

Formalizing Rust's Type System

Jacques-Henri Jourdan

February 14th, 2024

Abstract from last weeks

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

During the last three weeks, we had an introduction to Rust programming

- Type system enforces: “mutation XOR aliasing”,
- Traits: an abstraction mechanism, sometimes at zero-cost.
- Unsafe blocks/functions: workaround strong static type-checking constraints,
- Encapsulation: clients can safely use libraries written with unsafe code.
- Interior mutability (the ability to mutate through shared borrows): a typical example of well-encapsulated unsafe code.
- Rust is a good fit for multithreading.
 - Protects against data races (with `Send` and `Sync`).
 - Provides abstractions for multithreading.

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types
Lifetime logic

Supporting unsafe
code

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

I keep telling you Rust is type-safe.
How can I be so sure?

I.e., is this something one can prove formally?

1 Introduction

Introduction

Syntactic type system

λ_{Rust}
Type system

Building the logical relation

Setup
Interpreting simple
types
Lifetime logic

Supporting unsafe code

2 Syntactic type system

- λ_{Rust}
- Type system

3 Building the logical relation

- Setup
- Interpreting simple types
- Lifetime logic

4 Supporting unsafe code

The problem we are trying to solve

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

“Well-typed Rust programs do not go wrong.”

Really?

Introduction

Syntactic type
system

λ_{Rust}

Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

The problem we are trying to solve

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

“Well-typed Rust programs **not using unsafe** do not go wrong.”

Is that all?

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

The problem we are trying to solve

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

"Well-typed Rust programs **not using** `unsafe` do not go wrong."

A vast majority of real-life Rust programs (directly or indirectly) use unsafe code.
Example: `Vec`, interior mutability (e.g., `Cell`), low-level optimizations, ...

We want an **extensible** theorem to prove those programs safe too.

- Safe pieces of code are safe thanks to **syntactic typing rules**.
- Unsafe pieces are safe thanks to a **specialized proof**.

Both kinds of proofs will be linked together thanks to a **logical relation**.

Logical relations for type safety

The general approach

A type system is defined from:

- One (or several) **typing judgment(s)** $\dots \vdash \dots$
- Syntactic **typing rules**.

Ex. for lambda-calculus:
$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

The logical relation approach to type soundness (i.e., semantic type soundness):

- 1 Define a **semantic typing judgment** $\dots \models \dots$ from the operational semantics.
- 2 Prove the **fundamental theorem**: " $\dots \vdash \dots \Rightarrow \dots \models \dots$ "
- 3 Prove **adequacy** for the logical relation: " $\dots \models \dots \Rightarrow \text{safety}$ ".
 - "Semantically well-typed programs do not go wrong."
 - Usually easy from the definition of $\dots \models \dots$.

Why is this approach more extensible than subject reduction+progress?

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

Extending semantic type safety

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}

Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

The fundamental theorem: by induction over the typing tree using semantic typing rules:

$$\frac{\begin{array}{c} \dots \vdash \dots \\ \dots \vdash \dots \\ \vdots \qquad \dots \vdash \dots \\ \hline \dots \vdash \dots \end{array} \quad \dots \vdash \dots}{\dots \vdash \dots} \Rightarrow \frac{\begin{array}{c} \dots \models \dots \\ \dots \models \dots \\ \vdots \qquad \dots \models \dots \\ \hline \dots \models \dots \end{array} \quad \dots \models \dots}{\dots \models \dots}$$

Extending semantic type safety

Introduction

Syntactic type
system

λ_{Rust}

Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

The fundamental theorem: by induction over the typing tree using semantic typing rules:

$$\frac{\begin{array}{c} \dots \vdash \dots \\ \dots \vdash \dots \\ \vdots \qquad \dots \vdash \dots \\ \hline \dots \vdash \dots \end{array} \quad \dots \vdash \dots \quad \Rightarrow \quad \begin{array}{c} \dots \models \dots \\ \dots \models \dots \\ \vdots \qquad \dots \models \dots \\ \hline \dots \models \dots \quad \dots \models \dots \end{array}}{\dots \vdash \dots}$$

To prove $\dots \models \dots$ for the whole program, we don't need $\dots \vdash \dots$ everywhere.
We can fill some "holes" using custom proofs of $\dots \models \dots$.

Extending semantic type safety

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

`Cell` is not syntactically well-typed in safe Rust: “ $\dots \not\vdash \text{Cell}::\text{get} : \dots$ ”.

But we can prove “ $\dots \models \text{Cell}::\text{get} : \dots$ ”.

Combining with an otherwise syntactically well-typed program:

$$\frac{\begin{array}{c} \dots \vdash \dots \\ \hline \dots \vdash \dots \\ \vdots \\ \dots \models \text{Cell}::\text{get} : \dots \\ \hline \dots \models \dots & \dots \vdash \dots \\ \hline \dots \models \dots \end{array}}{\dots \models \dots}$$

And we can conclude safety thanks to adequacy!

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

IntroductionSyntactic type
system λ_{Rust}
Type systemBuilding the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

What's left to be done...

- Define formally the language and its syntactic type system.
- Define the logical relation and prove adequacy+fundamental theorem.
- Extend the logical relation for types like `Cell`...

That's a lot of work.

We will only see a **sketch here**.

