Formalizing Rust's Type System

Jacques-Henri Jourdan

February 23rd, 2022

Date of the exam

The date of the exam is March 9th, 2022 (12:45).

If you cannot be available (another exam...), speak now!

This is the last session for this course.
Next week (March 2nd) is free time for reviewing!

Abstract from last weeks

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.

Today

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

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

"Well-typed Rust programs do not go wrong."

Really?

The problem we are trying to solve

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

Is that all?

The problem we are trying to solve

"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) $\cdots \vdash \cdots$
- Syntactic typing rules.

  Ex. for lambda-calculus: $\dfrac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$

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

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

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

Extending semantic type safety

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

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{\cdots \vdash \cdots}{\cdots \vdash \cdots} \\ \vdots\end{array} \qquad \cdots \vdash \cdots}{\cdots \vdash \cdots} \qquad \cdots \vdash \cdots}{\cdots \vdash \cdots} \qquad \Rightarrow \qquad \dfrac{\dfrac{\begin{array}{c}\dfrac{\cdots \models \cdots}{\cdots \models \cdots} \\ \vdots\end{array} \qquad \cdots \models \cdots}{\cdots \models \cdots} \qquad \cdots \models \cdots}{\cdots \models \cdots}$$

Extending semantic type safety

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

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{\cdots\vdash\cdots}{\cdots\vdash\cdots}\\ \vdots\end{array} \qquad \cdots\vdash\cdots}{\cdots\vdash\cdots} \qquad \cdots\vdash\cdots}{\cdots\vdash\cdots} \quad\Rightarrow\quad \dfrac{\dfrac{\begin{array}{c}\dfrac{\cdots\models\cdots}{\cdots\models\cdots}\\ \vdots\end{array} \qquad \cdots\models\cdots}{\cdots\models\cdots} \qquad \cdots\models\cdots}{\cdots\models\cdots}$$

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

Extending semantic type safety

`Cell` is not syntactically well-typed in safe Rust: "$\cdots \nvdash$ `Cell::get` $: \cdots$".

But we can prove "$\cdots \vDash$ `Cell::get` $: \cdots$".

Combining with an otherwise syntactically well-typed program:

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{\cdots \vdash \cdots}{\cdots \vdash \cdots} \\ \vdots\end{array} \qquad \cdots \vDash \texttt{Cell::get} : \cdots}{\cdots \vDash \cdots} \qquad \cdots \vdash \cdots}{\cdots \vDash \cdots}$$

And we can conclude safety thanks to adequacy!

Extending semantic type safety

If the definition of $\cdots \models \cdots$ is well-chosen, then we can hope to prove the safety of a program using unsafe code:

1. Prove $\cdots \models \cdots$ for the safely encapsulated library using `unsafe`.
2. Deduce that the whole program is semantically well-typed.
3. Conclude safety thanks to adequacy.

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.

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

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.

The language: λ_{Rust}

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

$$\begin{aligned} \mathit{Val} \ni v &::= \texttt{false} \mid \texttt{true} \mid z \mid \ell \mid \texttt{funrec}\, f(\bar{x})\ \ \texttt{ret}\, k := F \\ \mathit{Path} \ni p &::= x \mid p.n \end{aligned}$$

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

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

The language: λ_{Rust}

Paths, elementary values

$$\mathit{Val} \ni v ::= \texttt{false} \mid \texttt{true} \mid z \mid \ell \mid \texttt{funrec}\, f(\bar{x})\ \ \texttt{ret}\, k := F$$
$$\mathit{Path} \ni p ::= x \mid p.n$$

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).

The language: λ_{Rust}

Instructions

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

Instructions are the elementary operations of λ_{Rust}.

They take paths as operands.

The language: λ_{Rust}

Instructions

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

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

The language: λ_{Rust}

Instructions

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

... 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: $p_1 :=_n {}^*p_2$.

The language: λ_{Rust}

Function bodies

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

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

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

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

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

Ghost instructions for creating/ending a lifetime.

