Using Ghost Ownership to Verify Union-Find and Persistent Arrays in Rust

Arnaud Golfouse, Jacques-Henri Jourdan, Armaël Guéneau

Inria

LMF Laboratoire Méthodes Formelles

université PARIS-SACLAY

Shared XOR mutable

Aliasing restrictions in Rust:

```
fn f(x: &mut i32, y: &mut i32) {
    *x = 1; *y = 2;
    assert!(*x == 1);
}
```

Shared XOR mutable

Aliasing restrictions in Rust:

```
fn f(x: &mut i32, y: &mut i32) {
    *x = 1; *y = 2;
    assert!(*x == 1);
}

let mut x: &mut i32 = &mut 1;
f(x, x); // ✗ compile error
```

Shared XOR mutable

Aliasing restrictions in Rust:

```
fn f(x: &mut i32, y: &mut i32) {
    *x = 1; *y = 2;
    assert!(*x == 1);
}

let mut x: &mut i32 = &mut 1;
f(x, x); // ✗ compile error
```

But not for unsafe code/raw pointers!

```
unsafe fn f(x: *mut i32, y: *mut i32) {
    *x = 1; *y = 2;
    assert!(*x == 1);
}
let mut x: *mut i32 = &mut 1;
unsafe { f(x, x); } // ✓ ok
```

Shared XOR mutable

Aliasing restrictions in Rust:

```
fn f(x: &mut i32, y: &mut i32) {
    *x = 1; *y = 2;
    assert!(*x == 1);
}

let mut x: &mut i32 = &mut 1;
f(x, x); // ✗ compile error
```

But not for unsafe code/raw pointers!

```
unsafe fn f(x: *mut i32, y: *mut i32) {
    *x = 1; *y = 2;
    assert!(*x == 1);
}
let mut x: *mut i32 = &mut 1;
unsafe { f(x, x); } // ✓ ok
```

These makes verification easier

Shared XOR mutable

Creusot  leverages this:

```
#[requires(true)]
#[ensures(^x == 1 && ^y == 2)]
fn f(x: &mut i32, y: &mut i32) {
    *x = 1; *y = 2;
    assert!(*x == 1);
}
```

Shared XOR mutable

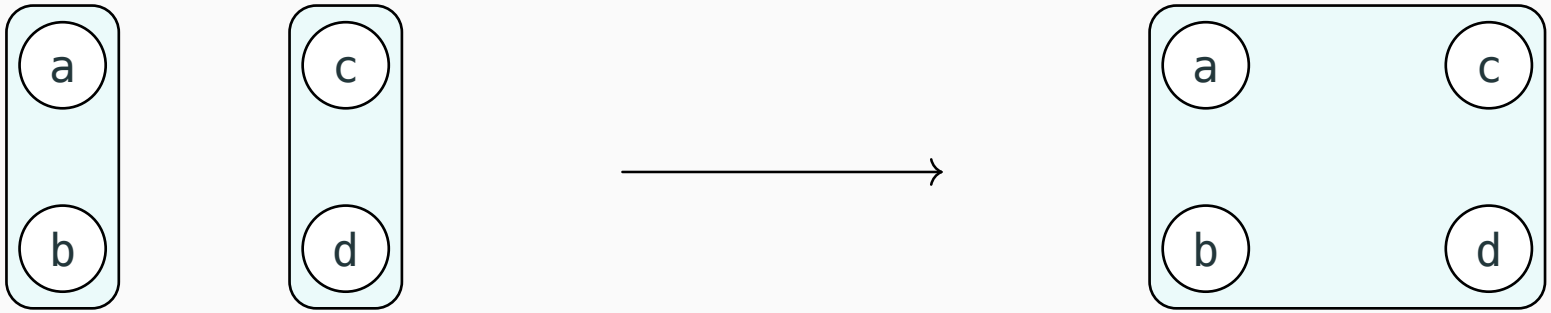
Creusot leverages this:

```
#[requires(true)]
#[ensures(^x == 1 && ^y == 2)]
fn f(x: &mut i32, y: &mut i32) {
    *x = 1; *y = 2;
    assert!(*x == 1);
}
```

We still want to implement and verify pointer-based data structures.

