The “latent spec” and its verification

Adventures in proving **domain-specific** correctness properties in real™ code

(For today, specifically in Tock OS)

I build automated verification methods for **latent specs**

What’s a “latent spec”?

A **domain-specific** property that must hold
for all* correct programs

Examples

- imperative programming ⇒
  No null dereferences.
- concurrency ⇒ No destructive data races.
- architecture ⇒ No impossible timing.

“But Mae, you do language design”

I do language design as a *way to enforce* a latent spec

- **Fearless concurrency**

“But Mae, you do language design”

I do language design as a *way to enforce* a latent spec

- Fearless concurrency
- **Safe uses of distributed consistency**

“But Mae, you do language design”

I do language design as a *way to enforce* a latent spec

- Fearless concurrency
- Safe uses of distributed consistency
- **Realizable hardware designs**

“But Mae, you do language design”

I do language design as a *way to enforce* a latent spec

- Fearless concurrency
- Safe uses of distributed consistency
- Realizable hardware designs
- **Efficiently compilable functional programs**

“But Mae, you do language design”

I do language design as a *way to enforce* a latent spec

- Fearless concurrency
- Safe uses of distributed consistency
- Realizable hardware designs
- Efficiently compilable functional programs
- **Stream processing**

“But Mae, you do language design”

I do language design as a *way to enforce* a latent spec

- Fearless concurrency
- Safe uses of distributed consistency
- Realizable hardware designs
- Efficiently compilable functional programs
- Stream processing
- **Choreographic high-performance code**

“But Mae, you do language design”

I do language design as a *way to enforce* a latent spec

- Fearless concurrency
- Safe uses of distributed consistency
- Realizable hardware designs
- Efficiently compilable functional programs
- Stream processing
- Choreographic high-performance code
- **Correct-by-construction microservices**

“But Mae, you do language design”

I do language design as a *way to enforce* a latent spec

- Fearless concurrency
- Safe uses of distributed consistency
- Realizable hardware designs
- Efficiently compilable functional programs
- Stream processing
- Choreographic high-performance code
- Correct-by-construction microservices
- **Keeping compositional intent in music generation**

“But Mae, you do language design”

I do language design as a *way to enforce* a latent spec

- Fearless concurrency
- Safe uses of distributed consistency
- Realizable hardware designs
- Efficiently compilable functional programs
- Stream processing
- Choreographic high-performance code
- Correct-by-construction microservices
- Keeping compositional intent in music generation
- **…**

The point of all this

Correctness becomes easier to establish:

- Domain experts can read the code more easily.
- Later verification can reuse the latent-spec proofs.

Part 2: Tock

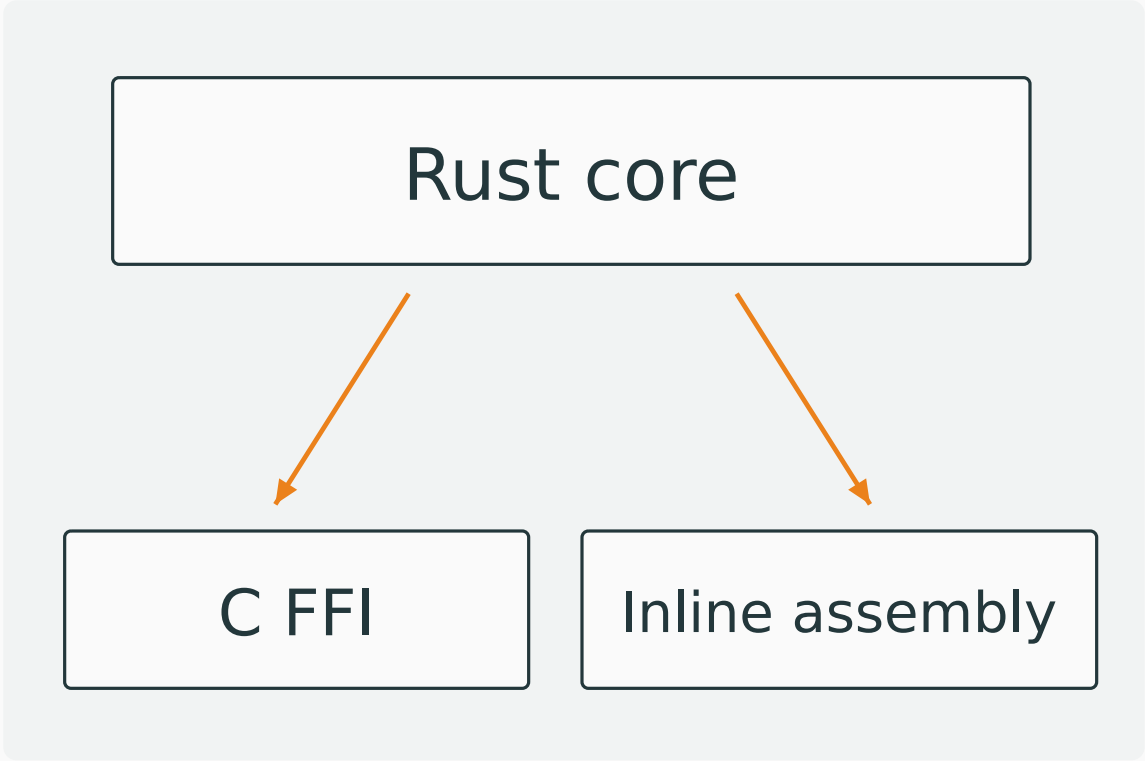
(and the C FFI)

Tock

Tock

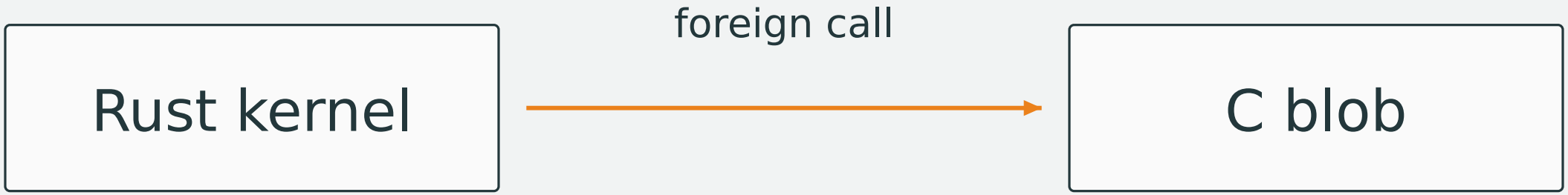
An embedded OS with
a well-typed Rust core.

