Verifying Rust code with ghost state and invariants

Travis Hance
Carnegie Mellon University
The state of Rust verification and Iris

RustBelt (2018)

Iris

Hard Unsafe Rust (Arc, RefCell, etc.)

Language type system and built-ins (&, &mut, Box, etc.)
The state of Rust verification and 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 $\bar{x} \gamma$ and invariants $\bar{I}^N$ have proved essential here!

**Iris**

RustBelt (2018)

RustHornBelt (2022)

Creusot

Automated first-order logic
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

- Safe Rust
- Hard Unsafe Rust (Arc, RefCell, etc.)
  - Verus primitives based on sep. logic propositions
  - Language type system and built-ins (&, &mut, Box, etc.)

Third-party libraries:
- RustBelt (2018)
- RustHornBelt (2022)
- Creusot

Automated first-order logic

Verus (our work)
Verus Overview

```rust
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 \( \ell \rightarrow \nu \)
- RA-based ghost state, analogue of \( \boxed{\chi}^\gamma \)
- Openable invariants, analogue of \( \boxed{I}^N \)
Verus Applications

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

- Classic cell and pointer utilities
  - \texttt{Rc, Arc, RefCell, RwLock} (though not the official standard lib versions)
  - Has limitations around \texttt{&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 \( \ell \rightarrow v \)
- RA-based ghost state, analogue of \( \chi^y \)
- Openable invariants, analogue of \( I^N \)

**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 $\ell \rightarrow v$
- **RA-based ghost state**, analogue of $\begin{array}{c} x \\ y \end{array}$
- **Openable invariants**, analogue of $I^N$
A resource algebra (RA) is a tuple $(M, \overline{V} : M \rightarrow \text{Prop}, |\cdot| : M \rightarrow M^2, (\cdot : M \times M \rightarrow M)$ satisfying:

\[
\forall a, b, c. (a \cdot b) \cdot c = a \cdot (b \cdot c) \quad \text{(RA-ASSOC)}
\]

\[
\forall a, b. a \cdot b = b \cdot a \quad \text{(RA-COMM)}
\]

\[
\forall a. |a| \in M \Rightarrow |a| \cdot a = a
\]

\[
\forall a. |a| \in M \Rightarrow |a|| = |a|
\]

\[
\forall a, b. |a| \in M \land a \leq b \Rightarrow |b| \in M \land |a| \leq |b|
\]

\[
\forall a, b. \overline{V}(a \cdot b) \Rightarrow \overline{V}(a)
\]

\[
\text{where } M^2 \triangleq M \uplus \{\bot\} \quad \text{with } a^2 \cdot \bot \triangleq \bot \cdot a^2 \triangleq a^2
\]

\[
a \bowtie b \triangleq \exists c \in M. b = a \cdot c
\]

\[
a \rightsquigarrow B \triangleq \forall c^2 \in M^2. \overline{V}(a \cdot c^2) \Rightarrow \exists b \in B. \overline{V}(b \cdot c^2)
\]

\[
a \rightsquigarrow b \triangleq a \rightsquigarrow \{b\}
\]

A unital resource algebra (uRA) is a resource algebra $M$ with an element $\varepsilon$ satisfying:

\[
\overline{V}(\varepsilon) \quad \forall a \in M. \varepsilon \cdot a = a \quad |\varepsilon| = \varepsilon
\]

**Fig. 3.** Resource algebras.
Resource Algebras

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

    spec fn valid(self) -> bool; // \forall
    spec fn op(self, other: Self) -> Self; // .
    spec fn unit() -> Self; // \epsilon

    // 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 $x^\gamma$

pub tracked type Resource<P>

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

    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
    And then they gain ownership of a resource

    GHOST_ALLOC
    $\tilde{\nu}(a)$
    True $\Rightarrow \exists \gamma. \overline{a}^{\gamma}$
**Proof Function `update`**

```plaintext
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;
```

**Definition**

\[ a \rightsquigarrow b \triangleq \forall c. \mathcal{V}(a \cdot c) \Rightarrow \mathcal{V}(b \cdot c) \]

**Spec Function `frame_preserving_update`**

```plaintext
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()
}
```
We use a **shared reference** so the ghost token isn’t destroyed

---

**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;

**pub proof fn validate**(tracked a: &Resource<P>)

**ensures**

a.value().valid();
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;

// Definition  $a \rightsquigarrow b \equiv \forall c. \ V(a \cdot c) \Rightarrow V(b \cdot c)$
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(),
  // $(a \cdot x) \rightsquigarrow (b \cdot x)$
  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

\[
\begin{array}{c}
  a \rightsquigarrow b \\
  \quad \Rightarrow \\
  b \rightsquigarrow b
\end{array}
\]

GHOST-UPDATE-SHARED?

\[
\begin{array}{c}
  (a \cdot x) \rightsquigarrow (b \cdot x) \\
  \quad \Rightarrow \\
  \quad \Rightarrow \\
  \quad b \rightsquigarrow b
\end{array}
\]
It also turns out that we really need a primitive with this type signature:

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

But what is this?
Consider an application ...

When you acquire a read-lock you get this `RwLockReadGuard` and you give it up to release the lock.

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

```rust
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<'b>(&'b mut self) -> &'b mut T;
    fn drop(self);
}
```
It also turns out that we really need a primitive with this type signature:

