

Supporting Weak Memory in Creusot and Verus

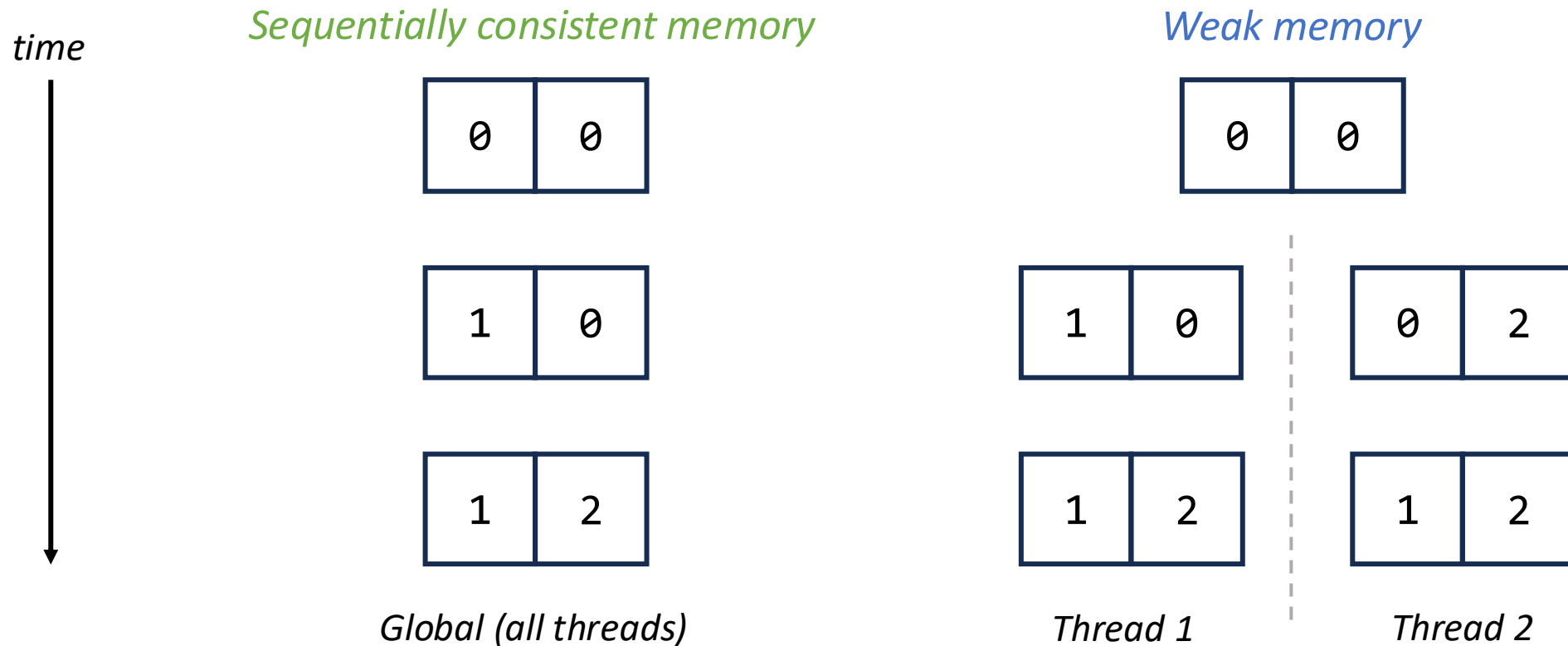
Rust Verification Workshop 2026

Natalie Neamtu^{*1}, *Vincent Lafeychine*^{*2}, *Laila Elbeheiry*³, *Elanor Tang*¹,
*Hoang-Hai Dang*⁴, *Derek Dreyer*³, *Travis Hance*³, *Chris Hawblitzel*⁵,
*Jacques-Henri Jourdan*², *Bryan Parno*¹

¹Carnegie Mellon University ²Université Paris-Saclay ³MPI-SWS ⁴BlueRock Security ⁵Microsoft Research

Weak Memory

- A **weak memory model** allows threads to have different views of memory



Weak Memory

- Weak memory can result in **more efficient code**
 - Permits more compiler and hardware reordering
- Weak memory can introduce **bugs**

Weak Memory in Rust

Concurrency primitives in `std::sync` and popular crates utilize Rust's [weak memory model](#).

