Verifying Safe Rust Clients of Internally-Unsafe Libraries

federico.poli@inf.ethz.ch

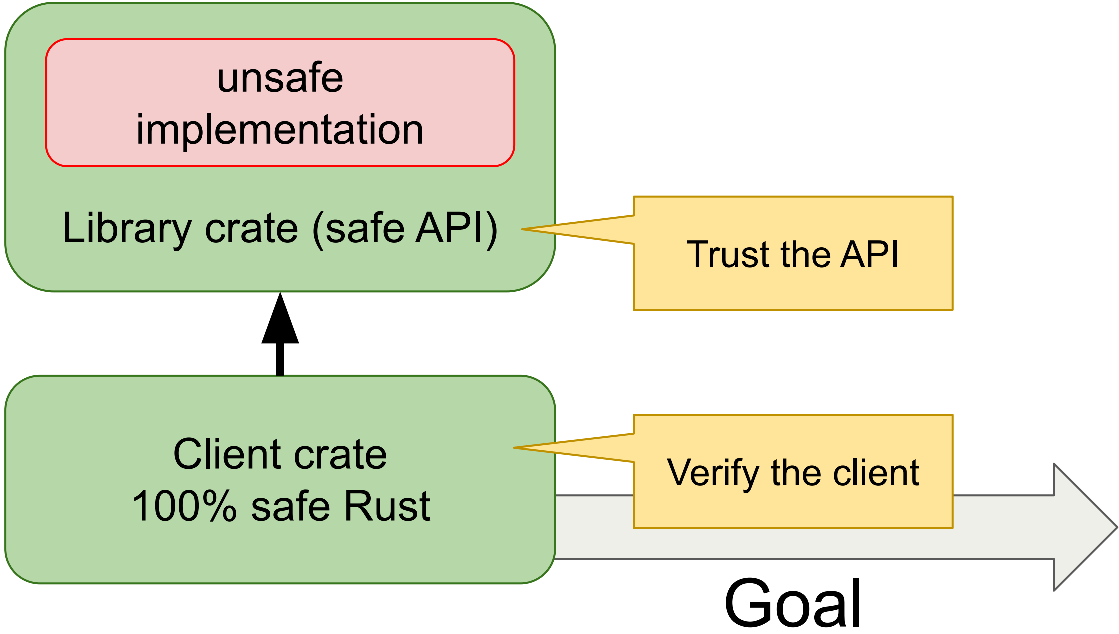
Federico Poli Peter Müller

Alexander J. Summers

ETH zürich

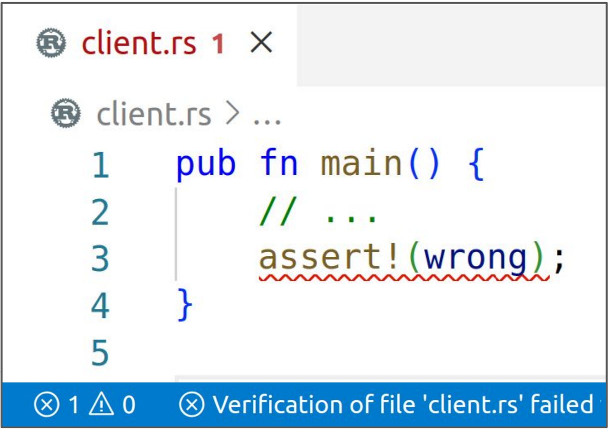
THE UNIVERSITY OF BRITISH COLUMBIA

Rust Verification Workshop 2023



P∗rust→∗i

http://prusti.org

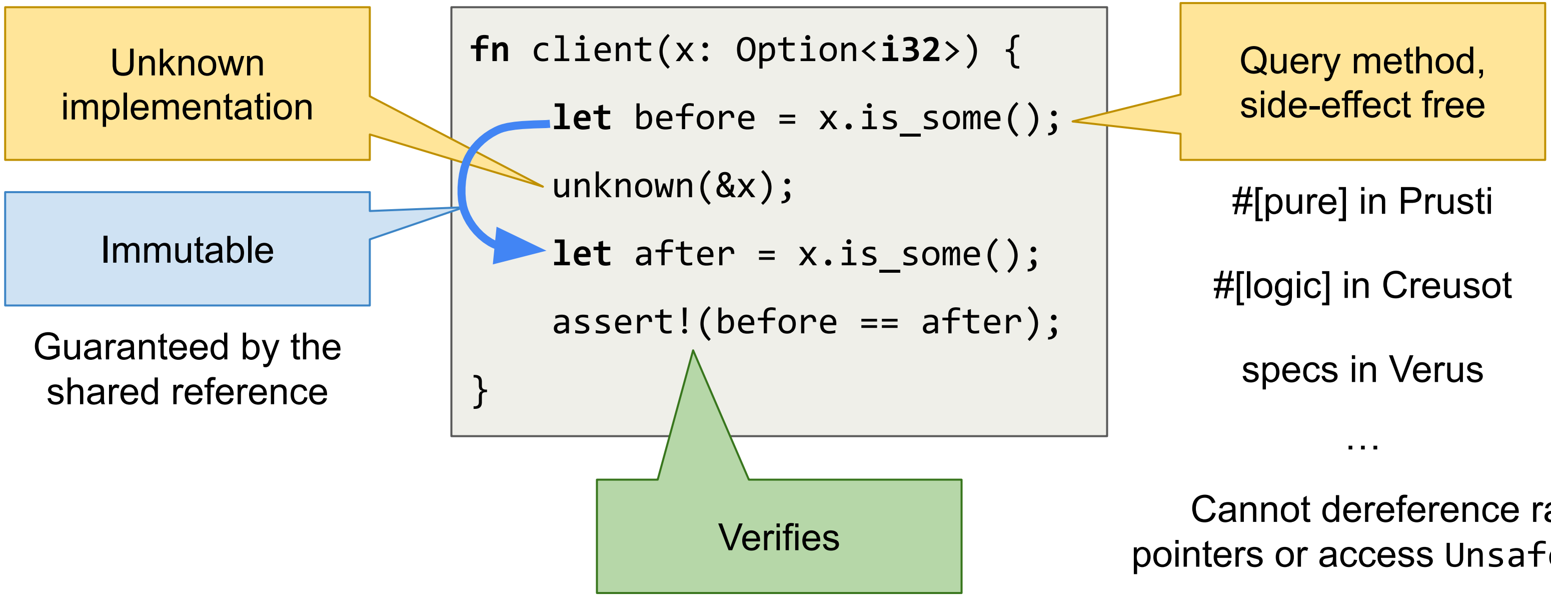


```
pub fn main() {
    // ...
    assert!(wrong);
}
```

⊗ 1 ⚠ 0 ⊗ Verification of file 'client.rs' failed

Deductive verifier for Rust

Background: fully safe code



```
fn client(x: Option<i32>) {
    let before = x.is_some();
    unknown(&x);
    let after = x.is_some();
    assert!(before == after);
}
```