Union-find

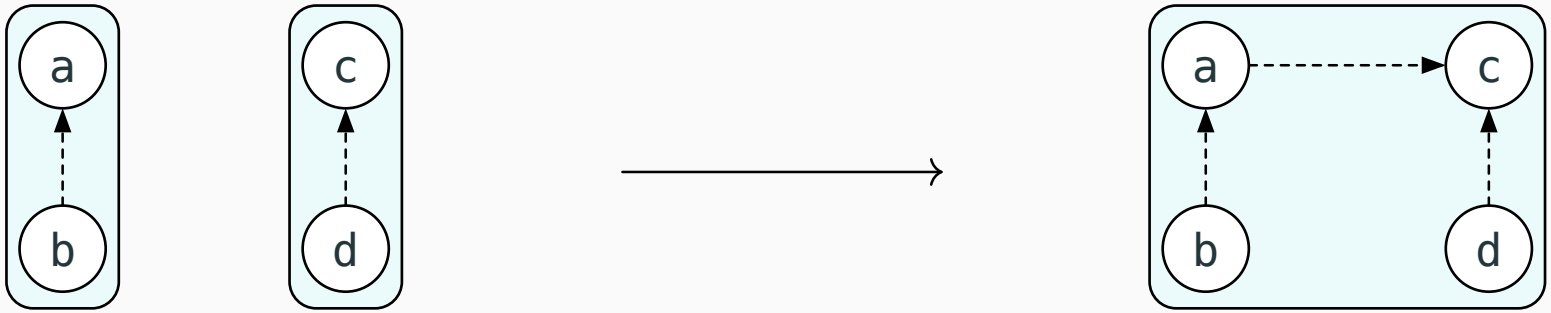
Categorize elements into sets




```
union(b, d);
```

Union-find

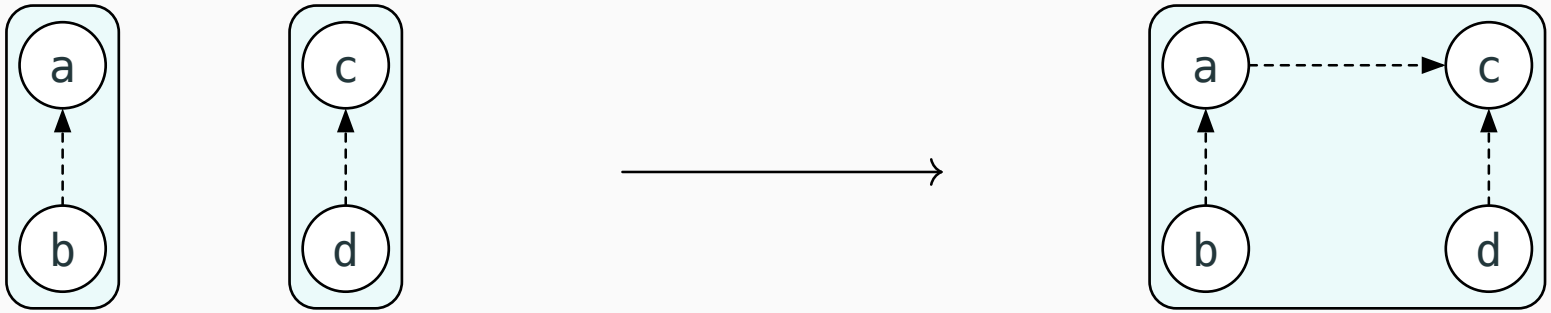
Categorize elements into sets




```
union(b, d);
```

Union-find

Categorize elements into sets




```
find(a) != find(c);
```

```
union(b, d);
```

```
find(a) == find(c)
```

Union-find

Alternative implementation: use an array, passed as parameter.

```
fn find(a: &mut Vec<Node>, elem: usize) -> usize;
```

Union-find

Alternative implementation: use an array, passed as parameter.

Our solution:

- Pointers in the code;
- Global state in **ghost code**, passed as parameter.

→ Ownership of this state needs to be tracked in ghost code.

```
fn find(gh: Ghost<&mut ...>, elem: *mut Node) -> *mut Node;
```

In this talk

Adapted ghost code with ownership (from Verus) in Creusot.

