Verification & Rust: Advanced Topics

Travis Hance (and the Verus Team)

CMU

MPI-SWS (to join soon)

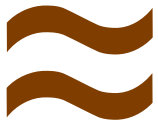
https://verus.rs/

Rust’s restrictions & verification

- Rust is widely seen as having a lot of “restrictions”
- These restrictions are actually good for verification! When you stick to the well-trodden path, verification is often efficient and relatively painless

The well-trodden path

```
fn example(x: &mut u64, y: &mut u64) {
    *x = *x + *y;
    *y = *y / *x;
}
```



```
fn example2(x: u64, y: u64) -> (u64, u64)
{
    return (x + y, y / (x + y));
}
```

Rust's reference system lets us make substantial simplifying assumptions

Life before Rust

```
method test6(a: Ptr, b: Ptr, c: Ptr, d: Ptr, e: Ptr, f: Ptr)
    requires a != b && b != c && a != c
        && a != d && b != d && c != d
        && a != e && b != e && c != e && d != e
        && a != f && b != f && c != f && d != f && e != f
{
  // ...
}
```

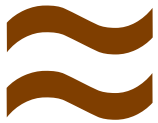
```
    function Repr() : set<object>
    reads this, persistentIndirectionTable, ephemeralIndirectionTable,
        frozenIndirectionTable, lru, cache, blockAllocator
    {
      {this} +
      persistentIndirectionTable.Repr +
      ephemeralIndirectionTable.Repr +
      (if frozenIndirectionTable != null then frozenIndirectionTable.Repr else {}) +
      lru.Repr +
      cache.Repr +
      blockAllocator.Repr
    }


    predicate ReprInv()
    reads this, persistentIndirectionTable, ephemeralIndirectionTable,
        frozenIndirectionTable, lru, cache, blockAllocator
    reads Repr()
    {
        && persistentIndirectionTable.Repr !! ephemeralIndirectionTable.Repr !!
lru.Repr !! cache.Repr !! blockAllocator.Repr
        && (frozenIndirectionTable != null ==>
            && frozenIndirectionTable.Repr !! persistentIndirectionTable.Repr
            && frozenIndirectionTable.Repr !! ephemeralIndirectionTable.Repr
            && frozenIndirectionTable.Repr !! lru.Repr
            && frozenIndirectionTable.Repr !! cache.Repr
            && frozenIndirectionTable.Repr !! blockAllocator.Repr
        )


        && this !in ephemeralIndirectionTable.Repr
        && this !in persistentIndirectionTable.Repr
        && (frozenIndirectionTable != null ==> this !in frozenIndirectionTable.Repr)
        && this !in lru.Repr
        && this !in cache.Repr
        && this !in blockAllocator.Repr
    }
```

The well-trodden path

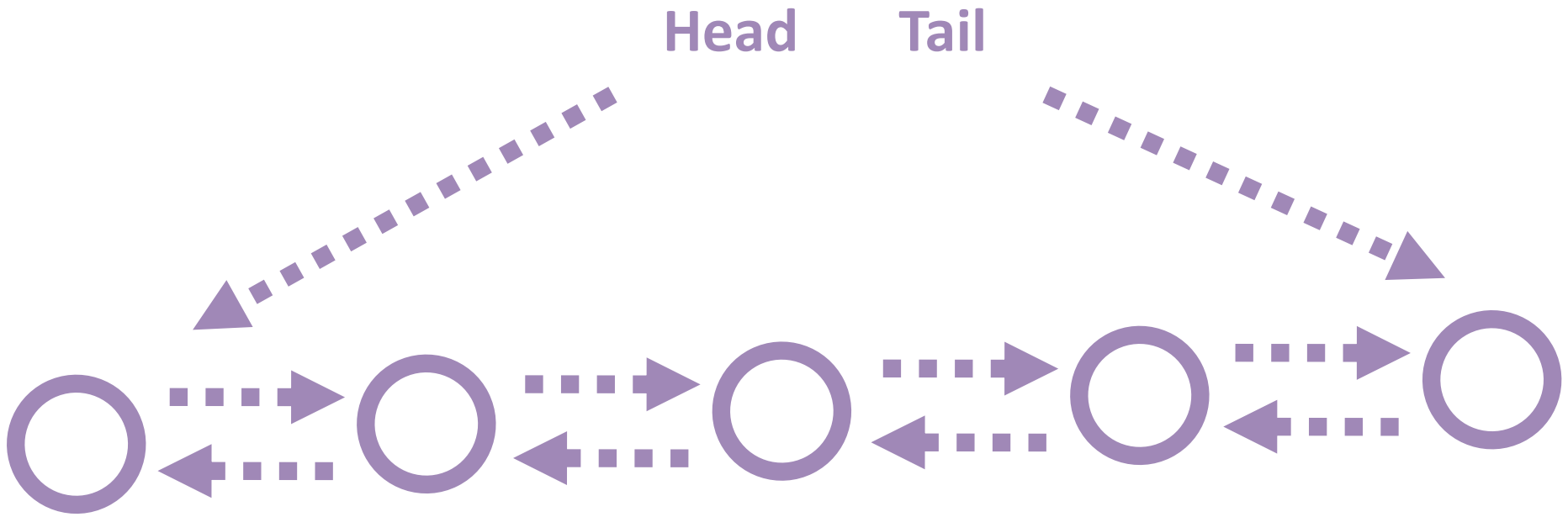
```
fn example(x: &mut u64, y: &mut u64) {
    *x = *x + *y;
    *y = *y / *x;
}
```



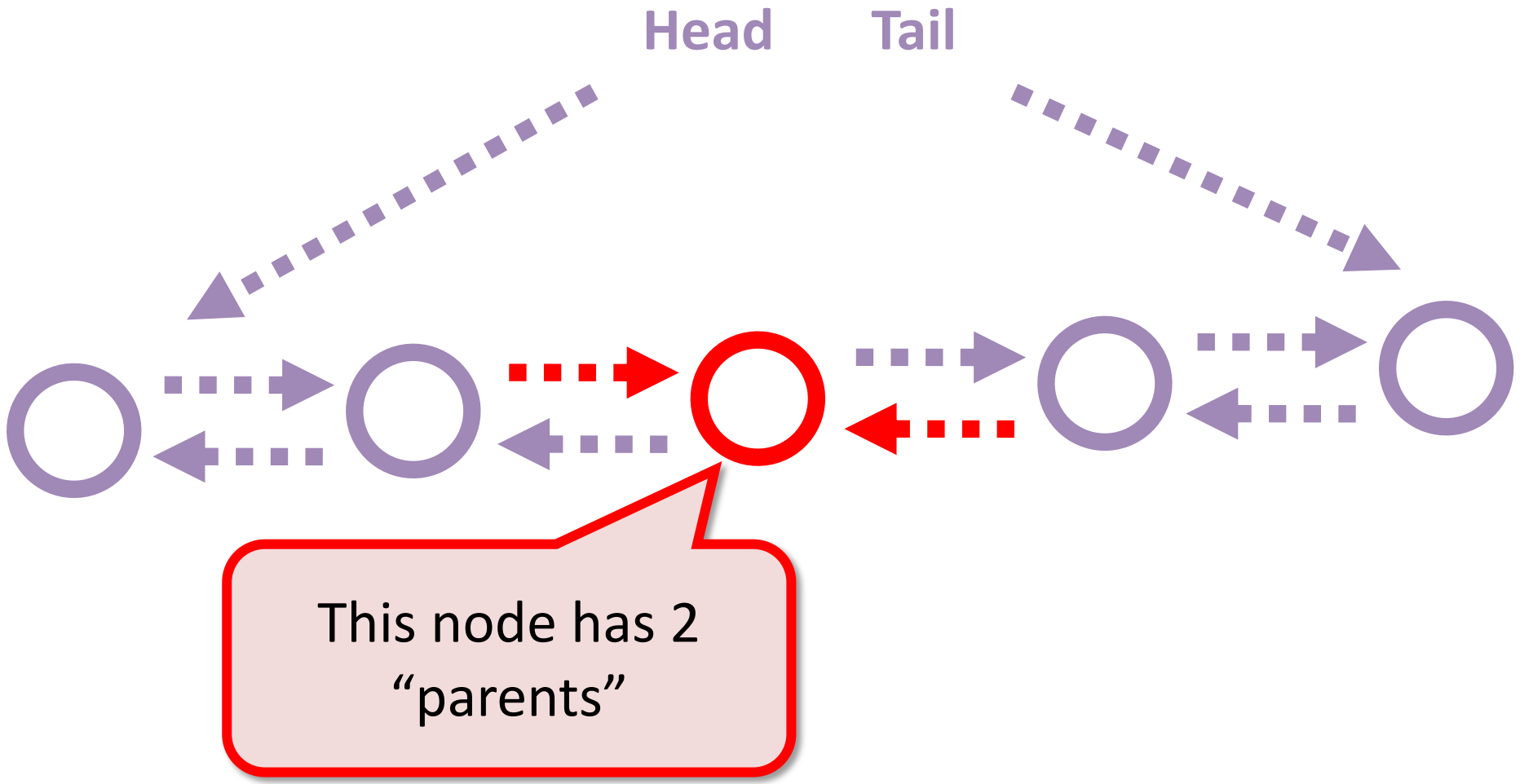
```
fn example2(x: u64, y: u64) -> (u64, u64)
{
    return (x + y, y / (x + y));
}
```

Rust's reference system lets us make substantial simplifying assumptions

The road less travelled



The road less travelled



Rules are bent, not broken

Unsafe code

Interior mutability

Concurrency

Unsafe Code

What is unsafe code?

Definition attempt #1:

Unsafe code is code that opts the programmer out of the memory-safety guaranteed by Rust's type system

```
fn oh_no() {
    let p: *mut u64 = std::ptr::null_mut();
    unsafe {
        *p = 20;
    }
}
```

What is unsafe code?

