Verus: Verifying Rust Programs Using Linear Ghost Types

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, Chris Hawblitzel

Sankalp Gambhir

September 16, 2025

Rust and Ownership

Rust has ownership and borrowing built into its type system, and it appears in a few different forms to the user:

Rust and Ownership

Rust has ownership and borrowing built into its type system, and it appears in a few different forms to the user:

Instance	Mutability	Semantics

Rust and Ownership

Rust has ownership and borrowing built into its type system, and it appears in a few different forms to the user:

Instance	Mutability	Semantics
`x: T`	✗	Affine

Rust and Ownership

Rust has ownership and borrowing built into its type system, and it appears in a few different forms to the user:

Instance	Mutability	Semantics
`x: T`	✗	Affine
`mut x: T`	✓	Affine

Rust and Ownership

Rust has ownership and borrowing built into its type system, and it appears in a few different forms to the user:

Instance	Mutability	Semantics
`x: T`	✗	Affine
`mut x: T`	✓	Affine
`x: &T`	✗	Shared with unrestricted (immutable) use

Rust and Ownership

Rust has ownership and borrowing built into its type system, and it appears in a few different forms to the user:

Instance	Mutability	Semantics
`x: T`	✗	Affine
`mut x: T`	✓	Affine
`x: &T`	✗	Shared with unrestricted (immutable) use
`x: &mut T`	✓	Exclusive reference, affine

Rust and Ownership

Rust has ownership and borrowing built into its type system, and it appears in a few different forms to the user:

Instance	**Mutability**	**Semantics**
`x: T`	✗	Affine
`mut x: T`	✓	Affine
`x: &T`	✗	Shared with unrestricted (immutable) use
`x: &mut T`	✓	Exclusive reference, affine

Others: `&a' T`, `&mut a' T`, `*const T`, `*mut T`, `Box<T>`, `Arc<T>`, `Mutex<T>`

Pop Quiz

Which of the following Rust programs should compile?

```rust
fn main() {
    let mut x = 5;
    let r1 = &mut x;
    let r2 = &mut x;
    println!("{} {}", r1, r2);
}
```

```rust
fn main() {
    let mut x = 5;
    let r1 = &mut x;
    let r2 = &mut x;
    println!("{} {}", r1, r2);
}
```

✗

```rust
fn main() {
    let s = String::from("hello");
    let s2 = s;
    println!("{}", s);
}
```

```rust
fn main() {
    let s = String::from("hello");
    let s2 = s;
    println!("{}", s);
}
```

✗

```rust
fn say_what() -> &String {
    let s = String::from("hello");
    &s
}
```

```
fn say_what() -> &String {
    let s = String::from("hello");
    &s
}
```

✗

```rust
fn main() {
    let mut x = 5;
    let r1 = &x;
    let r2 = &mut x;
    println!("{}", r1);
}
```

```rust
fn main() {
    let mut x = 5;
    let r1 = &x;
    let r2 = &mut x;
    println!("{}", r1);
}
```

✗

```rust


fn main() {
    let mut x = 5;
    {
        let r1 = &mut x;
        *r1 += 1;
    }
    let r2 = &mut x;
    *r2 += 1;
    println!("{}", x);
}
```

```rust


fn main() {
    let mut x = 5;
    {
        let r1 = &mut x;
        *r1 += 1;
    }
    let r2 = &mut x;
    *r2 += 1;
    println!("{}", x);
}
```

✓

```rust


fn main() {
    let mut x = 5;
    {
        let r1 = &mut x;
        *r1 += 1;
    }
    let r2 = &mut x;
    *r2 += 1;
    println!("{}", x);
}
```

✓

On the other hand...

```
struct Boxed(i32);

impl Boxed {
  fn inc(&mut self) -> () {
    self.0 += 1;
  }
  fn consume(self) -> () {
    ()
  }
  fn inc_immutable(self) -> Self {
    Boxed(self.0 + 1)
  }
}

fn main() {
  let x = Boxed(5); // immutable struct created
  let mut x = x.inc_immutable(); // consumed and bound to a new
    mutable variable
  x.consume(); // moved and consumed
  x.inc(); // error: use after move
}
```

