RustHornBelt:

A Semantic Foundation for Functional Verification of **Rust** Programs with Unsafe Code

Yusuke Matsushita
The University of Tokyo

Xavier Denis
Université Paris-Saclay, LMF

Jacques-Henri Jourdan
CNRS, LMF

Derek Dreyer
MPI-SWS

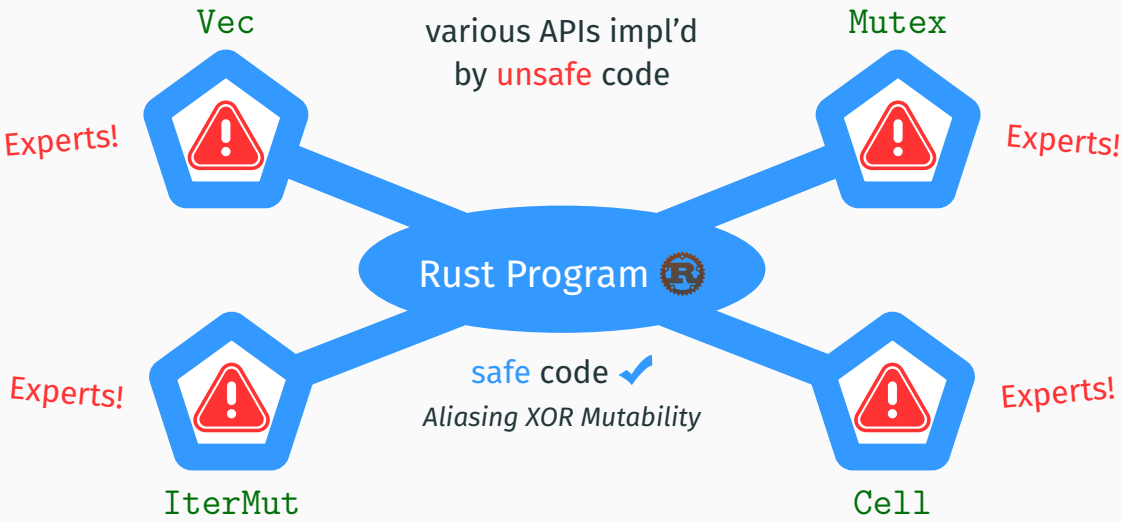
PLDI 2022, San Diego — June 17, 2022

Our High-level Goal: Rust Verification

safe code ✓

Aliasing XOR Mutability

Our High-level Goal: Rust Verification



Two Lines of Prior Work on Rust Verification

Two Lines of Prior Work on Rust Verification

RustBelt 

[Jung+ ‘18]

Safe & **Unsafe** 🏆

Manual in Iris ✳

Type Safety

Two Lines of Prior Work on Rust Verification

RustBelt

[Jung+ '18]

Safe & **Unsafe**

Manual in Iris ✱

Type Safety

RustHorn

[Matsushita+ '20],

Prusti P∗rust−∗i

[Astrauskas+ '19], ...

Safe

Automated

Func Correct

Two Lines of Prior Work on Rust Verification

RustHornBelt

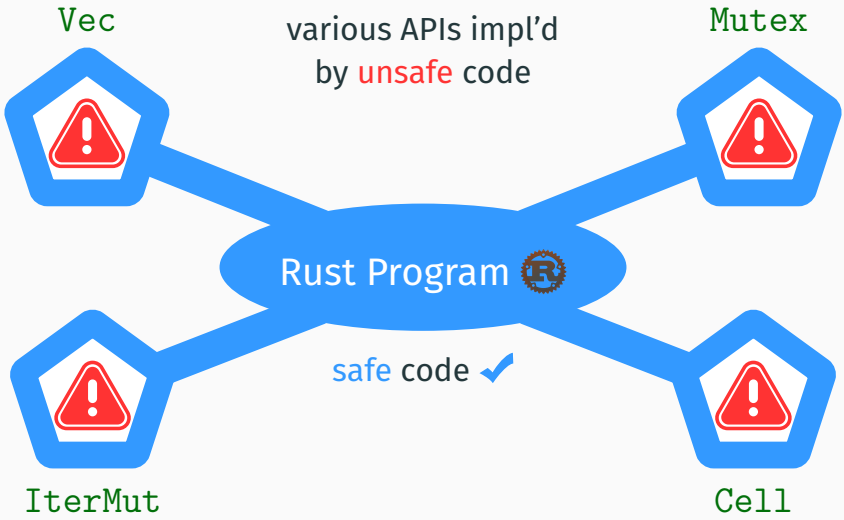
RustBelt
[Jung+ '18]