Definition attempt #1:

Unsafe code is code that opts the programmer out of the memory-safety guaranteed by Rust's type system

```
fn oh_no() {
    let p: *mut u64 = std::ptr::null_mut();
    unsafe {
        *p = 20;
    }
}
```

This seems pretty bad — doesn't Verus rely on this?

What is unsafe code?

Definition attempt #2:

Unsafe code is code that has **additional requirements** (beyond Rust's type checker) to guarantee memory safety

Example: `Option::unwrap_unchecked`

```
[−] pub unsafe fn unwrap_unchecked(self) -> T                    1.58.0 (const: unstable) · source
```

Returns the contained `Some` value, consuming the `self` value, without checking that the value is not `None`.

Safety

Calling this method on `None` is *undefined behavior*.

Examples

```
let x = Some("air");
assert_eq!(unsafe { x.unwrap_unchecked() }, "air");
```

```
let x: Option<&str> = None;
assert_eq!(unsafe { x.unwrap_unchecked() }, "air"); // Undefined behavior!
```

The subtext here is that behavior is defined **if** `self` is `Some(_)`

https://doc.rust-lang.org/std/option/enum.Option.html#method.unwrap_unchecked

What is unsafe code?

Unsafe code is really **Conditionally Safe** code — *if* the condition is satisfied, *then* it is safe

```
impl<T> Option<T> {
    fn unwrap_unchecked(self) -> (res: T)
        requires self.is_some()
        ensures self == Some(res)
    {
        /* ... */
    }
}
```

These conditions can be made **explicit and formal** using Verus preconditions

Unsafe Code


Unsafe Code


Unsafe pointers

Let’s check the docs again

Module std::ptr

1.0.0 · source · [–]

[–] Manually manage memory through raw pointers.

See also the pointer primitive types.

Safety

Many functions in this module take raw pointers as arguments and read from or write to them. For this to be safe, these pointers must be *valid* for the given access. Whether a pointer is valid depends on the operation it is used for (read or write), and the extent of the memory that is accessed (i.e., how many bytes are read/written) – it makes no sense to ask “is this pointer valid”; one has to ask “is this pointer valid for a given access”. Most functions use `*mut T` and `*const T` to access only a single value, in which case the documentation omits the size and implicitly assumes it to be `size_of::<T>()` bytes.

The precise rules for validity are not determined yet. The guarantees that are provided at this point are very minimal:

- For operations of size zero, *every* pointer is valid, including the null pointer. The following points are only concerned with non-zero-sized accesses.

ask “is this pointer valid for a given access”. Most functions use `*mut T` and `*const T` to access only a single value, in which case the documentation omits the size and implicitly assumes it to be `size_of::<T>()` bytes.

The precise rules for validity are not determined yet. The guarantees that are provided at this point are very minimal:

- For operations of size zero, *every* pointer is valid, including the null pointer. The following points are only concerned with non-zero-sized accesses.
- A null pointer is *never* valid.
- For a pointer to be valid, it is necessary, but not always sufficient, that the pointer be *dereferenceable*: the memory range of the given size starting at the pointer must all be within the bounds of a single allocated object. Note that in Rust, every (stack-allocated) variable is considered a separate allocated object.
- All accesses performed by functions in this module are *non-atomic* in the sense of atomic operations used to synchronize between threads. This means it is undefined behavior to perform two concurrent accesses to the same location from different threads unless both accesses only read from memory. Notice that this explicitly includes `read_volatile` and `write_volatile`: Volatile accesses cannot be used for inter-thread synchronization.
- The result of casting a reference to a pointer is valid for as long as the underlying object is live and no reference (just raw pointers) is used to access the same memory. That is, reference and pointer accesses cannot be interleaved.

These axioms, along with careful use of `offset` for pointer arithmetic, are enough to correctly implement many useful things in unsafe code. Stronger guarantees will be provided eventually, as the aliasing rules are being determined. For more information, see the book as well as the section in the reference devoted to undefined behavior.

We say that a pointer is “dangling” if it is not valid for any non-zero-sized accesses. This means out-of-bounds pointers, pointers to freed memory, null pointers, and pointers created with `NonNull::dangling` are all dangling.

Alignment

the book as well as the section in the reference devoted to undefined behavior.

We say that a pointer is "dangling" if it is not valid for any non-zero-sized accesses. This means out-of-bounds pointers, pointers to freed memory, null pointers, and pointers created with `NonNull::dangling` are all dangling.

Alignment

Valid raw pointers as defined above are not necessarily properly aligned (where "proper" alignment is defined by the pointee type, i.e., `*const T` must be aligned to `mem::align_of::<T>()`). However, most functions require their arguments to be properly aligned, and will explicitly state this requirement in their documentation. Notable exceptions to this are `read_unaligned` and `write_unaligned`.

When a function requires proper alignment, it does so even if the access has size 0, i.e., even if memory is not actually touched. Consider using `NonNull::dangling` in such cases.

Pointer to reference conversion

When converting a pointer to a reference (e.g. via `&*ptr` or `&mut *ptr`), there are several rules that must be followed:

- The pointer must be properly aligned.
- It must be non-null.
- It must be "dereferenceable" in the sense defined above.
- The pointer must point to a valid value of type `T`.
- You must enforce Rust's aliasing rules. The exact aliasing rules are not decided yet, so we only give a rough overview here. The rules also depend on whether a mutable or a shared reference is being created.
  - When creating a mutable reference, then while this reference exists, the memory it points to must not get accessed (read or

It keeps going, by the way

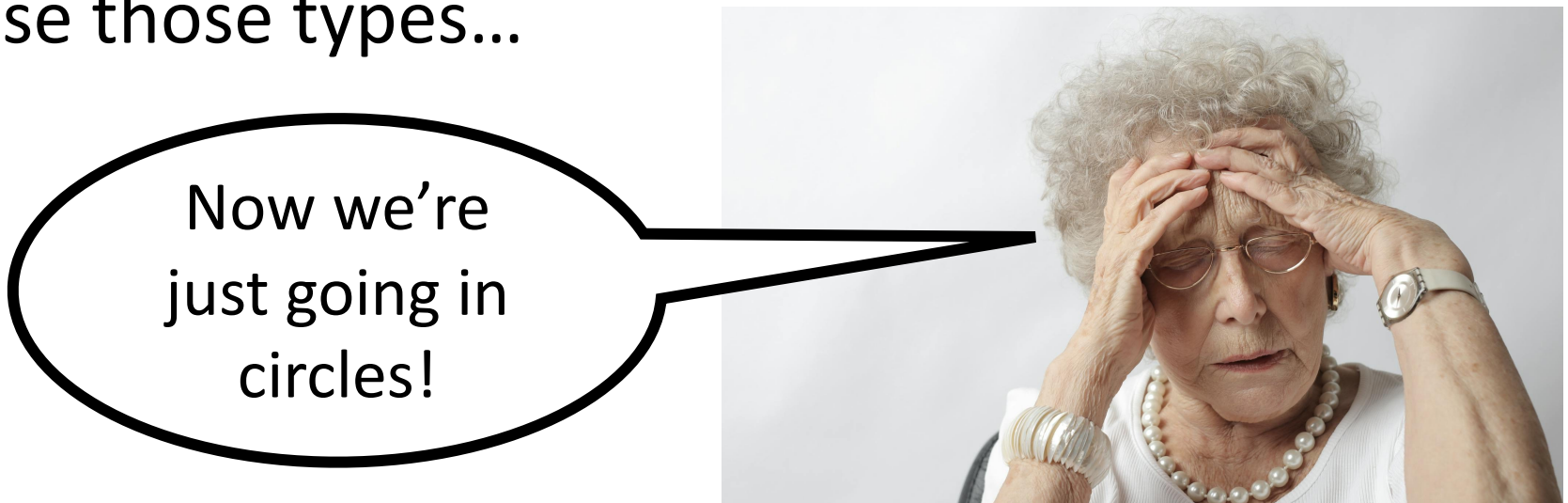
Pointers have a lot of conditions

- Accessing a pointer requires:
  - Memory is valid, allocated, well-aligned
  - Reads must be initialized
  - Accesses are data-race-free
  - Playing nice with reference system (`&T` and `&mut T`)
  - Handling “pointer provenance”

Can these conditions even be expressed via **Verus preconditions?**

Isn’t this what ownership types are for?

- Ordinarily, Rust’s type system, together with its clever standard library types (`&T`, `&mut T`, `Box<T>`, `Rc<T>`, `Arc<T>`, `RefCell<T>`, …), ensure that all the rules are followed
- But the whole point of using “raw pointers” is that we *can’t* use those types…



Isn’t this what ownership types are for?

