# Extending Verus with Logical Atomicity

Iris Workshop 2026 @ ISTA, Vienna/Austria

Aaron Bies, Travis Hance, Derek Dreyer

MPI-SWS & Saarland University

June 8, 2026

# Introduction to Verus

An SMT-based verification framework for Rust
(Lattuada et al. OOPSLA 2023)

Allows modular verification using specifications
e.g. function pre- and postconditions

```
fn min(x: i32, y: i32) -> (m: i32)
    ensures
        m == x || m == y,
        m <= x,
        m <= y, // compile-time error here
{
    if x <= y { x } else { x }
}                    //   ^^^ because of this
```

# Introduction to Verus

Verus has three modes:

| `exec` | ⊑ | `proof/tracked` | ⊑ | `spec/ghost` |
|---|---|---|---|---|
| Regular Rust code to be verified | | Erased at compile time but borrow checked | | Pure and total mathematical exprs. |
| **Executed at runtime** | | **Affine ghost state** | | **Checked by SMT solver** |

⇒ Verus is not SL, **but** supports Iris-style reasoning using affine ghost state

# Atomic Variables in Verus

**Iris proof rule:** $\ell \mapsto v \ * \ \triangleright (\ell \mapsto w \mathrel{-\!\!*} Q()) \ \vdash \mathsf{wp}\, \ell \leftarrow w \, \{w.\, Q(w)\}$

**Fwd. reason.:** $\{\ell \mapsto v\}\, \ell \leftarrow w \, \{\ell \mapsto w\}$

```
fn store(&self, perm: Tracked<&mut PermissionU64>, w: u64)
    requires
        self.id() == old(perm)@.patomic,
    ensures
        self.id() == final(perm)@.patomic,
        final(perm)@.value == w,
```

# Atomic Variables in Verus

```
1 let (var, Tracked(mut perm)) = PAtomicU64::new(3);
2 assert(perm@.patomic == var.id());
3 assert(perm@.value == 3);
```

$\{\top\}$
$\texttt{let}\ \ell = \texttt{ref}\ 3\ \texttt{in}$
$\{\ell \mapsto 3\}$

```
4 let x = var.load(Tracked(&perm));
5 assert(x == 3);
```

$\texttt{let}\ x = !\,\ell\ \texttt{in}$
$\{\ell \mapsto 3 \ * \ x = 3\}$

```
6 var.store(5, Tracked(&mut perm));
7 assert(perm@.value == 5);
```

$\ell \leftarrow 5$
$\{\ell \mapsto 5 \ * \ x = 3\}$

# Atomic Invariants

```
let tracked inv: AtomicInvariant<int, PermissionU64, IsEven> = ...;
```

Equivalent to $\boxed{\exists v.\, \ell \mapsto v \;*\; v \; mod \; 2 = 0}$

```
let Tracked(credit) = vstd::invariant::create_open_invariant_credit();
```

Verus does not have ▷ modality, but we have *later credits*

```
open_atomic_invariant!(credit => &inv => perm => {
    // assume perm@.value % 2 == 0
    var.store(4, Tracked(&mut perm));
    // assert perm@.value % 2 == 0
});
```

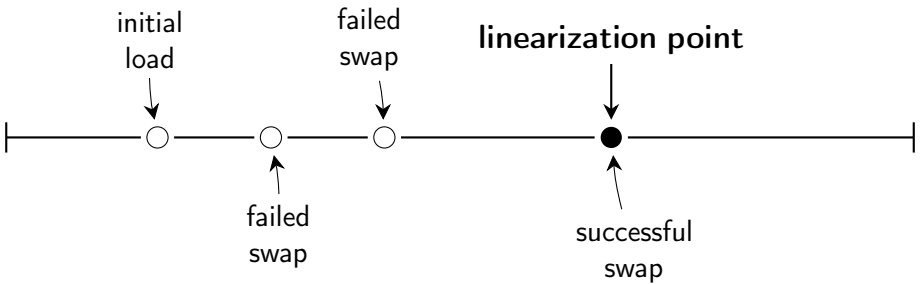
**Masks & namespaces**

No fancy updates, functions are annotated with masks instead

At most one *physically atomic* operation!

