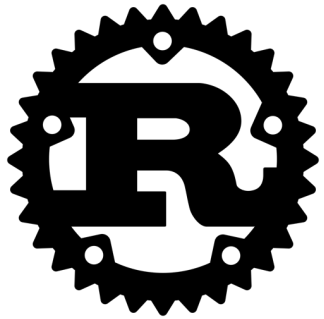
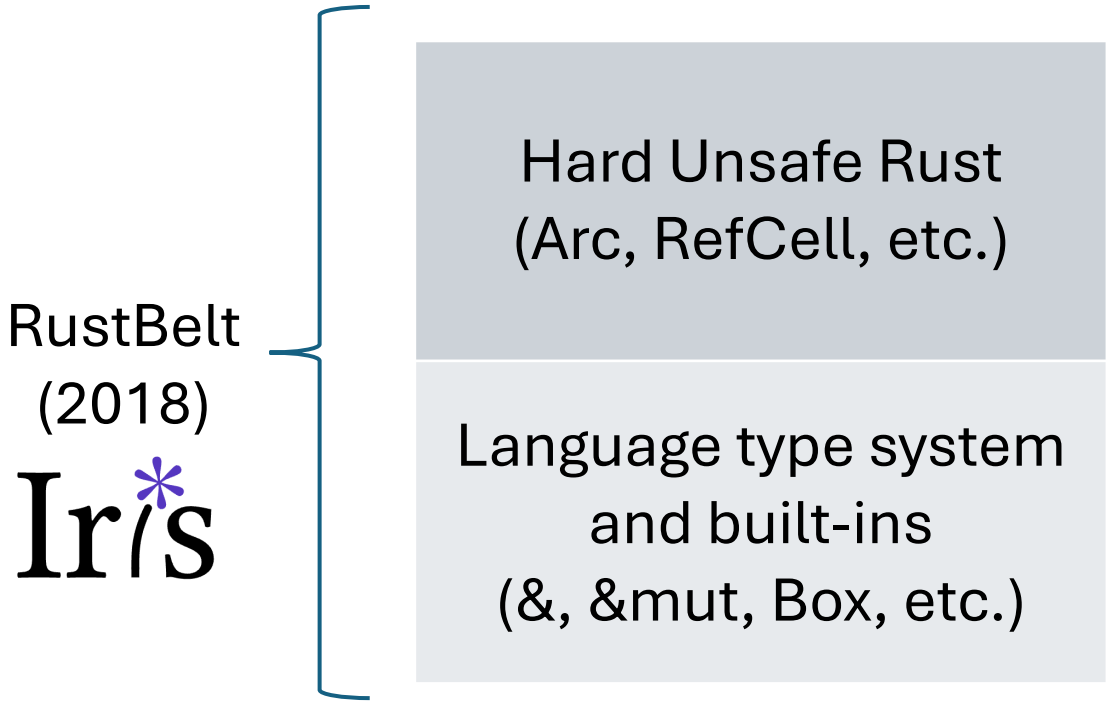
Verifying Rust code with ghost state and invariants

Travis Hance

Carnegie Mellon University



The state of Rust verification and Iris



The state of Rust verification and Iris

Creusot

Automated first-order logic

RustHornBelt
(2022)

RustBelt
(2018)

Iris

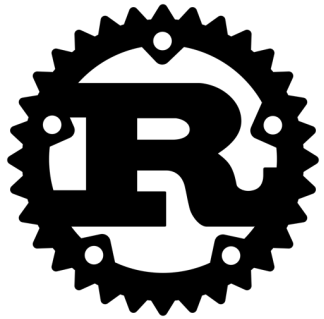
Safe Rust

Hard Unsafe Rust
(Arc, RefCell, etc.)

Language type system
and built-ins
(&, &mut, Box, etc.)

Can we also handle unsafe code with automated FOL techniques?

Sep. logic concepts like ghost state $\boxed{x}^{\gamma}$ and invariants $\boxed{I}^{\mathcal{N}}$ have proved essential here!