- Ordinarily, Rust’s type system, together with its clever standard library types (`&T`, `&mut T`, `Box<T>`, `Rc<T>`, `Arc<T>`, `RefCell<T>`, …), ensure that all the rules are followed
- But the whole point of using “raw pointers” is that we *can’t* use those types…
- Solution: **Decouple** the ownership reasoning from the physical pointers; reason about ownership in “proof code”

Pointers have a lot of conditions

- Acc
  - –
  - –
  - –
  - –
  - – Handling "pointer provenance"

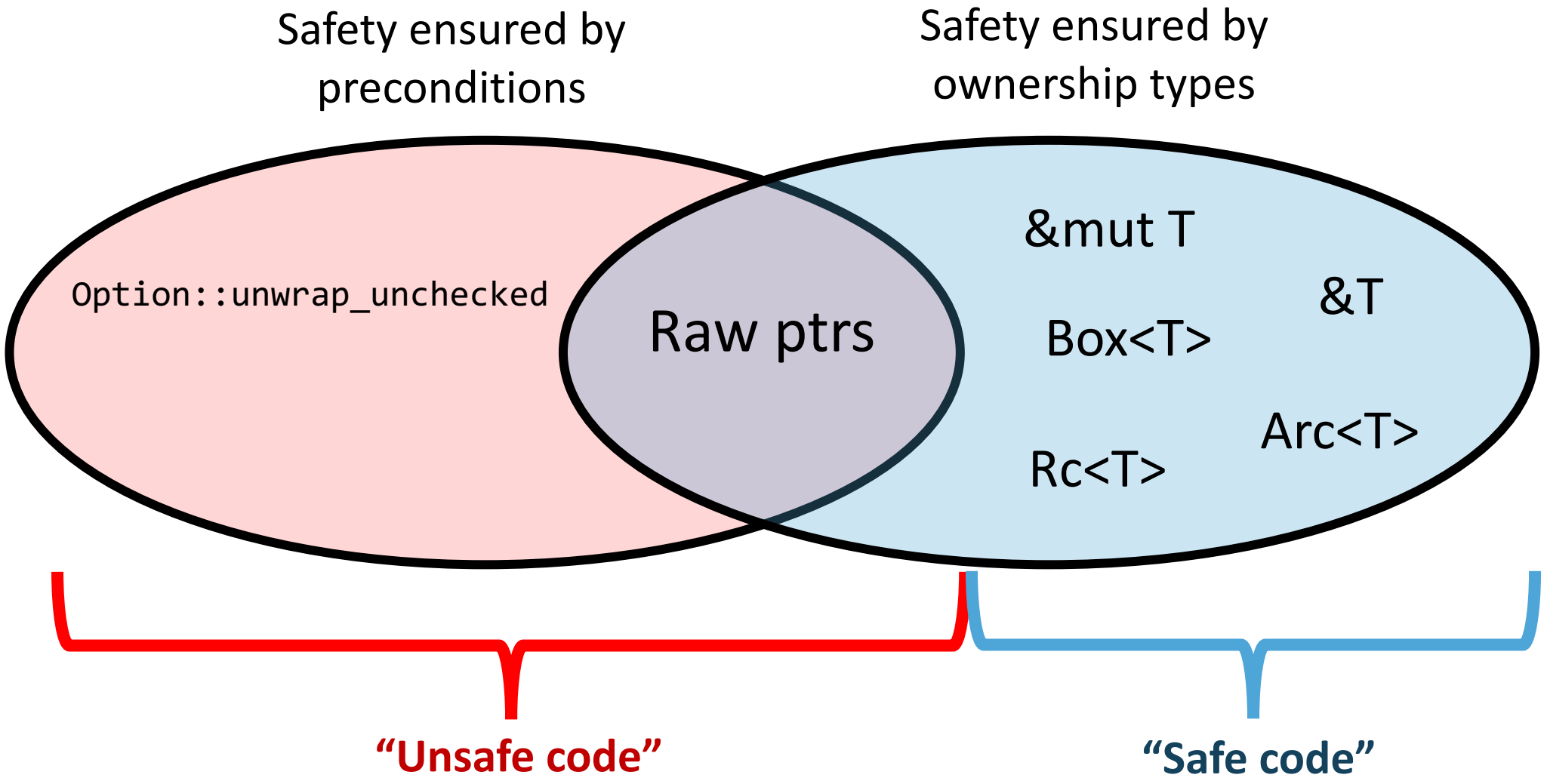
gned

&T and &mut T)

For the demo & exercise, we'll use Verus's **"Simple PPtr"** library, which sands down the rougher edges at the cost of generality — but it still covers the **key points**

[PPtr demo]

Summary


Safety ensured by preconditions
Safety ensured by ownership types
Option::unwrap_unchecked
Raw ptrs
&mut T
&T
Box<T>
Arc<T>
Rc<T>
“Unsafe code”
“Safe code”

Interior mutability

What is interior mutability?

Rust's references are usually described to new users as follows:

Type		
`&mut T`	**Unique**	**Mutable**
`&T`	**Shared**	**Immutable**

Unfortunately it's not entirely correct …

Sometimes described as "shared XOR mutable," this is an important property that enables Rust's memory safety & Verus's efficient SMT encoding

What is interior mutability?

- **Interior mutability** is a pattern wherein you can modify data behind a shared reference (&).
- However, this is only possible through specialized **“interior mutability”** types (e.g., Cell, RefCell, Mutex, RwLock, UnsafeCell)
- This is good, because unrestricted mutation-through-& would ruin all the theory behind Verus’s encoding
- As it is, we can restrict the additional complexity **only** to when these specialized types are involved

What's the problem?

```
fn example(cell1: &Cell<u64>, cell2: &Cell<u64>)
    -> (u64, u64)
{
    cell1.set(24);
    cell2.set(25);
    let x = cell1.get();
    let y = cell2.get();
    (x, y)
}
```

cell1 = 24

cell2 = 25

x = 24

y = 25

(x, y) = (24, 25)

What's the problem?

```
fn example(cell1: &Cell<u64>, cell2: &Cell<u64>)
    -> (u64, u64)
{
    cell1.set(24);
    cell2.set(25);
    let x = cell1.get();
    let y = cell2.get();
    (x, y)
}
```

cell1 = 24

cell2 = 25

x = 24

y = 25

~~(x, y) = (24, 25)~~

But (25, 25) is possible!

What's the problem?

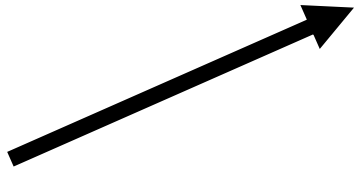
```
fn example(cell1: &Cell<u64>, cell2: &Cell<u64>)
    -> (u64, u64)
{
    cell1.set(24);
    cell2.set(25);
    let x = cell1.get();
    let y = cell2.get();
    (x, y)
}
```

Option 1: Maintain a "cell heap"

Heap = [cell1 |--> 24 ; cell2 |--> 25]
(and cell1 ≠ cell2)

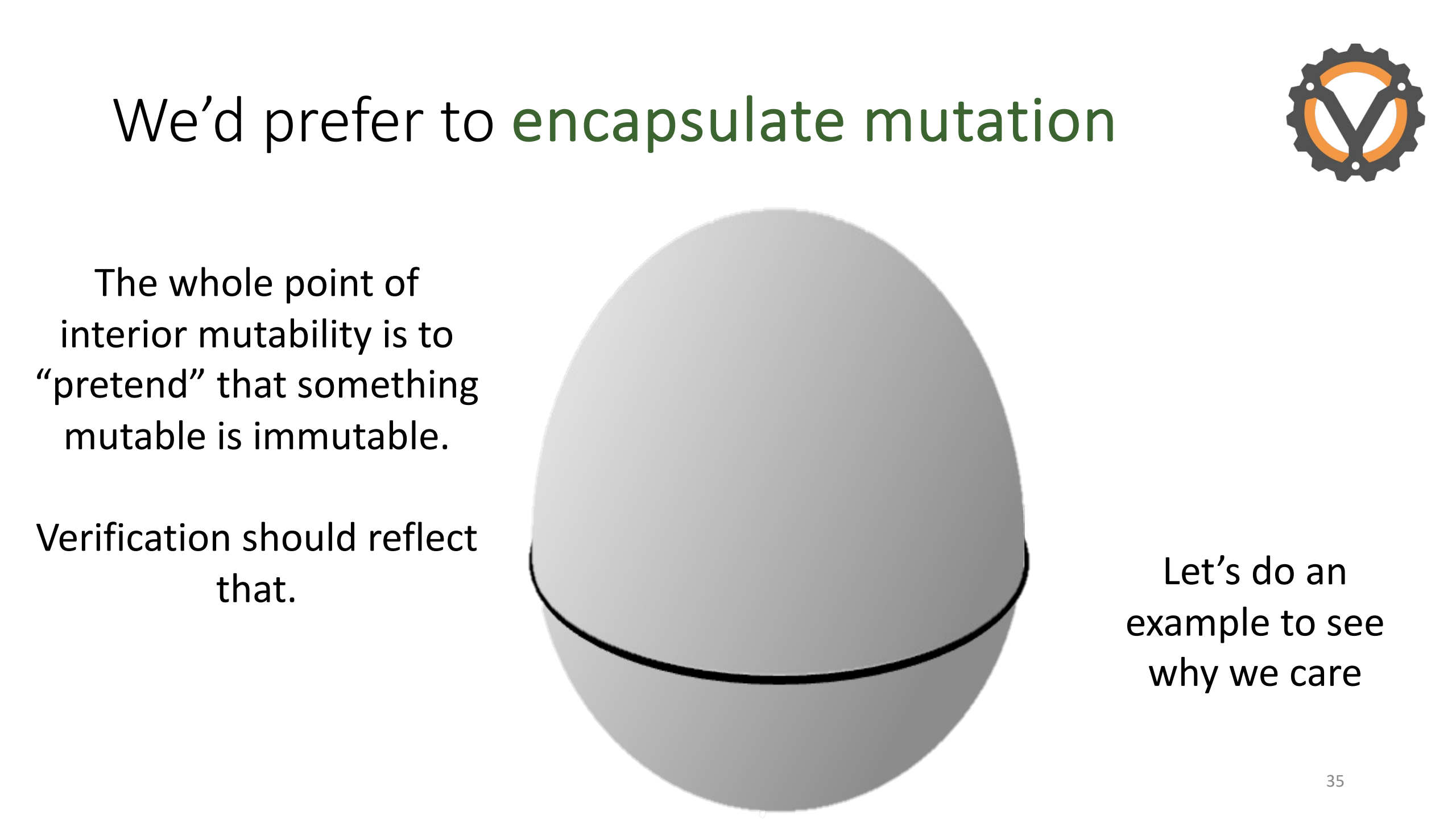
Or

Heap = [cell1 |--> 25]
(and cell1 = cell2)



This "leaks" the mutability everywhere

We'd prefer to encapsulate mutation



The whole point of interior mutability is to "pretend" that something mutable is immutable.

Verification should reflect that.

