Modeling Rust's type system using the Iris separation logic

Jacques-Henri Jourdan

Introduction

Programming languages are a **trade-off** between:

- Safety
  - OCaml, Java, Haskell, ...
  - "High-level" applications
- Control
  - C, C++, Assembly, ...
  - "Low-level" applications

Rust

Mozilla's replacement for C/C++

Systems programming language **focusing on safety**.

- Control over memory allocation & layout
- **Sound type system** with guarantees:
  - No **memory errors**
  - No undesired **data races**
  - No undesired **mutation of shared data**
  - Polymorphism/generics via type *traits* = type classes + associated types
  - First-class functions

Rust

Mozilla's replacement for C/C++

Systems programming language **focusing on safety**.

- Control over memory allocation & layout
- **Sound?** **type system** with guarantees:
  - No **memory errors**
  - No undesired **data races**
  - No undesired **mutation of shared data**
  - Polymorphism/generics via type *traits* = type classes + associated types
  - First-class functions

Our project: formally verify (a core of) Rust type system. **In Coq.**

Rust

Mozilla's replacement for C/C++

Systems programming language **focusing on safety**.

- Control over memory allocation & layout
- **Sound?** **type system** with guarantees:
  - No **memory errors**
  - No undesired **data races**
  - No undesired **mutation of shared data**
  - Polymorphism/generics via type *traits* = type classes + associated types
  - First-class functions

Our project: formally verify (a core of) Rust type system. **In Coq.**

The Iris concurrent separation logic

Motivation

Rust type system:

- **Ownership**, complex **sharing protocols**, in a **concurrent setting**

The Iris concurrent separation logic

Motivation

Rust type system:

- **Ownership**, complex **sharing protocols**, in a **concurrent setting**

The **Iris program logic** supports:

- Separation
- Concurrency
- Higher order
- Complex protocols via virtual ownership
  - Monoids & invariants

Introduction

Rust's ownership system

Overbiew of Iris

Modeling Rust in Iris

Conclusions

`Box<T>`

- Pointer to `T`, allocated on the heap
- Rust **prevents aliasing** of `Box<T>` values
  - To avoid: mutation of shared data & use after free
  - Ownership discipline:
    - `Box<T>` cannot be *copied*, can be *moved*

Need for a sharing mechanism

```
fn f(mut x : Box<i32>){ *x = 0 }

let mut x = Box::new(5); // We allocate the box
f(x);                    // We transfer (move) [x] to [f]
println!("{}", *x)       // Error: we do no longer own [x].
                         // [f] could have freed or leaked it!
```

Need for a sharing mechanism

```
fn f(mut x : Box<i32>){ *x = 0 }

let mut x = Box::new(5); // We allocate the box
f(x);                    // We transfer (move) [x] to [f]
println!("{}", *x)       // Error: we do no longer own [x].
                         // [f] could have freed or leaked it!
```

We need to:

- **Temporarily** transfer pointers
- Alias variables, in a **Controlled** way

Borrowing

```
fn f<'a>(x : &'a mut i32){ *x = 0 }
```

- `f` receives `x` as a **borrow**ed reference.
- `x` only used when the **lifetime** `'a` is *ongoing*
- Implicit: `'a` ends after `f`

Borrowing

```
fn f<'a>(x : &'a mut i32){ *x = 0 }
```

- `f` receives `x` as a **borrow**ed reference.
- `x` only used when the **lifetime** `'a` is *ongoing*
- Implicit: `'a` ends after `f`

```
let mut x = Box::new(5);
f(&mut *x);  // We pass to [f] only a borrow of [*x]
// ['a] ends here. We get back the full ownership of [x]
println!("{}", *x)
```

Shared borrowing

- `&i32` instead of `&mut i32`
- `Copy`able ⇒ **shared freely**
- What we lose: mutability
  - Invariant: no mutation of aliased data
  - **Reversible**: when the lifetime ends

Interior mutability

Sometimes we **need** aliased mutable state.

- For synchronization
- For memory managment (ref. counting)

Stdlib provides **interior mutability**

- For **some types**: mutation of shared data
- Written in `unsafe` blocks
- Library developers *believe* they are **safely encapsulated**

Interior mutability

Examples

`Cell<T>`

- **Mutable & shareable** memory cell
- `T` has to be `Copy`able

Memory managment: `Rc<T>`, `Arc<T>`

- (Atomically) reference counted pointer to `T`

Synchronization: `RwLock<T>`

- Protecting data of type `T`
- **Writer** lock $\Rightarrow$ `&mut T` $\Rightarrow$ **Can mutate**
- **Reader** lock $\Rightarrow$ `&T` $\Rightarrow$ **No mutation**

Interior mutability

Proof

Interior mutability is **very idiomatic**

- **Most Rust programs** use interior mutability

Our proof of soundness must **model these libraries**

- Program not fully well-typed
  - No syntactic proof
- Problem: complex sharing protocols!
- Need for a **powerful logic**

Introduction

Rust's ownership system

Overview of Iris

Modeling Rust in Iris

Conclusions

Overview of Iris

A concurrent separation logic built on **simple fundations**.