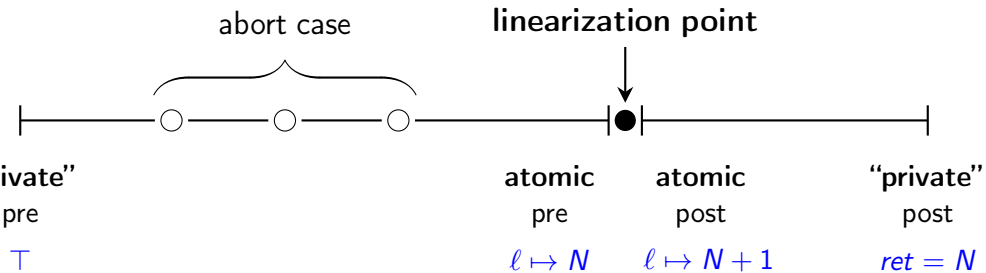
# Logical Atomicity

```
6  pub fn increment(var: &AtomicU64) -> u64 {
7      let mut curr = var.load(SeqCst);
8      loop {
9          match var.compare_exchange_weak(curr, curr + 1, SeqCst, SeqCst) {
10             Ok(_) => return curr,
11             Err(new) => curr = new,
12         }
13     }
14 }
```

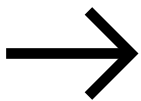


# Logical Atomicity

```
6  pub fn increment(var: &AtomicU64) -> u64 {
7      let mut curr = var.load(SeqCst);
8      loop {
9          match var.compare_exchange_weak(curr, curr + 1, SeqCst, SeqCst) {
10             Ok(_) => return curr,
11             Err(new) => curr = new,
12         }
13     }
14 }
```



Formalizations of logical atomicity come in **two flavors**:



## HOCAP-style

- Used by: VeriFast, HOCAP, Creusot
- **Current state of the art in Verus**

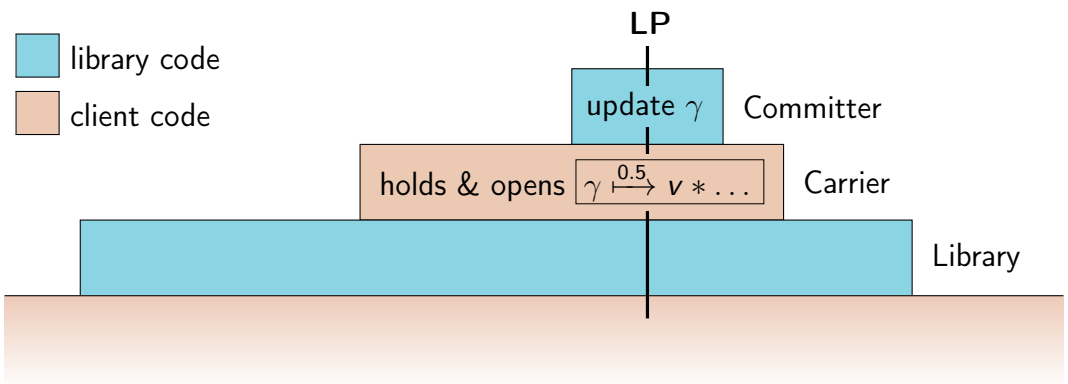
## Iris-style

- Inspired by TaDA
- **Our new extention to Verus**

# HOCAP-Style Logical Atomicity

**Idea:** "carrier" ghost callback provides permission at LP

Move ghost update at LP into "committer"



# HOCAP-Style Logical Atomicity

- VeriFast (Jacobs & Piessens POPL'11)
- HOCAP (Svendsen et al. ESOP'13)
- Creusot (Denis et al. ICFEM'22)