The language: λ_{Rust}

Handling sum types

$$
\begin{aligned}
\mathit{Instr} \ni I ::= {} & \ldots \\
& \mid p \overset{\text{inj } i}{:==} () \mid p_1 \overset{\text{inj } i}{:==} p_2 \mid p_1 \overset{\text{inj } i}{:==}_n {}^*p_2 \\
& \mid \ldots \\
\mathit{FuncBody} \ni F ::= {} & \ldots \\
& \mid \mathtt{case}\ {}^*p\ \ \mathtt{of}\ \bar{F}
\end{aligned}
$$

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

$$\mathit{Type} \ni \tau ::= \textbf{bool} \mid \textbf{int} \mid \textbf{own}\ \tau \mid \&^{\kappa}_{\textbf{mut}}\ \tau \mid \&^{\kappa}_{\textbf{shr}}\ \tau \mid \lightning_n$$
$$\mid \Pi\bar{\tau} \mid \Sigma\bar{\tau} \mid \forall\bar{\alpha}.\,\textbf{fn}(_{\mathsf{F}} : \mathsf{E}; \bar{\tau}) \to \tau \mid T \mid \mu\, T.\, \tau$$
$$\mid \ldots$$

Types for integer and Booleans.

Types

$$
\begin{aligned}
\mathit{Type} \ni \tau ::= {} & \mathbf{bool} \mid \mathbf{int} \mid \mathbf{own}\ \tau \mid \&^{\kappa}_{\mathbf{mut}}\ \tau \mid \&^{\kappa}_{\mathbf{shr}}\ \tau \mid \lightning_{n} \\
& \mid \Pi\bar{\tau} \mid \Sigma\bar{\tau} \mid \forall\bar{\alpha}.\, \mathbf{fn}(_{\mathsf{F}} : \mathsf{E}; \bar{\tau}) \to \tau \mid T \mid \mu\, T.\, \tau \\
& \mid \ldots
\end{aligned}
$$

Types for pointers: `Box` is written $\mathbf{own}\ \tau$.

Types

$$\mathit{Type} \ni \tau ::= \textbf{bool} \mid \textbf{int} \mid \textbf{own}\,\tau \mid \&^{\kappa}_{\textbf{mut}}\,\tau \mid \&^{\kappa}_{\textbf{shr}}\,\tau \mid \lightning_{n}$$
$$\mid \Pi\bar{\tau} \mid \Sigma\bar{\tau} \mid \forall\bar{\alpha}.\,\textbf{fn}(\mathsf{F} : \mathsf{E}; \bar{\tau}) \to \tau \mid T \mid \mu\,T.\,\tau$$
$$\mid \ldots$$

Type of "uninitialized memory".

- When memory is just initialized, or when non-`Copy` values are moved.
- Used to replace the "initializedness" analysis of the borrow checker.

Types

$$
\begin{aligned}
Type \ni \tau ::= &\ \mathbf{bool} \mid \mathbf{int} \mid \mathbf{own}\ \tau \mid \&^{\kappa}_{\mathbf{mut}}\ \tau \mid \&^{\kappa}_{\mathbf{shr}}\ \tau \mid \lightning_{n} \\
&\mid \Pi\bar{\tau} \mid \Sigma\bar{\tau} \mid \forall\bar{\alpha}.\, \mathbf{fn}(\mathsf{F} : \mathsf{E}; \bar{\tau}) \to \tau \mid T \mid \mu\, T.\, \tau \\
&\mid \ldots
\end{aligned}
$$

Complex types: sum, products.

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

Types

$$
\begin{aligned}
\mathit{Type} \ni \tau ::= &\ \mathbf{bool} \mid \mathbf{int} \mid \mathbf{own}\,\tau \mid \&^{\kappa}_{\mathbf{mut}}\,\tau \mid \&^{\kappa}_{\mathbf{shr}}\,\tau \mid \lightning_{n} \\
&\mid \Pi\bar{\tau} \mid \Sigma\bar{\tau} \mid \forall\bar{\alpha}.\,\mathbf{fn}(\mathsf{F} : \mathsf{E}; \bar{\tau}) \to \tau \mid T \mid \mu\,T.\,\tau \\
&\mid \ldots
\end{aligned}
$$

Type of function pointers, guarded equirecursive types...