What Verus is (and what it isn't)

Verus:

✓ is a verifier for Rust programs,

What Verus is (and what it isn't)

Verus:

- ✓ is a verifier for Rust programs,
- ✓ is a framework for writing specifications and proofs in Rust,

What Verus is (and what it isn't)

Verus:

- ✓ is a verifier for Rust programs,
- ✓ is a framework for writing specifications and proofs in Rust,
- ✓ provides SMT encoding for said code and proofs, and

What Verus is (and what it isn't)

Verus:

- ✓ is a verifier for Rust programs,
- ✓ is a framework for writing specifications and proofs in Rust,
- ✓ provides SMT encoding for said code and proofs, and
- ✓ provides semantics for ghost code and its elimination,

What Verus is (and what it isn't)

Verus:

- ✓ is a verifier for Rust programs,
- ✓ is a framework for writing specifications and proofs in Rust,
- ✓ provides SMT encoding for said code and proofs, and
- ✓ provides semantics for ghost code and its elimination,

What Verus is (and what it isn't)

Verus:

- ✓ is a verifier for Rust programs,
- ✓ is a framework for writing specifications and proofs in Rust,
- ✓ provides SMT encoding for said code and proofs, and
- ✓ provides semantics for ghost code and its elimination,

but:

- ✗ isn't a verifier or model for unsafe Rust, and

What Verus is (and what it isn't)

Verus:

- ✓ is a verifier for Rust programs,
- ✓ is a framework for writing specifications and proofs in Rust,
- ✓ provides SMT encoding for said code and proofs, and
- ✓ provides semantics for ghost code and its elimination,

but:

- ✗ isn't a verifier or model for unsafe Rust, and
- ✗ isn't even a model for safe Rust

What Verus is (and what it isn't)

Verus:

- ✓ is a verifier for Rust programs,
- ✓ is a framework for writing specifications and proofs in Rust,
- ✓ provides SMT encoding for said code and proofs, and
- ✓ provides semantics for ghost code and its elimination,

but:

- ✗ isn't a verifier or model for unsafe Rust, and
- ✗ isn't even a model for safe Rust

What Verus is (and what it isn't)

Verus:

- ✓ is a verifier for Rust programs,
- ✓ is a framework for writing specifications and proofs in Rust,
- ✓ provides SMT encoding for said code and proofs, and
- ✓ provides semantics for ghost code and its elimination,

but:

- ✗ isn't a verifier or model for unsafe Rust, and
- ✗ isn't even a model for safe Rust

Verus builds on the existing type and borrow checking done by the Rust compiler.

What Verus is (and what it isn't)

Verus:

- ✓ is a verifier for Rust programs,
- ✓ is a framework for writing specifications and proofs in Rust,
- ✓ provides SMT encoding for said code and proofs, and
- ✓ provides semantics for ghost code and its elimination,

but:

- ✗ isn't a verifier or model for unsafe Rust, and
- ✗ isn't even a model for safe Rust

Verus builds on the existing type and borrow checking done by the Rust compiler. A program is safe iff it passes the Rust compiler *and* Verus.

Writing a simple program with Verus + SMT encoding

	`spec`	`proof`	`exec`
Call spec	✓	✓	✓
Call proof	✗	✓	✓
Call exec	✗	✗	✓

	`spec`	`proof`	`exec`
Call spec	✓	✓	✓
Call proof	✗	✓	✓
Call exec	✗	✗	✓
Compiled to machine code	✗	✗	✓

	`spec`	`proof`	`exec`
Call spec	✓	✓	✓
Call proof	✗	✓	✓
Call exec	✗	✗	✓
Compiled to machine code	✗	✗	✓
Mutation	✗	✓	✓