`std::sync`

Struct Mutex

Since 1.0.0 · [Source](#)

`std::sync`

Struct RwLock

Since 1.0.0 · [Source](#)

`std::sync`

Struct Arc 

Since 1.0.0 · [Source](#)



crates.io

crossbeam v0.8.4

Tools for concurrent programming

↓ All-Time: 88,010,945

↓ Recent: 12,809,372

🔄 Updated: about 2 years ago

Weak Memory in Rust



When should one use `Relaxed` memory ordering

help



Ordering::Relaxed vs Acquire+Release?



rust-lang / rust

Q Type / to search

<> Code

Issues 5k+

Pull requests 1.1k

Actions

Projects

Security 6

Insights

Insufficient synchronization in `Arc::get_mut` #51780

Closed

Listed in #10

Weak Memory in Rust

- Practical (efficient) concurrency
- Source of developer confusion and bugs

Until now, Creusot and Verus did not support verification of programs utilizing weak memory

Supporting Weak Memory in Creusot & Verus

Roadmap to Weak Memory Support

Separation logics for weak memory

- Sound reasoning principles for weak memory

RustBelt Relaxed

- Verify safety of complex Rust libraries, under weak memory, using Iris

Our approach:
Integrate **weak memory reasoning principles** into **Rust type system**

Verification within Rust type system

Creusot & Verus

- Leverage **borrow-checked ghost types** to verify unsafe Rust
- Only sound for sequential consistency

Creusot & Verus
+ weak memory

This Talk

- Our Approach
 - Sound reasoning for **pointer permissions & invariants** in **Rust type system**
 - Simplifications to **Atomic API** using **borrow checker**
- Current Progress & Case Study
- Additional Ongoing Work

Why Verification Breaks Under Weak Memory

Example Unverified Program

```
let x = Ptr::new(0); let y = AtomicU8::new(0);
```

```
x.write(42);
```

```
y.fetch_add(1, Ordering::[...]);
```

```
||| if 1 == y.fetch_add(1, Ordering::[...]) {  
|||   x.free();  
||| }
```

atomic::Ordering

```
let atomic = AtomicU8::new(0);  
atomic.fetch_add(1, Ordering::[...]);
```

Ordering::SeqCst

*Provides **sequential consistency***

Ordering::RelAcq

Ordering::Release

Ordering::Acquire

Ordering::Relaxed

*Exposes **weak memory model***

Creusot & Verus: Ptr API

Borrow checker $\Rightarrow$ no data races!

```
Ptr<T> {  
    new(v: T) -> (Self, PtrPerm<T>)           Borrow-checked ghost permission  
    read(self, &PtrPerm<T>)  
    write(self, &mut PtrPerm<T>)  
    free(self, PtrPerm<T>)  
}  
  
impl<T> Copy for Ptr<T> {}
```

Creusot & Verus: Verification Under Sequential Consistency

```
let (x, P_x) = Ptr::new(0); let y = AtomicU8::new(0);  
    borrow-checked ghost permission
```

```
x.write(42, &mut P_x);  
y.fetch_add(1, SeqCst);
```

```
if 1 == y.fetch_add(1, SeqCst)  
{  
    x.free(P_x);  
}
```

Creusot & Verus: Invariants

```
let (x, P_x) = Ptr::new(0); let y = AtomicU8::new(0);  
      borrow-checked ghost permission
```

```
x.write(42, &mut P_x);
```

```
y.fetch_add(1, SeqCst);
```

```
inv_y = Some(P_x);
```

ghost update

invariant

inv_y

P_x

inv_y: y = 1 ==>

P_x

```
if 1 == y.fetch_add(1, SeqCst)
```

```
{ P_x = inv_y.unwrap();
```

```
x.free(P_x);
```

```
}
```