Guaranteed by the shared reference

#[pure] in Prusti

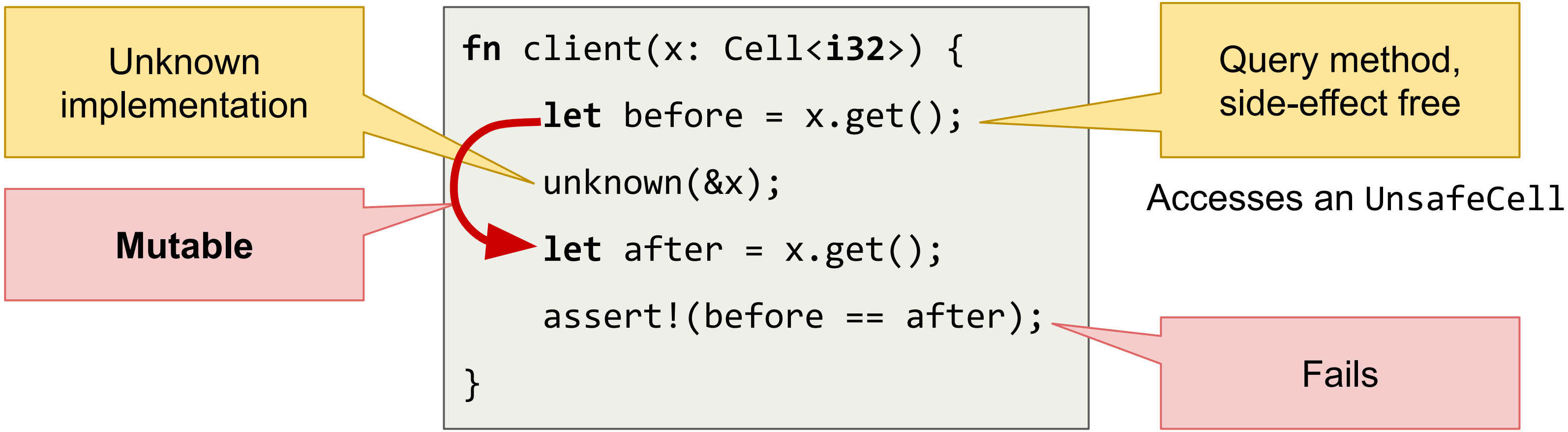
#[logic] in Creusot

specs in Verus

…

Cannot dereference raw pointers or access `UnsafeCell`

Now: libraries implemented with unsafe code



```
fn client(x: Cell<i32>) {
    let before = x.get();
    unknown(&x);
    let after = x.get();
    assert!(before == after);
}
```

Unknown implementation

Mutable

Query method, side-effect free

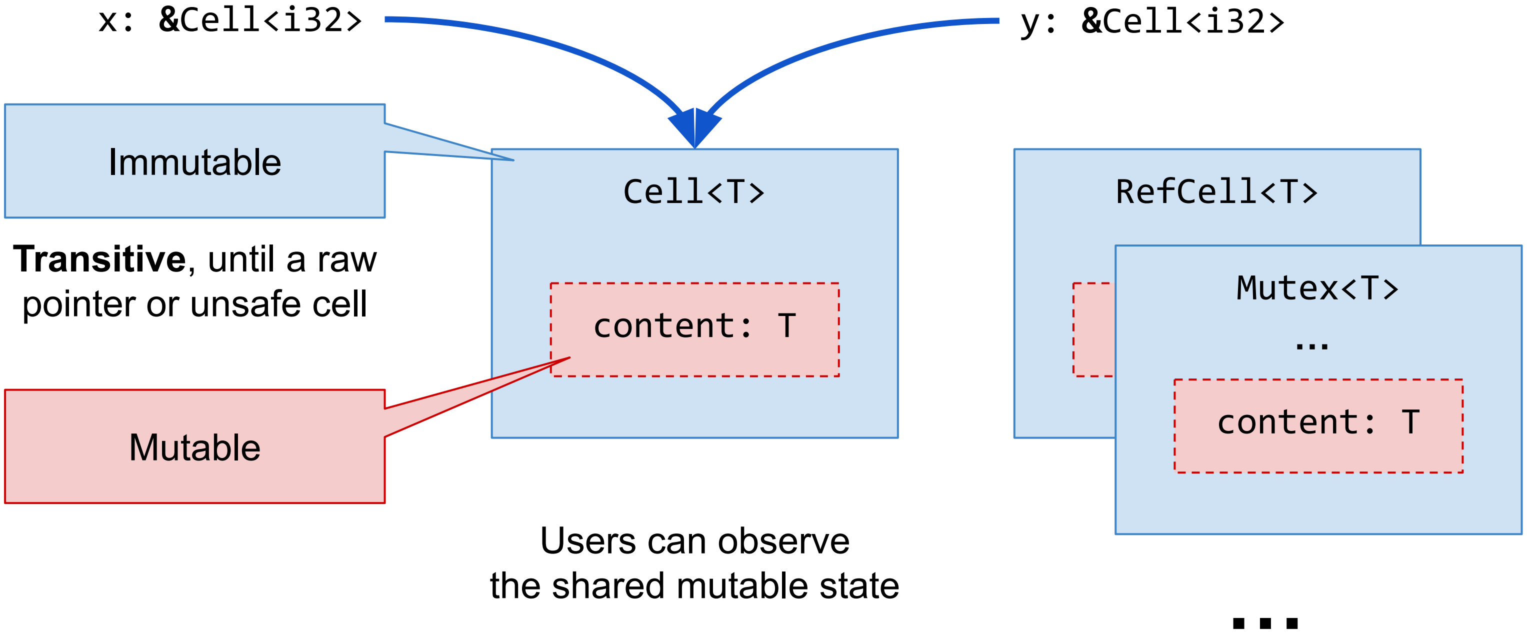
Accesses an `UnsafeCell`

Fails

`Cell` has **interior mutability.**
Query methods have weaker semantics.

Interior mutability

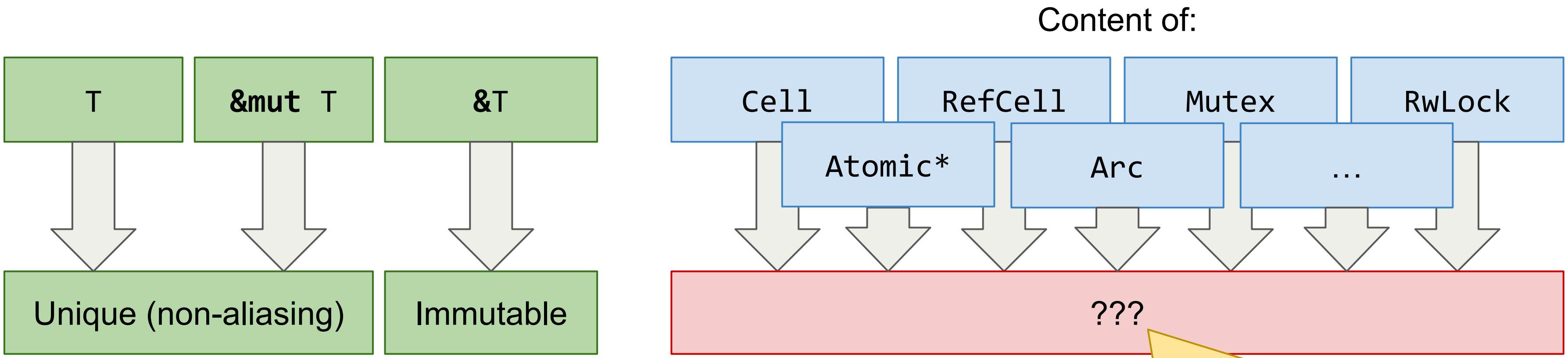
In this talk:
implemented with `UnsafeCell` or raw pointers



Transitive, until a raw pointer or unsafe cell

Users can observe the shared mutable state

Capabilities (non zero-size types)



T

&mut T

&T

Unique (non-aliasing)

Immutable

Content of:

Cell

RefCell

Mutex

RwLock

Atomic*

Arc

...