Let's do an example to see why we care

Example: Memoizing a deterministic fn

```
pub fn expensive_function(args: &Args) -> Output {
    /* ... */
}

struct Memoizer {
    map: HashMap<Args, Output>
}

impl Memoizer {
    pub fn get(&mut self, args: &Args) -> Output {
        match self.map.get(args) {
            Some(result) => result.clone(),
            None => {
                // Compute for the first time and store the result for next time
                let result = expensive_function(args);
                self.map.insert(args.clone(), result.clone());
                result
            }
        }
    }
}
```

Suppose we have some expensive but **deterministic** computation

We can **memoize** the results — lazily populate a lookup-table of results, as-needed

Example: Memoizing a deterministic fn

Now, what if we want to share the `Memoizer` across multiple subsystems? Maybe even across threads?

`Arc<Memoizer>` → `&Memoizer`

Whoops, now we can't call `get`:

```
fn get(&mut self, args: &Args)
```

Example: Memoizing a deterministic fn

```
pub fn expensive_function(args: &Args) -> Output {
    /* ... */
}

struct Memoizer {
    map: HashMap<Args, Output>
}

impl Memoizer {
    pub fn get(&mut self, args: &Args) -> Output {
        match self.map.get(args) {
            Some(result) => result.clone(),
            None => {
                // Compute for the first time and store the result for next time
                let result = expensive_function(args);
                self.map.insert(args.clone(), result.clone());
                result
            }
        }
    }
}
```

Example: Memoizing a deterministic fn

We really want to implement this instead:

```
fn get(&self, args: &Args)
```

For this we can use **interior mutability** …

Example: Memoizing a deterministic fn

```
pub fn expensive_function(args: &Args) -> Output {
    /* ... */
}

struct Memoizer {
    ref_cell: RefCell<HashMap<Args, Output>>
}

impl Memoizer {
    pub fn get(&self, args: &Args) -> Output {
        let map_handle = self.ref_cell.borrow_mut();
        match map_handle.get(args) {
            Some(result) => result.clone(), None => {
                // Compute for the first time and store the result for next time
                let result = expensive_function(args);
                map_handle.insert(args.clone(), result.clone());
                result
             }
        }
    }
}
```

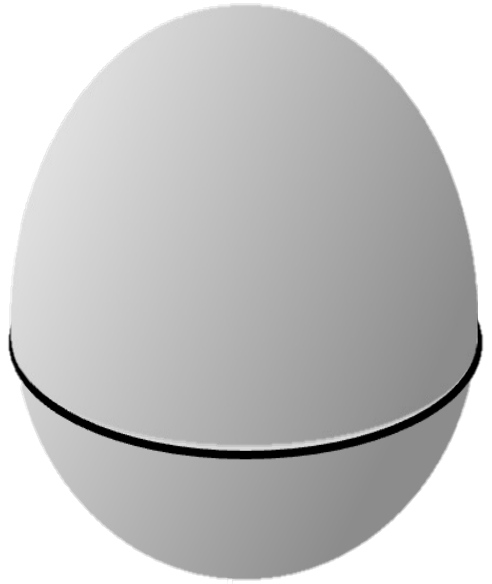
Example: Memoizing a deterministic fn

Now we have this beautiful type signature that hides the mutability:

```
fn get(&self, args: &Args)
```

The **Verus specification** should be equally concise:

- no “modifies” clause
- no passing in “write-permissions”
- etc.



Solution: What doesn't change?

If Verus can't treat like `&Cell<u64>` as a `u64` because the `u64` might change … let's treat it like something that **doesn't** change

What doesn't change? **An invariant.**

- An **invariant** = predicate describing the "allowed possible values"
- Forget about tracking the precise value; only track the invariant

[Invariants demo]

An encapsulating specification

Using cell invariants / lock invariants, we can prove the following spec:

```
pub spec fn func(args: &Args) -> Output { ... }

pub fn expensive_function(args: &Args) -> (output: Output)
    ensures output == func(args)
{ ... }

impl Memoizer {
    pub fn get(&self, args: &Args) -> (output: Output)
        ensures output == func(args)
    { ... }
}
```

Function must be deterministic

Memoizer is deterministic and computes the same function

An encapsulating specification

Using cell invariants / lock invariants, we can prove the following spec:

```
pub spec fn func(args: &Args) -> Output { ... }

pub fn expensive_function(args: &Args) -> (output: Output)
     ensures output == func(args)
{ ... }

impl Memoizer {
    pub fn get(&self, args: &Args) -> (output: Output)
        ensures output == func(args)
    { ... }
}
```

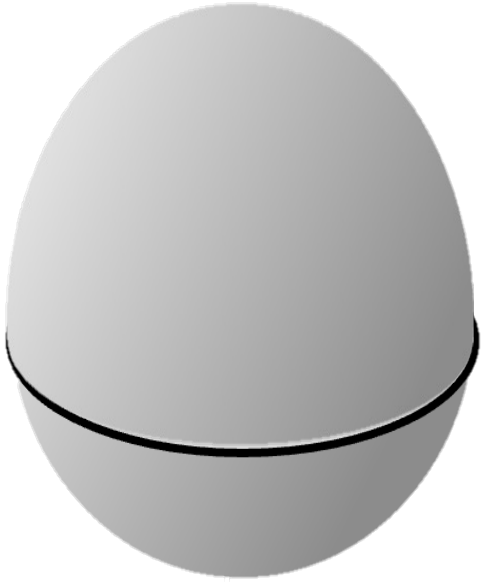
Function must be deterministic

Memoizer is deterministic and computes the same function

Observe again how the mutation is encapsulated

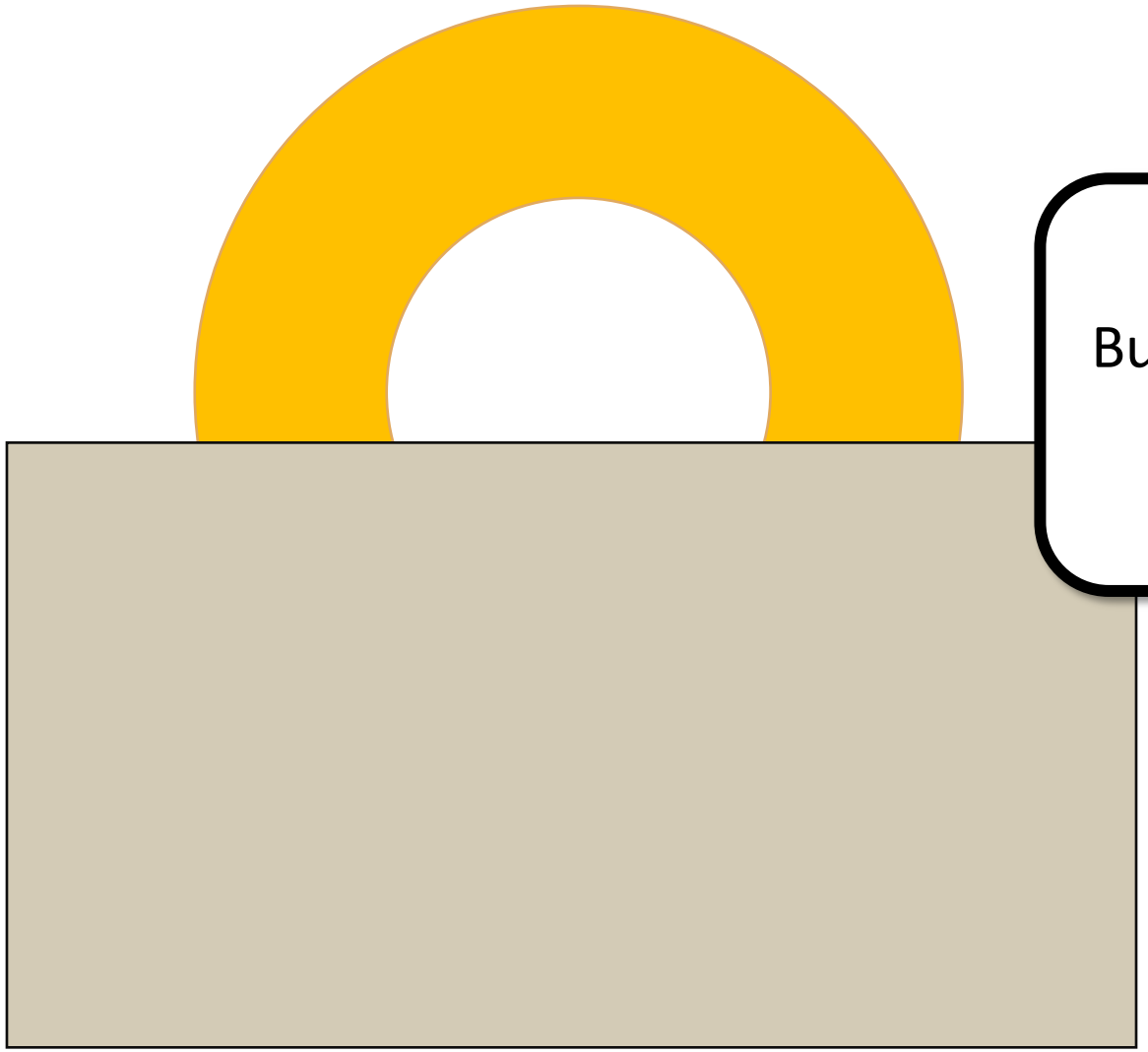
Recap

- **Interior mutability** carves an exception to the usual “sharing XOR mutability” dichotomy of Rust, which Verus relies on
- Thus, interior mutability adds complexity, but that complexity can be **encapsulated**
- To do this, we can use lock invariants / cell invariants