Creusot & Verus: Verification Under Sequential Consistency

```
let (x, P_x) = Ptr::new(0); let y = AtomicU8::new(0);  
      borrow-checked ghost permission
```

```
x.write(42, &mut P_x);
```

```
y.fetch_add(1, SeqCst);
```

```
  inv_y = Some(P_x);
```

ghost update

```
if 1 == y.fetch_add(1, SeqCst)
```

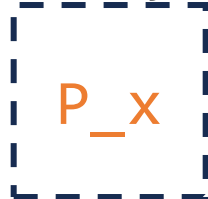
```
{  
  P_x = inv_y.unwrap();
```

```
  x.free(P_x);
```

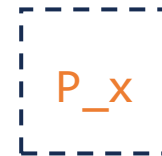
```
}
```

invariant

inv_y



inv_y: y = 1 ==>



Creusot & Verus: Verification Under Sequential Consistency

borrow-checked ghost permission
x.write(42, &mut P_x);

y.fetch_add(1, SeqCst); ★₁

inv_y = Some(P_x);

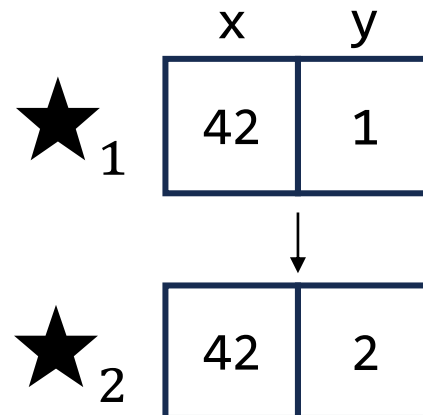
ghost update

if 1 == y.fetch_add(1, SeqCst) ★₂

{ P_x = inv_y.unwrap();

x.free(P_x);

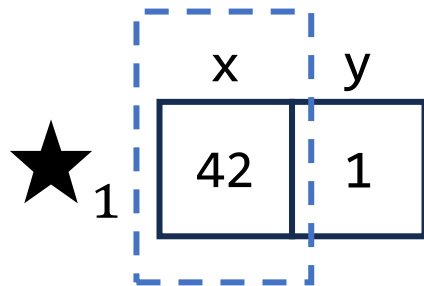
}



The Problem with Weak Memory

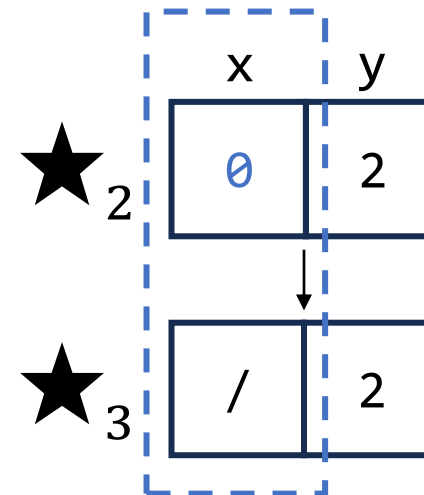
```
    borrow-checked ghost permission
x.write(42, &mut P_x);

y.fetch_add(1, Relaxed); ★1
    inv_y = Some(P_x);
ghost update
```



Problem: P_x no longer ensures memory safety if it can be used by threads with different **views** of memory

```
if 1 == y.fetch_add(1, Relaxed) ★2
{
    P_x = inv_y.unwrap();
    x.free(P_x); ★3
}
```