Types

$$\begin{aligned} \mathit{Type} \ni \tau ::= {} & \mathbf{bool} \mid \mathbf{int} \mid \mathbf{own}\,\tau \mid \&^{\kappa}_{\mathbf{mut}}\,\tau \mid \&^{\kappa}_{\mathbf{shr}}\,\tau \mid \lightning_{n} \\ & \mid \Pi\bar{\tau} \mid \Sigma\bar{\tau} \mid \forall\bar{\alpha}.\,\mathbf{fn}(\mathsf{F} : \mathsf{E}; \bar{\tau}) \to \tau \mid T \mid \mu\, T.\, \tau \\ & \mid \ldots \end{aligned}$$

Extensible for types providing safe abstraction over `unsafe`!

Typing judgments

There are two main typing judgments:

- One for instructions: $\mathrm{E}; \mathrm{L} \mid \mathrm{T}_1 \vdash I \dashv x.\, \mathrm{T}_2$
- One for function bodies: $\mathrm{E}; \mathrm{L} \mid \mathrm{K}; \mathrm{T} \vdash F$

They depend on four different kinds of contexts:

- T is the typing context. Elements:
  - $p \lhd \tau$: path p contains an elementary value of type τ.
  - $p \lhd^{\dagger\kappa} \tau$: 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: $k \lhd \mathbf{cont}(\mathrm{L}; \bar{x}.\, \mathrm{T})$.
    "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: $\mathsf{E}; \mathsf{L} \mid \mathsf{T}_1 \vdash I \dashv x.\, \mathsf{T}_2$
- One for function bodies: $\mathsf{E}; \mathsf{L} \mid \mathsf{K}; \mathsf{T} \vdash F$

They depend on four different kinds of contexts:

- T is the typing context. Elements:
  - $p \lhd \tau$: path p contains an elementary value of type τ.
  - $p \lhd^{\dagger\kappa} \tau$: 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: $k \lhd \mathbf{cont}(\mathsf{L}; \bar{x}.\, \mathsf{T})$.
    "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: $\mathsf{E}; \mathsf{L} \mid \mathsf{T}_1 \vdash I \dashv x.\, \mathsf{T}_2$
- One for function bodies: $\mathsf{E}; \mathsf{L} \mid \mathsf{K}; \mathsf{T} \vdash F$

They depend on four different kinds of contexts:

- T is the typing context. Elements:
  - $p \lhd \tau$: path p contains an elementary value of type τ.
  - $p \lhd^{\dagger\kappa} \tau$: 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: $k \lhd \mathbf{cont}(\mathsf{L}; \bar{x}.\, \mathsf{T})$.
    "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: $\mathrm{E}; \mathrm{L} \mid \mathrm{T}_1 \vdash I \dashv x.\, \mathrm{T}_2$
- One for function bodies: $\mathrm{E}; \mathrm{L} \mid \mathrm{K}; \mathrm{T} \vdash F$

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.
$x.\mathrm{T}_2$ binds a special variable x: to give a type to the result of the instruction.

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

Typing rules

Typing instructions

Selected rules:

$$\mathsf{E};\mathsf{L} \mid \emptyset \vdash z \dashv x.\, x \lhd \mathbf{int} \qquad \mathsf{E};\mathsf{L} \mid p_1 \lhd \mathbf{int}, p_2 \lhd \mathbf{int} \vdash p_1 \leq p_2 \dashv x.\, x \lhd \mathbf{bool}$$

$$\mathsf{E};\mathsf{L} \mid \emptyset \vdash \mathtt{new}(n) \dashv x.\, x \lhd \mathbf{own}\ \lightning_n \qquad \frac{n = \mathsf{size}(\tau)}{\mathsf{E};\mathsf{L} \mid p \lhd \mathbf{own}\ \tau \vdash \mathtt{delete}(n, p) \dashv \emptyset}$$

$$\frac{\tau\ \mathsf{copy} \qquad \mathsf{size}(\tau) = 1}{\mathsf{E};\mathsf{L} \mid p \lhd \mathbf{own}\ \tau \vdash {}^*p \dashv x.\, p \lhd \mathbf{own}\ \tau, x \lhd \tau} \qquad \frac{\mathsf{size}(\tau) = 1}{\mathsf{E};\mathsf{L} \mid p \lhd \mathbf{own}\ \tau \vdash {}^*p \dashv x.\, p \lhd \mathbf{own}\ \lightning_1, x \lhd \tau}$$