The details are in a paper, and the accompanying Coq development:

RustBelt: Securing the foundations of the Rust programming language. In POPL '18.

To make sure nothing is wrong, this is **formalized with Coq**. Particularly **useful** to design and maintain the proof.

1 Introduction

Introduction

Syntactic type system

λ_{Rust}
Type system

Building the logical relation

Setup
Interpreting simple types
Lifetime logic

Supporting unsafe code

2 Syntactic type system

- λ_{Rust}
- Type system

3 Building the logical relation

- Setup
- Interpreting simple types
- Lifetime logic

4 Supporting unsafe code

The language: λ_{Rust}

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

Rust is **way too complex** to be formalized as-is.

Should we formalize MIR (recall: the language behind the borrow checker)?

It still has a lot of technicalities unrelated to the type system, refers to traits...

Instead, we formalize a **core language**.

It has most of the features of MIR relevant for type soundness, but **idealized** to make the theory simpler.

The language: λ_{Rust}

Paths, elementary values

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}

Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

$\text{Val} \ni v ::= \text{false} \mid \text{true} \mid z \mid \ell \mid \text{funrec } f(\bar{x}) \text{ ret } k := F$

$\text{Path} \ni p ::= x \mid p.n$

Elementary values that can be stored in local registers x, \dots

Complex values (e.g., `struct`) are stored in memory, and accessed through pointers.

Note: function values take a continuation in parameter (language in CFG, see later).

The language: λ_{Rust}

Paths, elementary values

$Val \ni v ::= \text{false} \mid \text{true} \mid z \mid \ell \mid \text{funrec } f(\bar{x}) \text{ ret } k := F$

$Path \ni p ::= x \mid p.n$

Pointers to fields of complex values can be created with **paths**.

$p.n$: pointer to field at offset n of **struct** pointed to by p .

Operationally: just a pointer offset (i.e., an addition).

$$\begin{aligned} Instr \ni I ::= & v \mid p \mid p_1 + p_2 \mid p_1 - p_2 \mid p_1 \leq p_2 \mid p_1 == p_2 \\ & \mid \text{new}(n) \mid \text{delete}(n, p) \mid {}^*p \mid p_1 := p_2 \mid p_1 :=_n {}^*p_2 \\ & \mid \dots \end{aligned}$$

Instructions are the elementary operations of λ_{Rust} .

They take **paths** as operands.

Introduction

Syntactic type
system λ_{Rust}
Type systemBuilding the logical
relationSetup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code
$$\begin{aligned} \text{Instr} \ni I ::= & v \mid p \mid p_1 + p_2 \mid p_1 - p_2 \mid p_1 \leq p_2 \mid p_1 == p_2 \\ & \mid \text{new}(n) \mid \text{delete}(n, p) \mid {}^*p \mid p_1 := p_2 \mid p_1 :=_n {}^*p_2 \\ & \mid \dots \end{aligned}$$

λ_{Rust} 's instructions include values, paths, arithmetic operations...

The language: λ_{Rust}

Instructions

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

$$\begin{aligned} \text{Instr} \ni I ::= & v \mid p \mid p_1 + p_2 \mid p_1 - p_2 \mid p_1 \leq p_2 \mid p_1 == p_2 \\ & \mid \text{new}(n) \mid \text{delete}(n, p) \mid {}^*p \mid p_1 := p_2 \mid p_1 :=_n {}^*p_2 \\ & \mid \dots \end{aligned}$$

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

... and instruction to allocate/free memory, and read/write it.

Read/write one word from/to a register: *p , $p_1 := p_2$.

- Operational semantics defined so that races \Rightarrow undefined behavior.

Copy a complex value (several words) from one memory location to another: $p_1 :=_n {}^*p_2$.

$$\begin{aligned} \text{FuncBody} \ni F ::= & \text{letcont } k(\bar{x}) := F_1 \text{ in } F_2 \mid \text{jump } k(\bar{x}) \\ & \mid \text{let } x = l \text{ in } F \mid \text{if } p \text{ then } F_1 \text{ else } F_2 \\ & \mid \text{call } f(\bar{x}) \text{ ret } k \\ & \mid \text{newlft;} F \mid \text{endlft;} F \\ & \mid \dots \end{aligned}$$

Function bodies combine instructions to build functions.

Code is written in CPS: a way to model code written as a CFG.

We can declare a new continuation (i.e., label in the CFG), and jump to it.

The language: λ_{Rust}

Function bodies

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

$$\begin{aligned} \mathit{FuncBody} \ni F ::= & \text{letcont } k(\bar{x}) := F_1 \text{ in } F_2 \mid \text{jump } k(\bar{x}) \\ & \mid \text{let } x = I \text{ in } F \mid \text{if } p \text{ then } F_1 \text{ else } F_2 \\ & \mid \text{call } f(\bar{x}) \text{ ret } k \\ & \mid \text{newlft}; F \mid \text{endlft}; F \\ & \mid \dots \end{aligned}$$

Basic function bodies:

- An instruction (+ binding the result to a variable).
- If-then-else.
- Calling a function (passing a continuation).