Rust types express
OS-level safety invariants.



… plus some inline assembly and some C code :/

Why C code, tho?



For example, Bluetooth device firmware.

We may have no source code, and cannot assume a sensible software architecture.

Questions: FFI safety

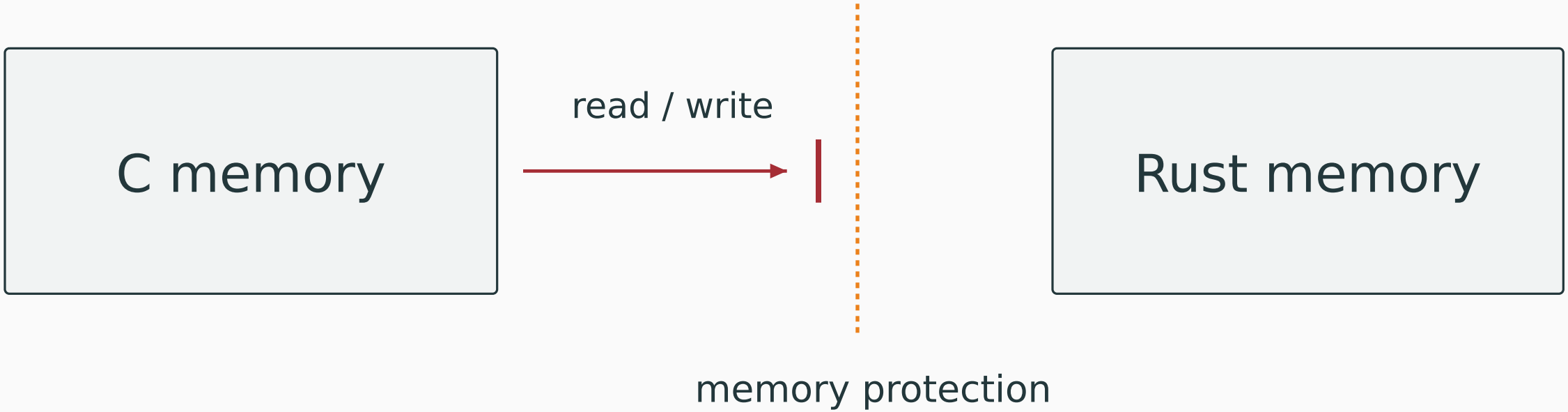
A Rust kernel calls a C blob.

What could go wrong?

Actual question—give me examples.

Ways Rust→C FFI Can Go Wrong

Fixing [many of] your suggestions



Restrict C’s accesses.

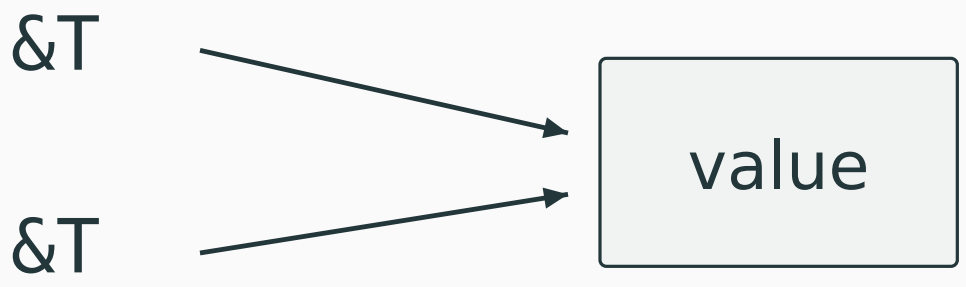
On a trapped memory fault, return to the FFI call with an error.

What about Rust’s **domain-specific** invariants?

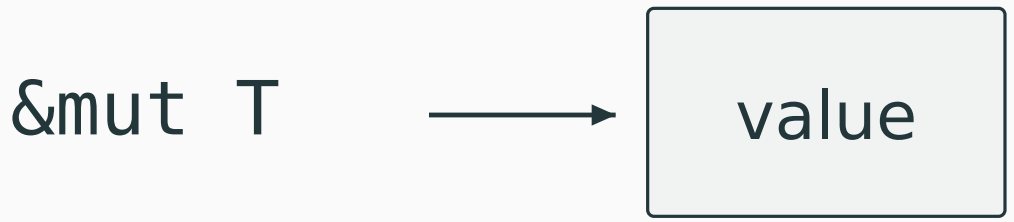
Rust ownership

Rust’s “aliasing xor mutability” discipline:

Shared references



One mutable reference



A mutable reference must be the only reference to that value.

Rust types and data validity

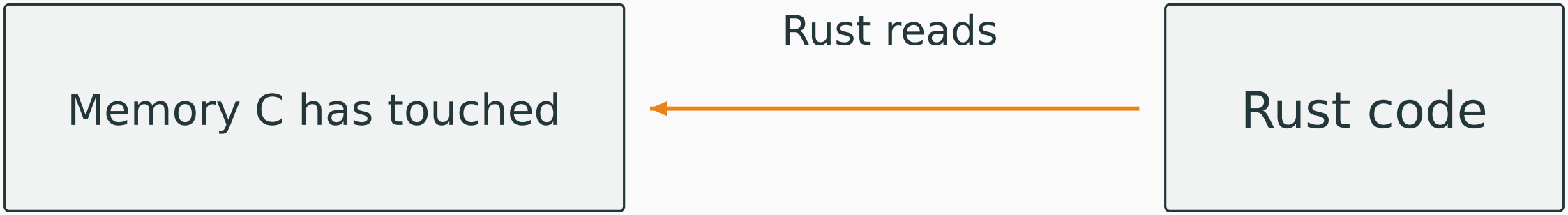
Fields contain values appropriate to their types.