Safe & **Unsafe**
Manual in Iris ∗
Type Safety

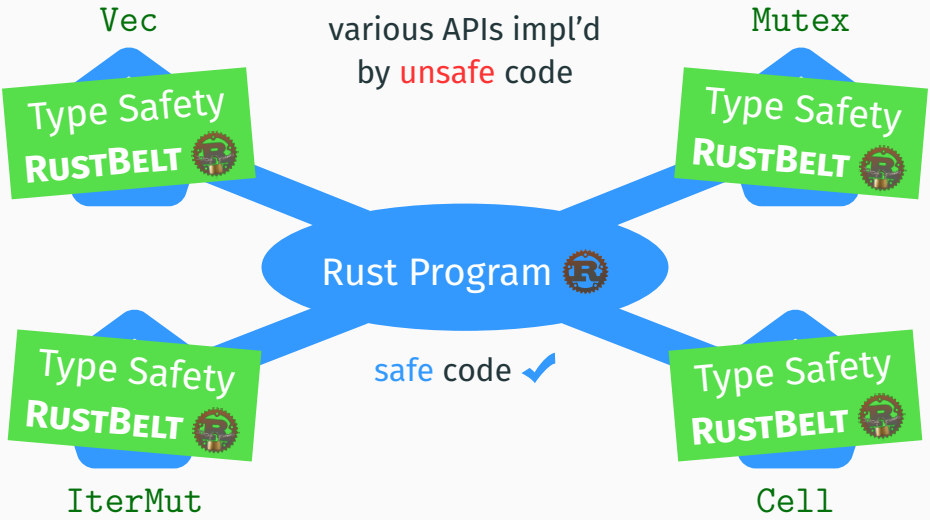
RustHorn
[Matsushita+ '20],
Prusti $P{*}\text{rust}{-\!*}i$
[Astrauskas+ '19], ...

Safe
Automated
Func Correct

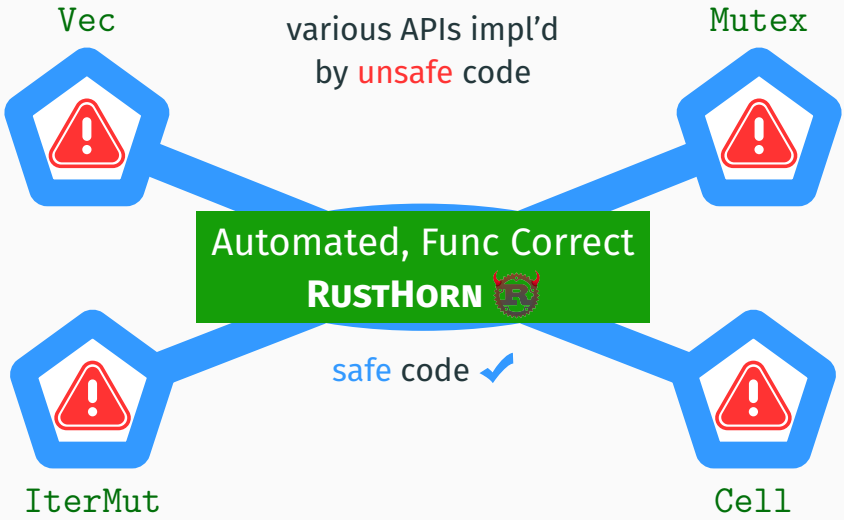
Our Goal: Marrying RustBelt with RustHorn