The language: λ_{Rust}

Function bodies

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

$$\begin{aligned} \mathit{FuncBody} \ni F ::= & \text{letcont } k(\bar{x}) := F_1 \text{ in } F_2 \mid \text{jump } k(\bar{x}) \\ & \mid \text{let } x = l \text{ in } F \mid \text{if } p \text{ then } F_1 \text{ else } F_2 \\ & \mid \text{call } f(\bar{x}) \text{ ret } k \\ & \mid \text{newlft;} F \mid \text{endlft;} F \\ & \mid \dots \end{aligned}$$

Ghost instructions for creating/ending a lifetime.

Instr $\ni I ::= \dots$
 $| p : \overset{\text{inj } i}{\equiv} () | p_1 : \overset{\text{inj } i}{\equiv} p_2 | p_1 : \overset{\text{inj } i}{\equiv}_n * p_2$
 $| \dots$

FuncBody $\ni F ::= \dots$
 $| \text{case } *p \text{ of } \bar{F}$

λ_{Rust} has a notion of sum types, stored in memory, with a tag in the first word.

There are special instructions to write such values together with the tag.

And a case statement for “pattern matching” the in-memory tag.

Types

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}

Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

$Type \exists \tau ::= \text{bool} \mid \text{int} \mid \text{own } \tau \mid \&_{\text{mut}}^{\kappa} \tau \mid \&_{\text{shr}}^{\kappa} \tau \mid \not\in n$
 $\mid \Pi \bar{\tau} \mid \Sigma \bar{\tau} \mid \forall \bar{\alpha}. \text{fn}(F : E; \bar{\tau}) \rightarrow \tau \mid T \mid \mu T. \tau$
 $\mid \dots$

Types for integer and Booleans.

Types

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}

Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

$Type \exists \tau ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{own} \tau \mid \&_{\mathbf{mut}}^{\kappa} \tau \mid \&_{\mathbf{shr}}^{\kappa} \tau \mid \not{\in} n$
 $\mid \Pi \bar{\tau} \mid \Sigma \bar{\tau} \mid \forall \bar{\alpha}. \mathbf{fn}(F : E; \bar{\tau}) \rightarrow \tau \mid T \mid \mu T. \tau$
 $\mid \dots$

Types for pointers: **Box** is written **own** τ .

Types

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}

Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

$$\begin{aligned} \text{Type } \exists \tau ::= & \text{bool} \mid \text{int} \mid \text{own } \tau \mid \&_{\text{mut}}^{\kappa} \tau \mid \&_{\text{shr}}^{\kappa} \tau \mid \notin n \\ & \mid \Pi \bar{\tau} \mid \Sigma \bar{\tau} \mid \forall \bar{\alpha}. \text{fn}(F : E; \bar{\tau}) \rightarrow \tau \mid T \mid \mu T. \tau \\ & \mid \dots \end{aligned}$$

Type of “uninitialized memory”.

- When memory is just allocated, or when non-Copy values are moved.
- Used to replace the “initializedness” analysis of the borrow checker.

Types

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}

Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

$$\begin{aligned} \text{Type } \exists \tau ::= & \text{bool} \mid \text{int} \mid \text{own } \tau \mid \&_{\text{mut}}^{\kappa} \tau \mid \&_{\text{shr}}^{\kappa} \tau \mid \not{\downarrow} n \\ & \mid \Pi \bar{\tau} \mid \Sigma \bar{\tau} \mid \forall \bar{\alpha}. \text{fn}(\bar{F} : E; \bar{\tau}) \rightarrow \tau \mid T \mid \mu T. \tau \\ & \mid \dots \end{aligned}$$

Complex types: sum, products.

Used to model `struct`, `enum` and tuples.

Types

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}

Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

Type $\exists \tau ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{own} \tau \mid \&_{\mathbf{mut}}^{\kappa} \tau \mid \&_{\mathbf{shr}}^{\kappa} \tau \mid \not\vdash n$
 $\mid \Pi \bar{\tau} \mid \Sigma \bar{\tau} \mid \forall \bar{\alpha}. \mathbf{fn}(F : E; \bar{\tau}) \rightarrow \tau \mid T \mid \mu T. \tau$
 $\mid \dots$

Type of function pointers, guarded equirecursive types...

Types

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}

Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

Type $\exists \tau ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{own} \tau \mid \&_{\mathbf{mut}}^{\kappa} \tau \mid \&_{\mathbf{shr}}^{\kappa} \tau \mid \not\in n$
 $\mid \Pi \bar{\tau} \mid \Sigma \bar{\tau} \mid \forall \bar{\alpha}. \mathbf{fn}(F : E; \bar{\tau}) \rightarrow \tau \mid T \mid \mu T. \tau$
 $\mid \dots$

Extensible for types providing safe abstraction over **unsafe**!

Typing judgments

There are two main typing judgments:

- One for instructions: $E; L \mid T_1 \vdash I \dashv x. T_2$
- One for function bodies: $E; L \mid K; T \vdash F$

They depend on four different kinds of contexts:

- T is the **typing context**. Elements:
 - $p \triangleleft \tau$: path p contains an elementary value of type τ .
 - $p \triangleleft^{\dagger\kappa} \tau$: same, but **frozen** until lifetime κ ends.
 - **Substructural**: elements cannot be duplicated (ownership tracking!).
- E and L are **lifetime contexts**. They contain:
 - information on **lifetime inclusion**, and
 - information on **local lifetimes**: which are they, and when they can be ended.
- K is the **continuation context**.
 - Describes the continuations we can jump to, and the required contexts.
 - Elements: $k \triangleleft \text{cont}(L; \bar{x}. T)$.
“We can jump to k , passing parameters \bar{x} , if the contexts contain L and T .”