The state of Rust verification and Iris

Creusot

Automated first-order logic

RustHornBelt
(2022)

RustBelt
(2018)

Iris

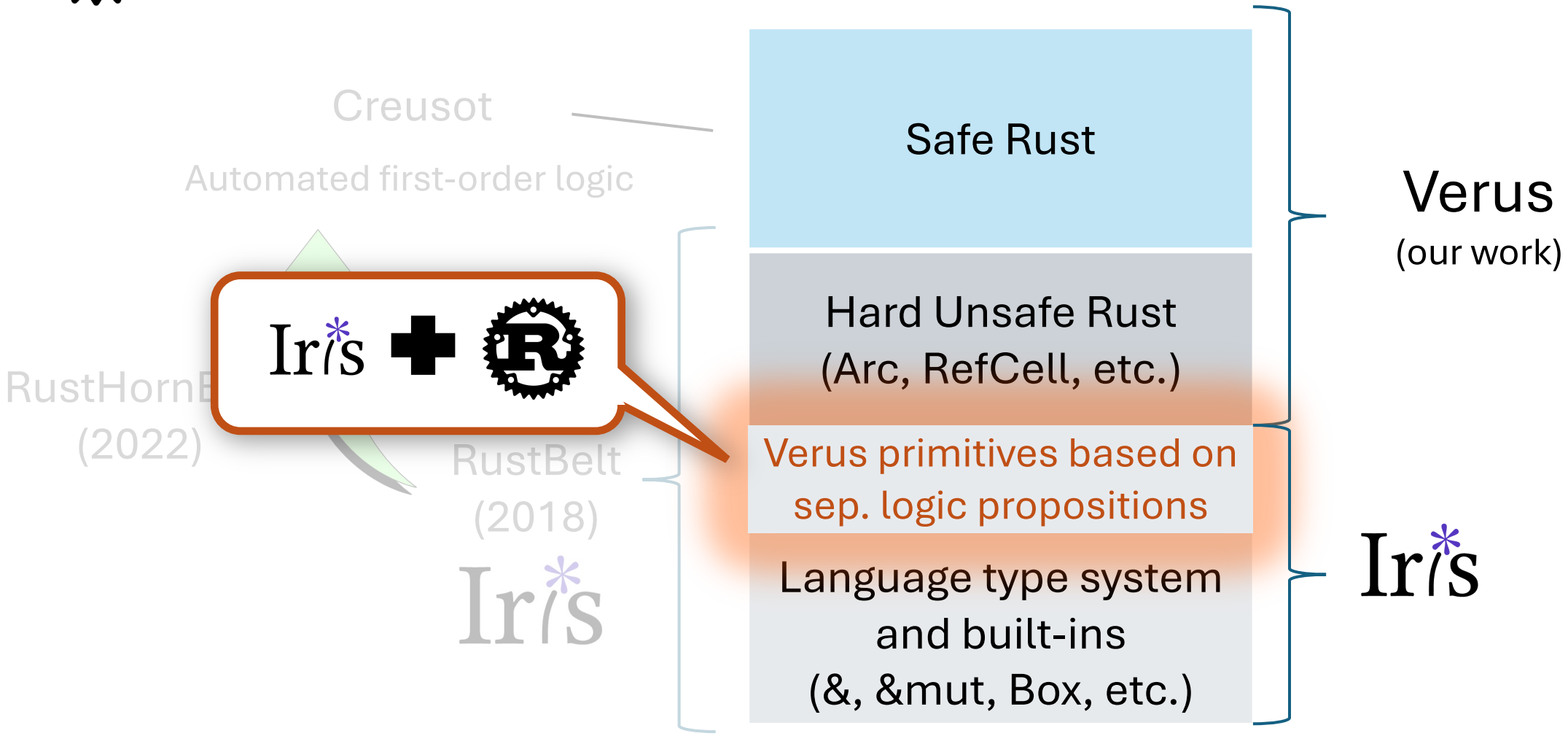
Safe Rust

Hard Unsafe Rust
(Arc, RefCell, etc.)

Language type system
and built-ins
(&, &mut, Box, etc.)

Verus
(our work)

The state of Rust verification and Iris



Verus Overview

```
fn insert<T>(vec: &mut Vec<T>, index: usize, element: T)
    requires 0 <= index && index <= old(vec).len()
    ensures
        vec.view() ==
            old(vec).view().subrange(0, index)
              .push(element)
              .concat(old(vec).view().subrange(index, old(vec).len()))
        //  vec ==
        //      old(vec) [ 0 .. index ]
        //         + [ element ]
        //         + old(vec) [ index .. ]
```

Verus allows Rust functions to be annotated with **pre-** and **post-conditions**

Verus includes:

- A specification language
- A proof language
- An SMT solver (Z3) to discharge proof obligations
- And …