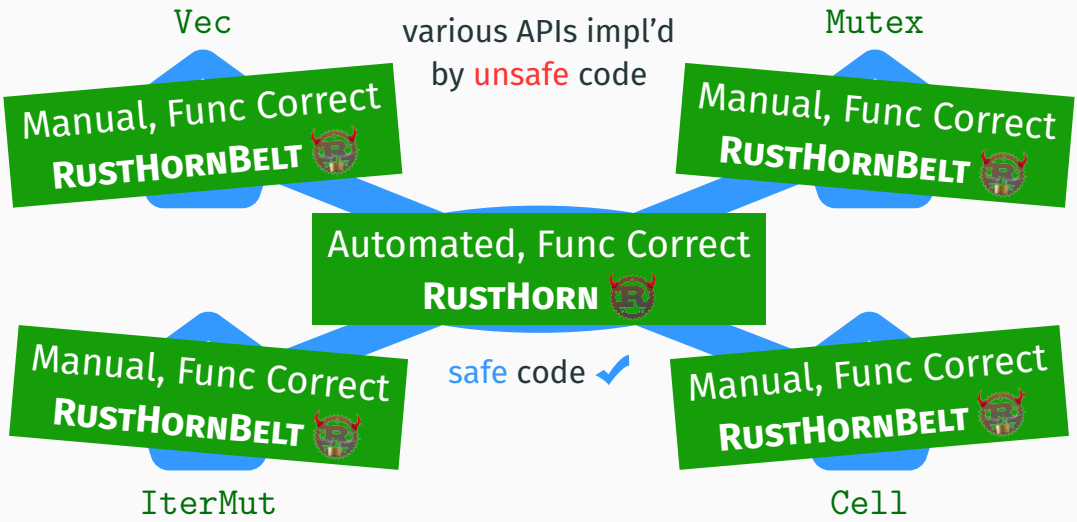
Our Goal: Marrying RustBelt with RustHorn



Our Goal: Marrying RustBelt with RustHorn



Our Goal: Marrying RustBelt with RustHorn



Our Work, RustHornBelt 

Marriage of **RustBelt**  with **RustHorn**  :

Our Work, RustHornBelt 

Marriage of **RustBelt** with **RustHorn**:

- **RustBelt**-based **semantic model** of Rust's **ownership types** in the separation logic Iris ✻ (Coq 🐓!), but extended with **RustHorn**-style functional **specs**

Our Work, RustHornBelt 

Marriage of **RustBelt**  with **RustHorn**  :

- **RustBelt**-based **semantic model** of Rust's **ownership types** in the separation logic Iris ✻ (Coq 🐓!), but extended with **RustHorn**-style functional **specs**
- Prove **soundness** of **RustHorn**-style functional **specs** of key **Rust APIs** ✔ with **unsafe** impl ⚠

Our Work, RustHornBelt 

Marriage of **RustBelt** with **RustHorn**:

- **RustBelt**-based **semantic model** of Rust's **ownership types** in the separation logic Iris ✻ (Coq 🐓!), but extended with **RustHorn**-style functional **specs**
- Prove **soundness** of **RustHorn**-style functional **specs** of key **Rust APIs** ✔ with **unsafe** impl ⚠
  - *Type-spec system* & *Parametric prophecies* 🔮

Our Work, RustHornBelt 

Marriage of **RustBelt**  with **RustHorn**  :

??

- **RustBelt**-based **semantic model** of Rust's **ownership types** in the separation logic Iris ✻ (Coq  !), but extended with **RustHorn**-style functional **specs**

??

- Prove **soundness** of **RustHorn**-style functional **specs** of key **Rust APIs** ✔ with **unsafe** impl 
  - *Type-spec system* & *Parametric prophecies* 

Motivating Example for RustHorn

What spec do you give to `max_or_incr`?

```
fn max_or_incr(ma: &mut int, mb: &mut int) -> &mut int {
    if *ma >= *mb {
        *mb += 42;  ma
    } else {
        *ma += 42;  mb
    }
}
```

Motivating Example for RustHorn

What spec do you give to `max_or_incr`?

```
fn max_or_incr(ma: &mut int, mb: &mut int) -> &mut int {
    if *ma >= *mb {
        *mb += 42;
        mb
    } else {
        *ma += 1;
        ma
    }
}
```

Mutable reference, ***borrowing*** ownership

Motivating Example for RustHorn

What spec do you give to `max_or_incr`?

```
fn max_or_incr(ma: &mut int, mb: &mut int) -> &mut int {
    if *ma >= *mb {
        *mb +=
    } else {
        *ma += 42;  mb
    }
}
```

Not aliased

Not aliased

Motivating Example for RustHorn

What spec do you give to `max_or_incr`?

```
fn max_or_incr(ma: &mut int, mb: &mut int) -> &mut int {
    if *ma >= *mb {
        *mb += 42;  ma
    } else {
        *ma += 42;  mb
    }
}
```

Motivating Example for RustHorn

What spec do you give to `max_or_incr`?

```
fn max_or_incr(ma: &mut int, mb: &mut int) -> &mut int {
    if *ma >= *mb {
        *mb += 42;  ma
    } else {
        *ma += 42;  mb
    }
}
```

Separation Logic?