References lead to valid, initialized memory.

> The same conditions apply recursively.

When dynamic enforcement isn’t enough

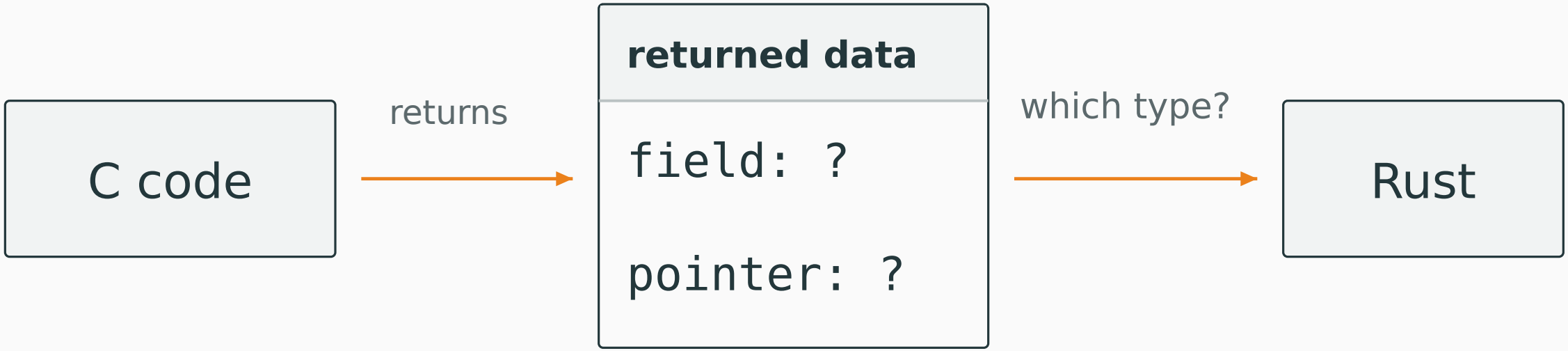
What can Rust assume about memory C has touched?



Accessible memory **may not be well-typed**

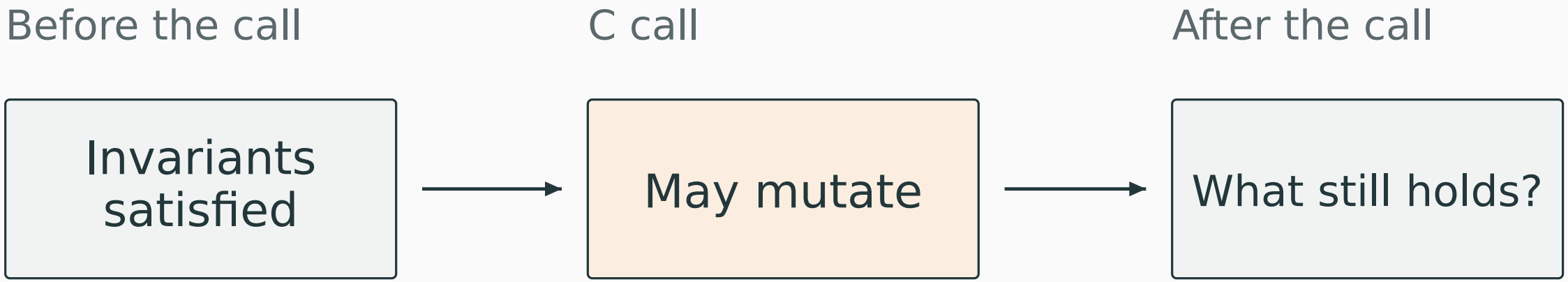
Question 1: data returned by C

C has chosen the contents of the returned memory.



How should Rust model the result without simply trusting C?

Question 2: memory passed to C



How should Rust model the same memory after C has changed it?

Side Note: Verification goal

Not the goal

Prove what the C code does.

The goal

> An FFI call cannot violate the safety of the surrounding Rust code.

Remember: ONLY the latent spec

We’re the OS authors. The checks have to fit how we build the code, without much extra programmer or system effort.

Out of scope

- Modeling each C module’s behavior.
- Adding pre/postconditions around C.
- Tracking more aliases than Rust does.
- Assembly-level verification of the modules.

Omniglot: dynamic and static checks

Checks at the boundary, combined with Rust’s borrowing rules.

Dynamic checks

Restrict C’s memory accesses.
Check pointers and values.

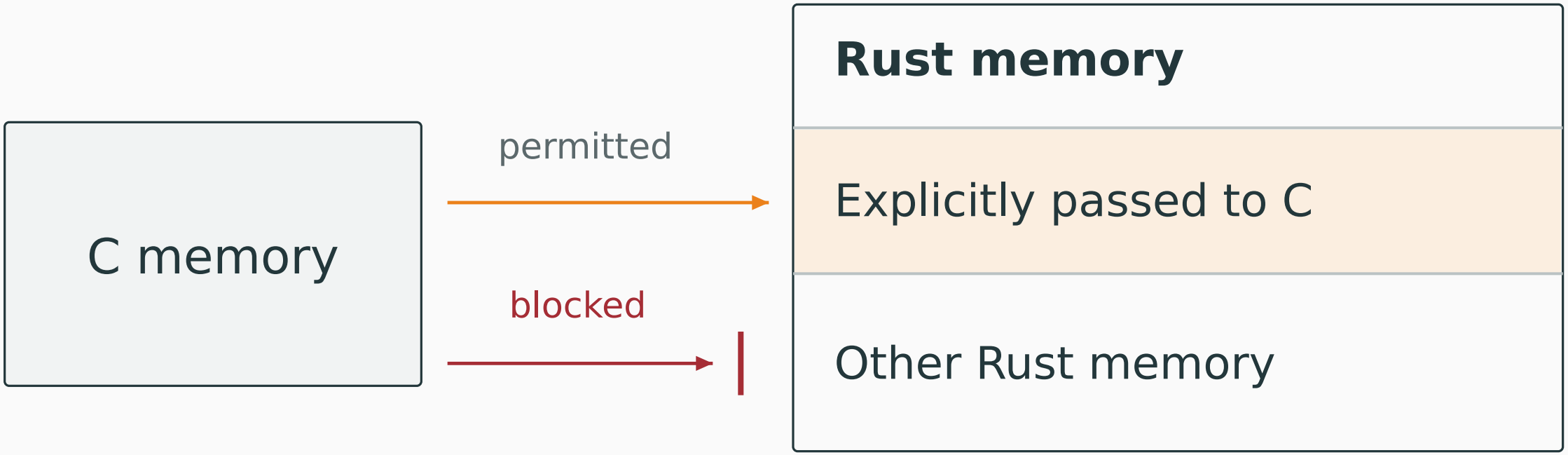
Static checks

Represent checked facts in types.
Limit how long they may be used.