Verus Special Primitives

Verus introduces **ghost objects** **in the Rust source code** to represent **separation logic propositions**

Verus's primitive ghost objects include:

- Memory permissions, analogue of $\ell \mapsto v$
- RA-based ghost state, analogue of $\boxed{x}^{\gamma}$
- Openable invariants, analogue of $\boxed{I}^{\mathcal{N}}$

Verus Applications

Using the ideas in this talk, we have used implemented and verified:

- Classic cell and pointer utilities
  - **Rc, Arc, RefCell, RwLock** (though not the official standard lib versions)
  - Has limitations around **&mut** references
- Concurrent code from the systems literature
  - Node replication algorithm (ASPLOS 2017) used in NrOS
  - A memory allocator based on Mimalloc (Microsoft 2019)

Verus Special Primitives

Verus introduces **ghost objects** **in the Rust source code** to represent **separation logic propositions**

Verus's primitive ghost objects include:

- Memory permissions, analogue of $\ell \mapsto v$
- RA-based ghost state, analogue of $\boxed{x}^{\gamma}$
- Openable invariants, analogue of $\boxed{I}^{\mathcal{N}}$

Focus of this talk

Verus Special Primitives

Verus introduces **ghost objects** **in the Rust source code** to represent **separation logic propositions**

Verus's primitive ghost objects include:

- Memory permissions, analogue of $\ell \mapsto v$
- RA-based ghost state, analogue of $\boxed{x}^{\gamma}$
- Openable invariants, analogue of $\boxed{I}^{\mathcal{N}}$

Resource Algebras

A *resource algebra* (RA) is a tuple $(M, \overline{\mathcal{V}} : M \to \mathit{Prop}, |-| : M \to M^?, (\cdot) : M \times M \to M)$ satisfying:

$$\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{(RA-ASSOC)}$$

$$\forall a, b.\ a \cdot b = b \cdot a \qquad \text{(RA-COMM)}$$

$$\forall a.\ |a| \in M \Rightarrow |a| \cdot a = a \qquad \text{(RA-CORE-ID)}$$

$$\forall a.\ |a| \in M \Rightarrow ||a|| = |a| \qquad \text{(RA-CORE-IDEM)}$$

$$\forall a, b.\ |a| \in M \land a \preccurlyeq b \Rightarrow |b| \in M \land |a| \preccurlyeq |b| \qquad \text{(RA-CORE-MONO)}$$

$$\forall a, b.\ \overline{\mathcal{V}}(a \cdot b) \Rightarrow \overline{\mathcal{V}}(a) \qquad \text{(RA-VALID-OP)}$$

where $M^? \triangleq M \uplus \{\bot\}$ with $a^? \cdot \bot \triangleq \bot \cdot a^? \triangleq a^?$

$$a \preccurlyeq b \triangleq \exists c \in M.\ b = a \cdot c \qquad \text{(RA-INCL)}$$

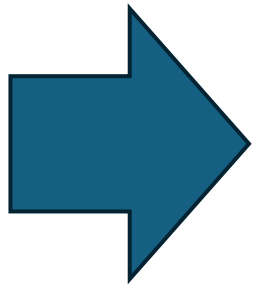
$$a \rightsquigarrow B \triangleq \forall c^? \in M^?.\ \overline{\mathcal{V}}(a \cdot c^?) \Rightarrow \exists b \in B.\ \overline{\mathcal{V}}(b \cdot c^?)$$

$$a \rightsquigarrow b \triangleq a \rightsquigarrow \{b\}$$

A *unital resource algebra* (uRA) is a resource algebra M with an element ε satisfying:

$$\overline{\mathcal{V}}(\varepsilon) \qquad \forall a \in M.\ \varepsilon \cdot a = a \qquad |\varepsilon| = \varepsilon$$

Fig. 3. Resource algebras.



$$\text{GHOST-ALLOC}\ \frac{\overline{\mathcal{V}}(a)}{\mathsf{True} \Rrightarrow_{\mathcal{E}} \exists \gamma.\ \boxed{a}^{\gamma}}$$

$$\text{GHOST-OP}\quad \boxed{a \cdot b}^{\gamma} \Leftrightarrow \boxed{a}^{\gamma} * \boxed{b}^{\gamma}$$

$$\text{GHOST-VALID}\quad \boxed{a}^{\gamma} \Rightarrow \overline{\mathcal{V}}(a)$$

$$\text{GHOST-UPDATE}\ \frac{a \rightsquigarrow B}{\boxed{a}^{\gamma} \Rrightarrow_{\mathcal{E}} \exists b \in B.\ \boxed{b}^{\gamma}}$$