Motivating Example for RustHorn

What spec do you give to `max_or_incr`?

```
fn max_or_incr(ma: &mut int, mb: &mut int) -> &mut int {
    if *ma >= *mb {
        *mb += 42;  ma
    } else {
        *ma += 42;  mb
    }
}
```

~~Separation Logic?~~

RustHorn — Automated Verification

RustHorn  *automates* **functional verification** of **safe** **Rust code** ✓, only using **first-order logic**!

RustHorn — Automated Verification

RustHorn  *automates* **functional verification** of **safe Rust code** ✓, only using **first-order logic**!

- Take advantage of **automation** techniques!
  - CHC (RustHorn), Why3 (Creusot [Denis+ '21]), ...

RustHorn — Automated Verification

RustHorn  *automates* **functional verification** of **safe Rust code** ✔, only using **first-order logic**!

- Take advantage of **automation** techniques!
  - CHC (RustHorn), Why3 (Creusot [Denis+ '21]), ...
- Key idea: **Mutable ref** `ma` repr'ed as a **value** (a, a')

RustHorn — Automated Verification

RustHorn  *automates* **functional verification** of **safe Rust code** ✓, only using **first-order logic**!

- Take advantage of **automation** techniques!
  - CHC (RustHorn), Why3 (Creusot [Denis+ '21]), ...
- Key idea: **Mutable ref** `ma` repr'ed as a **value** (a, a')
  - a — *current* value

RustHorn — Automated Verification

RustHorn  *automates* **functional verification** of **safe Rust code** ✔, only using **first-order logic**!

- Take advantage of **automation** techniques!
  - CHC (RustHorn), Why3 (Creusot [Denis+ '21]), ...
- Key idea: **Mutable ref** `ma` repr'ed as a **value** (a, a')
  - a — *current* value
  - a' — **prophecy**  of *final* value, returned to original owner

RustHorn's Answer to the Example

```
fn max_or_incr(ma: &mut int, mb: &mut int) -> &mut int {
    if *ma >= *mb {
        *mb += 42;  ma
    } else {
        *ma += 42;  mb
    }
}
```

$$\begin{aligned} &\mathit{max\_or\_incr}\ (a, a')\ (b, b')\ \mathit{res} \quad \triangleq \\ &\qquad \text{if}\ \ a \geq b\ \ \text{then}\ \ b' = b + 42\ \wedge\ \mathit{res} = (a, a') \\ &\qquad\qquad \text{else}\ \ a' = a + 42\ \wedge\ \mathit{res} = (b, b') \end{aligned}$$

RustHorn's Answer to the Example

```
fn max_or_incr(ma: &mut int, mb: &mut int) -> &mut int {
    if *ma >= *mb {
        *mb += 42;  ma
    } else {
        *ma += 42;  mb
    }
}
```

$$max\_or\_incr\ (a, a')\ (b, b')\ res \triangleq$$

$$\text{if } a \geq b \text{ then } b' = b + 42 \wedge res = (a, a')$$
$$\text{else } a' = a + 42 \wedge res = (b, b')$$

prophecy

RustHorn's Answer to the Example

```
fn max_or_incr(ma: &mut int, mb: &mut int) -> &mut int {
    if *ma >= *mb {
        *mb += 42;  ma
    } else {
        *ma += 42;  mb
    }
}
```

$$max\_or\_incr\ (a, a')\ (b, b')\ res \triangleq$$

$$\text{if } a \geq b \text{ then } b' = b + 42 \wedge res = (a, a')$$

$$\text{else } a' = a + 42 \wedge res = (b, b')$$

resolution

Type-Spec System

Type-Spec System

RustBelt's type system

$$\mathbf{T} \vdash \mathit{I} \dashv \mathtt{r}.\ \mathbf{T}'$$

typing judgment

Type-Spec System

RustBelt's type system + **Func Spec**

$$\mathbf{T} \vdash \mathit{I} \dashv \mathtt{r}.\ \mathbf{T}' \rightsquigarrow \mathbf{\Phi}$$

typing judgment

func spec

Type-Spec System

RustBelt's type system + **Func Spec**

$$\mathbf{T} \vdash \mathit{I} \dashv \mathtt{r}.\ \mathbf{T}' \rightsquigarrow \mathbf{\Phi}$$

typing judgment

func spec

predicate transformer

$$\lfloor \mathbf{T}' \rfloor \to \mathrm{Prop} \ni \mathit{post} \mapsto \mathit{pre} \in \lfloor \mathbf{T} \rfloor \to \mathrm{Prop}$$

RustHorn-style Representation of Rust Objects

Each **object** of *Rust* type `T` is represented by RustHorn as a **value** of *logic* sort $\lfloor \texttt{T} \rfloor$