(boldly stolen from Leon Schuermann’s talk on Omniglot)

Dynamic checks: memory isolation

C cannot access Rust memory unless it is explicitly passed to C.



(We use RISC-V PMP)

Dynamic checks: returned values

A value must be valid before Rust receives it as that type.

Foreign return bytes

`0x02`

invalid bool

Native Rust binding

```
fn aes_encrypt(...)
    -> bool
```

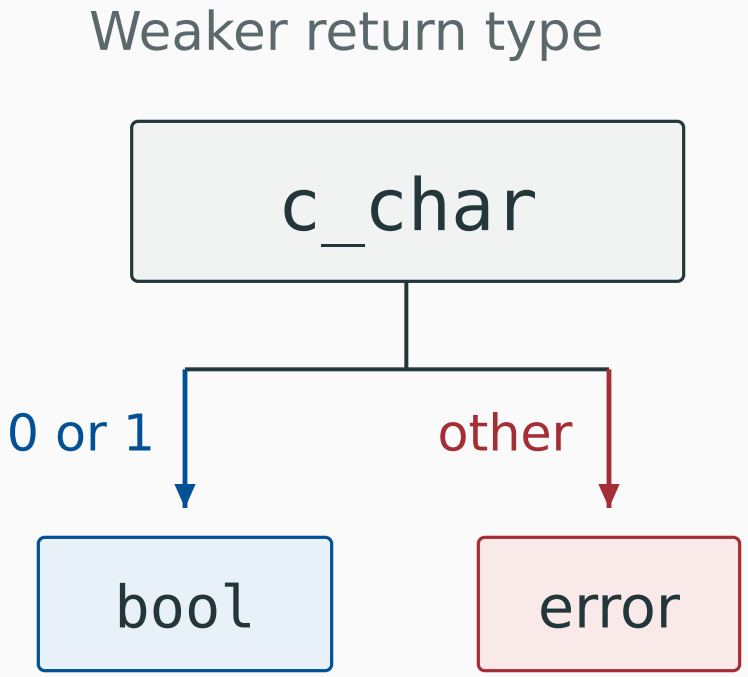
> Checking after receiving a Rust `bool` would already be too late.

Dynamic checks: validation wrapper

Call, then check the result.

Here: c_char has the same size/alignment, but no invalid bit patterns.

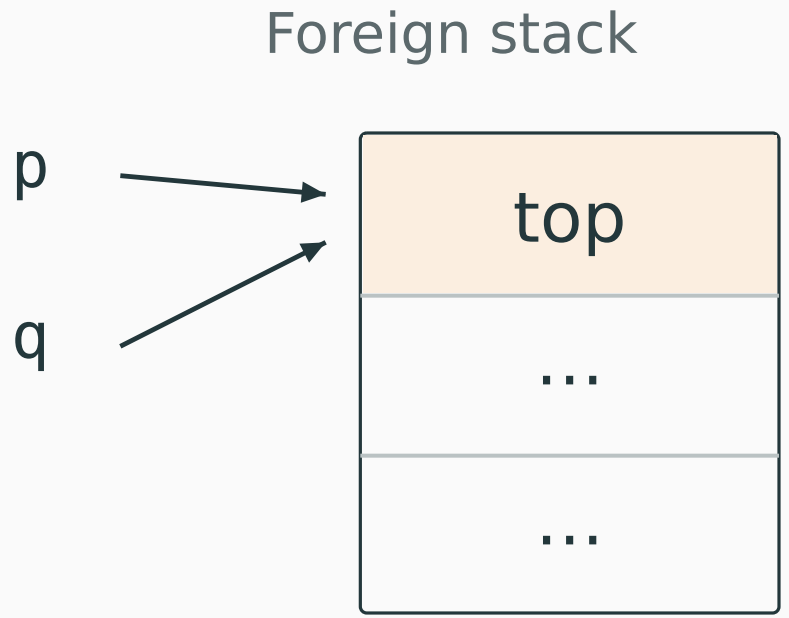
```
fn aes_encrypt_wrapped()
    -> Result<bool, Error> {
  let raw: c_char = aes_encrypt_weaker();
  match raw {
    0 => Ok(false),
    1 => Ok(true),
    _ => Err(ValidationFail),
  }
}
```



Foreign memory: aliasing example

Different calls may return overlapping pointers.

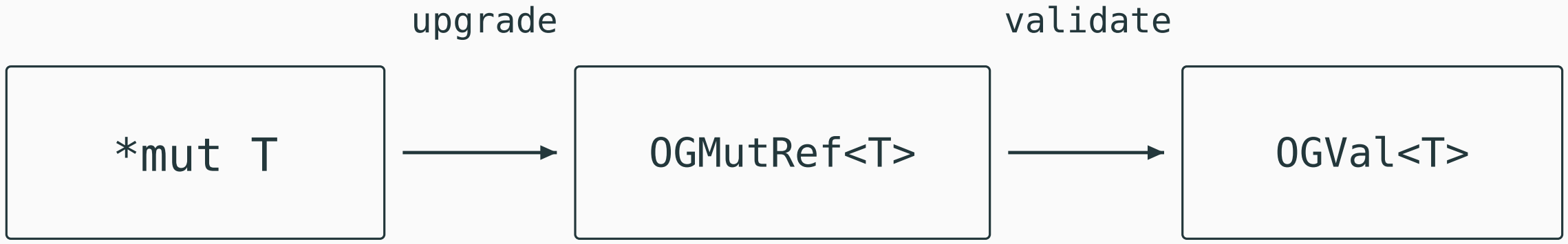
```
let p = peek();
let q = peek();
// p and q may be equal
```



These cannot simply become overlapping `&T` and `&mut T` references.