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

Typing rules

Typing function bodies

Selected rules:

$$\frac{\mathsf{E}; \mathsf{L} \mid \mathsf{T}_1 \vdash I \dashv x.\, \mathsf{T}_2 \qquad \mathsf{E}; \mathsf{L} \mid \mathsf{K}; \mathsf{T}_2, \mathsf{T} \vdash F}{\mathsf{E}; \mathsf{L} \mid \mathsf{K}; \mathsf{T}_1, \mathsf{T} \vdash \mathtt{let}\, x = I \;\; \mathtt{in}\, F} \qquad \frac{k \lhd \mathbf{cont}(\mathsf{L}; \bar{x}.\, \mathsf{T}) \in \mathsf{K}}{\mathsf{E}; \mathsf{L} \mid \mathsf{K}; \mathsf{T}[\bar{y}/\bar{x}] \vdash \mathtt{jump}\, k(\bar{y})}$$

$$\frac{\mathsf{E}; \mathsf{L} \vdash \mathsf{T} \overset{\mathsf{ctx}}{\Rrightarrow} \mathsf{T}' \qquad \mathsf{E}; \mathsf{L} \mid \mathsf{K}; \mathsf{T}' \vdash F}{\mathsf{E}; \mathsf{L} \mid \mathsf{K}; \mathsf{T} \vdash F}$$

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

Typing rules

Transforming typing environments

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

$$\frac{\tau \text{ copy}}{\mathsf{E}; \mathsf{L} \vdash p \lhd \tau \overset{\mathsf{ctx}}{\Rrightarrow} p \lhd \tau, p \lhd \tau}$$

$$\mathsf{E}; \mathsf{L} \vdash p \lhd \&^{\kappa}_{\mu} (\tau_1 \times \tau_2) \overset{\mathsf{ctx}}{\Rrightarrow} p.0 \lhd \&^{\kappa}_{\mu} \tau_1, \; p.\mathsf{size}(\tau_1) \lhd \&^{\kappa}_{\mu} \tau_2$$

$$\frac{\mathsf{E}; \mathsf{L} \vdash \kappa \text{ alive}}{\mathsf{E}; \mathsf{L} \vdash p \lhd \&^{\kappa}_{\mathbf{mut}} \tau \overset{\mathsf{ctx}}{\Rrightarrow} p \lhd \&^{\kappa}_{\mathbf{shr}} \tau} \qquad \mathsf{E}; \mathsf{L} \vdash p \lhd \mathbf{own}_n \, \tau \overset{\mathsf{ctx}}{\Rrightarrow} p \lhd \&^{\kappa}_{\mathbf{mut}} \tau, p \lhd^{\dagger\kappa} \mathbf{own}_n \, \tau$$

$$\frac{\mathsf{E}; \mathsf{L} \vdash \kappa' \sqsubseteq \kappa}{\mathsf{E}; \mathsf{L} \vdash p \lhd \&^{\kappa}_{\mathbf{mut}} \tau \overset{\mathsf{ctx}}{\Rrightarrow} p \lhd \&^{\kappa'}_{\mathbf{mut}} \tau, p \lhd^{\dagger\kappa'} \&^{\kappa}_{\mathbf{mut}} \tau}$$

Recap

We have presented syntactic typing judgments $\mathsf{E};\mathsf{L} \mid \mathsf{T}_1 \vdash I \dashv x.\, \mathsf{T}_2$ and $\mathsf{E};\mathsf{L} \mid \mathsf{K};\mathsf{T} \vdash F$.

We need to:

- Give them semantic counterparts $\mathsf{E};\mathsf{L} \mid \mathsf{T}_1 \models I \Dashv x.\, \mathsf{T}_2$ and $\mathsf{E};\mathsf{L} \mid \mathsf{K};\mathsf{T} \models F$
- Prove adequacy: if $\emptyset \mid \emptyset;\emptyset \mid \emptyset \models f \Dashv x.\, x \lhd \mathbf{fn}() \to \Pi[]$, 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 ownership. Any ideas how to models this?


Separation Logic
to the Rescue!

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!

We use Iris, a modern an expressive separation logic.