Typing judgments

There are two main typing judgments:

- One for instructions: $E; L \mid T_1 \vdash I \dashv x. T_2$
- One for function bodies: $E; L \mid K; T \vdash F$

They depend on four different kinds of contexts:

- T is the **typing context**. Elements:
 - $p \triangleleft \tau$: path p contains an elementary value of type τ .
 - $p \triangleleft^{\dagger\kappa} \tau$: same, but **frozen** until lifetime κ ends.
 - **Substructural**: elements cannot be duplicated (ownership tracking!).
- E and L are **lifetime contexts**. They contain:
 - **information on lifetime inclusion**, and
 - **information on local lifetimes**: which are they, and when they can be ended.
- K is the **continuation context**.
 - Describes the continuations we can jump to, and the required contexts.
 - Elements: $k \triangleleft \text{cont}(L; \bar{x}. T)$.
“We can jump to k , passing parameters \bar{x} , if the contexts contain L and T .”

Typing judgments

There are two main typing judgments:

- One for instructions: $E; L \mid T_1 \vdash I \dashv x. T_2$
- One for function bodies: $E; L \mid K; T \vdash F$

They depend on four different kinds of contexts:

- T is the **typing context**. Elements:
 - $p \triangleleft \tau$: path p contains an elementary value of type τ .
 - $p \triangleleft^{\dagger\kappa} \tau$: same, but **frozen** until lifetime κ ends.
 - **Substructural**: elements cannot be duplicated (ownership tracking!).
- E and L are **lifetime contexts**. They contain:
 - information on **lifetime inclusion**, and
 - information on **local lifetimes**: which are they, and when they can be ended.
- K is the **continuation context**.
 - Describes the continuations we can jump to, and the required contexts.
 - Elements: $k \triangleleft \text{cont}(L; \bar{x}. T)$.
"We can jump to k , passing parameters \bar{x} , if the contexts contain L and T ."

Typing judgments

There are two main typing judgments:

- One for instructions: $E; L \mid T_1 \vdash I \dashv x. T_2$
- One for function bodies: $E; L \mid K; T \vdash F$

The typing context for instructions has two typing contexts: **an input typing context T_1** and **an output typing context T_2** .

The instruction I **consumes T_1** and **produces T_2** .

$x. T_2$ binds a special variable x : to give a type to the result of the instruction.

Function bodies never return (CPS). They only consume T .

Introduction

Syntactic type
system λ_{Rust}

Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

Typing rules

Typing instructions

Selected rules:

$$E; L \mid \emptyset \vdash z \dashv x. x \triangleleft \mathbf{int}$$

$$E; L \mid p_1 \triangleleft \mathbf{int}, p_2 \triangleleft \mathbf{int} \vdash p_1 \leq p_2 \dashv x. x \triangleleft \mathbf{bool}$$

$$E; L \mid \emptyset \vdash \mathbf{new}(n) \dashv x. x \triangleleft \mathbf{own} \not\in n$$

$$\frac{n = \text{size}(\tau)}{E; L \mid p \triangleleft \mathbf{own} \tau \vdash \mathbf{delete}(n, p) \dashv \emptyset}$$

$$\tau \text{ copy} \quad \text{size}(\tau) = 1$$

$$\frac{}{E; L \mid p \triangleleft \mathbf{own} \tau \vdash {}^* p \dashv x. p \triangleleft \mathbf{own} \tau, x \triangleleft \tau}$$

$$\text{size}(\tau) = 1$$

$$\frac{}{E; L \mid p \triangleleft \mathbf{own} \tau \vdash {}^* p \dashv x. p \triangleleft \mathbf{own} \not\in 1, x \triangleleft \tau}$$

$$E; L \vdash \kappa \text{ alive}$$

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

Typing rules

Typing function bodies

Selected rules:

$$\frac{E; L \mid T_1 \vdash I \dashv x. T_2 \quad E; L \mid K; T_2, T \vdash F}{E; L \mid K; T_1, T \vdash \text{let } x = I \text{ in } F}$$

$$\frac{k \lhd \mathbf{cont}(L; \bar{x}. T) \in K}{E; L \mid K; T[\bar{y}/\bar{x}] \vdash \text{jump } k(\bar{y})}$$

$$\frac{E; L \vdash T \xrightarrow{\text{ctx}} T' \quad E; L \mid K; T' \vdash F}{E; L \mid K; T \vdash F}$$

Helper judgment for transforming a typing context: $E; L \vdash T \xrightarrow{\text{ctx}} T'$.

