RustBelt: Securing the Foundations of the Rust Programming Language

Ralf Jung
Jacques-Henri Jourdan
Robbert Krebbers
Derek Dreyer

December 18th, 2017
Séminaire GALLIUM

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* ≈ 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* ≈ Type classes + associated types

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

The key challenge

Superficially: **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: **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

λ_{Rust} : a small idealized Rust

A semantic model for λ_{Rust}

Conclusion

```
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

```
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
  - ☐ Remark: If `x : &'a (&'b T)` , then `&'b T` is valid during `'a`

    $\Longrightarrow$ Rust checks `'a` $\sqsubseteq$ `'b` (“**Outlives relation**”)

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

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())
```

Cell

Example of interior mutability

```
fn incr_cell<'a>(c : &'a Cell<i32>) {
    let x = c.get();
    // Can mutate through a shared borrow only
    c.set(x + 1)
}

fn main() {
    let c = Cell::new(0);

    incr_cell(&c);
}
```

Cell

Example of interior mutability

```
fn incr_cell<'a>(c : &'a Cell<i32>) {
    let x = c.get();
    // Can mutate through a shared borrow only
    c.set(x + 1)
}

fn main() {
    let c = Cell::new(0);

    // Data race => forbidden
    join(|| incr_cell(&c), || incr_cell(&c));
}
```

Cell

Example of interior mutability

```
fn inc
    le
    //
    c.
}

fn mai
    le

    // Data race => forbidden
    join(|| incr_cell(&c), || incr_cell(&c));
}
```

```
[...]
‘std::cell::Cell<i32>‘ cannot be shared
between threads safely
[...]
The trait ‘std::marker::Sync‘ is not
implemented for ‘std::cell::Cell<i32>‘
[...]
```

Cell

Example of interior mutability

`T` **shared accross threads** $\Longrightarrow$ `T : Sync`

- Automatic for basic types (structs, `i32`...)
- Ex: `Cell<T>` **not** `Sync`

`T` **moved accross threads** $\Longrightarrow$ `T : Send`

- Automatic for basic types (structs, `i32`...)
- Ex: `Rc<T>` (ref. counted smart pointer) **not** `Send`
- `T : Sync` $\Longleftrightarrow$ `&'a T : Send`