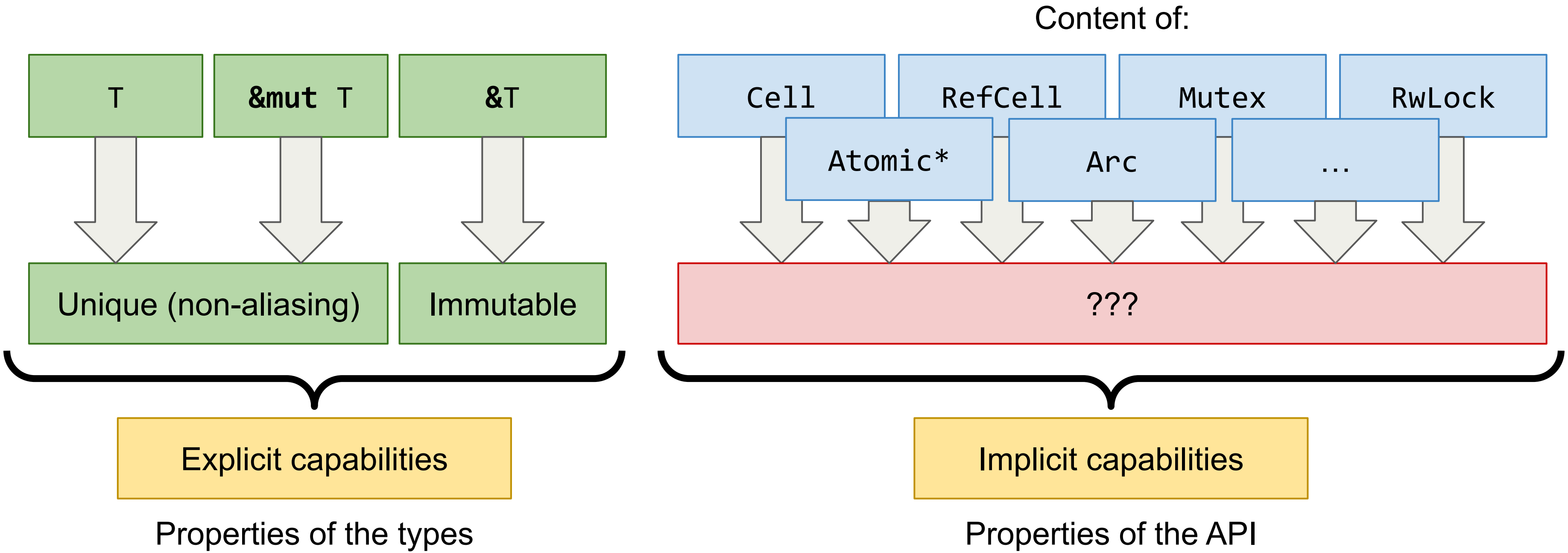
???

This talk:

They have capabilities

- Provided by auxiliary types (e.g. `MutexGuard`, `Ref`…)
- Depending on the state (e.g. reference count)
- Not always expressible using Rust types

Capabilities (non zero-size types)



Content of:

`T` `&mut T` `&T`

Unique (non-aliasing)

Immutable

`Cell` `RefCell` `Mutex` `RwLock` `Atomic*` `Arc` `...`

???

Explicit capabilities

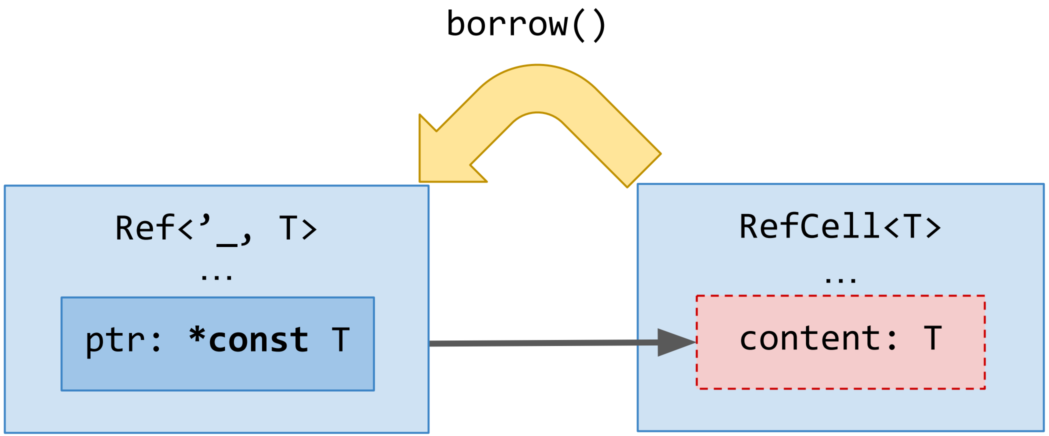
Properties of the types

Implicit capabilities

Properties of the API

What are them? How to declare them?
How can tools use them?