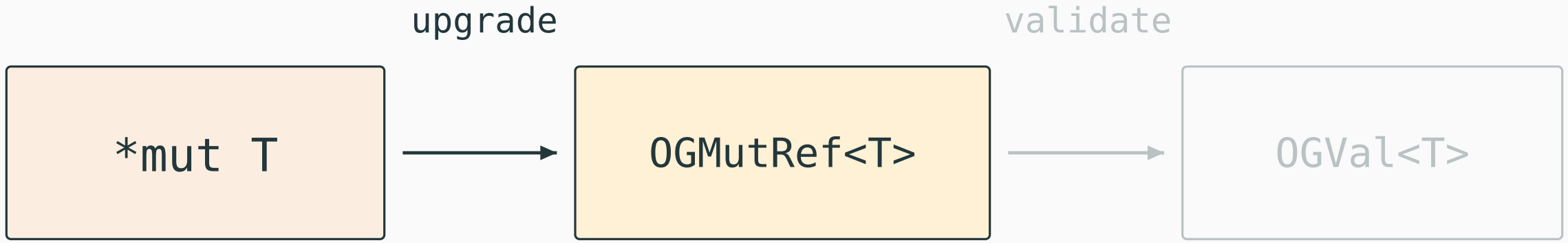
Foreign memory: reference types

Gain guarantees in stages.

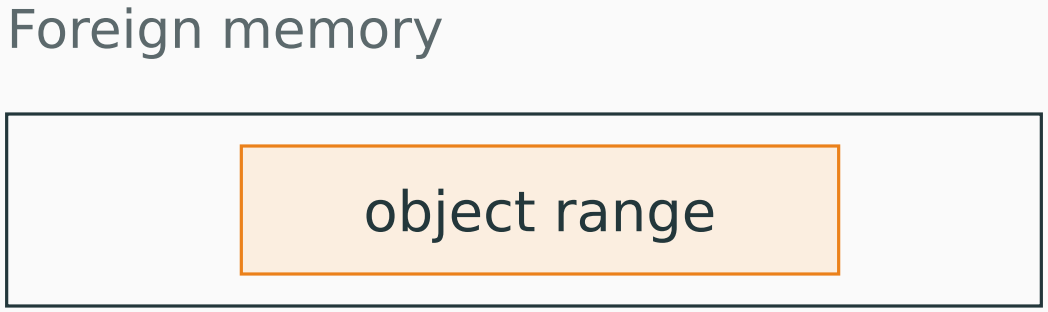


Dynamic checks: pointer upgrade

Establish access before using the pointer.



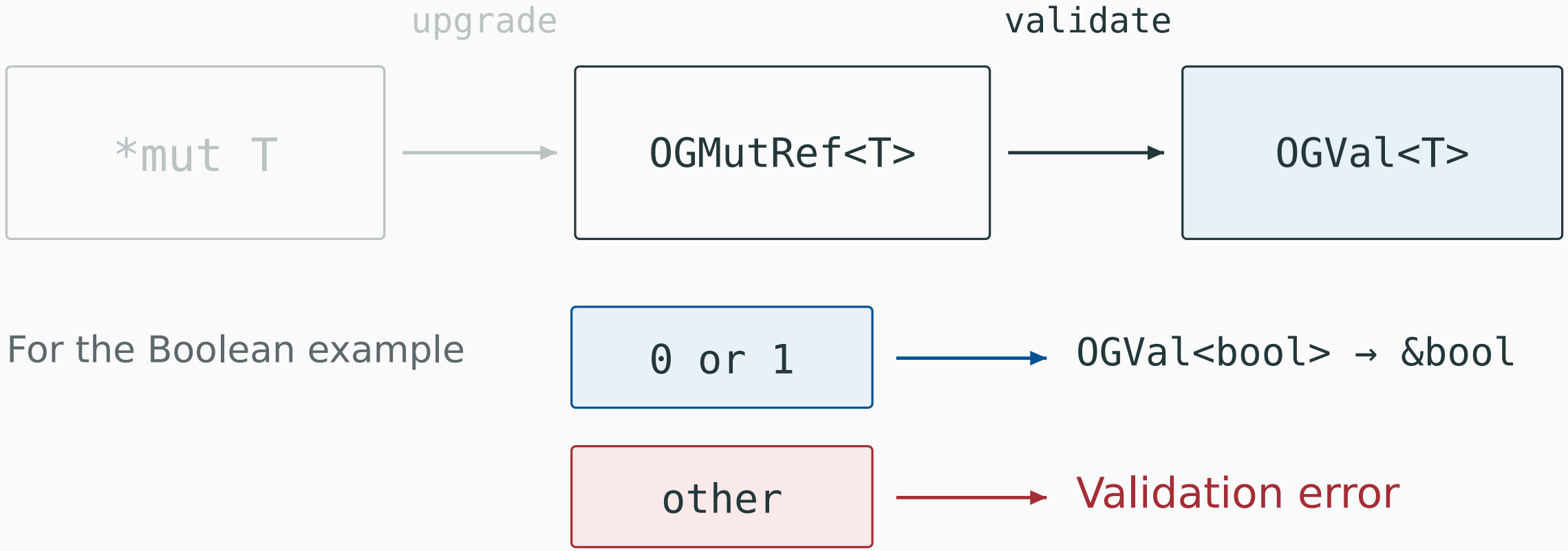
- The pointer is well-aligned.
- The whole object is accessible.



The range is accessible; its contents have not yet been checked as a T.