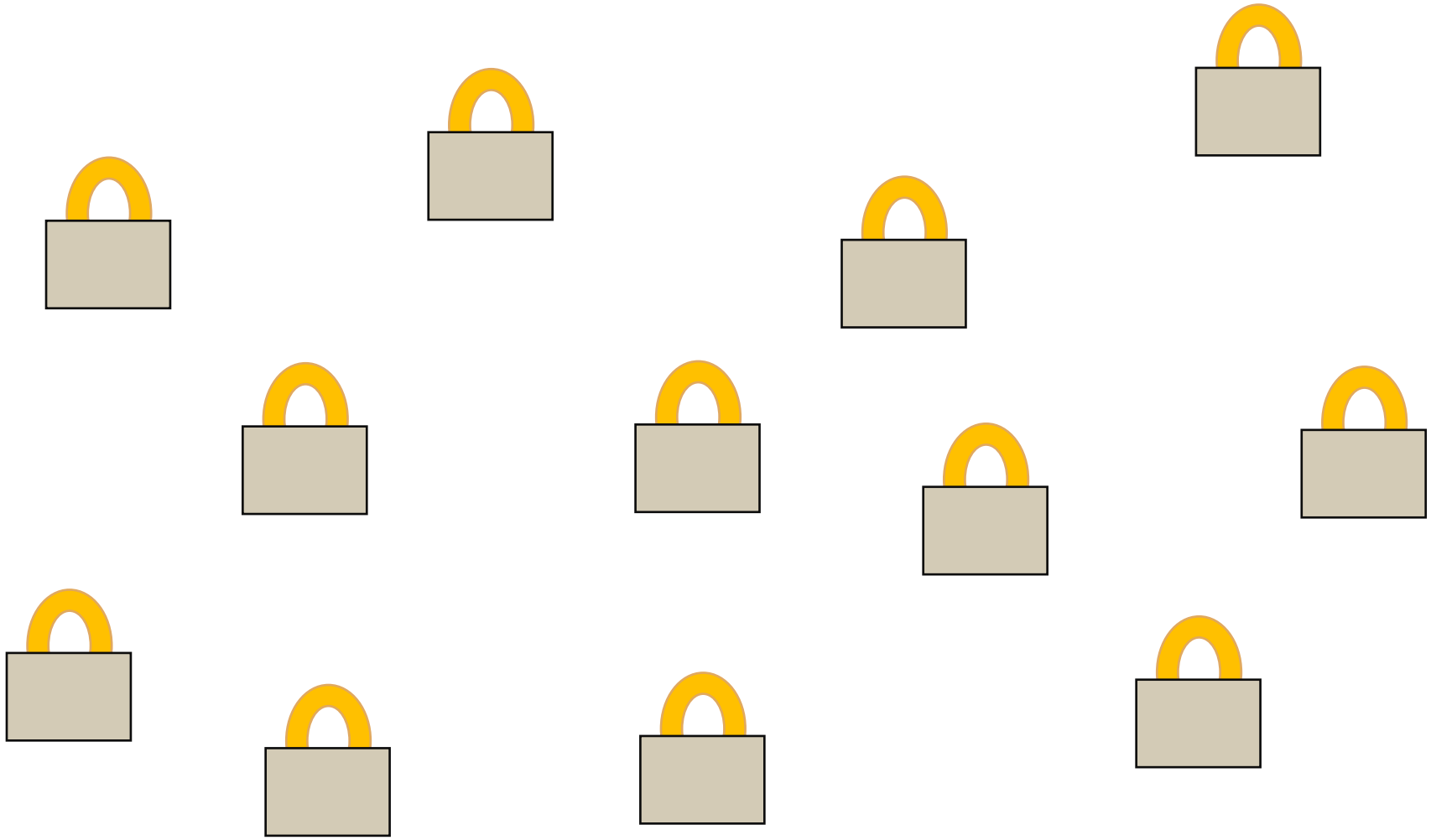

Concurrency

We already covered locks actually …



But one **big global lock** does not **concurrency** make!

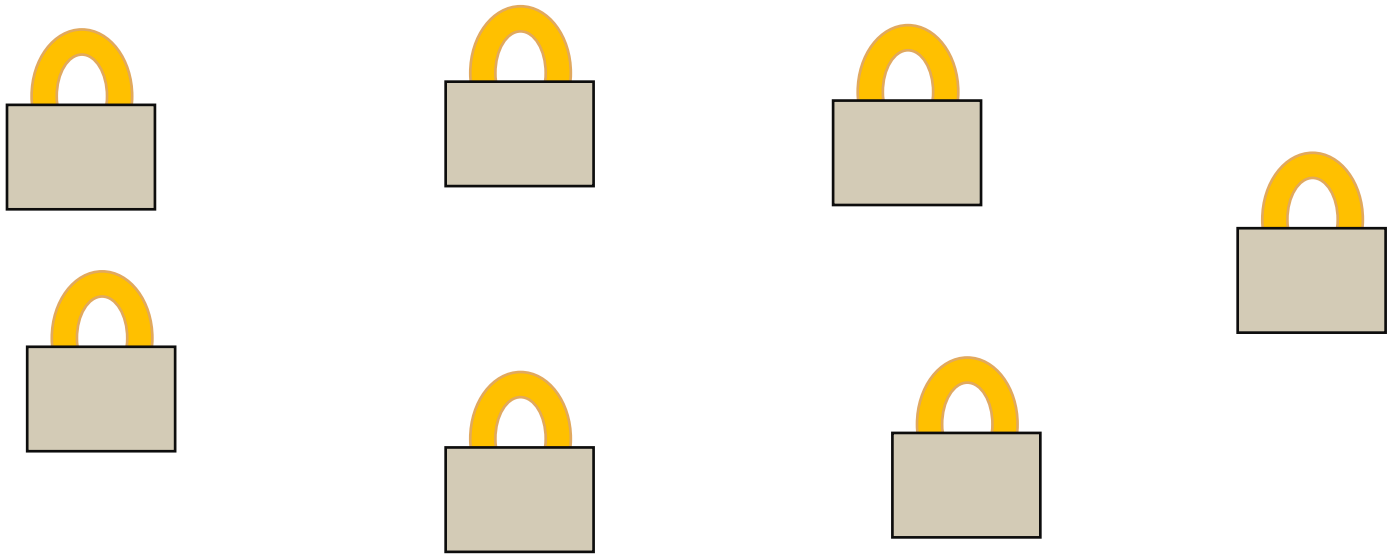
Realistic systems use fine-grained locks



Problem: split ownership

True Concurrency requires us to have **multiple** components that can be owned and operated **independently**

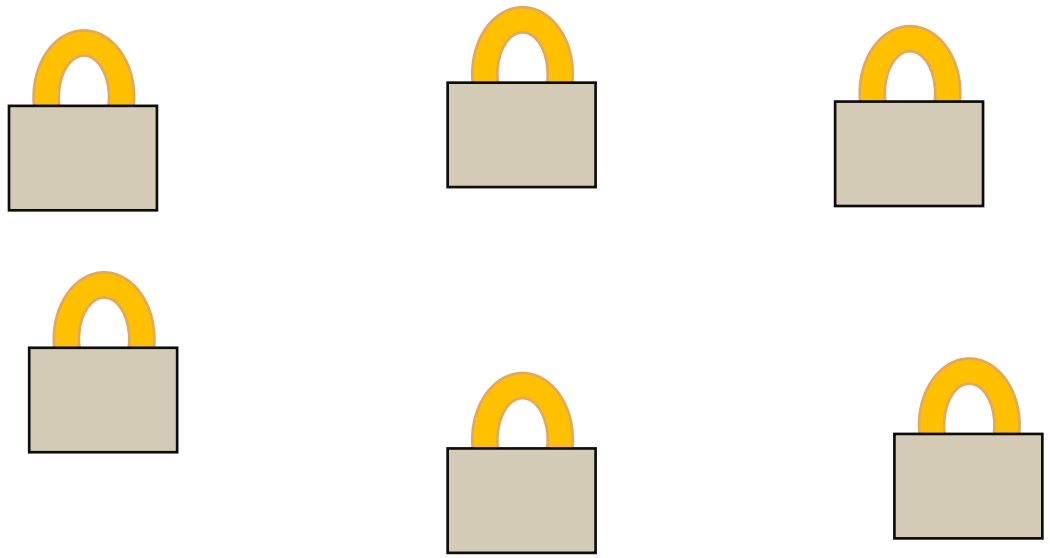
Problem: How do we reason about the system if we don't have ownership of the whole thing?

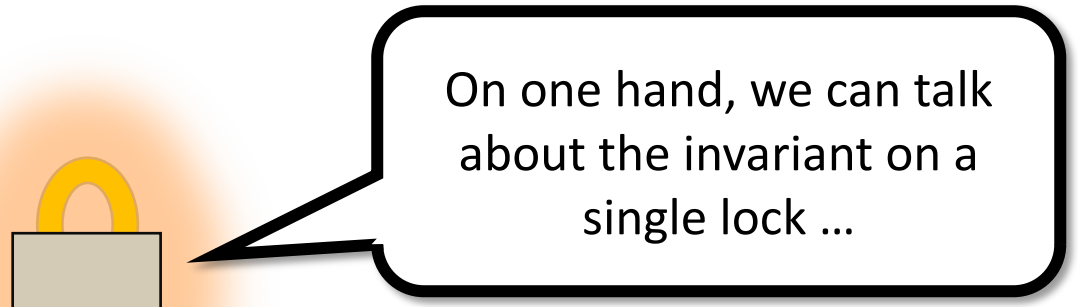


Problem: split ownership

True Concurrency requires us to have **multiple** components that can be owned and operated **independently**

Problem: How do we reason about the system if we don't have ownership of the whole thing?