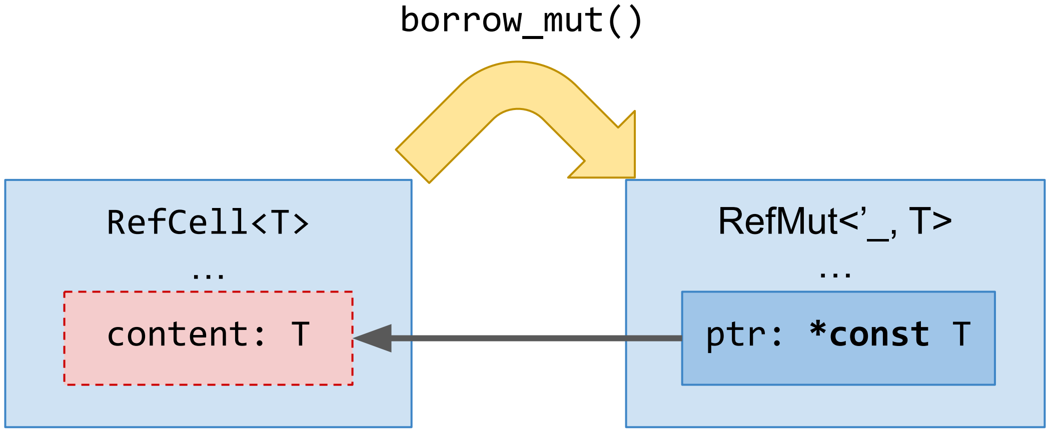
Example: RefCell



Read-only API

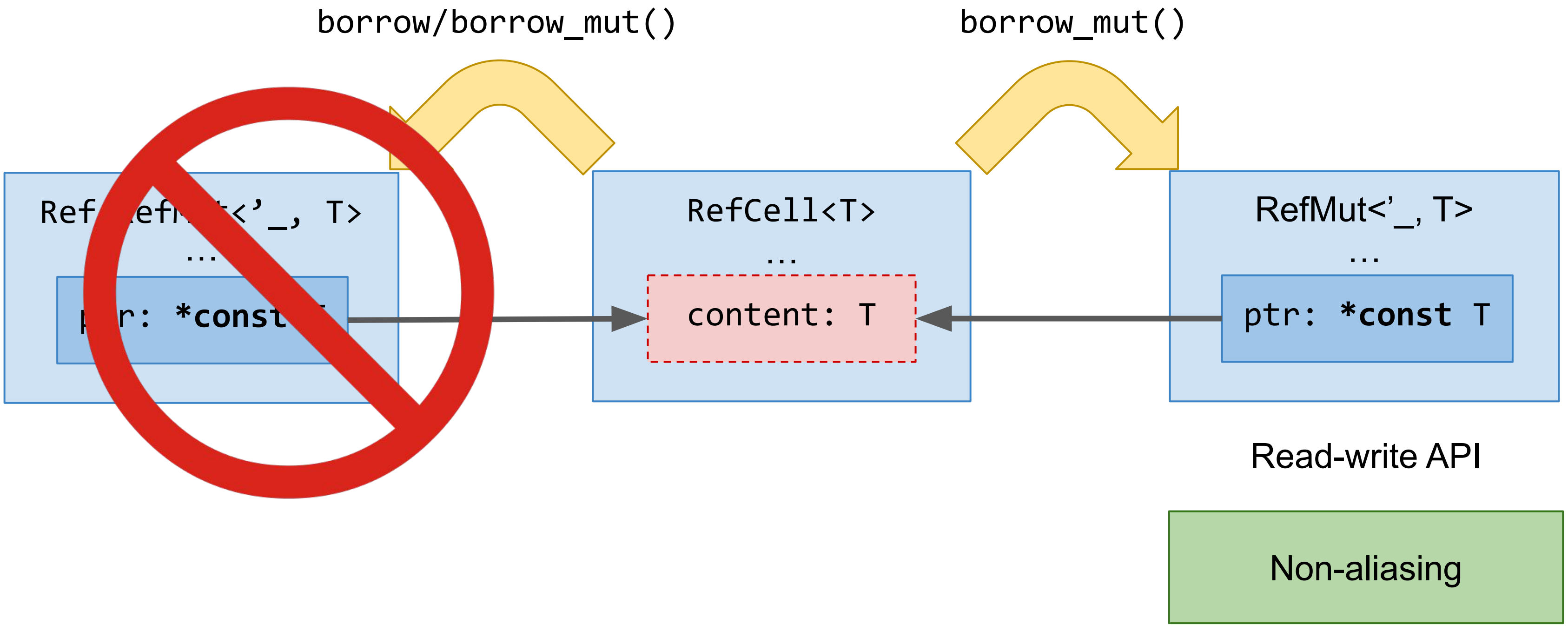
Immutability

Example: RefCell



Read-write API

Example: RefCell



Implicit capability annotation

Given &Ref

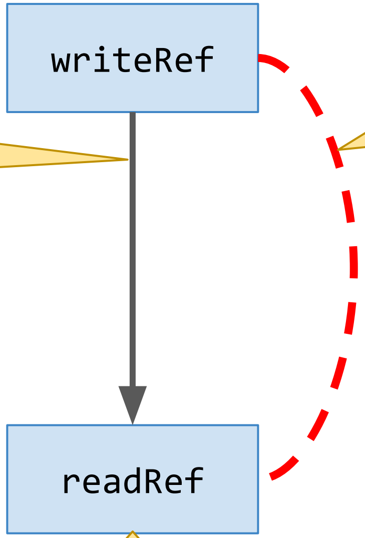
… one can always obtain a & to …

the content
(specified by address)

```
#[owns(&self => readRef(self.data_ptr()))]
impl<'b, T> Ref<'b, T> {}
```

```
#[owns(&mut self => writeRef(self.data_ptr()))]
impl<'b, T> RefMut<'b, T> {}
```

Properties of implicit capabilities



Implication
(exchange one for the other)

```
let x: &mut T = …
let y: &T = &*x;
```

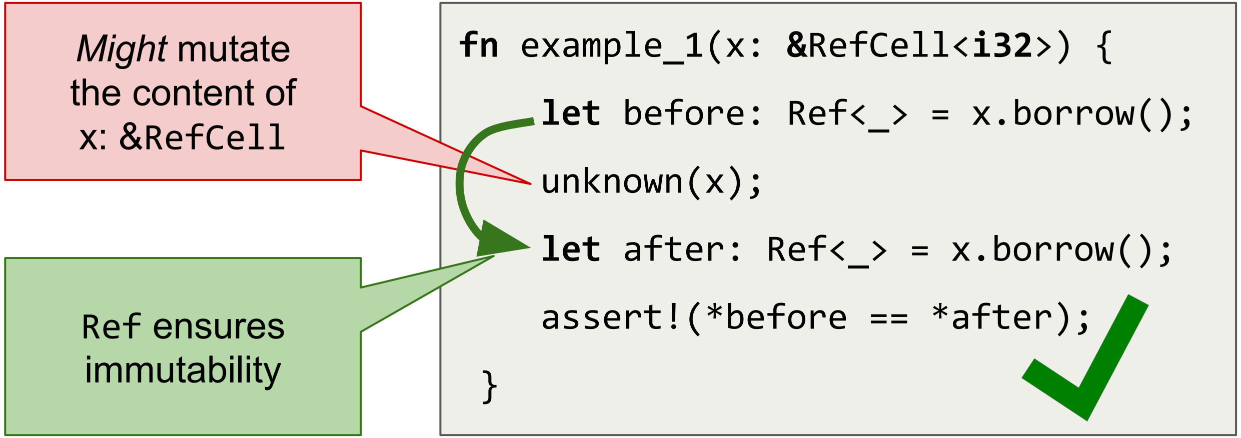
Incompatibility
(not usable at the same time)

Ensures **non-aliasing**

```
let mut x: T = …
let y: &mut T = &mut x;
let z: &T = &x;
using(z, y);
```

Ensures **immutability**

Verification example 1



Might mutate the content of x: `&RefCell`

Ref ensures immutability

```
fn example_1(x: &RefCell<i32>) {
    let before: Ref<_> = x.borrow();
    unknown(x);
    let after: Ref<_> = x.borrow();
    assert!(*before == *after);
 }
```

Verification example 2

The guards cannot refer to the content of the same `RwLock`

```
fn example_2(a: &RwLock<i32>) {
    let Ok(guard_1) = a.write() else { return; };
    let Ok(guard_2) = a.read() else { return; };
    unreachable!();
}
```

✓

Verification example 3

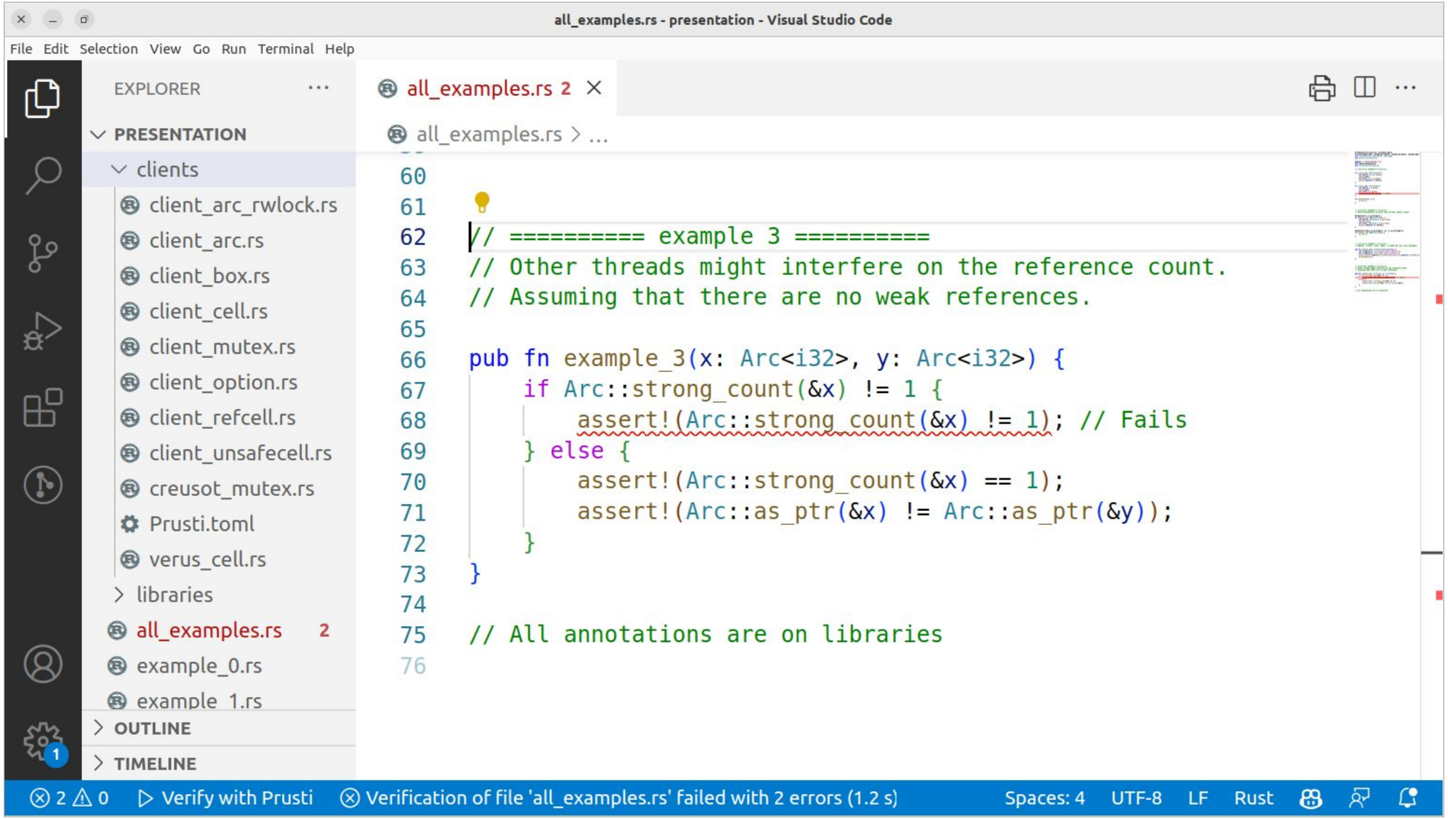
```
fn example_3(x: Arc<i32>, y: Arc<i32>) {
    if Arc::strong_count(&x) != 1 {
        assert!(Arc::strong_count(&x) != 1);
    } else {
        assert!(Arc::strong_count(&x) == 1);
        assert!(Arc::as_ptr(&x) != Arc::as_ptr(&y));
    }
}
```

What verifies and what does not?