**In Verus:** Has been used to verify CapybaraKV (LeBlanc et al. OSDI'25)

Traits used for this construction are in `vstd::logatom` module

# Pros and Cons

✓ Fully verified within Verus

✗ Inherently higher-order & complicated encoding

✗ Requires significant boilerplate for library & client

✗ Obfuscates function specification in layers of abstraction

✗ Inflexible, only ghost update in comitter

Formalizations of logical atomicity come in **two flavors**:

## HOCAP-style

- Used by: VeriFast, HOCAP, Creusot
- **Current state of the art in Verus**

## Iris-style

- Inspired by TaDA
- **Our new extention to Verus**

# Iris-Style Logical Atomicity

Introduced in Iris (Jung et al. POPL'15)
Inspired by TaDA (da Rocha Pinto et al. ECOOP'14)

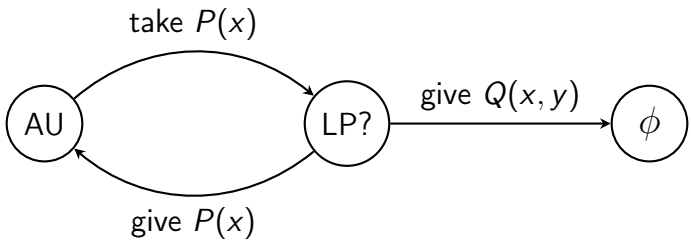
**Idea:** Atomic update (AU) object as abstraction for LP
Tightly coupled with atomic specification

- constructed by client
- destructed by library

# Atomic Specifications in Iris

$$\langle x.\, P(x)\rangle\ e\ \langle y.\, Q(x,y)\rangle^{Eo}_{Ei} \triangleq \forall \phi.\, AU \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\phi\}$$

$$AU \triangleq \nu U.\ {}_{Eo}\!\Rrightarrow_{Ei} \exists x.\, P(x) * \begin{pmatrix} \left(P(x)\ {}_{Ei}\!\Rrightarrow\!\!\!\!\!\!*\ {}_{Eo}\ U\right) \wedge \\ \left(\forall y.\, Q(x,y)\ {}_{Ei}\!\Rrightarrow\!\!\!\!\!\!*\ {}_{Eo}\ \phi(y)\right) \end{pmatrix}$$



# Atomic Specifications in Iris

$$\langle x.\, P(x)\rangle\ e\ \langle y.\, Q(x,y)\rangle^{Eo}_{Ei} \triangleq \forall \phi.\, AU \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\phi\}$$

$$AU \triangleq \nu U.\ {}_{Eo}\!\Rrightarrow_{Ei} \exists x.\, P(x) * \begin{pmatrix} \left(P(x)\ {}_{Ei}\!\Rrightarrow\!\!\!\!\!*_{Eo}\ U\right) \wedge \\ \left(\forall y.\, Q(x,y)\ {}_{Ei}\!\Rrightarrow\!\!\!\!\!*_{Eo}\ \phi(y)\right) \end{pmatrix}$$

Verus is missing the following logical constructs:

- greatest fixpoint ($\nu$) $\Rightarrow$ emulate with unbounded loop
- fancy update ($\Rrightarrow$) $\Rightarrow$ functions & built-in macros
- separating conjunction ($*$) $\Rightarrow$ tuples of tracked types
- magic wand ($-\!\!*$) $\Rightarrow$ proof-mode functions

# Atomic Specifications in Verus

```
1  fn function(px: Px) -> (py: Py) // P' Q' resources
2      atomically (atomic_update) {
3          (ax: Ax) -> (ay: Ay),   // P Q resources
4          requires atomic_pre_condition(px, ax),      // P
5          ensures atomic_post_condition(px, ax, ay),  // Q
6          outer_mask [...],  // Eo
7          inner_mask [...],  // Ei
8      },
9      requires private_pre_condition(px),              // P'
10     ensures private_post_condition(px, ax, ay, py),  // Q'
```

$$\langle P \rangle \{ P' \}\ e\ \langle Q \rangle \{ Q' \}^{Eo}_{Ei}$$

# Atomic Specifications in Verus

```
1  fn function(px: Px) -> (py: Py) // P' Q' resources
2      atomically (atomic_update) {
3          (ax: Ax) -> (ay: Ay),   // P Q resources
4              requires atomic_pre_condition(px, ax),      // P
5              ensures atomic_post_condition(px, ax, ay),  // Q
6              out
7              inn
8      },
9      require
10     ensures
```