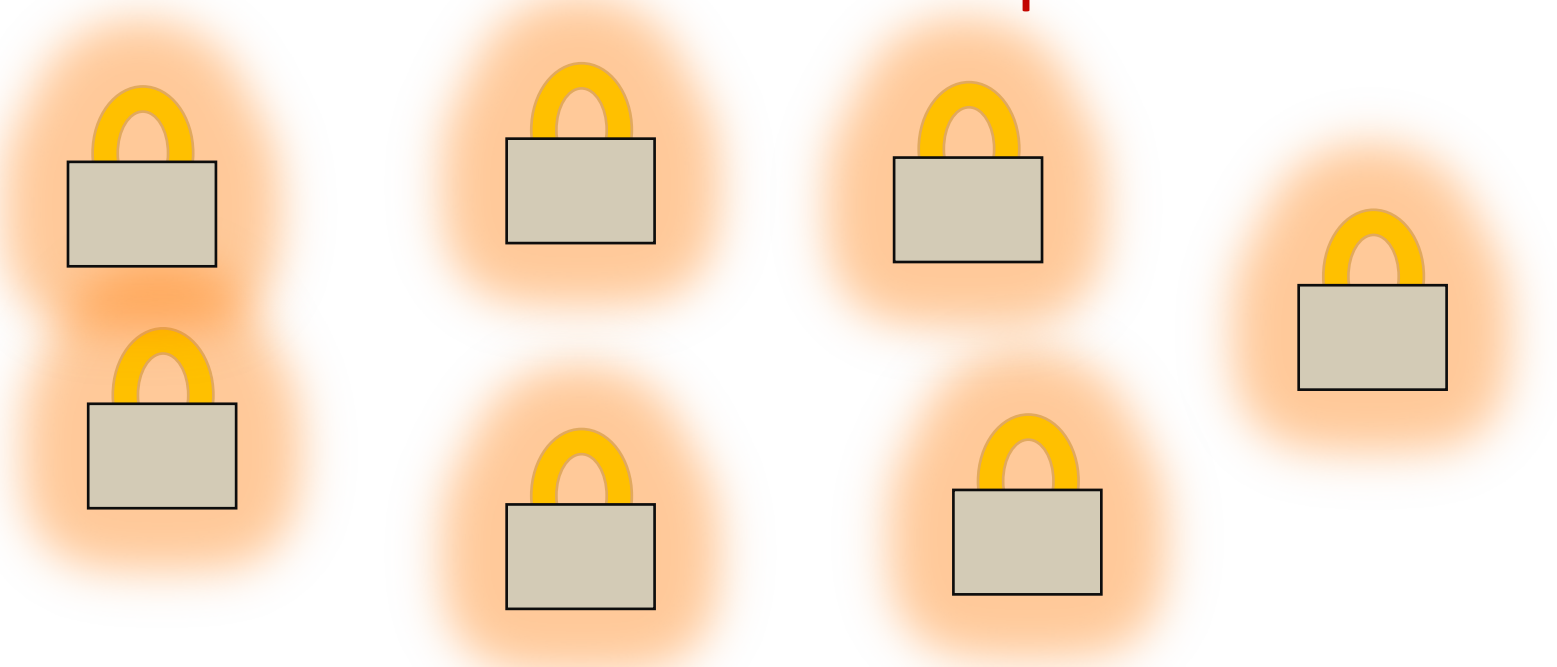




Problem: split ownership

True Concurrency requires us to have **multiple** components that can be owned and operated **independently**

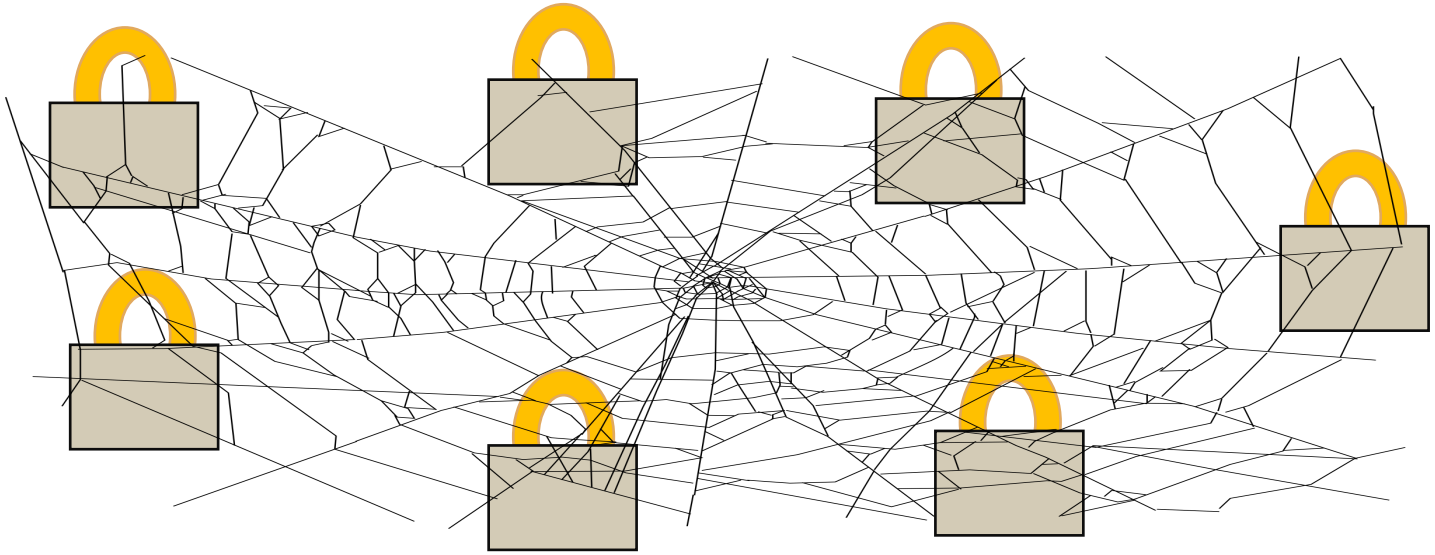
Problem: How do we reason about the system if we don't have ownership of the whole thing?



On one hand, we can talk about the invariant on a single lock …

But how do we relate them all **together**?

Problem: split ownership

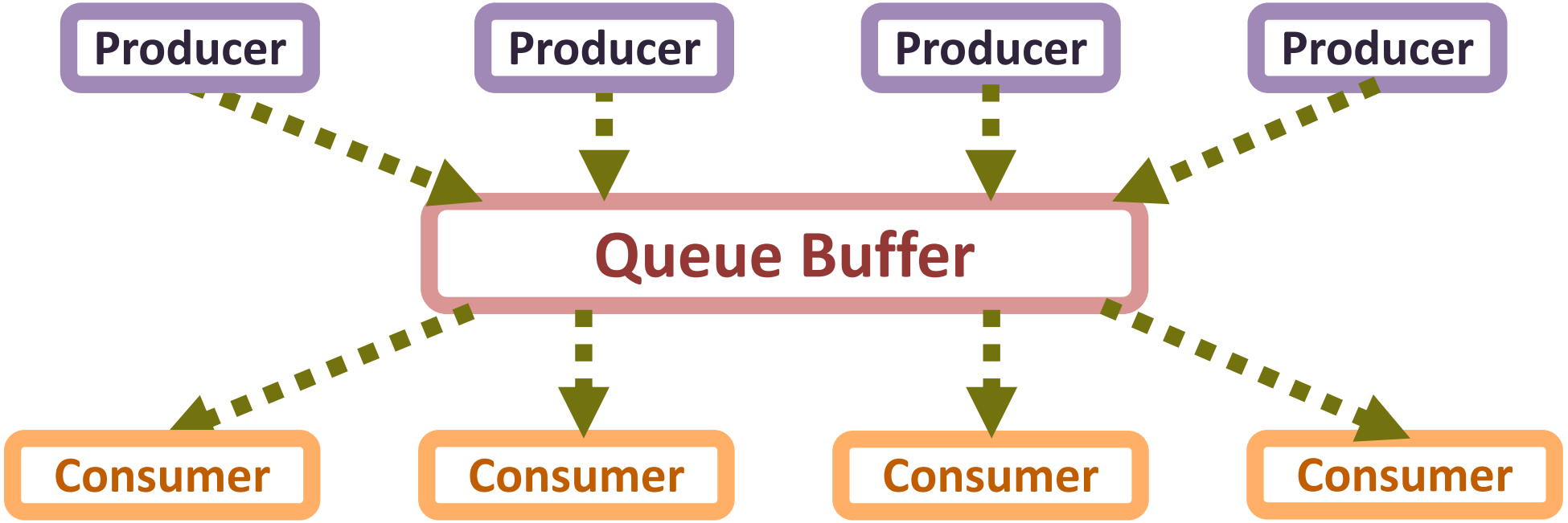


True Concurrency requires us to have **multiple** components that can be owned and operated **independently**