It features:

- Standard separation logic mechanisms
- Support for concurrency
- Built-in support for step-indexing (equirecursive types...)
- A powerful mechanism to create new mechanisms of ownership (borrows, ...)

Semantic interpretation of types

Define an ownership predicate $[\![\tau]\!].\mathrm{own}(t,\bar{v})$ 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

Define an ownership predicate $[\![\tau]\!].\mathrm{own}(t, \bar{v})$ 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 $[\![\tau]\!].\mathrm{own}(t, \bar{v})$ do not depend on t.
$\Rightarrow$ The interpretation will still be valid in another thread.

Interpreting semantic environments

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

$$\llbracket p \lhd \tau, \mathsf{T} \rrbracket := \llbracket \tau \rrbracket.\mathrm{own}(t, p) * \llbracket T \rrbracket$$
$$\llbracket \emptyset \rrbracket := \mathsf{True}$$

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

Semantic typing judgments

$$\mathsf{E};\mathsf{L} \mid \mathsf{T}_1 \models I \dashv x.\,\mathsf{T}_2 := \\ \forall t.\,\{[\![\mathsf{E}]\!] * [\![\mathsf{L}]\!] * [\mathrm{Na} : t] * [\![\mathsf{T}_1]\!](t)\}\ \ I\ \ \{v.\,[\![\mathsf{L}]\!] * [\mathrm{Na} : t] * [\![\mathsf{T}_2]\!]_{[x \leftarrow v]}(t)\}$$

Semantic typing judgment for function bodies is similar.

(I have omitted the definition of $[\![\mathsf{E}]\!]$ and $[\![\mathsf{L}]\!]$ – 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.

$[\mathrm{Na} : t]$ is a special resource owned by one thread only.
Non-thread-safe types need $[\mathrm{Na} : t]$. Thus their interpretation depends on t.

Semantic typing judgments

$$\begin{aligned}&\mathsf{E}; \mathsf{L} \mid \mathsf{T}_1 \models I \mathrel{=\!\!\mid} x.\, \mathsf{T}_2 := \\ &\quad \forall t.\, \{[\![\mathsf{E}]\!] * [\![\mathsf{L}]\!] * [\mathrm{Na} : t] * [\![\mathsf{T}_1]\!](t)\}\ \ I\ \ \{v.\, [\![\mathsf{L}]\!] * [\mathrm{Na} : t] * [\![\mathsf{T}_2]\!]_{[x \leftarrow v]}(t)\}\end{aligned}$$

Semantic typing judgment for function bodies is similar.

Adequacy follows from the soundness of the separation logic.

- If we have $\emptyset \mid \emptyset; \emptyset \mid \emptyset \models f \mathrel{=\!\!\mid} x.\, x \lhd \mathbf{fn}() \to \Pi[]$, 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.

Semantic typing judgments

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

Semantic typing

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

Basic types

$$\llbracket \textbf{bool} \rrbracket.\mathrm{own}(t, \bar{v}) := ?$$
$$\llbracket \textbf{int} \rrbracket.\mathrm{own}(t, \bar{v}) := ?$$

Basic types

$$\llbracket \textbf{bool} \rrbracket.\mathrm{own}(t,\bar{v}) := \bar{v} = \texttt{true} \vee \bar{v} = \texttt{false}$$
$$\llbracket \textbf{int} \rrbracket.\mathrm{own}(t,\bar{v}) := \exists z \in \mathbb{Z}.\bar{v} = z$$

So far, so good...

Products

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

Products

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

Okay, that seems to work...

Fully owned pointer

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

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

Fully owned pointer

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

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

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 $[\![\tau]\!].\mathrm{own}(t, \bar{w})$ are separated.
If `x: Box<T>`, then nothing in `*x` can point back to w.
A form of non-aliasing.

Unique borrows

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

Unique borrows

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

Problem: we do not claim full ownership.
We claim ownership up to κ. We can use it only when κ is alive.

The lifetime logic

We extend the separation logic with new connectives:

- $\&_{\mathsf{full}}^{\kappa} P$: 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):

$$\mathsf{True} \Rrightarrow \exists\kappa.\, [\kappa] * ([\kappa] \Rrightarrow [\dagger\kappa]) \qquad \triangleright P \Rrightarrow \&_{\mathsf{full}}^{\kappa} P * ([\dagger\kappa] \Rrightarrow \triangleright P)$$

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

($\Rrightarrow$ is a kind of implication in Iris separation logic.)