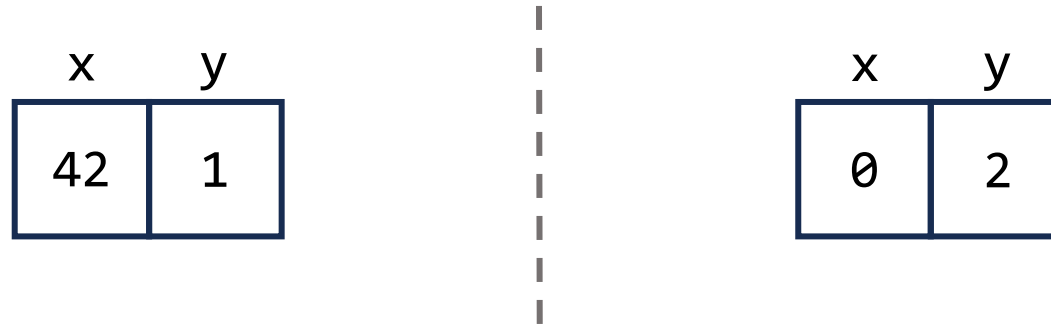
Possible use-after-free:
`ptr::write(x, 42)`

Repairing Verification Under Weak Memory

Subjectivity & Objectivity

Ghost state can be **subjective** or **objective**.

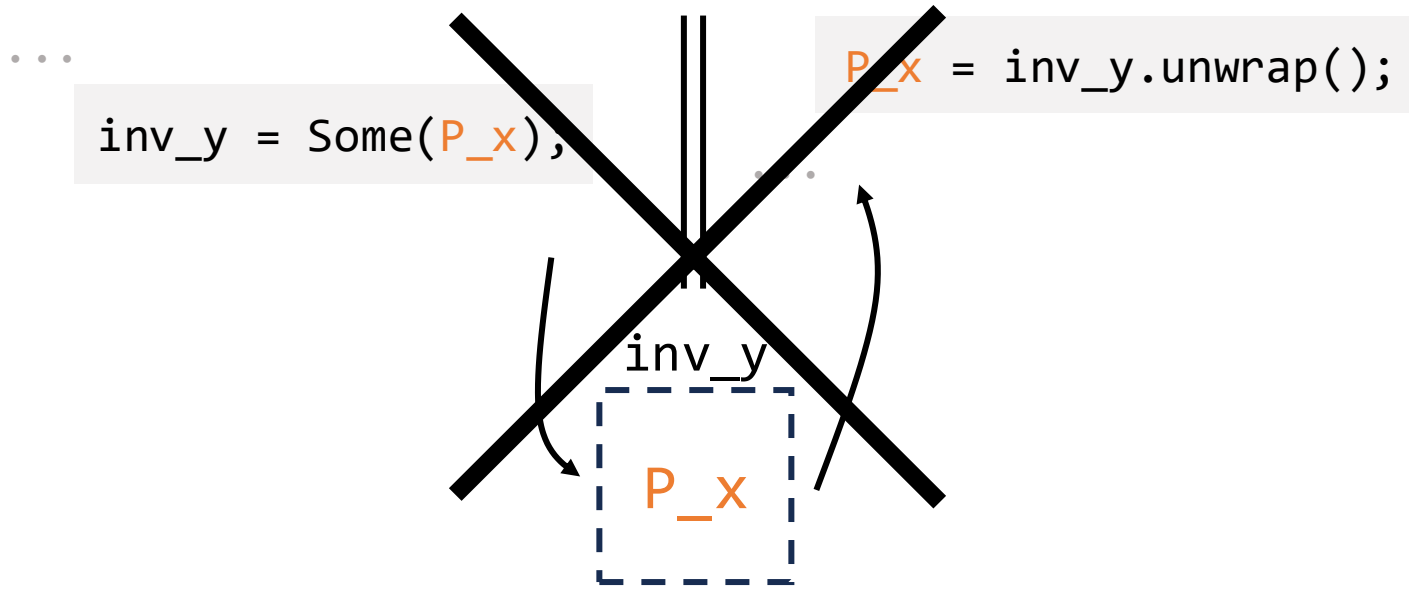
Subjective = safety depends on thread's view of memory



Objective = safety does not depend on thread's view of memory

Subjectivity & Objectivity

Pointer permissions are **subjective**,
and invariants can only hold **objective** ghost state.