Iris from the Ground Up (2018)

Resource Algebras

```
pub trait RA {
    // Definition of a (unital) Resource Algebra

    spec fn valid(self) -> bool;           // 𝒱
    spec fn op(self, other: Self) -> Self; // ·
    spec fn unit() -> Self;                // ϵ

    // Well-formedness conditions for a resource algebra

    proof fn closed_under_incl(a: Self, b: Self)
        requires Self::op(a, b).valid(),
        ensures a.valid();

    proof fn commutative(a: Self, b: Self)
        ensures Self::op(a, b) == Self::op(b, a);

    proof fn associative(a: Self, b: Self, c: Self)
        ensures Self::op(a, Self::op(b, c)) == Self::op(Self::op(a, b), c);

    proof fn op_unit(a: Self)
        ensures Self::op(a, Self::unit()) == a;

    proof fn unit_valid()
        ensures Self::valid(Self::unit());
}
```

User provides RA definition

And proves that it's a valid RA

Resource Algebras

```
// Ghost state representing x^γ
pub tracked type Resource<P>

impl<P: RA> Resource<P> {
    // Spec encoding of a Resource
    pub spec fn value(self) -> P;   // a
    pub spec fn loc(self) -> Loc;   // γ

    pub proof fn alloc(a: P) -> (tracked out: Self)
        requires
            a.valid(),
        ensures
            out.value() == a;
```

The caller/instantiater specifies a value *a*

Which has to be a "valid" element

$$\frac{\overline{\mathcal{V}(a)}}{\text{True} \Rrightarrow_{\mathcal{E}} \exists \gamma.\ \boxed{a}^{\gamma}}\ \text{GHOST-ALLOC}$$

And then they gain ownership of a resource

```
pub proof fn update(tracked a: Resource<P>, b_value: P)
      -> (tracked b: Resource<P>)
    requires
        frame_preserving_update(a.value(), b_value),
    ensures
        b.loc() == a.loc(),
        b.value() == b_value;
```

GHOST-UPDATE

$$\frac{a \rightsquigarrow b}{\boxed{a}^{\gamma} \Rrightarrow \boxed{b}^{\gamma}}$$

```
// Definition  a ⇝ b  ≜  ∀c. 𝒱(a · c) ⇒ 𝒱(b · c)
pub spec fn frame_preserving_update<P: RA>(a: P, b: P) -> bool {
    forall|c| P::op(a, c).valid() ==> P::op(b, c).valid()
}
```

```
pub proof fn join(tracked a: Resource<P>, tracked b: Resource<P>)
    -> (tracked a_op_b: Resource<P>)
  requires
      a.loc() == b.loc(),
  ensures
      a_op_b.loc() == a.loc(),
      a_op_b.value() == P::op(a.value(), b.value());

pub proof fn split(tracked a_op_b: Resource<P>, a_value: P, b_value: P)
    -> (tracked out: (Resource<P>, Resource<P>))
  requires
      a_op_b.value() == P::op(a_value, b_value),
  ensures
      out.0.loc() == a_op_b.loc(),
      out.1.loc() == a_op_b.loc(),
      out.0.value() == a_value,
      out.1.value() == b_value;
```

$$\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \vdash \boxed{a \cdot b}^{\gamma}$$

$$\boxed{a \cdot b}^{\gamma} \vdash \boxed{a}^{\gamma} * \boxed{b}^{\gamma}$$

We use a **shared reference** so the ghost token isn't destroyed

```
pub proof fn validate(tracked a: &Resource<P>)
  ensures
      a.value().valid();
```

GHOST-VALID

$$\boxed{a}^{\gamma} \Rightarrow \overline{\mathcal{V}}(a)$$

```
pub proof fn update(tracked a: Resource<P>, b_value: P)
      -> (tracked b: Resource<P>)
    requires
        frame_preserving_update(a.value(), b_value),
    ensures
        b.loc() == a.loc(),
        b.value() == b_value;
```