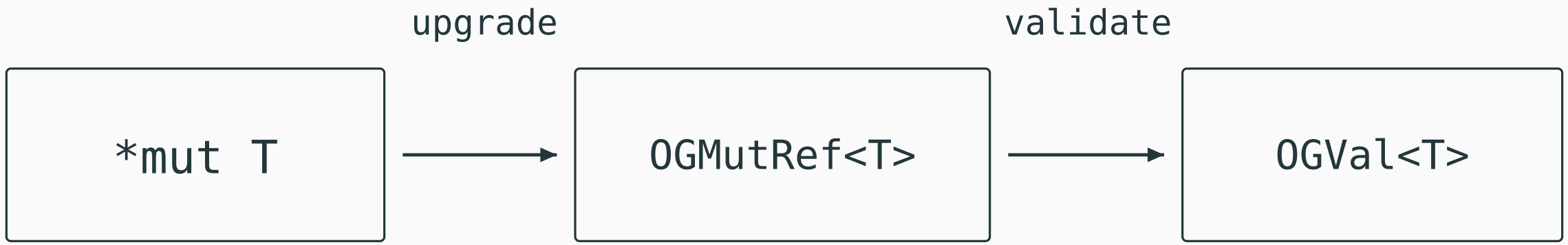
Dynamic checks: pointee validation

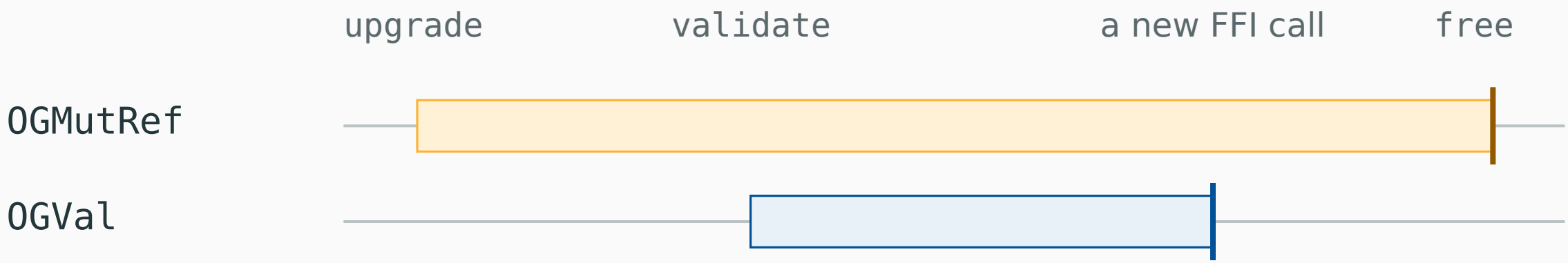
Check that the pointee is a valid value of the intended type.



(the bytes stay in foreign memory)

Static checks: invalidation





Static checks: scope markers

Think of these like static capabilities, one per library.

Allocation scope

Validating **writeability** borrows □

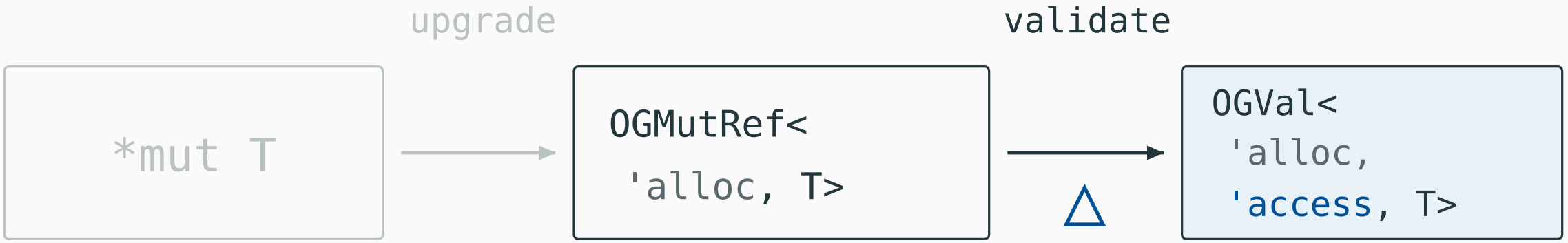
Access scope

Validating **readability** borrows ▵ (and also □)

> Invalidating operations require a unique borrow of the corresponding marker.

Static checks: access scope

△ Validation takes a shared borrow of the access marker.



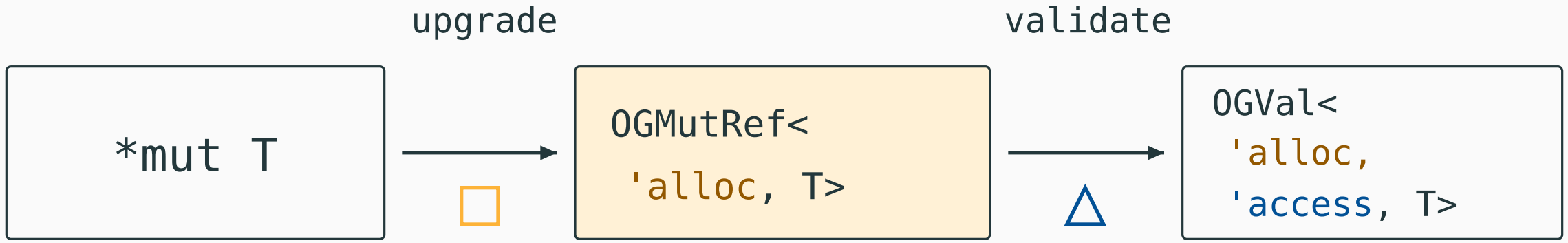

```
write(..., &mut access)
invoke(..., &mut access)
```

▲ These need a unique borrow of that marker.

Rust rejects them while a previously validated view still needs the shared borrow.

Static checks: allocation scope

- Upgrade takes a shared borrow of the allocation marker.




```
free(..., &mut alloc)
```

- Revocation needs a unique borrow.

Neither wrapper can remain usable across that revocation.

Static checks: operation requirements

The same Rust borrowing rules govern all of these operations.