Encoding in Creusot & Verus

New trait `unsafe trait Objective`

New trait bound `Invariant<P: Objective>`

New impl `impl<T> !Objective for PtrPerm<T> {}`

Transferring Permissions Between Threads Under Weak Memory

```
// thread1 safely uses subjective Perm  
// thread1's view = view1  
// objective: Perm is safe at view1
```

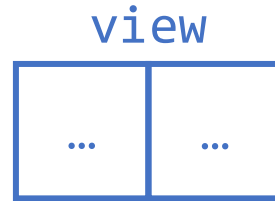
```
// objective: Perm is safe at view1  
// thread2's view = view2  $\supseteq$  view1  
// thread2 safely uses subjective Perm
```



ViewSeen: Encoding in Creusot & Verus

```
// thread has seen this view
```

```
ghost struct ViewSeen;
```



```
impl !Objective for ViewSeen {}
```

ViewSeen: Encoding in Creusot & Verus

Atomic operations advance thread's **view of memory**.

```
AtomicU8 {  
    fetch_add(&self, ..., VS: &mut ViewSeen) -> (...)  
    ensures ...  $\subseteq$  final(VS)  
}
```

ViewAt<P>: Encoding in Creusot & Verus

```
// permission is safe to use at the given view
```

```
ghost struct ViewAt<P>;
```



```
impl<P> Objective for ViewAt<P> {}
```

`ViewAt<P>` and `ViewSeen`: Encoding in Creusot & Verus

```
impl<P> ViewAt<P> {  
  
  new(Perm: P) -> ViewAt<P>  
  
  into_inner(VA: ViewAt<P>, VS_sync: ViewSeen) -> P  
    requires VA  $\subseteq$  VS_sync  
  
}
```

Sketch: Verification Under Weak Memory with **ViewAt<P>** and **ViewSeen**

```
// thread1 safely uses subjective Perm
```

```
// Perm ->
```

```
ViewAt<Perm>: Perm is safe at view1
```

```
ViewSeen: thread1's view = view1
```

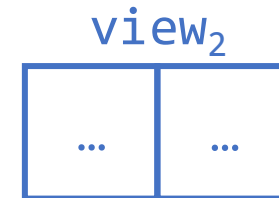
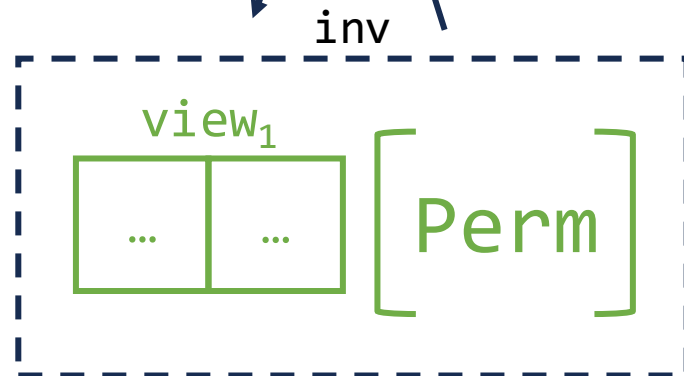
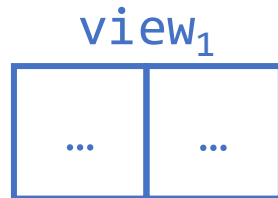
```
// ViewAt<Perm>: Perm is safe at view1
```

```
// atomic ops ->
```

```
ViewSeen: thread2's view = view2  $\supseteq$  view1
```

```
// ViewAt<Perm>, ViewSeen -> Perm
```

```
// thread2 safely uses subjective Perm
```



Leveraging the Borrow Checker

Ghost Permissions in `Atomic::load` API

```
y.load(Ordering::Acquire)
```

Our API:

```
load(..., &mut ViewSeen, &AtomicPerm)
```

Direct translation from RustBelt Relaxed:

```
load(&self, &mut ViewSeen, &mut HistorySeen, &mut ViewAt<AtomicPerm>)
```

Ghost Permissions in `Atomic::load` API

```
y.load(Ordering::Acquire)
```

Our API:

```
load(..., &mut ViewSeen, &AtomicPerm)
```

Direct translation from RustBelt Relaxed:

```
load(&self, &mut ViewSeen, &mut HistorySeen, &mut ViewAt<AtomicPerm>)
```

RustBelt Relaxed used these additional permissions.