$$\langle P \rangle \{P'\}\ e\ \langle Q \rangle \{Q'\}^{Eo}_{Ei}$$

`Ay = Result<T, E>` for AU with abort case

`Ay = Commit<T>` for "commit-only" AU

$\Rightarrow$ "abort type" can be different from `Ax`

user chooses relation between both types

```
let res: Result<(), Tracked<AtomicUpdate>>
  = try_open_atomic_update!(atomic_update, ax => {
    // assume atomic pre
    // ...
    // assert atomic post
    Tracked(ay)
});
```

**Destruction**

open AU, take `ax`, give `ay`

we get back AU when aborted

```
// assert private pre
let py = function(px) atomically loop |update| {
    // ...
    // assert atomic pre
    let ay = update(ax);
    // assume atomic post
    // ...
    break/continue
};
// assume private post
```

**Construction**

give `ax`, take `ay`, repeat on abort

`loop` to emulate greatest fixpoint

`break`/`continue` for commit/abort

# Resolves Check

**Problem:** Client assumes AU is committed, library might not

**Solution:** Prophecy variables!

- `au.resolves()` initially unknown value
- becomes `true` when `try_open_atomic_update!(au, ...)` commits
- library must prove `au.resolves() == true` at function exit

# Verified Increment Function

```
pub fn increment(var: &PAtomicU64) -> (out: u64)
    atomically (atomic_update) {
        (perm: PermissionU64)
            -> (res: Result<PermissionU64, (PermissionU64, OpenInvariantCredit)>),
        requires perm@.patomic == var.id(),
        ensures match res {
            Err((p, _)) => p@ == perm@,
            Ok(p) => p@.patomic == perm@.patomic
                  && p@.value == perm@.value.wrapping_add(1),
        },
        outer_mask any,
        inner_mask none,
    },
    ensures out == perm@.value,
{ /* next slide */ }
```

```
15 {
16     let Tracked(credit) = vstd::invariant::create_open_invariant_credit();
17     let tracked mut au = atomic_update;

18     let mut curr;
19     let wrapped_au = try_open_atomic_update!(au, perm => {
20         curr = var.load(Tracked(&perm));
21         Tracked(Err((perm, credit)))
22     });

23     proof { au = wrapped_au.get().tracked_unwrap_err() };

24     loop invariant au == atomic_update {
25         /* next slide */
42     }
43 }
```

```
24  loop invariant au == atomic_update {
25      let Tracked(credit) = vstd::invariant::create_open_invariant_credit();
26      let next = curr.wrapping_add(1);
27      let res;
28      let maybe_au = try_open_atomic_update!(au, mut perm => {
29          res = var.compare_exchange_weak(Tracked(&mut perm), curr, next);
30          Tracked(match res {
31              Ok(_) => Ok(perm),
32              Err(_) => Err((perm, credit)),
33          })
34      });
35      match res {
36          Ok(_) => return curr,
37          Err(new) => {
38              proof { au = maybe_au.get().tracked_unwrap_err() };
39              curr = new;
40          }
41      }
42  }
```

# Example Client

```
let Tracked(mut credit) = vstd::invariant::create_open_invariant_credit();

let out = increment(&var) atomically loop |update| {
    let tracked mut spare = None;
    open_atomic_invariant!(credit => &inv => perm => {
        match update(perm) {
            Ok(p) => perm = p,
            Err((p, c)) => { perm = p; spare = Some(c) }
        }
    });

    match spare {
        None => break,
        Some(c) => credit = c,
    }
};
```

# Closing Remarks

- Implementation: in review (PR #2326)
- Two case studies (so far):
  - Simple flag example with "helping"
  - Concurrent hash table with fine-grain locking
- Future & ongoing work:
  - Atomic specifications in traits
  - Integrate into `vstd`

Thank you!