Iris' slogan:

"Monoids
and invariants
are all you need"

Overview of Iris

A concurrent separation logic built on **simple fundations**.

Iris' slogan:

"User-defined ghost resources

and invariants

are all you need"

Overview of Iris

A concurrent separation logic built on **simple fundations**.

Iris' slogan:

"User-defined ghost resources
and invariants
are all you need"

What are user-defined resources?

Iris: a **logic of ressources**

What are resources?

Examples:

- **Heap fragments** $(x \mapsto_q v)$, as in usual separation logic
- Exclusive **tokens**, abilities (right to do something)
- **Agreement**
  - If I own ag(1), nobody can own ag(0).

Can be chosen **by the user**!

Iris: a **logic of ressources**

What are resources?

Examples:

- **Heap fragments** $(x \mapsto_q v)$, as in usual separation logic
- Exclusive **tokens**, abilities (right to do something)
- **Agreement**
  - If I own ag(1), nobody can own ag(0).

Can be chosen **by the user**!

Iris: a **logic of ressources**

What are resources?

Examples:

- **Heap fragments** $(x \mapsto_q v)$, as in usual separation logic
- Exclusive **tokens**, abilities (right to do something)
- **Agreement**
  - If I own ag(1), nobody can own ag(0).

Can be chosen **by the user**!

Iris: a **logic of ressources**

What are resources?

Examples:

- **Heap fragments** $(x \mapsto_q v)$, as in usual separation logic
- Exclusive **tokens**, abilities (right to do something)
- **Agreement**
  - If I own ag(1), nobody can own ag(0).

Can be chosen **by the user**!

Iris: a **logic of ressources**

What are resources?

Examples:

- **Heap fragments** $(x \mapsto_q v)$, as in usual separation logic
- Exclusive **tokens**, abilities (right to do something)
- **Agreement**
  - If I own ag(1), nobody can own ag(0).

Can be chosen **by the user**!

Iris: a **logic of ressources**

What are resources?

An **algebraic structure** with

- A associative, commutative **composition operator**
  - What resource is owned by several parties owning a fragment?
  - Proof rule:

$$\boxed{a \cdot b} \Leftrightarrow \boxed{a} * \boxed{b}$$

- A set of **valid elements**
  - Not all compositions are valid
    - A token is exclusive, $\mathrm{ag}(\cdot)$ must agree, only one owns a heap fragment...
  - Proof rule:

$$\boxed{a} \Rightarrow \mathcal{V}(a)$$

- Can be **higher-order** (step-indexing)

Iris: a **logic of ressources**

What are resources?

An **algebraic structure** with

- A associative, commutative **composition operator**
  - What resource is owned by several parties owning a fragment?
  - Proof rule:

$$\boxed{a \cdot b} \Leftrightarrow \boxed{a} * \boxed{b}$$

- A set of **valid elements**
  - Not all compositions are valid
    - A token is exclusive, $\mathrm{ag}(\cdot)$ must agree, only one owns a heap fragment...
  - Proof rule:

$$\boxed{a} \Rightarrow \mathcal{V}(a)$$

- Can be **higher-order** (step-indexing)

Iris: a **logic of ressources**

What are resources?

An **algebraic structure** with

- A associative, commutative **composition operator**
  - What resource is owned by several parties owning a fragment?
  - Proof rule:

$$\boxed{a \cdot b} \Leftrightarrow \boxed{a} * \boxed{b}$$

- A set of **valid elements**
  - Not all compositions are valid
    - A token is exclusive, $\mathrm{ag}(\cdot)$ must agree, only one owns a heap fragment...
  - Proof rule:

$$\boxed{a} \Rightarrow \mathcal{V}(a)$$

- Can be **higher-order** (step-indexing)

Iris: a **logic of ressources**

What are resources?

Resources can be **updated**

- The frame a_{f} must **remain compatible**:

$$a \rightsquigarrow b \quad \triangleq \quad \forall a_{\mathrm{f}}.\, \mathcal{V}(a \cdot a_{\mathrm{f}}) \Rightarrow \mathcal{V}(b \cdot a_{\mathrm{f}}) \qquad \frac{a \rightsquigarrow b}{\boxed{a} \Rrightarrow \boxed{b}}$$

- Non-deterministic version:

$$a \rightsquigarrow B \quad \triangleq \quad \forall a_{\mathrm{f}}.\, \mathcal{V}(a \cdot a_{\mathrm{f}}) \Rightarrow \exists b \in B.\, \mathcal{V}(b \cdot a_{\mathrm{f}}) \qquad \frac{a \rightsquigarrow B}{\boxed{a} \Rrightarrow \exists b \in B.\, \boxed{b}}$$

Iris: a **logic of ressources**

What are resources?