We can replace them with the borrow checker!

Ghost Permissions in `Atomic::load` API

```
x.load(Ordering::Acquire)
```

Our API:

```
load(..., &mut View, &AtomicPerm)
```

Direct translation from RustBelt Relaxed:

```
load(&self, &mut View, &mut History, &mut ViewAt<AtomicPerm>)
```



Why does the borrow checker
let us make `AtomicPerm`
objective?

AtomicPerm is Objective

Atomic permits racy writes, but:

- Uses of **&Atomic** must not race with destruction of the **Atomic**

`new() -> (Self, AtomicPerm)`

borrow &self

`store(&self, ...)`

borrow ends ★

...

RustBelt Rlx: no data races across end of borrow

`into_inner(self, AtomicPerm)`

Ownership of **self**
guarantees absence of races

Ownership of **AtomicPerm** does not
need to guarantee absence of races!

AtomicPerm is Objective

```
y.load(Ordering::Acquire)
```

Our API:

```
load(..., &mut View, &AtomicPerm)
```



AtomicPerm can safely
be objective!

Direct translation from RustBelt Relaxed:

```
load(&self, &mut View, &mut History, &mut ViewAt<AtomicPerm>)
```

Current Progress & Case Study

Currently Supported Operations

fence: { Release, Acquire }

load, store, fetch_add:
 { Release, Acquire, Relaxed }

We foresee supporting all remaining atomic operations for:

 { RelAcq, Release, Acquire, Relaxed }

Verified Weak Memory Case Study: `fetch_add`

```
x.write(42);
```

```
fence(Release);
```

```
y.fetch_add(1, Relaxed);
```

```
if 1 == y.fetch_add(1, Relaxed) {
```

```
    fence(Acquire);
```

```
    x.free();
```

```
}
```

Verified Weak Memory Case Study: `fetch_add`

Tool	Proof / Code Ratio (% Δ SeqCst)	Verification Time (% Δ SeqCst)
Verus	6.0 (+40%)	0.36s (+16%)
Creusot	2.8 (+27%)	0.87s (+40%)

Weak memory adds **up to 40% overhead** compared to the same program under sequential consistency.

Verified Weak Memory Case Study: `fetch_add`

- Views are opaque
 - Only need to reason about partial order: $\text{view}_1 \subseteq \text{view}_2 \subseteq \dots$
- One additional quantifier is introduced for weak memory
 - AtomicPerm contains entire history of updates:
`forall |t: timestamp| perm.history.contains(t) ==> ...`

Additional Ongoing Work

Additional Ongoing Work

- Abstracting view-based reasoning with higher-level principles
 - Verus: Supporting storage protocols under weak memory
 - Verus: Proving soundness in VerusBelt under weak memory
 - Mixing SeqCst and { Relaxed, Release, Acquire, ... } accesses
- 

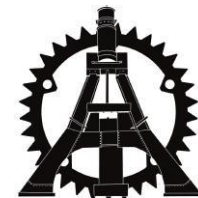
Summary

- **Weak memory** is used for practical, efficient concurrency in Rust
- Support weak memory in Creusot and Verus by bringing **reasoning principles** into **Rust type system**



Thank you!

Natalie Neamtu
nneamtu@andrew.cmu.edu



This material is based on work supported by the National Science Foundation CISE Graduate Fellowships under Grant No. 2313998, the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE2140739, National Science Foundation Grant No. CCF-2318953, the CyLab Future Enterprise initiative, and a gift from Amazon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.