GHOST-UPDATE

$$\frac{a \rightsquigarrow b}{\boxed{a}^{\gamma} \Rrightarrow \boxed{b}^{\gamma}}$$

```
// Definition  a ⇝ b  ≜  ∀c. V(a · c) ⇒ V(b · c)
pub spec fn frame_preserving_update<P: RA>(a: P, b: P) -> bool {
    forall|c| P::op(a, c).valid() ==> P::op(b, c).valid()
}
```

```
pub proof fn update_with_shared(
    tracked a: Resource<P>, tracked x: &Resource<P>,
    b_value: P,
) -> (tracked b: Resource<P>)
    requires
        a.loc() == x.loc(),
        // (a · x) ⇝ (b · x)
        frame_preserving_update(
            P::op(a.value(), x.value()),
            P::op(b_value, x.value())),
    ensures
        b.loc() == a.loc(),
        b.value() == b_value;
```

GHOST-UPDATE-SHARED?

$$\frac{(a \cdot x) \rightsquigarrow (b \cdot x)}{\left(\text{“shared ref.”}\ \boxed{x}^{\gamma}\right) * \boxed{a}^{\gamma} \Rrightarrow \boxed{b}^{\gamma}}$$

But what is this?

```
pub proof fn update_with_shared(
    tracked a: Resource<P>, tracked x: &Resource<P>,
    b_value: P,
) -> (tracked b: Resource<P>)
    requires
        a.loc() == x.loc(),
        // (a · x) ⇝ (b · x)
        frame_preserving_update(
            P::op(a.value(), x.value()),
            P::op(b_value, x.value())),
    ensures
        b.loc() == a.loc(),
        b.value() == b_value;
```

But what is this?

GHOST-UPDATE-SHARED?

$$\frac{(a \cdot x) \rightsquigarrow (b \cdot x)}{\left(\text{“shared ref.”}\ \boxed{x}^{\gamma}\right) * \boxed{a}^{\gamma} \Rrightarrow \boxed{b}^{\gamma}}$$

It also turns out that we really need a primitive with this type signature:

```
pub proof fn shared_to_shared<'a>(tracked &'a Resource<P>) -> (tracked &'a Other)
    requires
        ???
    ensures
        ???
```

Consider an application …

When you acquire a read-lock you get this RwLockReadGuard

And you give it up to release the lock

```
impl<T> RwLock<T> {
    fn new(t: T) -> RwLock<T>;

    fn read<'a>(&'a self) -> RwLockReadGuard<'a, T>;
    fn write<'a>(&'a self) -> RwLockWriteGuard<'a, T>;

    fn into_inner(self) -> T;
}

impl<'a, T> RwLockReadGuard<'a, T> {
    fn deref<'b>(&'b self) -> &'b T;
    fn drop(self);
}

impl<'a, T> RwLockWriteGuard<'a, T> {
    fn deref_mut<'a>(&'b mut self) -> &'b mut T;
    fn drop(self);
}
```

While you have the RwLockReadGuard object, you can access the read-protected T object

It also turns out that we really need a primitive with this type signature:

```
pub proof fn shared_to_shared<'a>(tracked &'a Resource<P>) -> (tracked &'a Other)
    requires
        ???
    ensures
        ???
```

It also turns out that we really need a primitive with this type signature:

```
pub proof fn shared_to_shared<'a>(tracked &'a Resource<P>) -> (tracked &'a PointsTo)
    requires
        ???
    ensures
        ???
```

What should these conditions be?

Read-only state with **Leaf**

Leaf is our Iris library that defines a variation on the RA that answers these questions (OOPSLA 2023)

Leaf defines a variation on the Resource Algebra called a **Storage Protocol** whose laws describe the relationships among “temporarily shared state.”

Storage Protocols

A **storage protocol** consists of:
A *storage monoid*, that is, a partial commutative monoid $(S, \cdot, \mathcal{V})$, where,

$$\forall a.\ a \cdot \epsilon = a$$

$$\forall a, b.\ a \cdot b = b \cdot a$$

$$\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

$$\mathcal{V}(\epsilon)$$

$$\forall a, b.\ a \preceq b \wedge \mathcal{V}(b) \Rightarrow \mathcal{V}(a)$$

A *protocol monoid*, that is, a (total) commutative monoid $(P, \cdot)$, with an arbitrary predicate $C : P \to \text{Bool}$ and function $\mathcal{S} : P|_C \to S$ (i.e., the domain of $\mathcal{S}$ is restricted to the subset of P where C holds) where,

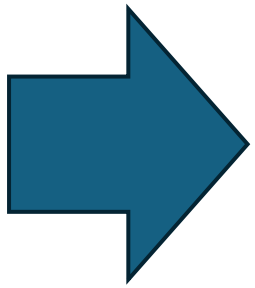
$$\forall a.\ a \cdot \epsilon = a$$

$$\forall a, b.\ a \cdot b = b \cdot a$$

$$\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

$$\forall a.\ C(a) \Rightarrow \mathcal{V}(\mathcal{S}(a))$$

Note that C (unlike $\mathcal{V}$) is *not* necessarily closed under $\preceq$.