Assume no weak references

Demo

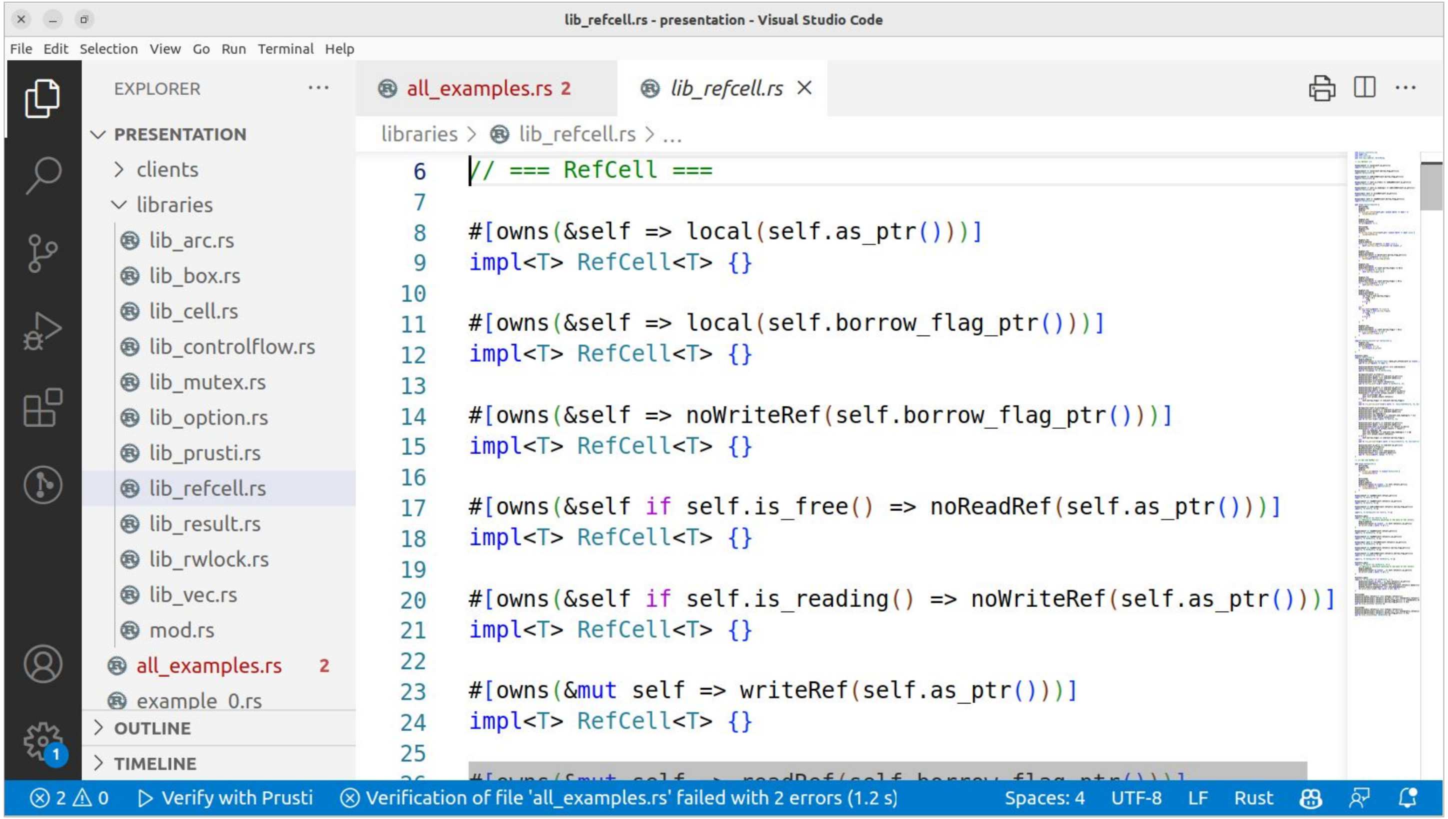
Demo: clients



```rust
// ========== example 3 ==========
// Other threads might interfere on the reference count.
// Assuming that there are no weak references.

pub fn example_3(x: Arc<i32>, y: Arc<i32>) {
    if Arc::strong_count(&x) != 1 {
        assert!(Arc::strong_count(&x) != 1); // Fails
    } else {
        assert!(Arc::strong_count(&x) == 1);
        assert!(Arc::as_ptr(&x) != Arc::as_ptr(&y));
    }
}

// All annotations are on libraries
```

⊗ Verification of file 'all_examples.rs' failed with 2 errors (1.2 s)

Demo: library annotations



```rust
// === RefCell ===

#[owns(&self => local(self.as_ptr()))]
impl<T> RefCell<T> {}

#[owns(&self => local(self.borrow_flag_ptr()))]
impl<T> RefCell<T> {}

#[owns(&self => noWriteRef(self.borrow_flag_ptr()))]
impl<T> RefCell<T> {}

#[owns(&self if self.is_free() => noReadRef(self.as_ptr()))]
impl<T> RefCell<T> {}

#[owns(&self if self.is_reading() => noWriteRef(self.as_ptr()))]
impl<T> RefCell<T> {}

#[owns(&mut self => writeRef(self.as_ptr()))]
impl<T> RefCell<T> {}

#[owns(&mut self => readRef(self.borrow_flag_ptr()))]
```

ⓧ 2 ⚠ 0 ▷ Verify with Prusti ⓧ Verification of file 'all_examples.rs' failed with 2 errors (1.2 s)

More types…

RefCell

Mutex

RwLock

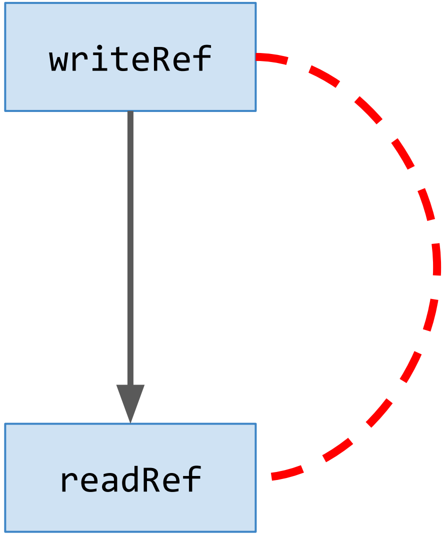
Cell, Arc, Rc, Atomic, …

```
#[owns(&self => rea Ref(self.data_p r()))]
impl<'b, T> RefMut< b, T> {}

#[owns(&mut self
impl<'b, T> RefMu

…
```