Typing rules

Transforming typing environments

Some transformations can happen on environments before typing a piece of code:

$$\frac{\tau \text{ copy}}{E; L \vdash p \triangleleft \tau \xrightarrow{\text{ctx}} p \triangleleft \tau, p \triangleleft \tau}$$

$$E; L \vdash p \triangleleft \&_{\mu}^{\kappa} (\tau_1 \times \tau_2) \xrightarrow{\text{ctx}} p.0 \triangleleft \&_{\mu}^{\kappa} \tau_1, p.\text{size}(\tau_1) \triangleleft \&_{\mu}^{\kappa} \tau_2$$

$$\frac{E; L \vdash \kappa \text{ alive}}{E; L \vdash p \triangleleft \&_{\text{mut}}^{\kappa} \tau \xrightarrow{\text{ctx}} p \triangleleft \&_{\text{shr}}^{\kappa} \tau} \quad E; L \vdash p \triangleleft \text{own}_n \tau \xrightarrow{\text{ctx}} p \triangleleft \&_{\text{mut}}^{\kappa} \tau, p \triangleleft^{\dagger \kappa} \text{own}_n \tau$$

$$\frac{E; L \vdash \kappa' \sqsubseteq \kappa}{E; L \vdash p \triangleleft \&_{\text{mut}}^{\kappa} \tau \xrightarrow{\text{ctx}} p \triangleleft \&_{\text{mut}}^{\kappa'} \tau, p \triangleleft^{\dagger \kappa'} \&_{\text{mut}}^{\kappa} \tau}$$

Introduction

Syntactic type
system λ_{Rust}

Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

1 Introduction

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types
Lifetime logic

Supporting unsafe
code

2 Syntactic type system

- λ_{Rust}
- Type system

3 Building the logical relation

- Setup
- Interpreting simple types
- Lifetime logic

4 Supporting unsafe code

Recap

We have presented **syntactic typing judgments** $E; L \mid T_1 \vdash I \dashv x. T_2$ and $E; L \mid K; T \vdash F$.

We need to:

- Give them **semantic counterparts** $E; L \mid T_1 \models I \models x. T_2$ and $E; L \mid K; T \models F$
- Prove adequacy: if $\emptyset \mid \emptyset; \emptyset \mid \emptyset \models f \models x. x \triangleleft \mathbf{fn}() \rightarrow \Pi[]$, then f cannot execute to a stuck state.
- Prove the **semantic typing rules**: same as syntactic rules, but replacing \vdash with \models .
- Extend the proof to types defined using **unsafe**.

The main difficulty is to find the definitions of the semantic judgments.

Once defined, their properties follow “naturally” (somewhat). In this course, we will **not give the proofs** and focus on the definitions.

Let's define a model for semantic judgments.

To do that, we need a semantic interpretation of types: $[\![\tau]\!]$.

Problem: we need to speak about ownership. Any ideas how to models this?

Separation Logic

to the Rescue!



Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

Separation Logic to the Rescue!

Extension of Hoare logic (O'Hearn-Reynolds-..., 1999)

- For reasoning about pointer-manipulating programs

Major influence on many verification & analysis tools

- e.g. Infer, VeriFast, Viper, Bedrock, jStar, ...

Separation logic = Ownership logic

- Perfect fit for modeling Rust's ownership types!

Introduction

Syntactic type system

λ_{Rust}
Type system

Building the logical relation

Setup

Interpreting simple types

Lifetime logic

Supporting unsafe code

We use **Iris**, a modern and expressive separation logic.

It features:

- standard separation logic mechanisms,
- support for concurrency,
- built-in support for step-indexing (equirecursive types, higher-order store, ...),
- a powerful mechanism to create new mechanisms of ownership (borrows, ...).

Semantic interpretation of types

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

We define an ownership predicate $[\![\tau]\!].\text{own}(t, \bar{v})$ in separation logic for every type τ .

- It represents what it means in separation logic that \bar{v} is an object of type τ in thread t .
- It not only states what \bar{v} should look like, it also describes the resources behind τ .

Note: it depends on the thread t : the interpretation of some types depend on it!
Why is this useful?

Semantic interpretation of types

We define an **ownership predicate** $[\![\tau]\!].\text{own}(t, \bar{v})$ in separation logic for every type τ .

- It represents what it **means** in separation logic that \bar{v} is an object of type τ in thread t .
- It not only states what \bar{v} should look like, it also describes the **resources** behind τ .

Note: it depends on the thread t : the interpretation of some types depend on it!
Why is this useful?

Semantic interpretation of **Send**: types for which $[\![\tau]\!].\text{own}(t, \bar{v})$ do **not** depend on t .
 \Rightarrow The interpretation will still be valid in another thread.

Interpreting semantic environments

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

We directly lift semantic types to semantic environments using the separating conjunction:

$$\begin{aligned} \llbracket p \triangleleft \tau, T \rrbracket &:= \llbracket \tau \rrbracket.\text{own}(t, p) * \llbracket T \rrbracket \\ \llbracket \emptyset \rrbracket &:= \text{True} \end{aligned}$$

Meaning: every element in the typing context correspond to a distinct piece of ownership.

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

Semantic typing judgements

$$E; L \mid T_1 \models I \models x. T_2 :=$$
$$\forall t. \{[E] * [L] * [Na : t] * [T_1](t)\} \mid \{v. [L] * [Na : t] * [T_2]_{[x \leftarrow v]}(t)\}$$

Semantic typing judgment for function bodies is similar.

(I have omitted the definition of $[E]$ and $[L]$ – not important for understanding.)

We use a separation logic **Hoare triple**.

- All we say is that I behaves well when placed in an environment described by the contexts, and return an environment described by the output contexts.

$[Na : t]$ is a special resource owned by one thread only.

Non-thread-safe types need $[Na : t]$. Thus their interpretation depends on t .

Semantic typing judgements

$$E; L \mid T_1 \models I \models x. T_2 :=$$
$$\forall t. \{[E] * [L] * [Na : t] * [T_1](t)\} \mid \{v. [L] * [Na : t] * [T_2]_{[x \leftarrow v]}(t)\}$$

Semantic typing judgment for function bodies is similar.