```rust
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)
```

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.”
A storage protocol consists of:

A storage monoid, that is, a partial commutative monoid \((S, \cdot, \mathcal{V})\), where,

\[
\forall a. a \cdot e = a
\]

\[
\forall a, b. a \cdot b = b \cdot a
\]

\[
\forall a, b, c. (a \cdot b) \cdot c = a \cdot (b \cdot c)
\]

\[
\mathcal{V}(e)
\]

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

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

\[
\forall a. a \cdot e = a
\]

\[
\forall a, b. a \cdot b = b \cdot a
\]

\[
\forall a, b, c. (a \cdot b) \cdot c = a \cdot (b \cdot c)
\]

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

Note that \(C\) (unlike \(\mathcal{V}\)) is not necessarily closed under \(\preceq\).
Read-only state with **Leaf**

```haskell
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**

\[
\text{sto}(\gamma, F) + \langle p \rangle^\gamma \Rightarrow \gamma (\Rightarrow F(s))
\]
Verus Special Primitives

Verus’s primitive ghost objects include:

- Memory permissions, analogue of $\ell \mapsto \nu$
- RA-based ghost state, analogue of $\vec{x}^y$
- Openable invariants, analogue of $\mathcal{I}^N$
Verus Special Primitives

Verus’s primitive **ghost objects** include:

- **Memory permissions**, analogue of \( \ell \mapsto \nu \)
- **RA-based ghost state**, analogue of \( \chi \mapsto \gamma \)
  - ResourceAlgebra trait
  - StorageProtocol trait
  - StorageProtocol declaration language
- **Openable invariants**, analogue of \( I \mapsto N \)

```rust
CountingPermissions {
    fields {
        #[sharding(storage_option)] pub stored: Option<T>,
        #[sharding(variable)] pub main_counter: Option<(nat, T)>,
        #[sharding(multiset)] pub read_ref: Multiset<T>,
    }
    init!{
        new() {
            init stored = None;
            init main_counter = None;
            init read_ref = Multiset::empty();
        }
    }
    transition!{
        writable_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 \(\ell \rightarrow \nu\)
- RA-based ghost state, analogue of \(\chi \rightarrow \gamma\)
  - ResourceAlgebra trait
  - StorageProtocol trait
  - StorageProtocol declaration language
- Openable invariants, analogue of \(I \rightarrow N\)

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!
    writable_to_readable(t: T) {
      require pre.main_counter.is_none();
      require count == 0;
      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

\[
\begin{align*}
\{\triangleright P \ast Q_1\} \in \{v. \triangleright P \ast Q_2\}_{E\setminus N} & \quad \text{atomic}(e) \quad N \subseteq E \\
\{P^N \ast Q_1\} \in \{v. P^N \ast Q_2\}_E
\end{align*}
\]

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

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

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

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

<table>
<thead>
<tr>
<th>AtomicInvariant</th>
<th>LocalInvariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread-safe</td>
<td>Restricted to a single thread</td>
</tr>
<tr>
<td>Restricted to atomic actions</td>
<td>Can open for arbitrary length</td>
</tr>
<tr>
<td>Like $\mathbf{p}^N$</td>
<td>Like $\text{NalInv}_{\text{threadId}}^N(P)$</td>
</tr>
</tbody>
</table>
Invariants and threads

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

<table>
<thead>
<tr>
<th>AtomicInvariant</th>
<th>LocalInvariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread-safe</td>
<td>Restricted to a single thread</td>
</tr>
<tr>
<td>Restricted to atomic actions</td>
<td>Can open for arbitrary length</td>
</tr>
<tr>
<td>Like $p^N$</td>
<td>Like $\text{Nalnv}_{\text{threadId}}^N(P)$</td>
</tr>
<tr>
<td>Uses <strong>Sync</strong> marker trait</td>
<td>Is not <strong>Sync</strong></td>
</tr>
</tbody>
</table>
Invariants and threads

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

<table>
<thead>
<tr>
<th>AtomicInvariant</th>
<th>LocalInvariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread-safe</td>
<td>Restricted to a single thread</td>
</tr>
<tr>
<td>Restricted to atomic actions</td>
<td>Can open for arbitrary length</td>
</tr>
<tr>
<td>Like $p^N$</td>
<td>Like $\text{Nalv}^{\text{threadId}.N}(P)$</td>
</tr>
<tr>
<td>Uses <strong>Sync</strong> marker trait</td>
<td>Is not <strong>Sync</strong></td>
</tr>
<tr>
<td>Useful for <strong>Arc</strong></td>
<td>Useful for <strong>Rc</strong></td>
</tr>
</tbody>
</table>
Invariants and masks

```rust
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
```
  open_atomic_invariant!(&inv => t1 => {
      ^^^ this invariant
```
```
  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.

\[
\{ \triangleright P \land Q_1 \} \in \{ v. \triangleright P \land Q_2 \}_{\mathcal{N}} \quad \text{atomic}(e) \quad \mathcal{N} \subseteq \mathcal{E}
\]

\[
\{ P^\mathcal{N} \land Q_1 \} \in \{ v. P^\mathcal{N} \land Q_2 \}_{\mathcal{E}}
\]

Functions can also specify which invariants they open:

```pub fn example()
    requires ...
    ensures ...
    opens_invariants [ AWESOME_NAMESPACE ]
{
    ...
}
```
Invariants and “later”

\[
\{\vartriangleright P \ast Q_1\} \in \{v. \vartriangleright P \ast Q_2\}_{E \setminus N} \quad \text{atomic}(e) \quad N \subseteq E
\]

\[
\{P^N \ast Q_1\} \in \{v. P^N \ast Q_2\}_{E}
\]

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:

\[
\begin{align*}
[\text{Resource}<P>] & \equiv GName \times [P] \\
[\text{Resource}<P>].\text{own}((y, x), \text{tid}, []) & \equiv \boxed{x}^Y
\end{align*}
\]

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