Support for most pointer operations → verify idiomatic Rust code.

Can verify union-find's implementation.

PtrOwn

Cannot dereference raw pointers as-is.

Instead, use `Ghost<PtrOwn<T>>`:

- Ghost-only
- Has ownership

Data stored in `Ghost<T>` is erased at runtime:

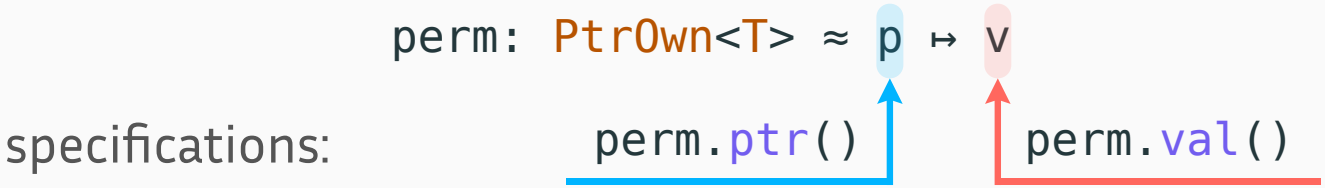
for Creusot	for rustc

`#[requires(**x == 1)]`
`fn f(x: Ghost<Box<i32>>) {`
`  let y1 = x;`
`  let y2 = x;  // ✗ compile error`
`}`
	`fn f(x: ()) {`
	`  let y1 = x;`
	`  let y2 = x;`
	`}`

· `Ghost<T>` ≈ `T` logically · borrow checked	· no memory footprint

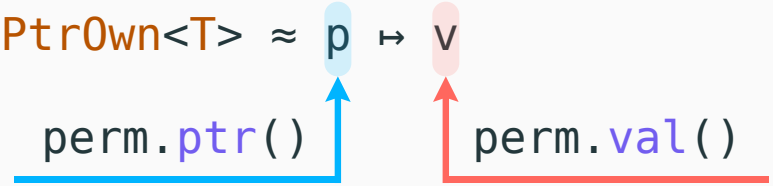
PtrOwn

```
perm: PtrOwn<T> ≈ p ↦ v
```


perm: PtrOwn<T> ≈ p ↦ v
specifications:
perm.ptr()
perm.val()

PtrOwn

specifications:



code:

Create	`let (p, perm): (*mut T, Ghost<PtrOwn<T>>) = PtrOwn::new(v);`
Read	`let x: &T = PtrOwn::as_ref(p, perm.borrow());`
Write	`let x: &mut T = PtrOwn::as_mut(p, perm.borrow_mut());`

Specification for as_mut

```
#[requires(ptr == perm.ptr())]
#[ensures((^perm).ptr() == perm.ptr())]
#[ensures(*result == perm.val())]
#[ensures((^perm).val() == ^result)]
pub unsafe fn as_mut(ptr: *mut T, perm: Ghost<&mut PtrOwn<T>>) -> &mut T;
```

```
// perm ≈ p ↦ v
let bor = PtrOwn::as_mut(ptr, perm.borrow_mut());
*bor = *bor + 1;
// perm ≈ p' ↦ v'
```

Specification for as_mut

```
#[requires(ptr == perm.ptr())]
#[ensures((^perm).ptr() == perm.ptr())]
#[ensures(*result == perm.val())]
#[ensures((^perm).val() == ^result)]
pub unsafe fn as_mut(ptr: *mut T, perm: Ghost<&mut PtrOwn<T>>) -> &mut T;
```

```
// perm ≈ p ↦ v
let bor = PtrOwn::as_mut(ptr, perm.borrow_mut());
*bor = *bor + 1;
// perm ≈ p' ↦ v'
```

Specification for as_mut

```
#[requires(ptr == perm.ptr())]
#[ensures((^perm).ptr() == perm.ptr())]
#[ensures(*result == perm.val())]
#[ensures((^perm).val() == ^result)]
pub unsafe fn as_mut(ptr: *mut T, perm: Ghost<&mut PtrOwn<T>>) -> &mut T;
```

```
// perm ≈ p ↦ v
let bor = PtrOwn::as_mut(ptr, perm.borrow_mut());
*bor = *bor + 1;
// perm ≈ p' ↦ v'
```