Adequacy follows from the soundness of the separation logic.

- If we have $\emptyset \mid \emptyset; \emptyset \mid \emptyset \models f \models x. x \triangleleft \mathbf{fn}() \rightarrow \Pi[]$, then we have a Hoare triple with a trivial precondition. Thus f is safe.

Proving the semantic typing rules amounts to proving a Hoare triple.

- We can use the rules of the program logic, just like when proving programs.
- Proof of the fundamental theorem: translating a typing derivation to a Hoare logic derivation.

Semantic typing judgements

$$E; L \mid T_1 \models I \models x. T_2 :=$$

$$\forall t. \{[E] * [L] * [Na : t] * [T_1](t)\} \mid \{v. [L] * [Na : t] * [T_2]_{[x \leftarrow v]}(t)\}$$

Semantic typing

Adequacy follows

Remaining problem: define semantic interpretation of types
 $[\tau].\text{own}(t, \bar{v})$.

- If we have $\emptyset \mid \emptyset; \emptyset \mid \emptyset \models f \models x. x \triangleleft \text{fn}() \rightarrow \Pi$, then we have a Hoare triple with a trivial precondition. Thus f is safe.

Proving the semantic typing rules amounts to proving a Hoare triple.

- We can use the rules of the program logic, just like when proving programs.
- Proof of the fundamental theorem: translating a typing derivation to a Hoare logic derivation.

Basic types

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

$[\![\text{bool}]\!].\text{own}(t, \bar{v}) := ?$

$[\![\text{int}]\!].\text{own}(t, \bar{v}) := ?$

Basic types

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

`[[bool]].own(t, v) := v = true ∨ v = false`

`[[int]].own(t, v) := ∃z ∈ ℤ. v = z`

So far, so good...

Products

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

$$[\![\tau_1 \times \tau_2]\!].\text{own}(t, \bar{v}) := ?$$

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

Products

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

$$[\![\tau_1 \times \tau_2]\!].\text{own}(t, \bar{v}) := \exists \bar{v}_1 \bar{v}_2. \bar{v} = \bar{v}_1 \bar{v}_2 * [\![\tau_1]\!].\text{own}(t, \bar{v}_1) * [\![\tau_2]\!].\text{own}(t, \bar{v}_2)$$

Okay, that seems to work...

Fully owned pointer

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Recall: **own** τ is λ_{Rust} equivalent of `Box<T>`.

$$[\![\mathbf{own} \, \tau]\!].\text{own}(t, \bar{v}) := ?$$

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

[Introduction](#)[Syntactic type
system](#) [\$\lambda_{\text{Rust}}\$
Type system](#)[Building the logical
relation](#)[Setup](#)[Interpreting simple
types](#)[Lifetime logic](#)[Supporting unsafe
code](#)

Fully owned pointer

Recall: **own** τ is λ_{Rust} equivalent of `Box<T>`.

$$\llbracket \mathbf{own} \, \tau \rrbracket.\text{own}(t, \bar{v}) := \exists \ell \bar{w}. \bar{v} = \ell * \ell \mapsto \bar{w} * \triangleright \llbracket \tau \rrbracket.\text{own}(t, \bar{w})$$

We claim the ownership of memory at ℓ containing \bar{w} , but also the ownership related to τ , recursively.

Important remark: $\mapsto \bar{w}$ and $\llbracket \tau \rrbracket.\text{own}(t, \bar{w})$ are **separated**.

If `x: Box<T>`, then nothing in `*x` can point back to w .

A form of non-aliasing.

Unique borrows

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

$$[\![\&_{\mathbf{mut}}^{\kappa} \tau]\!].\text{own}(t, \bar{v}) := ?$$

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

Unique borrows

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

$$[\![\&_{\mathbf{mut}}^{\kappa} \tau]\!].\text{own}(t, \bar{v}) := ?$$

Problem: we do not claim full ownership.

We claim ownership up to κ . We can use it only when κ is alive.

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

The lifetime logic

We extend the separation logic with new connectives:

- $\&_{\text{full}}^{\kappa} P$: ownership of P when κ is still alive.
- $[\kappa]$: resource witnessing that κ is still alive.
- $[\dagger\kappa]$: assertion witnessing that κ is dead.
 - **persistent** assertion: not a resource, duplicable.

And prove a few rules (using Iris mechanisms):

$$\text{True} \Rightarrow \exists \kappa. [\kappa] * ([\kappa] \Rightarrow [\dagger\kappa]) \quad \triangleright P \Rightarrow \&_{\text{full}}^{\kappa} P * ([\dagger\kappa] \Rightarrow \triangleright P)$$

$$\&_{\text{full}}^{\kappa} P * [\kappa] \Rightarrow \triangleright P * (\triangleright P \Rightarrow \&_{\text{full}}^{\kappa} P * [\kappa])$$

(\Rightarrow and \Rightarrow are kinds of implication in Iris separation logic.)