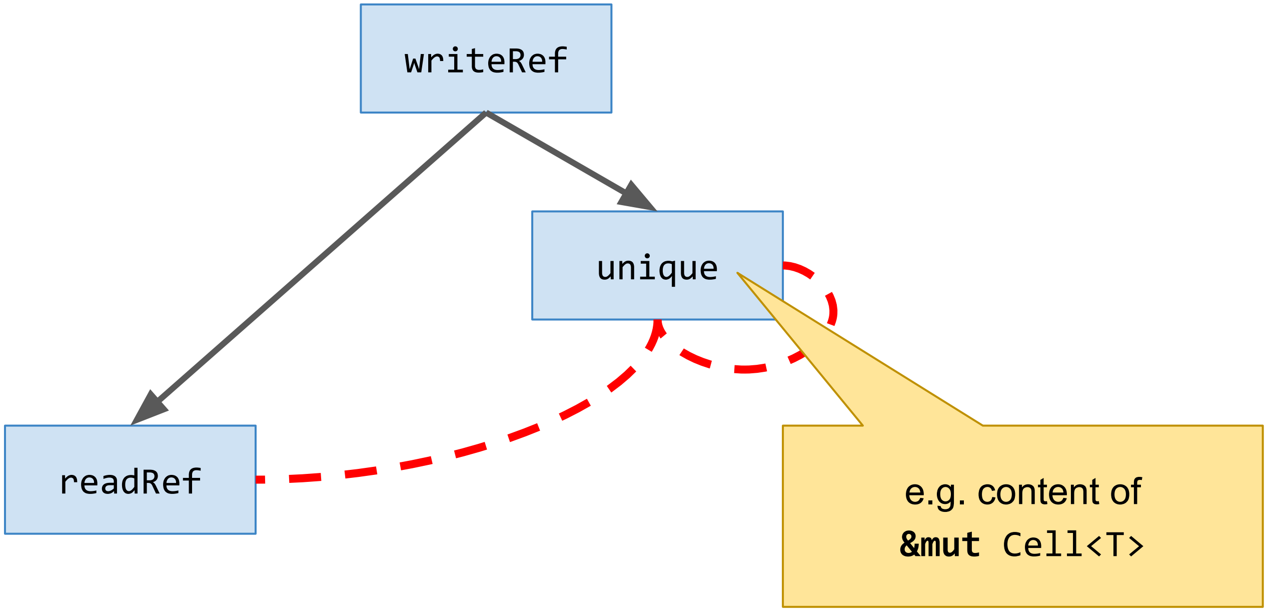
```
#[owns(&mut self => writeRef(self.data_ptr()))]
impl<'a, T> MutexGu rd<'a, T> {}

#[owns(&self =>
impl<'a, T> Mut

…
```

```
#[owns(&mut self => writeRef(self.data_ptr()))]
impl<'a, T> RwLock<'a, T> {}

#[owns(&self => readRef(self.data_ptr()))]
impl<'a, T> RwLockReadGuard<'a, T> {}

#[owns(&mut self => writeRef(self.data_ptr()))]
impl<'a, T> RwLockWriteGuard<'a, T> {}

…
```