Resources can be **updated**

- The frame a_{f} must **remain compatible**:

$$a \rightsquigarrow b \quad \triangleq \quad \forall a_{\mathrm{f}}.\, \mathcal{V}(a \cdot a_{\mathrm{f}}) \Rightarrow \mathcal{V}(b \cdot a_{\mathrm{f}}) \qquad \frac{a \rightsquigarrow b}{\boxed{a} \Rrightarrow \boxed{b}}$$

- Non-deterministic version:

$$a \rightsquigarrow B \quad \triangleq \quad \forall a_{\mathrm{f}}.\, \mathcal{V}(a \cdot a_{\mathrm{f}}) \Rightarrow \exists b \in B.\, \mathcal{V}(b \cdot a_{\mathrm{f}})$$

$$\frac{a \rightsquigarrow B}{\boxed{a} \Rrightarrow \exists b \in B.\, \boxed{b}}$$

Iris: a **logic for (concurrent) programs**

For any programming language, define *in the logic*:

$$\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$$

Weakest precondition of e for postcondition Φ:

- Physical state transitions (from operational semantics):
  - Modeled by updates $\Rrightarrow$
  - Physical state: tied to a resource algebra "via agreement"
- Maintains **invariants**:

$$\frac{\triangleright P \vdash \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\, \{v.\, \triangleright P * \Phi(v)\} \qquad \mathsf{atomic(e)} \qquad \iota \in \mathcal{E}}{\boxed{P}^{\iota} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

Iris: a **logic for (concurrent) programs**

For any programming language, define *in the logic*:

$$\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$$

Weakest precondition of e for postcondition Φ:

- Physical state transitions (from operational semantics):
  - Modeled by updates $\Rrightarrow$
  - Physical state: tied to a resource algebra "via agreement"
- Maintains **invariants**:

$$\frac{\triangleright P \vdash \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\, \{v.\ \triangleright P * \Phi(v)\} \qquad \text{atomic(e)} \qquad \iota \in \mathcal{E}}{\boxed{P}^{\iota} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

Introduction

Rust's ownership system

Overview of Iris

Modeling Rust in Iris

Conclusions

Modeling lifetimes

Reminder:

```
fn f<'a>(x : &'a mut i32){ *x = 0 }

let mut x = Box::new(5);
f(&mut *x);
println!("{}", *x)
```

- What is the resource modeling `&'a mut i32`?
- It cannot be "$x \mapsto$ `i32`":
  - `f` could throw it away, or free the block

Lifetime logic

Usually: we split ownership with respect to space

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

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

Lifetime logic

Usually: we split ownership with respect to space

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

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

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

Lifetime logic

Usually: we split ownership with respect to space

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

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

A *borrowed* part, that can be used when κ is ongoing

- Must be returned unchanged when κ ends
- Modeled by a *virtual* resource

Lifetime logic

Usually: we split ownership with respect to space

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

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

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

Lifetime tokens

- Birth & death of a lifetime

$$\mathsf{True} \Rrightarrow \exists \kappa.\, [\kappa]_1 * \left([\kappa]_1 \Rrightarrow [\dagger \kappa] \right)$$

  We get both the **lifetime token** $[\kappa]_1$ and a "weapon" to kill it.

- To access a borrow $\&^\kappa P$, we give a *fraction* $[\kappa]_q$ in **deposit**.
  - κ cannot be ended before giving back P

Other features

- Lifetimes can be **ordered by borrowing tokens**:

$$\&^{\kappa}\,[\kappa']_q \quad \Rrightarrow \quad \kappa \sqsubseteq \kappa'$$

- Borrows can be **unnested**:

$$\&^{\kappa'}(\&^{\kappa}\,P) \quad \Rrightarrow \quad \&^{\kappa \sqcap \kappa'}\,P$$

- Borrows are **compatible with some logical combinators**:

$$\&^{\kappa}(P * Q) \quad \Lrightarrow \quad \&^{\kappa}\,P * \&^{\kappa}\,Q$$
$$(\&^{\kappa}\,\exists x.\,P) \quad \Rrightarrow \quad \exists x.\,\&^{\kappa}\,P$$

Introduction

Rust's ownership system

Overview of Iris

Modeling Rust in Iris

Conclusions

Conclusions

- Rust is an interesting programming language
- **Groundbreaking type system**
  - **Several soundess bugs** already found
  - Not easy to prove correct
  - $\Rightarrow$ Interesting
- Our project:
  - Formally verify it in **Coq**...
  - ... using **Iris** separation logic ...
  - ... by building a **lifetime logic**
    - On-paper proof, ongoing formalization
- Related question: **weak memory in Rust**?

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

Questions?