Specification for as_mut

```
#[requires(ptr == perm.ptr())]
#[ensures((^perm).ptr() == perm.ptr())]
#[ensures(*result == perm.val())]
#[ensures((^perm).val() == ^result)]
pub unsafe fn as_mut(ptr: *mut T, perm: Ghost<&mut PtrOwn<T>>) -> &mut T;
```

```
// perm ≈ p ↦ v
let bor = PtrOwn::as_mut(ptr, perm.borrow_mut());
*bor = *bor + 1;
// perm ≈ p' ↦ v'
```

Specification for as_mut

```
#[requires(ptr == perm.ptr())]
#[ensures((^perm).ptr() == perm.ptr())]
#[ensures(*result == perm.val())]
#[ensures((^perm).val() == ^result)]
pub unsafe fn as_mut(ptr: *mut T, perm: Ghost<&mut PtrOwn<T>>) -> &mut T;
```

```
// perm ≈ p ↦ v
let bor = PtrOwn::as_mut(ptr, perm.borrow_mut());
*bor = *bor + 1;
// perm ≈ p' ↦ v'
```

Specification for as_mut

```
#[requires(ptr == perm.ptr())]
#[ensures((^perm).ptr() == perm.ptr())]
#[ensures(*result == perm.val())]
#[ensures((^perm).val() == ^result)]
pub unsafe fn as_mut(ptr: *mut T, perm: Ghost<&mut PtrOwn<T>>) -> &mut T;
```

```
// perm ≈ p ↦ v
let bor = PtrOwn::as_mut(ptr, perm.borrow_mut());
*bor = *bor + 1;
// perm ≈ p ↦ v + 1
```

Union-find

```
enum Node { Root, Link(*mut Node) }

struct UF { // wrapped in Ghost
    perms: FMap<*mut Node, PtrOwn<Node>>,
    // ...
}

impl Invariant for UF {
    #[logic]
    fn invariant(self) -> bool {
        forall<p: *mut Node> self.perms.contains(p) ==>
          self.perms[p].ptr() == p &&
          match self.perms[p].val() {
              // ...
          }
    }
}
```

Proof of `find`

```
fn find(                          ptr: *mut Node) -> *mut Node {

    match unsafe { &*ptr } {
        // ...
    }
}
```

Proof of `find`

```
struct UF {
  perms: FMap<*mut Node, PtrOwn<Node>>
}
```

```
#[requires(uf.contains(ptr))]
#[ensures(result == uf.root_of(ptr))]
fn find(mut uf: Ghost<&mut UF>, ptr: *mut Node) -> *mut Node {
    let perm: Ghost<&PtrOwn<Node>> = ?;
    match *PtrOwn::as_ref(ptr, perm) {
        // ...
    }
}
```

Proof of `find`

```
struct UF {
  perms: FMap<*mut Node, PtrOwn<Node>>
}
```

```
#[requires(uf.contains(ptr))]
#[ensures(result == uf.root_of(ptr))]
fn find(mut uf: Ghost<&mut UF>, ptr: *mut Node) -> *mut Node {
    let perm: Ghost<&PtrOwn<Node>> = ghost!(&uf.perms[ptr]);
    match *PtrOwn::as_ref(ptr, perm) {
        // ...
    }
}
```

Ghost code

```
let x:  Box<i32>         = Box::new(1);
let gh: Ghost<Box<i32>> = ghost!(Box::new(1));
let inner = *gh;                  ✗
ghost! {
    let x_ref = &x;               ✓
    let x_mutref = &mut x;        ✗
    let x_owned = x;              ✗

    let inner = *gh;              ✓
    let gh_owned = gh;            ✓
    let gh_ref = &gh;             ✓
    let gh_mutref = &mut gh;      ✓

    return; break; continue;      ✗
};
```

Conclusion

- In this talk:
  - Ghost code with ownership with `Ghost` and `ghost!`.
  - Support for pointers via `PtrOwn`.
  - Proof of union-find's implementation.
- In the paper:
  - Resource algebras and local invariants.
  - Proof of persistent arrays' implementation.

Proofs are in Creusot's repository: https://github.com/creusot-rs/creusot