Modeling unique borrows

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

$$[\![\&_{\mathbf{mut}}^{\kappa} \tau]\!].\text{own}(t, \bar{v}) := ?$$

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

Modeling unique borrows

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

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

Introduction

Syntactic type
system

λ_{Rust}

Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

Modeling unique borrows

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

Creating a borrow:

- We use $\triangleright P \not\models \&_{\text{full}}^{\kappa} P * ([\dagger \kappa] \not\models \triangleright P)$.
- We place $[\dagger \kappa] \not\models \triangleright P$ in the interpretation of the frozen typing context element:

$$[\![p \triangleleft^{\dagger \kappa} \tau]\!] := [\dagger \kappa] \not\models [\![\tau]\!].\text{own}(t, p)$$

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types
Lifetime logic

Supporting unsafe
code

Modeling unique borrows

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

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

Using a borrow:

- After getting a lifetime token $[\kappa]$, we temporarily “open” $\&_{\text{full}}^{\kappa} P$ using the rule:

$$\&_{\text{full}}^{\kappa} P * [\kappa] \not\equiv \triangleright P * (\triangleright P \not\equiv \&_{\text{full}}^{\kappa} P * [\kappa])$$

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

Modeling unique borrows

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

Reborrowing:

- Recall the syntactic rule:

$$\frac{E; L \vdash \kappa' \sqsubseteq \kappa}{E; L \vdash p \triangleleft \&_{\text{mut}}^{\kappa} \tau \xrightarrow{\text{ctx}} p \triangleleft \&_{\text{mut}}^{\kappa'} \tau, p \triangleleft^{\dagger \kappa'} \&_{\text{mut}}^{\kappa} \tau}$$

We use a lifetime logic rule for reborrowing:

$$\kappa' \sqsubseteq \kappa * \&_{\text{full}}^{\kappa} P \Rightarrow \&_{\text{full}}^{\kappa'} P * ([\dagger \kappa'] \Rightarrow \&_{\text{full}}^{\kappa} P)$$

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

1 Introduction

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types
Lifetime logic

Supporting unsafe
code

2 Syntactic type system

- λ_{Rust}
- Type system

3 Building the logical relation

- Setup
- Interpreting simple types
- Lifetime logic

4 Supporting unsafe code

The plan

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

We have seen how to prove the syntactic type system sound.

How do we extend it to, say, `Cell`?

The plan:

- 1 Define $\llbracket \text{cell}(\tau) \rrbracket$
- 2 Prove the semantic typing judgments on functions on cell (`new`, `from_inner`, `get`, `set`, ...).

Example, for Cell::new

Example, for Cell::new:

```
impl<T> Cell<T>{
    fn new(x: T) -> Cell<T> { <code> }
}
```

Where <code> is essentially `return x`.

Unfolding the definitions of the semantic judgment, we should have (roughly):

$$\{\llbracket \tau \rrbracket.\text{own}(x, t)\} <\text{code}> \{v. \llbracket \text{cell}(\tau) \rrbracket.\text{own}(v, t)\}$$

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

Example, for `Cell::new`

Formalizing Rust's
Type System
Jacques-Henri
Jourdan

Example, for `Cell::new`:

```
impl<T> Cell<T>{
    fn new(x: T) -> Cell<T> { <code> }
}
```

Where `<code>` is essentially `return x`.

Unfolding the definitions of the semantic judgment, we should have (roughly):

$$\{\llbracket \tau \rrbracket.\text{own}(x, t)\} <\!\!\text{code}\!\!> \{v. \llbracket \mathbf{cell}(\tau) \rrbracket.\text{own}(v, t)\}$$

And similarly for `Cell::into_inner`...

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

Example, for `Cell::new`

```
impl<T> Cell<T>{  
    fn new(x: T) ->  
}
```

Where `<code>` is:

Unfolding the def.

We can go from $[\tau].\text{own}(v, t)$ to $[\text{cell}(\tau)].\text{own}(v, t)$ and back, without changing anything in the memory state...

⇒ We really don't have a choice, we must use:

$$[\text{cell}(\tau)].\text{own}(v, t) := [\tau].\text{own}(v, t)$$

But... does that mean that semantically $\text{cell}(\tau)$ and τ are equivalent?

No!

And similarly for `Cell::into_inner`...

Sharing predicates

Even though we want $\llbracket \text{cell}(\tau) \rrbracket.\text{own}(v, t)$ and $\llbracket \tau \rrbracket.\text{own}(v, t)$ to be equivalent,
we do **not want** $\llbracket \&_{\text{shr}}^{\kappa} \text{cell}(\tau) \rrbracket.\text{own}(v, t)$ and $\llbracket \&_{\text{shr}}^{\kappa} \tau \rrbracket.\text{own}(v, t)$ to be equivalent

Introduction

Syntactic type
system

λ_{Rust}

Type system

Building the logical
relation

Setup

Interpreting simple
types

Lifetime logic

Supporting unsafe
code

Sharing predicates

Even though we want $\llbracket \mathbf{cell}(\tau) \rrbracket.\text{own}(v, t)$ and $\llbracket \tau \rrbracket.\text{own}(v, t)$ to be equivalent, we do **not want** $\llbracket \&_{\text{shr}}^{\kappa} \mathbf{cell}(\tau) \rrbracket.\text{own}(v, t)$ and $\llbracket \&_{\text{shr}}^{\kappa} \tau \rrbracket.\text{own}(v, t)$ to be equivalent

- The ownership predicate of $\&_{\text{shr}}^{\kappa} \tau$ is not defined from the ownership predicate of τ .
- Each type has its **own definition** of $\llbracket \&_{\text{shr}}^{\kappa} \tau \rrbracket.\text{own}(v, t)$.
 - We call this $\llbracket \tau \rrbracket.\text{shr}(\kappa, t, \ell)$, and define:

$$\llbracket \&_{\text{shr}}^{\kappa} \tau \rrbracket.\text{own}(t, \bar{v}) := \exists \ell. \bar{v} = \ell * \llbracket \tau \rrbracket.\text{shr}(\kappa, t, \ell)$$

- $\llbracket \tau \rrbracket.\text{shr}(\kappa, t, \ell)$ is the **sharing predicate** of τ . It describes the sharing protocol of τ .

