;
    join(|| *fst *= 2,
         || *snd += 1);
}
```

```
fn f<'a>(pair : &'a mut (i32, i32)) {
    let fst : &'a mut i32 = &mut pair.0;
    let snd : &'a mut i32 = &mut pair.1;
    join(|| *fst *= 2,
         || *snd += 1);
}
```

We need to **split borrows**:

$$\&^{\alpha}(P * Q) \Longleftrightarrow \&^{\alpha} P * \&^{\alpha} Q$$

```
fn f<'a>(pair : &'a mut (i32, i32)) {
    let fst : &'a mut i32 = &mut pair.0;
    let snd : &'a mut i32 = &mut pair.1;
    join(|| *fst *= 2,
         || *snd += 1);
}
```

Both threads witness that the lifetime is alive.
$\Longrightarrow$ We make the lifetime token **fractional**:

$$[\alpha]_{q+q'} \Leftrightarrow [\alpha]_q * [\alpha]_{q'}$$

Fractional lifetime tokens

How to witness that α is alive?

We use **lifetime tokens** $[\alpha]_q$

- Fractional: $[\alpha]_{q+q'} \Leftrightarrow [\alpha]_q * [\alpha]_{q'}$
- **Full token** needed to **terminate** α:

$$[\alpha]_1 \Rrightarrow [\dagger\alpha]$$

- **Fraction** left in deposit when opening a borrow:

$$\&^\alpha P * [\alpha]_q \quad \Rrightarrow \quad \triangleright P \; * \; \left(\triangleright P \Rrightarrow \&^\alpha P * [\alpha]_q\right)$$

Sharing protocols

$$[\![\texttt{\&'a T}]\!].\mathrm{own}([l]) \triangleq [\![\texttt{T}]\!].\mathrm{shr}([\![\texttt{'a}]\!], l) \triangleq ?$$

Depends on `T`. Common idea:

- **Share a borrow** using an invariant:

$$[\![\texttt{T}]\!].\mathrm{shr}([\![\texttt{'a}]\!], l) \triangleq \boxed{\&^{\alpha}\,[\text{Protocol}]}^{\mathcal{N}}$$

  - Technical problems with step-indexing

- Specific construction: **persistent borrows**: $\&^{\alpha/\mathcal{N}}_{\mathbf{at}}\,[\text{Protocol}]$
  - Behave like **cancellable invariant**

Conclusion

And also...

- Model of most of Rust's types with **interior mutability**
  - `Cell<T>`, `RefCell<T>`, `Rc<T>`, `Arc<T>`, `Mutex<T>`, `RwLock<T>`
- **Subtyping**/lifetime inclusions
- Some types cannot be **sent** or **shared** between threads
  - In Rust: `Send`/`Sync`
  - Our model: $[\![\texttt{T}]\!].\mathrm{own}/[\![\texttt{T}]\!].\mathrm{shr}$ may depend on the thread identifier

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