SP-Guard
$$\frac{p \twoheadrightarrow s}{\text{sto}(\gamma, F) \vdash \langle p \rangle^{\gamma} \twoheadrightarrow\!\!\!\!\twoheadrightarrow_{\gamma} (\triangleright F(s))}$$

SP-Alloc
$$\frac{\text{RespectsComposition}(F) \quad C(p) \quad \mathcal{N} \text{ infinite}}{F(\mathcal{S}(p)) \Rrightarrow \exists \gamma.\ \text{sto}(\gamma, F) * \langle p \rangle^{\gamma} * (\gamma \in \mathcal{N})}$$

SP-Exchange
$$\frac{(p, s) \dashrightarrow\!\!\!\!\rightsquigarrow (p', s')}{\text{sto}(\gamma, F) \vdash (\triangleright F(s)) * \langle p \rangle^{\gamma} \Rrightarrow_{\gamma} (\triangleright F(s')) * \langle p' \rangle^{\gamma}}$$

SP-Sep
$$\langle p \cdot q \rangle^{\gamma} \dashv\vdash \langle p \rangle^{\gamma} * \langle q \rangle^{\gamma}$$

SP-Deposit
$$\frac{(p, s) \dashrightarrow\!\!\!\!\rightsquigarrow p'}{\text{sto}(\gamma, F) \vdash (\triangleright F(s)) * \langle p \rangle^{\gamma} \Rrightarrow_{\gamma} \langle p' \rangle^{\gamma}}$$

SP-Withdraw
$$\frac{p \rightsquigarrow\!\!\!\!\rightsquigarrow (p', s')}{\text{sto}(\gamma, F) \vdash \langle p \rangle^{\gamma} \Rrightarrow_{\gamma} (\triangleright F(s')) * \langle p' \rangle^{\gamma}}$$

SP-Unit
$$\text{sto}(\gamma, F) \vdash \langle \epsilon \rangle^{\gamma}$$

SP-Valid
$$\langle p \rangle^{\gamma} \vdash \exists q.\ C(p \cdot q)$$

SP-And
$$\frac{\forall t.\ ((x \preccurlyeq t) \wedge (y \preccurlyeq t) \wedge C(t)) \Rightarrow (z \preccurlyeq t)}{\langle x \rangle^{\gamma} \wedge \langle y \rangle^{\gamma} \vdash \langle z \rangle^{\gamma}}$$

SP-Update
$$\frac{p \rightsquigarrow p'}{\text{sto}(\gamma, F) \vdash \langle p \rangle^{\gamma} \Rrightarrow_{\gamma} \langle p' \rangle^{\gamma}}$$

SP-PointProp
$$\text{point}(\langle p \rangle^{\gamma})$$

SP-Exchange-Guarded-Nondeterministic
$$\frac{(p \cdot q, s) \dashrightarrow\!\!\!\!\rightsquigarrow \{(p' \cdot q, s') \mid (p', s') \in Z\} \quad \gamma \in \mathcal{E}}{\text{sto}(\gamma, F) * (G \twoheadrightarrow\!\!\!\!\twoheadrightarrow_{\mathcal{E}} \langle q \rangle^{\gamma}) \vdash (\triangleright F(s)) * \langle p \rangle^{\gamma} * G \Rrightarrow_{\mathcal{E}} \exists p', s'.\ ((p', s') \in Z) * (\triangleright F(s')) * \langle p' \rangle^{\gamma} * G}$$

Read-only state with **Leaf**

```
pub spec fn guards<K, V, P: Protocol<K, V>>(p: P, b: Map<K, V>) -> bool {
    forall |q: P| P::op(p, q).inv() ==> b.submap_of(P::op(p, q).interp())
}
```

```
pub proof fn guard<'a>(tracked x: &'a StorageResource<K, V, SP>), b: Map<K, V>)
    -> (tracked out: &'a Map<K, V>)
    requires
        guards(x.value(), b),
    ensures
        out == b;
```