More capabilities…

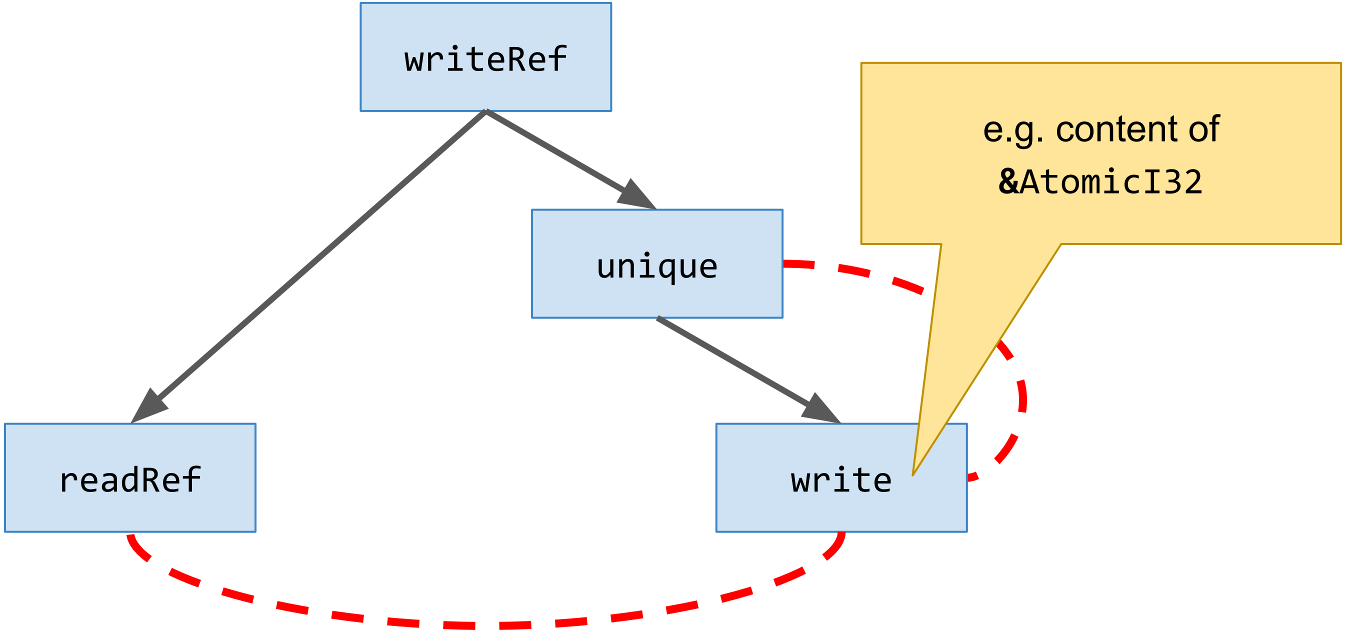


More capabilities…



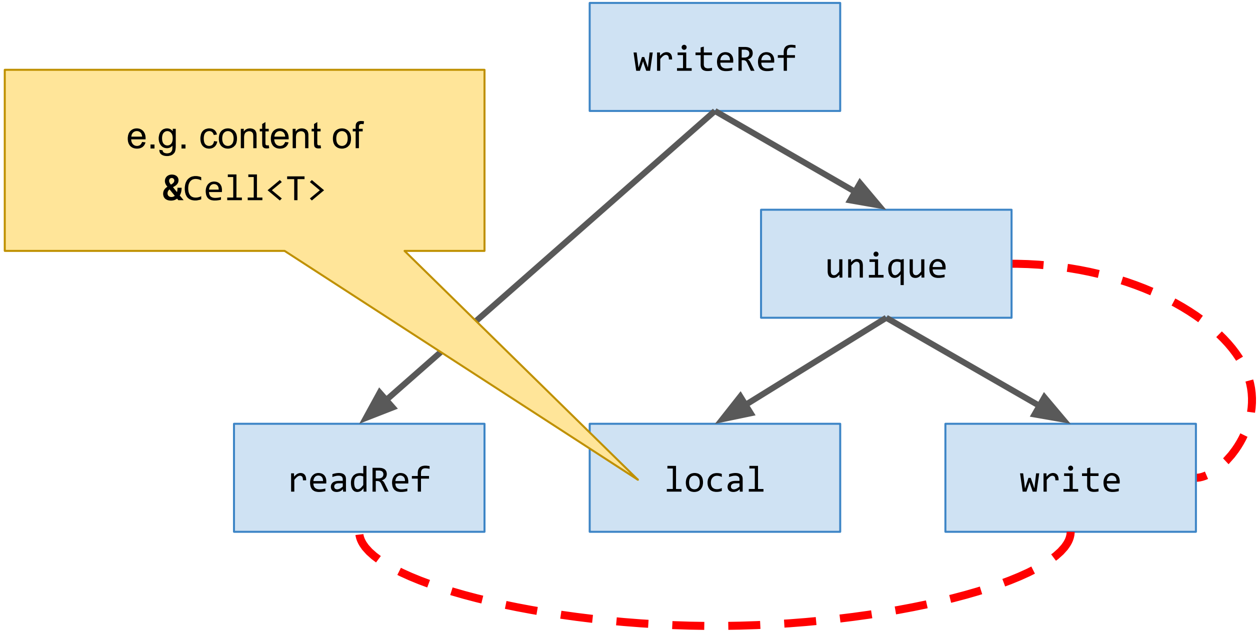
e.g. content of
&mut Cell<T>

More capabilities…



e.g. content of
&AtomicI32

More capabilities…

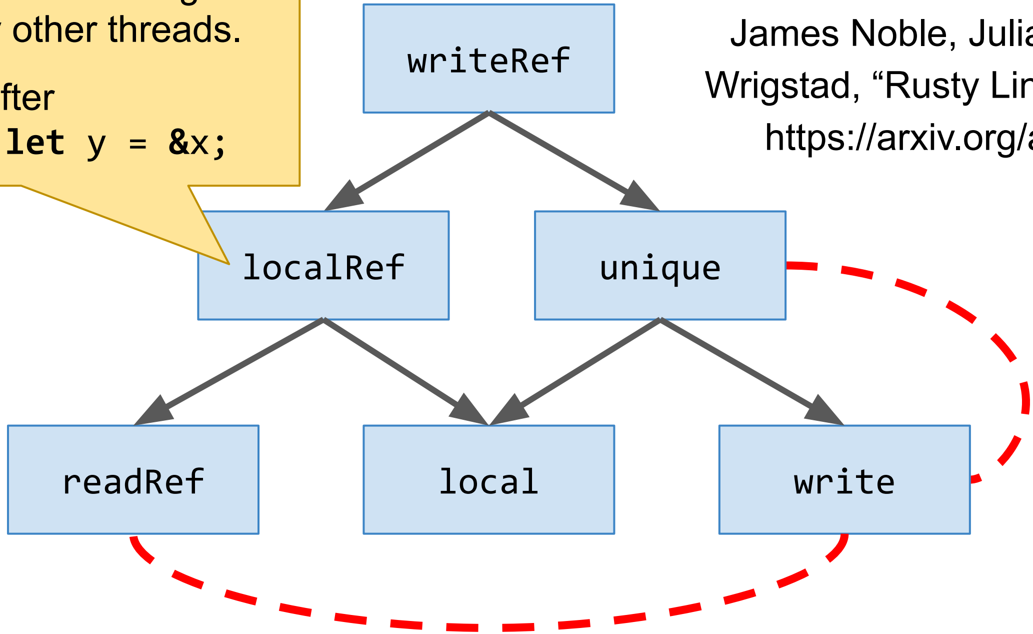


e.g. content of
`&Cell<T>`

More capabilities…

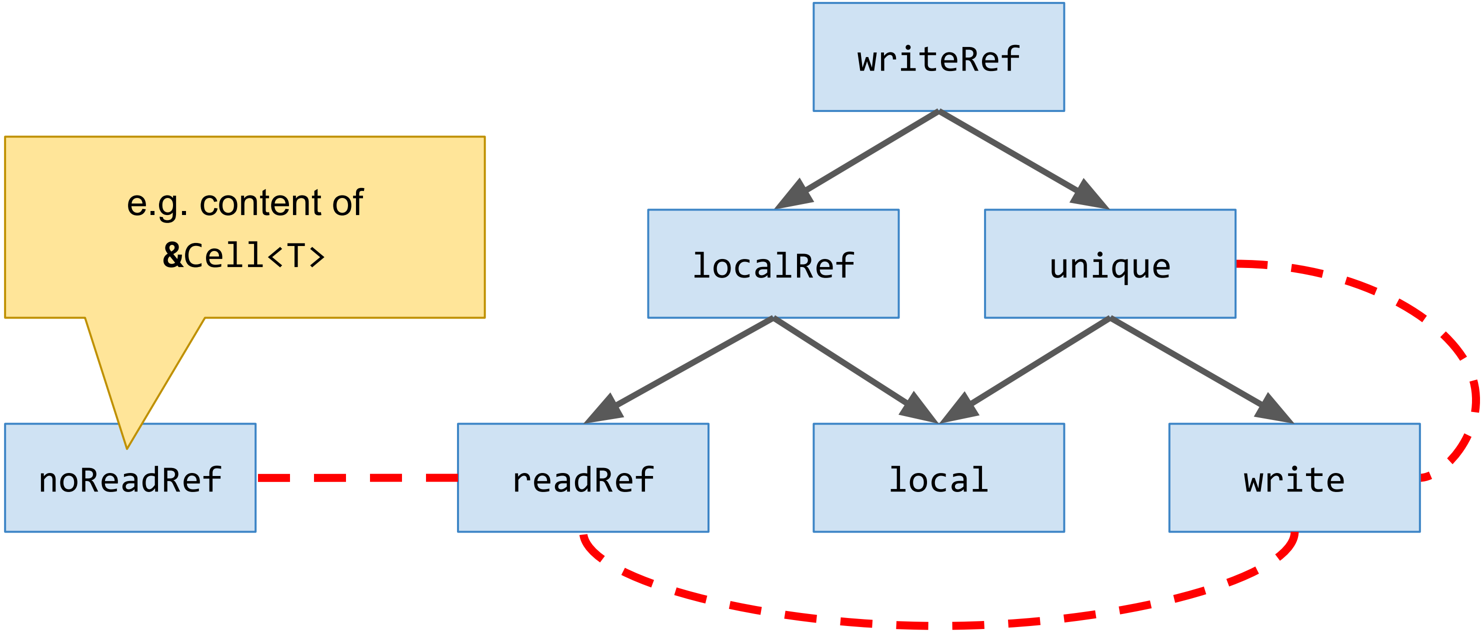
Shared references whose target is not reachable by other threads.

e.g. y after
`let` `x: T = …;` **`let`** `y = &x;`



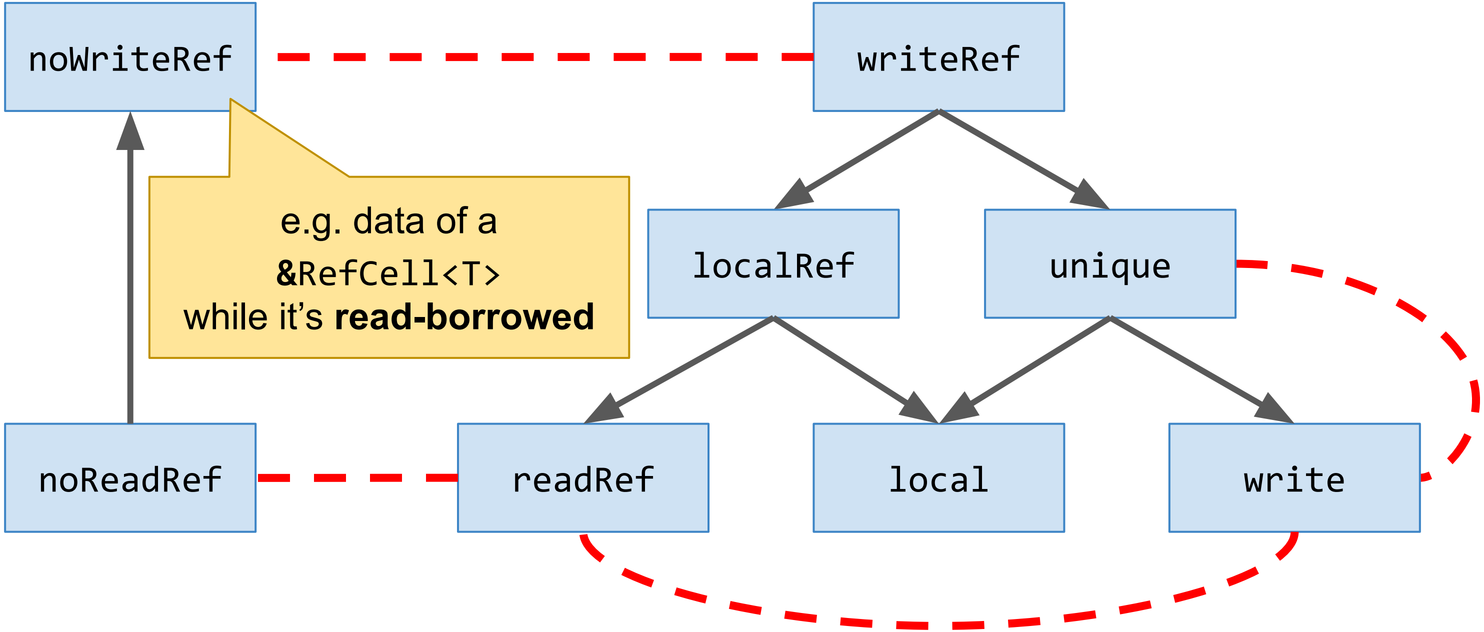
It's the "**`&loc`** `T`" of James Noble, Julian Mackay, Tobias Wrigstad, "Rusty Links in Local Chains" https://arxiv.org/abs/2205.00795