Modeling unique borrows

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

Modeling unique borrows

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

Modeling unique borrows

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

Creating a borrow:

- We use $\triangleright P \Rrightarrow\!\!\ast\; \&^{\kappa}_{\mathsf{full}}\, P * \left([\dagger\kappa] \Rrightarrow\!\!\ast \triangleright P\right)$.
- We place $[\dagger\kappa] \Rrightarrow\!\!\ast \triangleright P$ in the interpretation of the frozen typing context element:

$$[\![p \lhd^{\dagger\tau}]\!] := [\dagger\kappa] \Rrightarrow\!\!\ast\; [\![\tau]\!].\mathrm{own}(t,p)$$

Modeling unique borrows

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

Using a borrow:

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

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

Modeling unique borrows

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

Reborrowing:

- Recall the syntactic rule:

$$\frac{\mathsf{E}; \mathsf{L} \vdash \kappa' \sqsubseteq \kappa}{\mathsf{E}; \mathsf{L} \vdash p \lhd \&^{\kappa}_{\textbf{mut}}\,\tau \overset{\textbf{ctx}}{\Rrightarrow} p \lhd \&^{\kappa'}_{\textbf{mut}}\,\tau, p \lhd^{\dagger\kappa'} \&^{\kappa}_{\textbf{mut}}\,\tau}$$

We use a lifetime logic rule for reborrowing:

$$\kappa' \sqsubseteq \kappa * \&^{\kappa}_{\mathsf{full}}\,P \Rrightarrow\!\!\ast\; \&^{\kappa'}_{\mathsf{full}}\,P * ([\dagger\kappa'] \Rrightarrow\!\!\ast\; \&^{\kappa}_{\mathsf{full}}\,P)$$

The plan

We have seen how to prove the syntactic type system sound.
How do we extend it to, say, `Cell`?

The plan:

1. Define $[\![\textbf{cell}(\tau)]\!]$
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> }
}
```

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

$$\{[\![\tau]\!].\mathrm{own}(\mathrm{x}, t)\}\ \texttt{<code>}\ \{v.\ [\![\textbf{cell}(\tau)]\!].\mathrm{own}(v, t)\}$$

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

Example, for `Cell::new`

Example, for `Cell::new`:

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

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

$$\{[\![\tau]\!].\mathrm{own}(\mathrm{x}, t)\}\ \texttt{<code>}\ \{v.\ [\![\textsf{\textbf{cell}}(\tau)]\!].\mathrm{own}(v, t)\}$$

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

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

Example, for `Cell::new`

Example, for Ce

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

Unfolding the d

Where <code> i

And similarly fo

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

$\Rightarrow$ We really don't have a choice, we must use:

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

But... does that mean that semantically **cell**(τ) and τ are equivalent? No!

Sharing predicates

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

Sharing predicates

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

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

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

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

Sharing predicates

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

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

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

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

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

Sharing predicates

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

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

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

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

Recall: `Send` types are those for which $[\![\tau]\!].\mathrm{own}(t, \bar{v})$ do not depend on thread t.
`Sync` types are those for which $[\![\tau]\!].\mathrm{shr}(\kappa, t, \ell)$ do not depend on thread t.

- This perfectly matches `T: Sync` $\Leftrightarrow$ `&T: Send`.

Sharing predicates

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..

- $[\![\textbf{int}]\!].\mathrm{shr}(\kappa, t, \ell)$ and $[\![\textbf{bool}]\!].\mathrm{shr}(\kappa, t, \ell)$ only allow read access during κ.
- $[\![\textbf{cell}(\tau)]\!].\mathrm{shr}(\kappa, t, \ell)$ allow reading and writing during κ:
  - but only in thread t (token $[\mathrm{Na} : t]$ 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.
- Add a specification part to the judgments to make it possible to compute specifications to programs while typing...

Last words

Did you learn things in this course?
Please comment in the course evaluation. We would be happy to improve it!

Reminder: no course next week (March, 2nd), exam on March, 9th.

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