Operation	Marker borrow	What follows
`upgrade`	 `&alloc`	Accessible, writable wrapper
`validate`	 `&access`	Validated, readable wrapper
`write / invoke`	 `&mut access`	No old validated view in use
`free`	 `&mut alloc`	No old allocation-bound view

Outline shapes = shared borrows; filled shapes = unique borrows.

Stack example: rejected sequence

```
let a = p.upgrade(&alloc);
let b = q.upgrade(&alloc);

let v = a.validate(&access);
b.write(42, &mut access);
println!("{}", v);
```

Stack example: rejected sequence

```
let a = p.upgrade(&alloc);
let b = q.upgrade(&alloc);

let v = a.validate(&access);
b.write(42, &mut access);
println!("{}", v);
```

Shared □ borrow

Stack example: rejected sequence

```
let a = p.upgrade(&alloc);
let b = q.upgrade(&alloc);

let v = a.validate(&access);
b.write(42, &mut access);
println!("{}", v);
```

Shared □ borrow

→ **Rejected**

Both need &mut ▲

Stack example: rejected sequence

Suppose p and q point to the same stack element.

```
let a = p.upgrade(&alloc);
let b = q.upgrade(&alloc);

let v = a.validate(&access);
b.write(42, &mut access);
println!("{}", v);
```

Shared □ borrow

→ **Rejected**

Both need &mut ▲

Stack example: rejected sequence

Suppose p and q point to the same stack element.

```
let a = p.upgrade(&alloc);
let b = q.upgrade(&alloc);

let v = a.validate(&access);
b.write(42, &mut access);
println!("{}", v);
```

Shared □ borrow

→ **Rejected**

Both need &mut ▲

The later use of v keeps its shared borrow live across the write.

Stack example: permitted sequence

Finish using the old view, write, then validate again.

```
{
    let v = a.validate(&access);
    println!("{}", v);
}
b.write(42, &mut access);
let v = a.validate(&access);
println!("{}", v);
```

C-touched memory: the two cases

For memory accessed through the interface just described:

C returns data

Do not assume the returned representation is a valid T.

Check values; upgrade and validate returned pointers.

C changes memory

The call needs `&mut access`. Old checked views cannot remain in use across it.

Validate again before reading.

Dynamic and static checks: summary

Dynamic checks

Establish accessible memory and values that satisfy their types' requirements.

Static checks

Shared marker borrows keep views usable. Invalidating operations need unique borrows.

> This does not prove what the C computation is supposed to do.

Questions: dynamic and static checks

Questions about the value checks
or the allocation/access scopes?

Which parts of this boundary would you want
to see specified more precisely?

Part 3: Tock

(and the inline assembly)

Question: inline assembly

Should the FFI approach work for inline assembly?

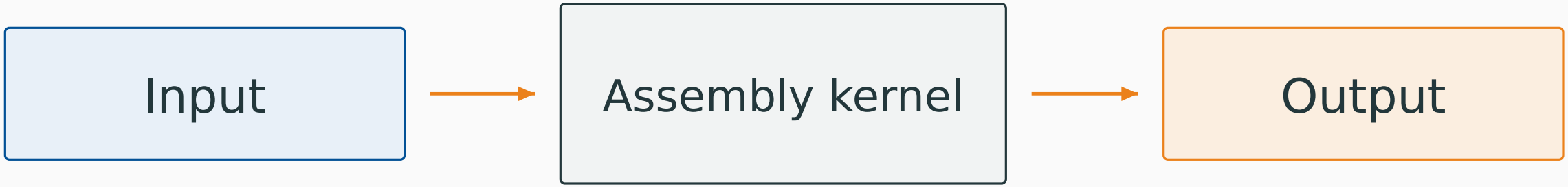
Same approach.

More work to do.

The assembly we’re targeting

Small computational kernels

Cryptography, vectorized loops, input and output buffers.



What we model

Arithmetic, loads and stores, tests and branches.

What we exclude

Privileged instructions, function calls and system calls.

Verification goal

Verify that inline assembly satisfies Rust’s memory-safety requirements.

Rust wrapper

```rust
fn copy_words(
    src: &[u32],
    dst: &mut [MaybeUninit<u32>],
) {
    assert_eq!(src.len(), dst.len());
    unsafe {
        asm!(include_str!("copy-loop.s"),
            inout("a0") src.as_ptr() => _,
            inout("a1") dst.as_mut_ptr() => _,
            in("a2") src.len(),
            out("t0") _, out("t1") _,
            options(nostack),
        );
    }
}
```

Memory contract

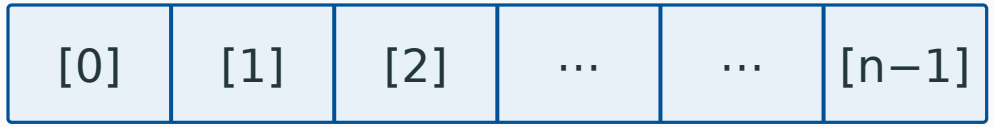
`n = src.len() = dst.len()`

`src: &[u32]`

Shared input

Initialized u32 values.

Read only.