More capabilities…



e.g. content of
&Cell<T>

More capabilities…



e.g. data of a
&RefCell<T>
while it’s **read-borrowed**

Implicit capabilities with runtime conditions

Given a &RefCell

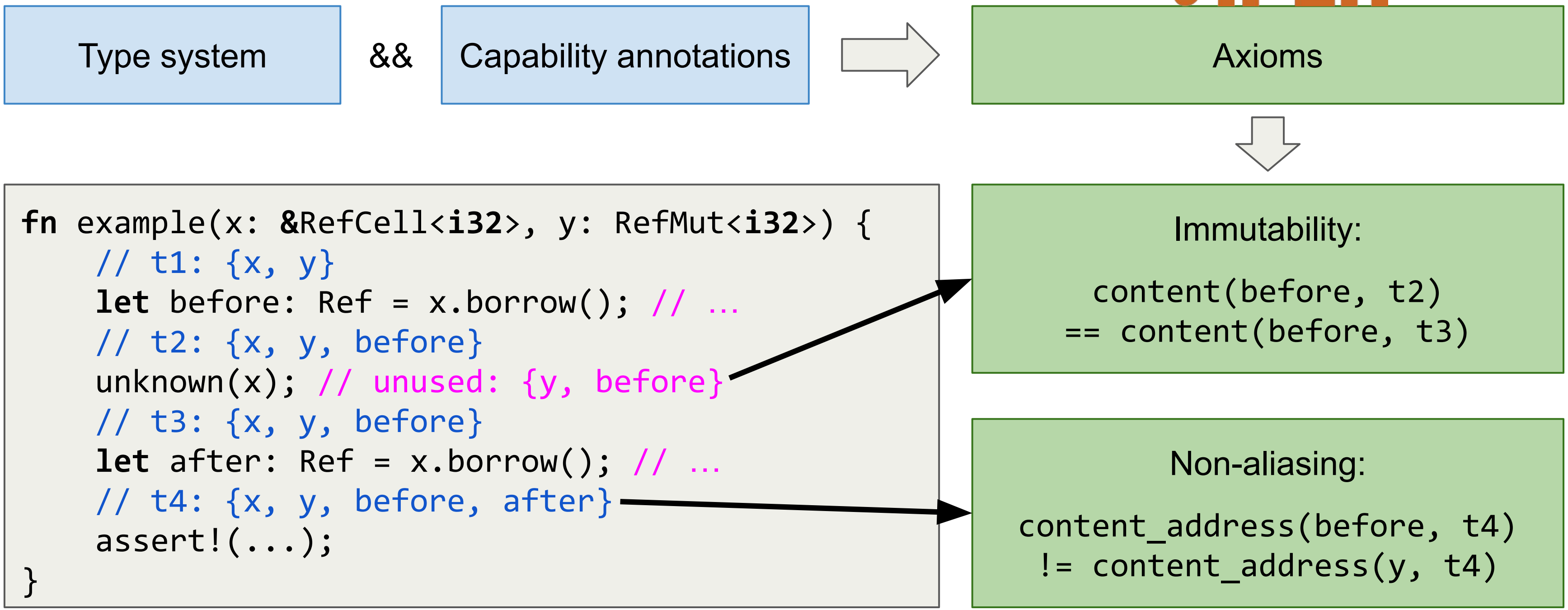
… while it’s read-borrowed …

… one cannot obtain a &mut to …

the content

```
#[owns(&self if self.is_reading() => noWriteRef(self.data_ptr()))]
impl<T> RefCell<T> {}
```

Encoding sketch



Type system && Capability annotations → Axioms

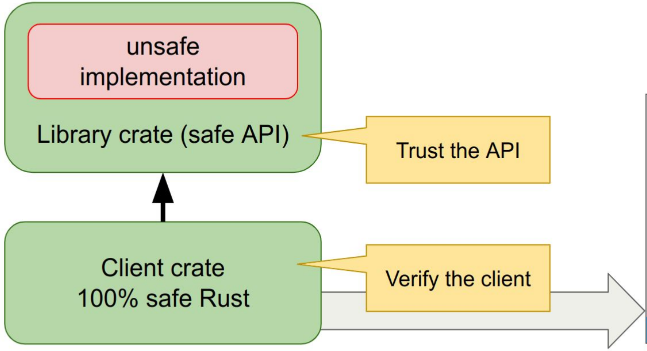
```
fn example(x: &RefCell<i32>, y: RefMut<i32>) {
    // t1: {x, y}
    let before: Ref = x.borrow(); // …
    // t2: {x, y, before}
    unknown(x); // unused: {y, before}
    // t3: {x, y, before}
    let after: Ref = x.borrow(); // …
    // t4: {x, y, before, after}
    assert!(...);
}
```

Immutability:

```
content(before, t2)
== content(before, t3)
```

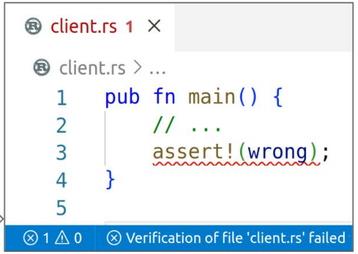
Non-aliasing:

```
content_address(before, t4)
!= content_address(y, t4)
```



P∗rust→∗i

http://prusti.org

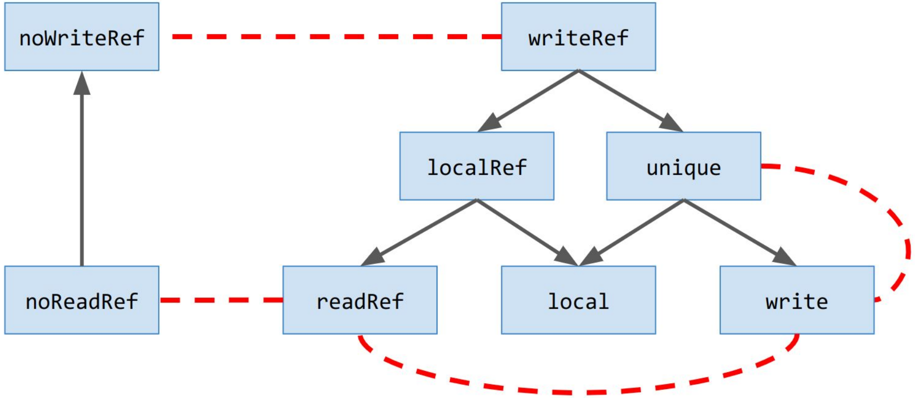


```
pub fn main() {
    // ...
    assert!(wrong);
}
```

⊗ Verification of file 'client.rs' failed

Deductive verifier for Rust

Thanks! Questions?


noWriteRef
writeRef
localRef
unique
noReadRef
readRef
local
write