SP-GUARD

$$\frac{p \twoheadrightarrow s}{\mathrm{sto}(\gamma, F) \vdash \langle p \rangle^{\gamma} \twoheadrightarrow\!\!\!\!\twoheadrightarrow_{\gamma} (\triangleright F(s))}$$

Verus Special Primitives

Verus's primitive ghost objects include:

- Memory permissions, analogue of $\ell \mapsto v$
- RA-based ghost state, analogue of $\boxed{x}^{\gamma}$
- Openable invariants, analogue of $\boxed{I}^{\mathcal{N}}$

Verus Special Primitives

Verus's primitive ghost objects include:

- Memory permissions, analogue of $\ell \mapsto v$
- RA-based ghost state, analogue of $\boxed{x}^{\gamma}$
  - ResourceAlgebra trait
  - StorageProtocol trait
  - StorageProtocol declaration language
- Openable invariants, analogue of $\boxed{I}^{\mathcal{N}}$

```
CountingPermissions {
    fields {
        #[sharding(storage_option)] pub stored: Option<T>,
        #[sharding(variable)] pub main_counter: Option<(nat,
        #[sharding(multiset)] pub read_ref: Multiset<T>,
    }

    init!{
        new() {
            init stored = None;
            init main_counter = None;
            init read_ref = Multiset::empty();
        }
    }

    transition!{
        writeable_to_readable(t: T) {
            require pre.main_counter.is_none();
            update main_counter = Some((0, t));
            deposit stored += Some(t);
        }
    }

    transition!{
        readable_to_writeable() {
            require let Some((count, t)) = pre.main_counter;
            require count == 0;
            update main_counter = None;
            withdraw stored -= Some(t);
        }
    }

    property!{
        read_ref_guards(t: T) {
            have read_ref >= { t };
            guard stored >= Some(t);
        }
    }

    transition!{
        new_ref() {
            require let Some((count, t)) = pre.main_counter;
            update main_counter = Some((count + 1, t));
            add read_ref += { t };
        }
    }
}
```

Verus Special Primitives

Verus's primitive ghost objects include:

- Memory permissions, analogue of $\ell \mapsto v$
- RA-based ghost state, analogue of $\boxed{x}^{\gamma}$
  - ResourceAlgebra trait
  - StorageProtocol trait
  - StorageProtocol declaration language
- Openable invariants, analogue of $\boxed{I}^{\mathcal{N}}$

```
CountingPermissions {
    fields {
        #[sharding(storage_option)] pub stored: Option<T>,
        #[sharding(variable)] pub main_counter: Option<(nat,
        #[sharding(multiset)] pub read_ref: Multiset<T>,
    }

    init!{
        new() {
            init stored = None;
            init main_counter = None;
            init read_ref = Multiset::empty();
        }
    }

    transition!{
        writeable_to_readable(t: T) {
            require pre.main_counter.is_none();
            update main_counter = Some((0, t));
            deposit stored += Some(t);
        }
    }

    transition!{
        readable_to_writeable() {
            require let Some((count, t)) = pre.main_counter;
            require count == 0;
            update main_counter = None;
            withdraw stored -= Some(t);
        }
    }

    property!{
        read_ref_guards(t: T) {
            have read_ref >= { t };
            guard stored >= Some(t);
        }
    }

    transition!{
        new_ref() {
            require let Some((count, t)) = pre.main_counter;
            update main_counter = Some((count + 1, t));
            add read_ref += { t };
        }
    }
}
```

Invariants

$$\frac{\{\triangleright P * Q_1\}\ e\ \{v.\ \triangleright P * Q_2\}_{\mathcal{E}\setminus\mathcal{N}} \quad \text{atomic}(e) \quad \mathcal{N} \subseteq \mathcal{E}}{\{\boxed{P}^{\mathcal{N}} * Q_1\}\ e\ \{v.\ \boxed{P}^{\mathcal{N}} * Q_2\}_{\mathcal{E}}}$$

Verus also has an **AtomicInvariant** type that you can "open" to gain temporary ownership of its ghost state contents:

```
tracked inv: &AtomicInvariant<_, T, _>

open_atomic_invariant!(inv, t => {
    // In this block we have ownership of `t: T`
});
```

Invariants and threads

We actually have **two** kinds of invariants:

AtomicInvariant	LocalInvariant
Thread-safe	Restricted to a single thread
Restricted to atomic actions	Can open for arbitrary length
Like $\boxed{P}^{\mathcal{N}}$	Like $\mathrm{NaInv}^{threadId.\mathcal{N}}(P)$

Invariants and threads

We actually have **two** kinds of invariants:

AtomicInvariant	LocalInvariant
Thread-safe	Restricted to a single thread
Restricted to atomic actions	Can open for arbitrary length
Like $\boxed{P}^{\mathcal{N}}$	Like $\mathrm{NaInv}^{threadId.\mathcal{N}}(P)$
Uses **Sync** marker trait	Is not **Sync**

Invariants and threads

We actually have **two** kinds of invariants:

AtomicInvariant	LocalInvariant
Thread-safe	Restricted to a single thread
Restricted to atomic actions	Can open for arbitrary length
Like $\boxed{P}^{\mathcal{N}}$	Like $\mathrm{NaInv}^{threadId.\mathcal{N}}(P)$
Uses **Sync** marker trait	Is not **Sync**
Useful for **Arc**	Useful for **Rc**

Invariants and masks

```
open_atomic_invariant!(&inv => t1 => {
    open_atomic_invariant!(&inv => t2 => {
        /* do something contradictory with t1 and t2 */
    });
});
```

```
error: possible invariant collision
  --> tinv.rs:25:29
   |
25 |     open_atomic_invariant!(&inv => t1 => {
   |                            ^^^ this invariant
26 |         open_atomic_invariant!(&inv => t2 => {
   |                                ^^^ might be the same as this invariant
```

Invariants and masks

Verus uses a **namespace** system similar to Iris's to track the invariants open at any program point

$$\frac{\{\triangleright P * Q_1\}\ e\ \{v.\ \triangleright P * Q_2\}_{\mathcal{E}\setminus\mathcal{N}} \quad \text{atomic}(e) \quad \mathcal{N} \subseteq \mathcal{E}}{\{\boxed{P}^{\mathcal{N}} * Q_1\}\ e\ \{v.\ \boxed{P}^{\mathcal{N}} * Q_2\}_{\mathcal{E}}}$$

Functions can also specify which invariants they open

```
pub fn example()
    requires ...
    ensures ...
    opens_invariants [ AWESOME_NAMESPACE ]
{
    ...
}
```

Invariants and “later”

$$\frac{\{\triangleright P * Q_1\}\ e\ \{v.\ \triangleright P * Q_2\}_{\mathcal{E}\setminus\mathcal{N}} \quad \text{atomic}(e) \quad \mathcal{N} \subseteq \mathcal{E}}{\{\boxed{P}^{\mathcal{N}} * Q_1\}\ e\ \{v.\ \boxed{P}^{\mathcal{N}} * Q_2\}_{\mathcal{E}}}$$

As in Iris, naïve invariants in Verus would be unsound together with certain higher-order features:

- For example, Rust’s **dyn** objects (existential types) can be used to recreate the “the invariant paradox” with naïve invariants

To resolve this, we are introducing a credit system similar to **later credits**

What about soundness?

How can we be sure that the Verus primitives and all their interactions with Rust's type system are sound?

Ongoing work: Semantic Type Soundness

RustHornBelt (PLDI 2022) shows how to use RustBelt's semantic models of types to prove the soundness of *type-spec judgments*

Can we apply this framework to prove the soundness of Verus's primitives?

Ongoing work: Semantic Type Soundness

Verus primitives are based closely on Iris propositions, so they have obvious semantic interpretations in Iris:

$$\lfloor \texttt{Resource<P>} \rfloor = GName \times \lfloor P \rfloor$$

$$[\![\texttt{Resource<P>}]\!].\mathrm{own}((\gamma, x), tid, []) = \boxed{x}^{\gamma}$$

We can also use **Leaf** to handle shared references (**&**) and their lifetimes.

Open question: Is this idea compatible with RustHornBelt's approach to *mutable* references (**&mut**)?

Conclusion

- Rust's ownership type system allows us to reason in Iris-style ways
- The interaction with Rust results in rich new structure:
  - Use shared references (**&**) instead of fractional permissions for read-only state
  - Ghost objects track key properties through **Send** and **Sync** marker traits
- Demonstrably powerful in conjunction with automated SMT reasoning

thance@andrew.cmu.edu

https://github.com/verus-lang/verus

https://github.com/secure-foundations/leaf