RustHorn-style Representation of Rust Objects

Each **object** of *Rust* type `T` is represented by RustHorn as a **value** of *logic* sort $\lfloor \texttt{T} \rfloor$

$$\lfloor \texttt{int} \rfloor \triangleq \mathbb{Z} \qquad \lfloor \texttt{(T, U)} \rfloor \triangleq \lfloor \texttt{T} \rfloor \times \lfloor \texttt{U} \rfloor$$

$$\lfloor \texttt{\&mut T} \rfloor \triangleq \lfloor \texttt{T} \rfloor \times \underbrace{\lfloor \texttt{T} \rfloor}_{\textbf{prophecy}}$$

Interpreting Rust's Types & RustHorn in Iris

Interpreting Rust's Types & RustHorn in Iris

Semantics of the typing judgment in Iris $*$, roughly:

$$[\![\mathbf{T} \vdash \mathit{l} \dashv \mathtt{r}.\, \mathbf{T}']\!] \triangleq \{ [\![\mathbf{T}]\!] \}\ \mathit{l}\ \{ \mathtt{r}.\ [\![\mathbf{T}']\!] \}$$

RustBelt 

Interpreting Rust's Types & RustHorn in Iris

Semantics of the type-spec judgment in Iris ✻, roughly:

$$\llbracket \mathbf{T} \vdash \mathit{l} \dashv \mathtt{r}.\, \mathbf{T}' \rightsquigarrow \mathbf{\Phi} \rrbracket \quad \triangleq \quad \forall\, \mathit{post}.$$

$$\{\, \exists\, \overline{a}.\ \mathbf{\Phi}\ \mathit{post}\ \overline{a} \ast \llbracket \mathbf{T} \rrbracket(\overline{a}) \,\}\ \ \mathit{l}\ \ \{\, \mathtt{r}.\ \exists\, \overline{b}.\ \mathit{post}\ \overline{b} \ast \llbracket \mathbf{T}' \rrbracket(\overline{b}) \,\}$$

RustHornBelt  !

Example — Rust's Vec API

`Vec<T>` — Growable, heap-allocated array

Example — Rust's Vec API

`Vec<T>` — Growable, heap-allocated array

```
fn push(mv: &mut Vec<T>, a: T)
```

Example — Rust's Vec API

`Vec<T>` — Growable, heap-allocated array

```
fn push(mv: &mut Vec<T>, a: T)
```

```
fn index_mut(mv: &mut Vec<T>, i: int) -> &mut T
```

Example — Rust's Vec API

`Vec<T>` — Growable, heap-allocated array

```
fn push(mv: &mut Vec<T>, a: T)
```

```
fn index_mut(mv: &mut Vec<T>, i: int) -> &mut T
```

$$\lfloor \texttt{Vec<T>} \rfloor \triangleq \mathit{List}\ \lfloor \texttt{T} \rfloor$$

RustHorn's Spec for push

```
mv: &mut Vec<T>, a: T ⊢ push(mv, a) ⊣
```

RustHorn's Spec for push

$$\texttt{mv}\colon \texttt{\&mut Vec<T>},\ \texttt{a}\colon \texttt{T} \ \vdash\ \texttt{push}(\texttt{mv}, \texttt{a}) \ \dashv$$

$$\rightsquigarrow\ \lambda\, post, ((v, v'), a).\ v' = v +\!\!+ [a] \ \to\ post\ ()$$

RustHorn's Spec for push

$$\texttt{mv}\colon \texttt{\&mut Vec<T>},\ \texttt{a}\colon \texttt{T} \ \vdash\ \texttt{push}(\texttt{mv}, \texttt{a}) \ \dashv$$

$$\rightsquigarrow\ \lambda\, post, ((v, v'), a).\ v' = v +\!\!+ [a] \ \to\ post\ ()$$

prophecy

RustHorn's Spec for push

$$\texttt{mv: \&mut Vec<T>, a: T} \vdash \texttt{push(mv, a)} \dashv$$

$$\rightsquigarrow \lambda\, post, ((v, v'), a).\ \underbrace{v' = v +\!\!+ [a]}_{\textbf{resolution}} \to post\ ()$$

v': **prophecy**

RustHorn's Spec for index_mut

$$\texttt{mv}\colon \texttt{\&mut Vec<T>},\ \texttt{i}\colon \texttt{int} \vdash \texttt{index\_mut}(\texttt{mv}, \texttt{i}) \dashv \texttt{ma}.\ \texttt{ma}\colon \texttt{\&mut T}$$