Sharing predicates

Even though we want $[\![\text{cell}(\tau)]].\text{own}(v, t)$ and $[\![\tau]\].\text{own}(v, t)$ to be equivalent, we do **not want** $[\!&_{\text{shr}}^{\kappa} \text{cell}(\tau)]].\text{own}(v, t)$ and $[\!&_{\text{shr}}^{\kappa} \tau]\].\text{own}(v, t)$ to be equivalent

- The ownership predicate of $\&_{\text{shr}}^{\kappa} \tau$ is not defined from the ownership predicate of τ .
- Each type has its **own definition** of $[\!&_{\text{shr}}^{\kappa} \tau]\].\text{own}(v, t)$.
 - We call this $[\![\tau]\].\text{shr}(\kappa, t, \ell)$, and define:

$$[\!&_{\text{shr}}^{\kappa} \tau]\].\text{own}(t, \bar{v}) := \exists \ell. \bar{v} = \ell * [\![\tau]\].\text{shr}(\kappa, t, \ell)$$

- $[\![\tau]\].\text{shr}(\kappa, t, \ell)$ is the **sharing predicate** of τ . It describes the sharing protocol of τ .

Recall: **Send** types are those for which $[\![\tau]\].\text{own}(t, \bar{v})$ do not depend on thread t . How should we semantically interpret **Sync**?

Sharing predicates

Even though we want $\llbracket \mathbf{cell}(\tau) \rrbracket.\text{own}(v, t)$ and $\llbracket \tau \rrbracket.\text{own}(v, t)$ to be equivalent, we do **not want** $\llbracket \&_{\text{shr}}^{\kappa} \mathbf{cell}(\tau) \rrbracket.\text{own}(v, t)$ and $\llbracket \&_{\text{shr}}^{\kappa} \tau \rrbracket.\text{own}(v, t)$ to be equivalent

- The ownership predicate of $\&_{\text{shr}}^{\kappa} \tau$ is not defined from the ownership predicate of τ .
- Each type has its **own definition** of $\llbracket \&_{\text{shr}}^{\kappa} \tau \rrbracket.\text{own}(v, t)$.
 - We call this $\llbracket \tau \rrbracket.\text{shr}(\kappa, t, \ell)$, and define:

$$\llbracket \&_{\text{shr}}^{\kappa} \tau \rrbracket.\text{own}(t, \bar{v}) := \exists \ell. \bar{v} = \ell * \llbracket \tau \rrbracket.\text{shr}(\kappa, t, \ell)$$

- $\llbracket \tau \rrbracket.\text{shr}(\kappa, t, \ell)$ is the **sharing predicate** of τ . It describes the sharing protocol of τ .

Recall: **Send** types are those for which $\llbracket \tau \rrbracket.\text{own}(t, \bar{v})$ do not depend on thread t .

Sync types are those for which $\llbracket \tau \rrbracket.\text{shr}(\kappa, t, \ell)$ do not depend on thread t .

- This perfectly matches $\mathbf{T}: \text{Sync} \Leftrightarrow \&\mathbf{T}: \text{Send}$.

Sharing predicates

Formalizing Rust's
Type System

Jacques-Henri
Jourdan

Introduction

Syntactic type
system

λ_{Rust}
Type system

Building the logical
relation

Setup
Interpreting simple
types

Lifetime logic

Supporting unsafe
code

We will not give sharing predicates for every types we have seen...

The idea is to define the **sharing protocol** during κ for each of these types..

- **[[int]].shr(κ, t, ℓ)** and **[[bool]].shr(κ, t, ℓ)** only allow read access during κ .
- **[[cell(τ)].shr(κ, t, ℓ)** allow reading and writing during κ :
 - but only in thread t (token $[\text{Na} : t]$ required),
 - without extracting ownership.
- ...

Conclusion

I hope I gave you a taste of what it is to define a logical relation for a large language!

As you have seen, this is quite technical.

Formalizing in Coq not only improves trust. It also **helps** to manage the various details.

We can add many extensions to this proof. Examples:

- Add other unsafe libraries (`RefCell`, `Rc`, `Mutex`, `Arc`, ...).
- Support for weak memory model.
- Use a **binary** logical relation instead of a unary logical relation.
 - We prove that a (simpler) program is equivalent to a well-typed Rust program.
 - \implies It is possible to use typing to guide the translation to a purely functional program, easier to prove.

Last words

Did you learn things in this course?

Please comment by email or in the course evaluation (if this still exists?). We would be happy to improve it!

Reminder: course with Pierre-Evariste Dagand next week (February, 21st), exam on March, 6th.

Good luck for your exams, and I wish you all the best for the rest of your studies!