	`spec`	`proof`	`exec`
Call spec	✓	✓	✓
Call proof	✗	✓	✓
Call exec	✗	✗	✓
Compiled to machine code	✗	✗	✓
Mutation	✗	✓	✓
"SMT effects"	✗	✓	✓

	`spec`	`proof`	`exec`
Call spec	✓	✓	✓
Call proof	✗	✓	✓
Call exec	✗	✗	✓
Compiled to machine code	✗	✗	✓
Mutation	✗	✓	✓
"SMT effects"	✗	✓	✓
Borrow-checking	✗	✓	✓

	`spec`	`proof`	`exec`
Call spec	✓	✓	✓
Call proof	✗	✓	✓
Call exec	✗	✗	✓
Compiled to machine code	✗	✗	✓
Mutation	✗	✓	✓
"SMT effects"	✗	✓	✓
Borrow-checking	✗	✓	✓
SMT Types	✓	✓	✗
Pre/post-conditions	✗	✓	✓

Mutexes in Rust

```rust
const N: usize = 10;
let data = Arc::new(Mutex::new(0));

let (tx, rx) = channel();
for _ in 0..N {
    let (data, tx) = (Arc::clone(&data), tx.clone());
    thread::spawn(move || {
        let mut data = (*data).lock().unwrap();
        *data += 1;
        if *data == N {
            tx.send(()).unwrap();
        }
    });
}

rx.recv().unwrap();
```

Interior Mutability and the Myth of Safe Rust

```rust
impl<T> Mutex<T> {
  // ...

  pub fn lock(&self) -> LockResult<MutexGuard<'_, T>> {
    unsafe {
        self.inner.lock();
        MutexGuard::new(self)
    }
  }
}

```

Unsafe code in Verus

What does Verus do with this code?

Unsafe code in Verus

What does Verus do with this code? Nothing.

Unsafe code in Verus

What does Verus do with this code? Nothing.

Verus does not attempt to (directly) verify unsafe code.

Interior Mutability via Linear Ghost Permissions

Instead, it provides primitives that allow proving safety of code like the Mutex example before.

Interior Mutability via Linear Ghost Permissions

Instead, it provides primitives that allow proving safety of code like the Mutex example before. This is made possible in part by the fact that proof code is linearity checked.

```
open_local_invariant!(&self.perm_inv => perm => {
  r = self.pcell.replace(&mut perm, val);
});
```

Interior Mutability via Linear Ghost Permissions

Instead, it provides primitives that allow proving safety of code like the Mutex example before. This is made possible in part by the fact that proof code is linearity checked.

```
    open_local_invariant!(&self.perm_inv => perm => {
      r = self.pcell.replace(&mut perm, val);
    });
```

Once open, the permission is used like a capability, and cannot be copied or leaked due to borrow checking. Double-opening is guarded against by the verifier specifically.

Interior Mutability via Linear Ghost Permissions

Instead, it provides primitives that allow proving safety of code like the Mutex example before. This is made possible in part by the fact that proof code is linearity checked.

```
    open_local_invariant!(&self.perm_inv => perm => {
      r = self.pcell.replace(&mut perm, val);
    });

```

Once open, the permission is used like a capability, and cannot be copied or leaked due to borrow checking. Double-opening is guarded against by the verifier specifically.

These objects can be verified as destroyed by consuming the permission object (see PPtr).

On (Local) Invariants

It is called an invariant because it additionally ensures that the value stored in the cell satisfies the type invariant:

```
impl<K, V, Pred: InvariantPredicate<K, V>> LocalInvariant<K, V, Pred>
  pub proof fn new(k: K, tracked v: V, ns: int) -> tracked i :
    LocalInvariant<K, V, Pred>
     requires
       Pred::inv(k, v),
     ensures
       i.constant() == k,
       i.namespace() == ns,

```

The invariant checks are in `proof` mode and are erased. `InvariantPredicate` is a typeclass just providing the `inv` function.

Thanks! Questions?

If we have more time we'll discuss and look at how certain things are implemented in the (Rust or Verus) standard library or play with the verifier.