RustHorn's Spec for index_mut

$$\texttt{mv}: \texttt{\&mut Vec<T>},\ \texttt{i}: \texttt{int} \vdash \texttt{index\_mut}(\texttt{mv}, \texttt{i}) \dashv \texttt{ma}.\ \texttt{ma}: \texttt{\&mut T}$$

$$\rightsquigarrow \quad \lambda\, post, ((v, v'), i).\quad 0 \le i < |v| \ \wedge$$

$$\ldots$$

RustHorn's Spec for index_mut

$$\texttt{mv}\colon \texttt{\&mut Vec<T>},\ \texttt{i}\colon \texttt{int} \vdash \texttt{index\_mut}(\texttt{mv}, \texttt{i}) \dashv \texttt{ma}.\ \texttt{ma}\colon \texttt{\&mut T}$$

$$\rightsquigarrow \quad \lambda\, post, ((v, v'), i).\ \ 0 \le i < |v| \ \wedge$$

$$\underbrace{v' = v\{i \coloneqq ?\}}_{\textbf{resolution}} \to post\,(v[i], ?)$$

RustHorn's Spec for index_mut

$$\texttt{mv}\colon \texttt{\&mut Vec<T>},\ \texttt{i}\colon \texttt{int} \vdash \texttt{index\_mut}(\texttt{mv}, \texttt{i}) \dashv \texttt{ma}.\ \texttt{ma}\colon \texttt{\&mut T}$$

$$\rightsquigarrow \quad \lambda\, post, ((v, v'), i).\ \ 0 \le i < |v| \ \wedge$$

$$\forall a'.\ v' = v\{i := a'\} \to post\,(v[i], a')$$

resolution **prophecy**

Parametric Prophecies

RustHorn  — **Mutable ref** as (a, a'), a' is **prophecy** 

← Semantically model this in separation logic Iris?

Parametric Prophecies

RustHorn  — **Mutable ref** as (a, a'), a' is **prophecy** 

← Semantically model this in separation logic Iris?

Prior work [Jung+ '20] — *Not flexible enough*

Parametric Prophecies

RustHorn — **Mutable ref** as (a, a'), a' is **prophecy**

← Semantically model this in separation logic Iris?

Prior work [Jung+ '20] — *Not flexible enough*

Our solution — **Clairvoyant monad** $\mathit{Clair}\ A \triangleq \mathit{Future} \to A$

i.e., reader monad over every possible **future** π

Semantics in Iris — Now Prophetic!

$$\llbracket \mathbf{T} \vdash I \dashv \mathtt{r}.\, \mathbf{T}' \rightsquigarrow \mathbf{\Phi} \rrbracket \triangleq \forall\, post.$$
$$\{\, \exists\, \overline{a}. \quad \mathbf{\Phi}\ post \quad \overline{a} \quad * \llbracket \mathbf{T} \rrbracket(\overline{a}\ ) \,\}$$
$$I \quad \{\, \mathtt{r}.\, \exists\, \overline{b}. \quad post \quad \overline{b} \quad * \llbracket \mathbf{T}' \rrbracket(\overline{b}\ ) \,\}$$

Semantics in Iris — Now Prophetic!

$$[\![\mathbf{T} \vdash I \dashv \mathtt{r}.\, \mathbf{T}' \rightsquigarrow \mathbf{\Phi}]\!] \triangleq \forall \hat{post}.$$

$$\{ \exists \overline{\hat{a}}.\, \langle \lambda\, \pi.\, \mathbf{\Phi}\, (\hat{post}\, \pi)\, \overline{\hat{a}}\, \pi \rangle * [\![\mathbf{T}]\!](\overline{\hat{a}\, \pi}) \}$$

$$I \quad \{ \mathtt{r}.\, \exists \overline{\hat{b}}.\, \langle \lambda\, \pi.\, \hat{post}\, \pi\, (\overline{\hat{b}\, \pi}) \rangle * [\![\mathbf{T}']\!](\overline{\hat{b}\, \pi}) \}$$

In the Paper, Also...

- More on type-spec system & **key Rust APIs** (`Vec`, `IterMut`, `Mutex`, ...) (§2)
- More on **parametric prophecies** & Model of **mutable ref** `&mut T` & **Proof sketch** of key type-spec rules (§3)
- **Coq** **mechanization** details & Automation **benchmarks** in Creusot (§4)