```
fn inc
    le
    //
    c.
}

fn mai
    le

    //
    join(|| incr_cell(&c), || incr_cell(&c));
}
```

Protocols

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

- `&'a i32`
  - ⟹ **Read-only**
  - Safety: trivial
- `&'a Mutex<i32>`
  - ⟹ Read-write **by taking the lock**
  - Safety: ensured by proper synchronization
- `&'a Cell<i32>`
  - ⟹ Read-Write **via** `get()` **and** `set(...)`
  - Safety: single threaded (no `Sync`), no inner `&'a mut i32`

Introduction

Overview of Rust

λ_{Rust} : a small idealized Rust

A semantic model for λ_{Rust}

Conclusion

λ_{Rust}: Main goal

Formalize Rust's type system without its main complications.

- Close to MIR (`rustc` intermediate language)
- Omitted: traits, polymorphism, panics, weak memory*, IO, destructors...

*Work in progress

λ_{Rust}: Syntax and operational semantics

Lambda-calculus with extensions:

- Integers and Boolean
- Heap (manual (de)allocation) and pointers (block ID + offset)
- Concurrency (`fork`, atomic memory accesses, `CAS`)

Operational semantics:

- Small-step style
- Stuck:
  - Type errors
  - Incorrect memory accesses (incl. double free, ...)
  - Data races

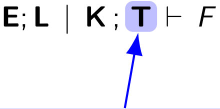
λ_{Rust}: Type system

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

Programs written in **continuation-passing style**

- MIR programs: control flow graphs
- $\Longrightarrow$ No output type

λ_{Rust}: Type system



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

Type context

Each variable: integer or pointer.
Associated with **type + ownership**. Examples:

- $x \lhd \textbf{int}$
- $p \lhd \textbf{box}\,(\textbf{int} \times \textbf{int} + ())$
- $p \lhd \&^{\alpha}_{\textbf{mut}}\,(\textbf{int} \times \textbf{int} + ())\ ;\ p \lhd^{\dagger\alpha} \textbf{box}\,(\textbf{int} \times \textbf{int} + ())$
- $p \lhd \&^{\alpha}_{\textbf{shr}}\,\textbf{int}$

λ_{Rust}: Type system

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

Lifetime contexts

Contain information for:

- Lifetime currently alive: $\mathbf{E};\mathbf{L} \vdash \alpha$ alive
- Lifetime inclusions: $\mathbf{E};\mathbf{L} \vdash \alpha \sqsubseteq \beta$
  - We know *in advance* that α is shorter than β
- Allowing to end a lifetime
  - Check: no promise to end another lifetime earlier.

λ_{Rust}: Type system

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

Continuation context

- Calling a continuation ⇔ Jumping to another block in CFG
- May require constraints on typing context and lifetime context
- Example:

$$k \lhd \mathbf{cont}(\text{“}\alpha \text{ alive”} \;;\; r \lhd \mathbf{box}\,(() + \&^{\alpha}_{\mathbf{mut}}\,\mathbf{int}), x \lhd \mathbf{box}\,\mathbf{int})$$

Introduction

Overview of Rust

λ_{Rust} : a small idealized Rust

A semantic model for λ_{Rust}

Conclusion

Challenge #1: the right approach

- One can write **unsafe code** in a **safely encapsulated** manner
  - New types that are safe for **new reasons**
- Our goal: prove that these library are safe
  - $\Longrightarrow$ **Syntactic approaches will not work**

We build a **logical relation** for Rust's type system

Choosing the right logic

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!

Proof method

- We define a **logical relation in Iris**:

$$\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \models \mathtt{F} \quad \triangleq \quad \{[\![\mathbf{E}]\!] * [\![\mathbf{L}]\!] * [\![\mathbf{K}]\!] * [\![\mathbf{T}]\!]\}\ \mathtt{F}\ \{\mathsf{True}\}$$

- The relation is **compatible** with type-checking rules:

$$\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash \mathtt{F} \implies \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \models \mathtt{F}$$

- The relation is **adequate**:

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

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

Logical interpretation of types

Example 1: **int**

What values are integers?

$$[\![\textbf{int}]\!].\mathrm{own}(v) \quad \triangleq \quad \exists n \in \mathbb{Z}.\, v = n$$

Logical interpretation of types

Example 2: **box** τ

We must state **ownership** of memory and inner type:

$$[\![\mathbf{box}\,\tau]\!].\mathrm{own}(v) \quad \triangleq \quad \exists \ell \in \mathcal{L}.\ \ v = \ell \ * \ \exists v'.\ \ell \mapsto v' \ * \ [\![\tau]\!].\mathrm{own}(v')$$

Logical interpretation of types

Example 3: $\tau_1 \times \tau_2$

Actually, types refer to **list of values**

$$\begin{aligned}\llbracket \tau_1 \times \tau_2 \rrbracket.\mathsf{own}(\bar{v}) \quad &\triangleq \\ &\exists \bar{v}_1 \bar{v}_2.\quad \bar{v} = \bar{v}_1 +\!\!+ \bar{v}_2 \;*\; \llbracket \tau_1 \rrbracket.\mathsf{own}(\bar{v}_1) \;*\; \llbracket \tau_2 \rrbracket.\mathsf{own}(\bar{v}_2)\end{aligned}$$

This also ensures **no aliasing** between $\bar{v}_1$ and $\bar{v}_2$

Challenge #2: interpreting borrows

Mutable borrows $\&^{\alpha}_{\mathbf{mut}}\,\tau$

Pointer with **temporary ownership**

Recall:

$$\begin{aligned}\llbracket \mathbf{box}\,\tau \rrbracket.\mathrm{own}(\bar{v}) \quad &\triangleq \\ &\exists \ell \in \mathcal{L}.\quad \bar{v} = [\ell] \;*\; \exists \bar{w}.\ \ell \mapsto \bar{w} \;*\; \llbracket \tau \rrbracket.\mathrm{own}(\bar{w})\end{aligned}$$

Challenge #2: interpreting borrows

Mutable borrows $\&^{\alpha}_{\mathbf{mut}}\,\tau$

Pointer with **temporary ownership**

For mutable borrows, we use the full borrow assertion $\&^{\kappa}_{\mathbf{full}}\,\_$:

$$\begin{aligned}\llbracket \&^{\alpha}_{\mathbf{mut}}\,\tau \rrbracket.\mathsf{own}(\bar{v}) \quad &\triangleq \\ &\exists \ell \in \mathcal{L}.\quad \bar{v} = [\ell] \;*\; \&^{\llbracket \alpha \rrbracket}_{\mathbf{full}}(\exists \bar{w}.\ \ell \mapsto \bar{w} \;*\; \llbracket \tau \rrbracket.\mathsf{own}(\bar{w}))\end{aligned}$$

Challenge #2: interpreting borrows

Mutable borrows $\&^{\alpha}_{\textbf{mut}}\ \tau$

Pointer

$\&^{\kappa}_{\textbf{full}}\ P$ is a separation logic assertion

- Intuition: ownership of P only if κ is *alive*
- Part of the **lifetime logic** (logical library defined in Iris)

Challenge #2: interpreting borrows

Shared borrows $\&^{\alpha}_{\textsf{shr}}\,\tau$

$$[\![\&^{\alpha}_{\textsf{shr}}\,\tau]\!].\mathsf{own}(\bar{v}) \quad \triangleq \quad \exists \ell \in \mathcal{L}.\;\; \bar{v} = [\ell] \;*\quad ???$$

Challenge #2: interpreting borrows

Shared borrows $\&^{\alpha}_{\textbf{shr}}\,\tau$

$$[\![\&^{\alpha}_{\textbf{shr}}\,\tau]\!].\mathrm{own}(\bar{v}) \quad \triangleq \quad \exists \ell \in \mathcal{L}. \quad \bar{v} = [\ell] \;*\quad ???$$

A type choses its sharing protocol.

Examples:

- Read-only for `&'a i32`,
- With a lock for `&'a Mutex<i32>`
- Thread-local + accessors for `&'a Cell<i32>`

Challenge #2: interpreting borrows

Shared borrows $\&^{\alpha}_{\mathsf{shr}}\,\tau$

$$[\![\&^{\alpha}_{\mathsf{shr}}\,\tau]\!].\mathsf{own}(\bar{v}) \quad \triangleq \quad \exists \ell \in \mathcal{L}.\;\; \bar{v} = [\ell] \; * \quad ???$$

A type choses its sharing protocol.

In our model, types have **two interpretations**

- *Ownership predicate* $[\![\tau]\!].\mathsf{own}(\bar{v})$: when exclusively owned
- *Sharing predicate* $[\![\tau]\!].\mathsf{shr}(\kappa, \ell)$: when shared (under a shared borrow)

Challenge #2: interpreting borrows

Shared borrows $\&_{\mathsf{shr}}^{\alpha}\,\tau$

$$[\![\&_{\mathsf{shr}}^{\alpha}\,\tau]\!].\mathsf{own}(\bar{v}) \quad \triangleq \quad \exists \ell \in \mathcal{L}.\ \ \bar{v} = [\ell] \ * \ [\![\tau]\!].\mathsf{shr}([\![\alpha]\!], \ell)$$

A type choses its sharing protocol.

In our model, types have **two interpretations**

- *Ownership predicate* $[\![\tau]\!].\mathsf{own}(\bar{v})$: when exclusively owned
- *Sharing predicate* $[\![\tau]\!].\mathsf{shr}(\kappa, \ell)$: when shared (under a shared borrow)

Challenge #2: interpreting borrows

Shared b

$[\![\tau]\!]$.shr(κ, ℓ) has to be **persistent**.

- I.e., duplicable, does not contain resources
- So that $\&^{\alpha}_{\mathbf{shr}}\,\tau$ can be shared

Lifetime logic provides tools for defining $\&^{\alpha}_{\mathbf{shr}}\,\tau$

- $\&^{\kappa}_{\mathbf{frac}}\,\Phi$: access to only a fraction of Φ
  - Read-only protocol
- $\&^{\kappa}_{\mathbf{at}}\,P$: access to P only for atomic operations
  - Mutex...
- ...

In our

- *Own*
- *Shar*
  borrow)

Challenge #3: thread safety

Reminder, in Rust:

- `T : Send` $\iff$ `T` can be **moved to another thread**
- `T : Sync` $\iff$ `T` can be **shared with another thread**

How to model this?

Challenge #3: thread safety

Reminder, in Rust:

- `T : Send` $\iff$ `T` can be **moved to another thread**
- `T : Sync` $\iff$ `T` can be **shared with another thread**

$[\![\tau]\!]$.own and $[\![\tau]\!]$.shr **can depend on thread ID**:

$$[\![\tau]\!].\mathsf{own}(t, \bar{v}) = \dots \qquad [\![\tau]\!].\mathsf{shr}(\kappa, t, \ell) = \dots$$

- τ is `Send` : $[\![\tau]\!].\mathsf{own}(t, \bar{v})$ independent of t
- τ is `Sync` : $[\![\tau]\!].\mathsf{shr}(\kappa, t, \ell)$ independent of t
- In the lifetime logic: $\&_{\mathbf{na}}^{\kappa/t}\, P$
  - Shared P while κ is alive, limited to thread t

Introduction

Overview of Rust

λ_{Rust} : a small idealized Rust

A semantic model for λ_{Rust}

Conclusion

Conclusion

And also. . .

- **Lifetime logic**: a library in Iris
  - Logical model of lifetimes and borrows
  - Main idea: **split ownership over time** (instead of "over space")
- 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

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

Borrows and inheritance

Usually: we split ownership with respect to space

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

$$\triangleright P \Rrightarrow \&^{\alpha}_{\textbf{full}}\, P * \left([\dagger\alpha] \Rrightarrow \triangleright P\right)$$

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

Borrows and inheritance

Usually: we split ownership with respect to space

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

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

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**:

$$\triangleright P \quad \Rrightarrow \quad \&^{\alpha}_{\textsf{full}}\, P \ * \ \big([\dagger\alpha] \Rrightarrow \triangleright P\big)$$

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:

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

- 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**:

$$\&^{\alpha}_{\mathbf{full}}(P * Q) \Longleftrightarrow \&^{\alpha}_{\mathbf{full}}\, P * \&^{\alpha}_{\mathbf{full}}\, Q$$

```
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:

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

Sharing protocols

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

Depends on `T`. Common idea:

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

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

  - Technical problems with step-indexing

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