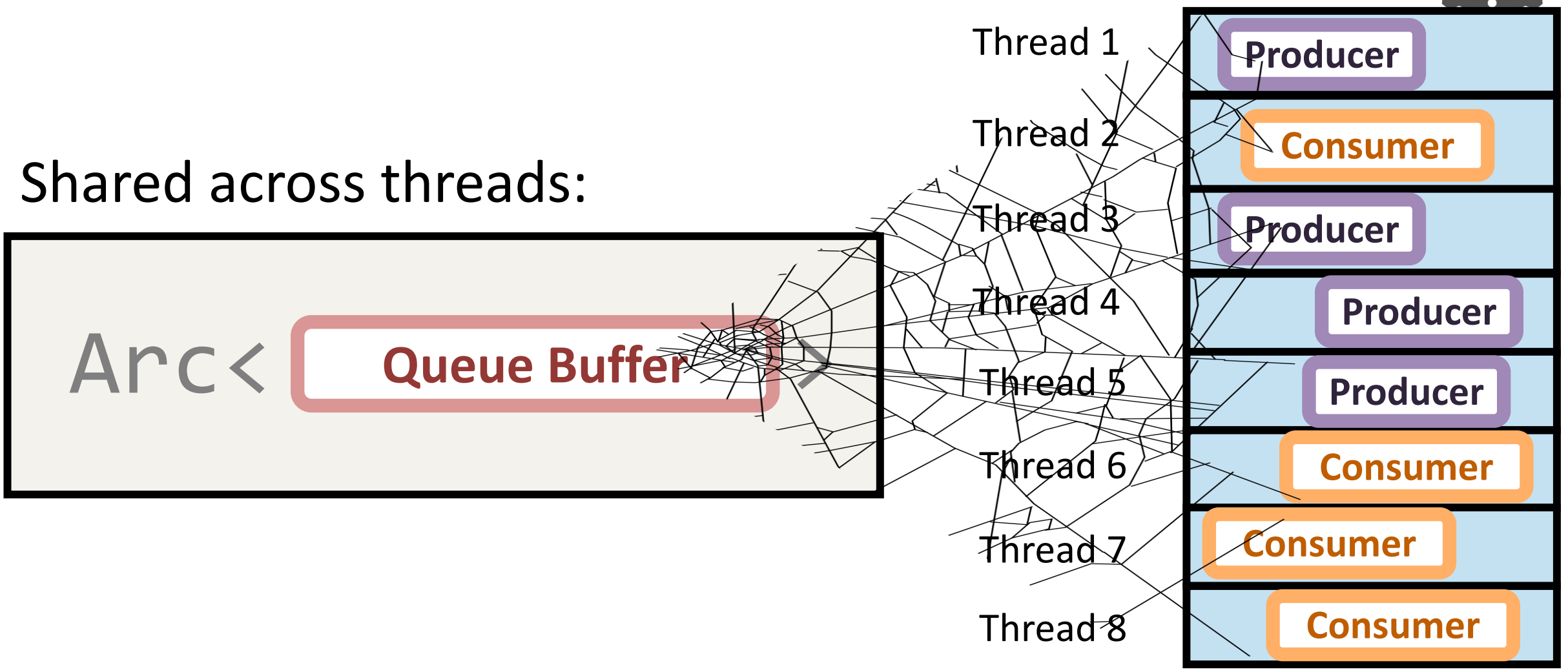
Problem: How do we reason about the system if we don't have ownership of the whole thing?

We need an invariant that "reaches across space"

Example 2: Producer/consumer queue



Example 2: Producer/consumer queue



Example 3: “Agreement”

```
struct Token {
    id: Id,
    value: u64
}


fn new_pair(value: u64) -> (pair: (Token, Token))
    ensures
        pair.0.id == pair.1.id,
        pair.0.value == value,
        pair.1.value == value,
{
    let id = fresh_id();
    (Token { id: id, value: value },
     Token { id: id, value: value })
}
```

```
fn update_pair(mut token0: Token,
  mut token1: Token, new_value: u64
) -> (pair: (Token, Token))
    requires token0.id == token1.id
    ensures
        pair.0.value == new_value,
        pair.1.value == new_value,
        pair.0.id == token0.id,
        pair.1.id == token1.id,
{
    token0.value = new_value;
    token1.value = new_value;
    (token0, token1)
}


fn validate_pair(token0: &Token, token1: &Token)
    requires token0.id == token1.id
    ensures token0.value == token1.value
{
    /* ? */
}
```

Example 3: “Agreement”

```
struct Token {
    id: Id,
    value: u64
}
```

```
fn new_pair(value: u64) -> (pair: (Token, Token))
    ensures
        pair.0.id == pair.1.id,
        pair.0.value == value,
        pair.1.value == value,
{
    let id = fresh_id();
    (Token { id: id, value: value },
     Token { id: id, value: value })
}
```

```
fn update_pair(mut token0: Token,
  mut token1: Token, new_value: u64
) -> (pair: (Token, Token))
    requires token0.id == token1.id
    ensures
        pair.0.value == new_value,
        pair.1.value == new_value,
        pair.0.id == token0.id,
        pair.1.id == token1.id,
{
    token0.value = new_value;
    token1.value = new_value;
    (token0, token1)
}
```

```
fn validate_pair(token0: &Token, token1: &Token)
    requires token0.id == token1.id
    ensures token0.value == token1.value
{
    /* ? */
}
```

Example 3: “Agreement”

```
struct Token {
    id: Id,
    value: u64
}
```

```
fn new_pair(value: u64) -> (pair: (Token, Token))
    ensures
        pair.0.id == pair.1.id,
        pair.0.value == value,
        pair.1.value == value,
{
    let id = fresh_id();
    (Token { id: id, value: value },
     Token { id: id, value: value })
}
```

```
fn update_pair(mut token0: Token,
  mut token1: Token, new_value: u64
) -> (pair: (Token, Token))
    requires token0.id == token1.id
    ensures
        pair.0.value == new_value,
        pair.1.value == new_value,
        pair.0.id == token0.id,
        pair.1.id == token1.id,
{
    token0.value = new_value;
    token1.value = new_value;
    (token0, token1)
}
```

```
fn validate_pair(token0: &Token, token1: &Token)
    requires token0.id == token1.id
    ensures token0.value == token1.value
{
    /* ? */
}
```

Example 3: “Agreement”

```
struct Token {
    id: Id,
    value: u64
}


fn new_pair(value: u64) -> (pair: (Token, Token))
    ensures
        pair.0.id == pair.1.id,
        pair.0.value == value,
        pair.1.value == value,
{
    let id = fresh_id();
    (Token { id: id, value: value },
     Token { id: id, value: value })
}
```

```
fn update_pair(mut token0: Token,
  mut token1: Token, new_value: u64
) -> (pair: (Token, Token))
    requires token0.id == token1.id
    ensures
        pair.0.value == new_value,
        pair.1.value == new_value,
        pair.0.id == token0.id,
        pair.1.id == token1.id,
{
    token0.value = new_value;
    token1.value = new_value;
    (token0, token1)
}
```

```
fn validate_pair(token0: &Token, token1: &Token)
    requires token0.id == token1.id
    ensures token0.value == token1.value
{
    /* ? */
}
```

Example 3: “Agreement”

```
struct Token {
    id: Id,
    value: u64
}


fn new_pair(value: u64) -> (pair: (Token, Token))
    ensures
        pair.0.id == pair.1.id,
        pair.0.value == value,
        pair.1.value == value,
{
    let id = fresh_id();
    (Token { id: id, value: value },
     Token { id: id, value: value })
}
```

```
fn update_pair(mut token0: Token,
  mut token1: Token, new_value: u64
) -> (pair: (Token, Token))
    requires token0.id == token1.id
    ensures
        pair.0.value == new_value,
        pair.1.value == new_value,
        pair.0.id == token0.id,
        pair.1.id == token1.id,
{
    token0.value = new_value;
    token1.value = new_value;
    (token0, token1)
}
```

```
fn validate_pair(token0: &Token, token1: &Token)
    requires token0.id == token1.id
    ensures token0.value == token1.value
{
    /* ? */
}
```

Example 3: “Agreement”

```
struct Token {
    id: Id,
    value: u64
}


fn new_pair(value: u64) -> (pair: (Token, Token))
    ensures
        pair.0.id == pair.1.id,
        pair.0.value == value,
        pair.1.value == value,
{
    let id = fresh_id();
    (Token { id: id, value: value },
     Token { id: id, value: value })
}
```

Who maintains this invariant?

```
fn update_pair(mut token0: Token,
  mut token1: Token, new_value: u64
) -> (pair: (Token, Token))
    requires token0.id == token1.id
    ensures
        pair.0.value == new_value,
        pair.1.value == new_value,
        pair.0.id == token0.id,
        pair.1.id == token1.id,
{
    token0.value = new_value;
    token1.value = new_value;
    (token0, token1)
}


fn validate_pair(token0: &Token, token1: &Token)
    requires token0.id == token1.id
    ensures token0.value == token1.value
{
    /* ? */
}
```

Verus's System: **VerusSync**

- Provides a means for "space-reaching invariants"
- Acknowledges that real systems need invariants a *little* more complicated than two-party agreement.

Verus's System: **VerusSync**

VerusSync provides:

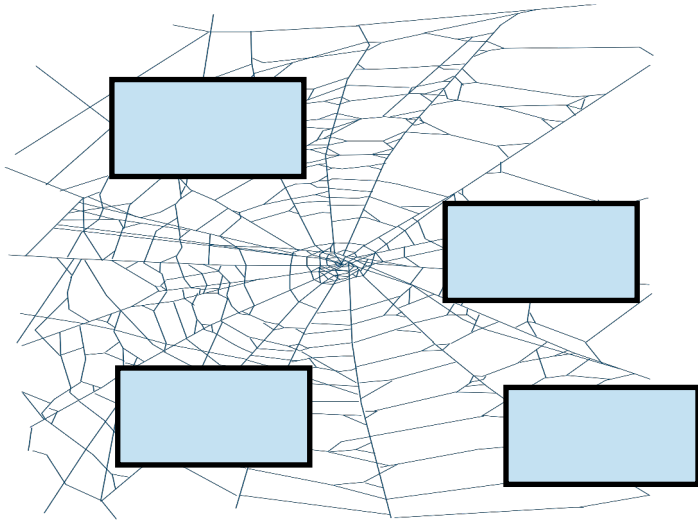
Ghost objects (like PointsTo)

Developer defines:

State

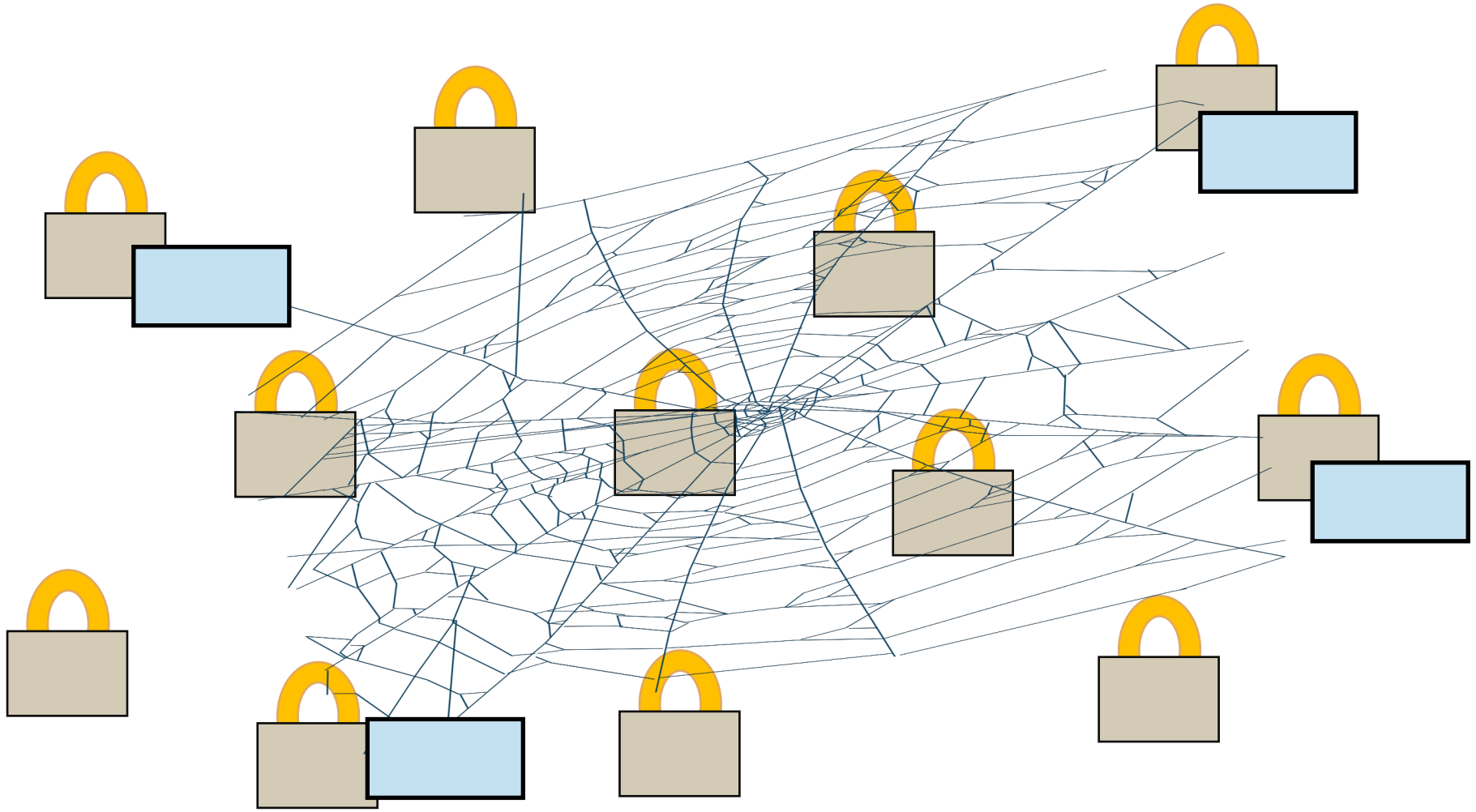
Operations

Invariants

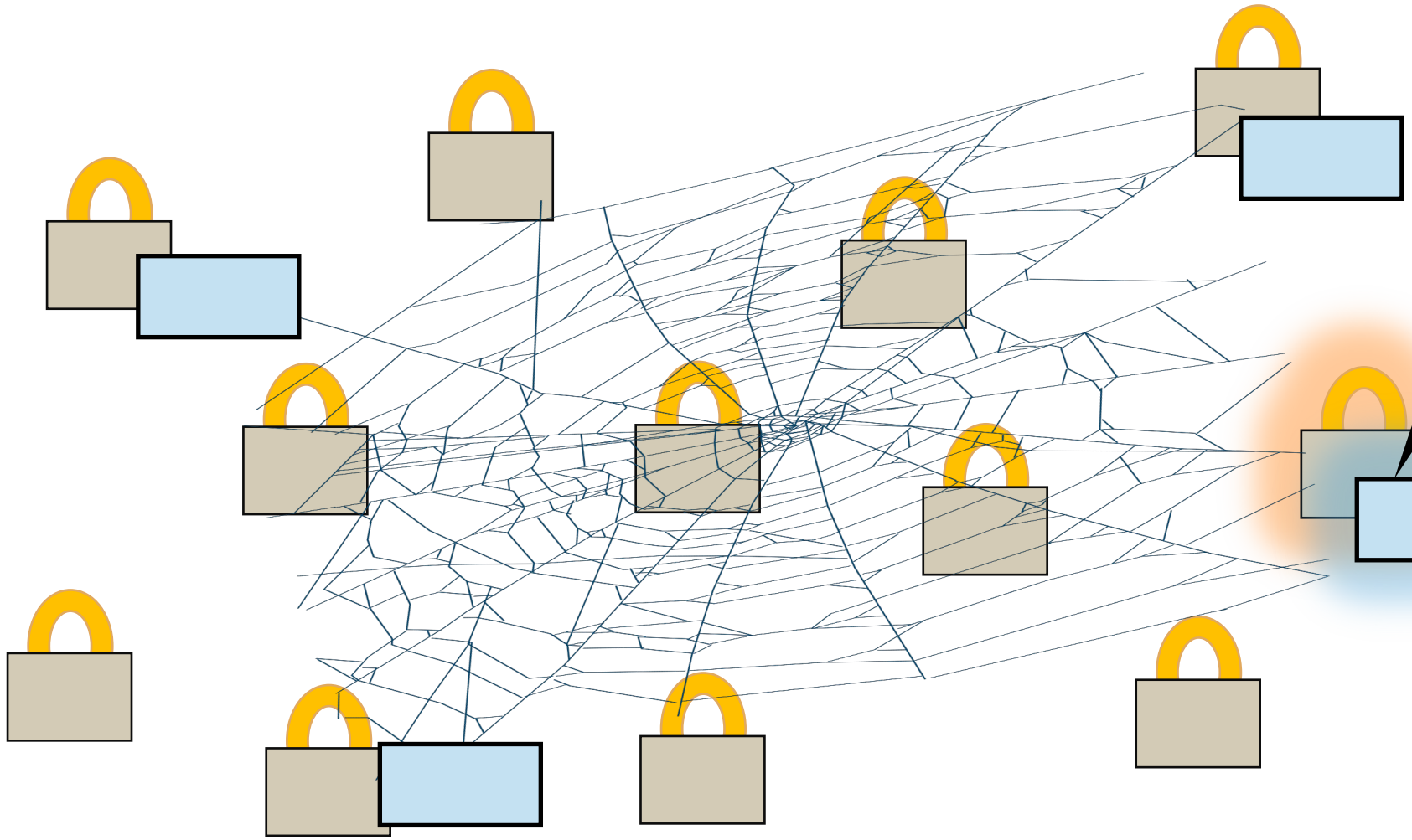


With a space-reaching invariants

Verus's System: **VerusSync**



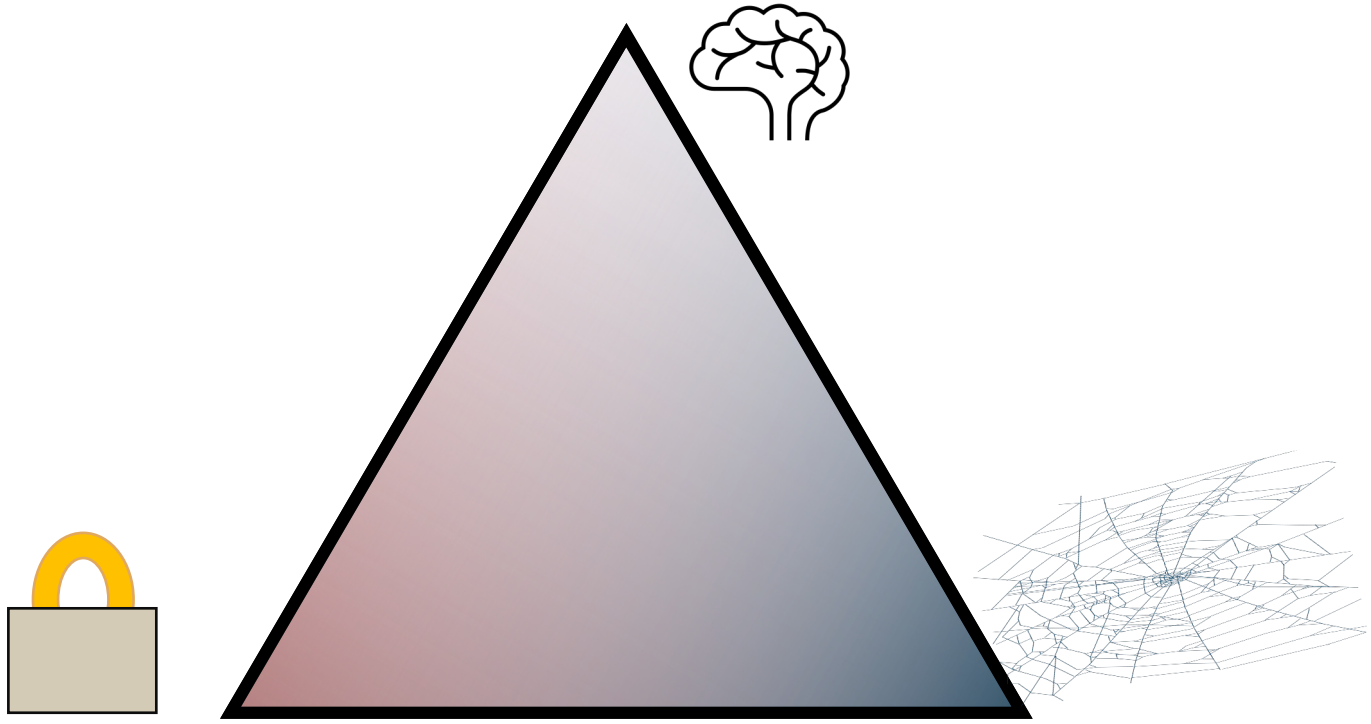
Verus's System: **VerusSync**



You can connect the ghost state to other state you care about via more lock invariants

[Demo: VerusSync]

Recap


Memory permissions
Lock invariants
Space-reaching invariants