`a0 = S · [S, S + 4n)`

`dst:`

`&mut [MaybeUninit<u32>]`

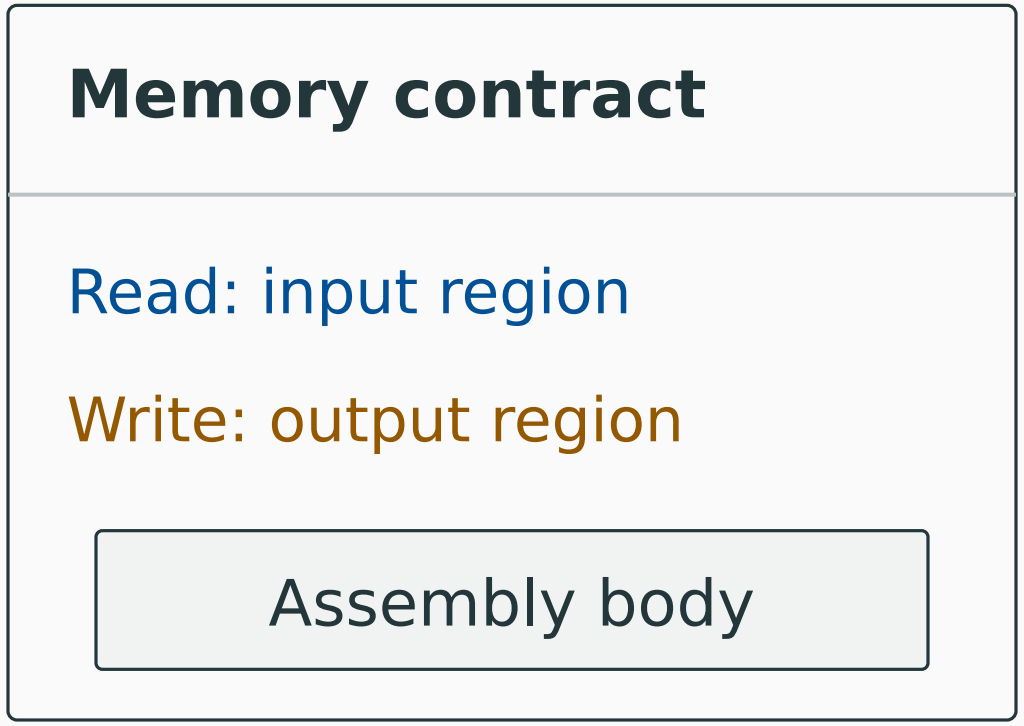
May be uninitialized.

Exclusive writes.

`a1 = D · [D, D + 4n)`

Assembly analysis

Check the assembly body against the memory contract.



- Find loop candidates.
- Accumulate SMT formulas; check memory accesses.
- Infer an index recurrence.
- Prove the invariant.
- Check the access bounds.

Loop candidates

Backward jumps must be loops over input data.

```
li t0, 0
beq a2, zero, 3f
2:
  lw t1, 0(a0)
  sw t1, 0(a1)
  addi a0, a0, 4
  addi a1, a1, 4
  addi t0, t0, 1
  bltu t0, a2, 2b
3:
```

Candidate starts at label 2:

Abstract interpretation

Accumulate SMT formulas and check each memory access.

```
li t0, 0
beq a2, zero, 3f
2:
  lw t1, 0(a0)
  sw t1, 0(a1)
  addi a0, a0, 4
  addi a1, a1, 4
  addi t0, t0, 1
  bltu t0, a2, 2b
3:
```

Entry on the `n > 0` path

`a0=S, a1=D, t0=0`

Abstract interpretation

Accumulate SMT formulas and check each memory access.

```
li t0, 0
beq a2, zero, 3f
2:
  lw t1, 0(a0)
  sw t1, 0(a1)
  addi a0, a0, 4
  addi a1, a1, 4
  addi t0, t0, 1
  bltu t0, a2, 2b
3:
```

Entry on the `n > 0` path

`a0=S, a1=D, t0=0`

Load · allowed input read

```
t1 = load32(M0, S)
```

Abstract interpretation

Accumulate SMT formulas and check each memory access.

```
li t0, 0
beq a2, zero, 3f
2:
  lw t1, 0(a0)
  sw t1, 0(a1)
  addi a0, a0, 4
  addi a1, a1, 4
  addi t0, t0, 1
  bltu t0, a2, 2b
3:
```

Entry on the `n > 0` path

`a0=S, a1=D, t0=0`

Load · allowed input read

`t1 = load32(M0, S)`

Store · allowed output write

`M1 = store32(M0, D, t1)`

Abstract interpretation

Accumulate SMT formulas and check each memory access.

```
li t0, 0
beq a2, zero, 3f
2:
  lw t1, 0(a0)
  sw t1, 0(a1)
  addi a0, a0, 4
  addi a1, a1, 4
  addi t0, t0, 1
  bltu t0, a2, 2b
3:
```

Entry on the `n > 0` path

`a0=S, a1=D, t0=0`

Load · allowed input read

`t1 = load32(M0, S)`

Store · allowed output write

`M1 = store32(M0, D, t1)`

Advance and test the bound

`a0′=S+4, a1′=D+4`

`t0′=1; back if 1<n`

Finding an invariant

For n > 2, two backward traversals give:

Iterations (k)	t0	a0	a1
0	0	S	D

Finding an invariant

For n > 2, two backward traversals give:

Iterations (k)	t0	a0	a1
0	0	S	D
1	1	S + 4	D + 4

Finding an invariant

For `n > 2`, two backward traversals give:

Iterations (k)	t0	a0	a1
0	0	S	D
1	1	S + 4	D + 4
2	2	S + 8	D + 8

Candidate after k iterations

```
t0=k      a0=S+4k      a1=D+4k
```

Checking the invariant

Ask SMT to prove the base case and the step.

$I(k)$: `t0=k, a0=S+4k, a1=D+4k`

$0 \le k < n$

First loop entry

$n > 0, \quad k = 0$

`a0=S, a1=D, t0=0`

Next loop entry

$k' = k + 1$

`a0′ = S + 4(k+1)`

`a1′ = D + 4(k+1)`

Back edge gives $k' < n$.

Checking the bounds

```
beq a2, zero, 3f
bltu t0, a2, 2b
```

Empty input: no memory access.

At each loop entry: $0 \le k < n$.

$$0 \le k < n \quad \Rightarrow \quad 0 \le 4k \quad \text{and} \quad 4k + 4 \le 4n$$

Read $[S + 4k, S + 4k + 4) \subseteq [S, S + 4n)$

Write $[D + 4k, D + 4k + 4) \subseteq [D, D + 4n)$

Implementation choice

Performance

Analysis cost

Why keep it this small?

- Small computational kernels.
- Bounded formula generation.
- Reject code outside the target class.

milliseconds

On our small examples.

Question: existing tools

What could we have used off the shelf?

Summary

C FFI	Reflecting **dynamic guarantees** into **typestate** enables pay-as-you-go validation!
Inline assembly	OS inline ASM is heavily **structured**, which enables even naive **automated** memory safety verification
Latent specs	You can **materially improve** code correctness, **without** keeping the modelers employed