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

Arnaud Golfouse, Armaël Guéneau, Jacques-Henri Jourdan

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

Arnaud Golfouse
Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles
91190 Gif-sur-Yvette, France
arnaud.golfouse@inria.fr

Armaël Guéneau
Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles
91190 Gif-sur-Yvette, France
armael.gueneau@inria.fr

Jacques-Henri Jourdan
Université Paris-Saclay, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles
91190 Gif-sur-Yvette, France
jacques-henri.jourdan@cnrs.fr

Abstract

The type system of Rust enforces the "shared xor mutable" principle, which forbids mutation of shared memory. This principle eases verification in Rust, but certain programs require circumventing it with the mechanism of *interior mutability*. Thus, supporting interior mutability in a deductive verification tool is difficult. The Verus [22] tool demonstrated the use of *ghost resources* to that end. So far, this mechanism has only been applied to Verus in order to verify primarily system code.

We extend the deductive verification tool Creusot with support for linear ghost resources. We show how Creusot's full support for mutable borrows enables better specifications for primitives of linear ghost code. We apply this methodology to the verification of two data structures using sharing and mutation: union-find and persistent arrays.

CCS Concepts: • **Theory of computation** → **Program verification**; *Program specifications*; *Pre- and post-conditions*; • **Software and its engineering** → Imperative languages.

Keywords: Deductive verification, Rust, ghost code

ACM Reference Format:
Arnaud Golfouse, Armaël Guéneau, and Jacques-Henri Jourdan. 2026. Using Ghost Ownership to Verify Union-Find and Persistent Arrays in Rust. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '26), January 12–13, 2026, Rennes, France.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3779031.3779086

1 Introduction

When verifying programs, we strive to make our proofs as modular as possible. In particular, it is easier to reason about a function if the mutation is local: that is, variables handled by the function may only be modified by it. More generally, tracking which aliases currently exist for those variables is essential. There are two main solutions to this aliasing problem.

The first solution is to use separation logic: this allows precise knowledge of which variables are aliased or not, and flexible proofs can be written. However, proofs are typically hard to automate in separation logic, restricting its usage to more manual proof tools.

Another solution is to restrict the set of accepted programs to ones where aliasing is absent, or at least statically known. This is the solution used by Maalej et al. [25] in Ada SPARK, and in Linear Dafny [24], which use a rudimentary borrowing mechanism inspired by Rust; or Why3 [12], which uses a region-based type system to track aliasing. This is also a property of the type system of Rust. In Rust, the type system is substructural: values may not be freely duplicated, and a *borrow checker* ensures that mutating a memory location is only possible through non-aliased references: this is the "shared xor mutable" principle. For a verification tool, this principle makes it significantly easier to reason on Rust programs. For example, we can statically deduce that the assertion below always succeeds in a type-safe context:

```
1 fn writes_3(x: &mut i32) {
2     *x = 3;
3     f();
4     assert!(x == 3);
5 }
```

Here, the type modifier `&mut` indicates that `x` is a *mutable borrow*, *i.e.*, a non-aliased reference. Thus, no other code (including `f`) can modify the value of `x` between the write `*x = 3` and the assertion `x == 3`.

Deductive verification tools exploit this to ease verification [2, 3, 11, 15, 22, 23, 28]: because of the "shared xor mutable" principle, these tools can see Rust as a pure functional language, which allows easier proofs using *e.g.*, SMT solvers.

1.1 Interior Mutability

However, restricting programs to those where aliasing can be statically typechecked makes some data structures impossible to write. For instance, we may want to implement a

*This research was supported by the Décysif project funded by the Île-de-France region and by the French government in the context of "Plan France 2030".

CPP '26, Rennes, France

ACM ISBN 979-8-4007-2341-4/2026/01
https://doi.org/10.1145/3779031.3779086

union-find data structure using pointers, but this requires both aliasing *and* sharing. Indeed, an element of the data structure can be pointed to by several other elements, and can also be mutated during merges and path compression.

To implement union-find in Rust, we must use *interior mutability*. Interior mutability is the name given to a family of libraries in Rust that allow mutation using shared references. They are implemented using "unsafe" features of Rust, allowing unrestricted access to memory through, *e.g.*, raw pointers, but ultimately present a safe interface to the client.

The type `Cell` is a simple example of interior mutability. It allows read and write accesses using shared references:

```
let x: Cell<i32> = Cell::new(1);
let (r1, r2) = (&x, &x);   // Two aliased references
r1.set(2);
assert!(r2.get() == 2);
```

Here the value of the cell can be set through `r1`, even though `r2` is an alias to the cell. It is, however, impossible to get a reference to the *contents* of the cell, which is enough to keep the whole type system safe [16].

The types `RefCell` and `Mutex` are other examples of interior mutability, using other mechanisms to ensure safety.

In the case of union-find, using a `RefCell` together with some managed shared pointer like `Rc` is enough to implement the data structure. But verifying such a structure is now made harder, as the aliasing discipline enforced by the type system is compromised.

1.2 Verification and Interior Mutability

Interior mutability compromises the "shared xor mutable" promise made by the Rust type system, re-introducing the combination of aliasing and mutation. This poses a major challenge for the aforementioned verification tools that view Rust as a functional language and avoid leveraging the full power of separation logic.

A significant contribution towards that end comes from Verus [21] and its introduction of *linear ghost code* [22]. This technique proposes to use "ghost code" (proof-relevant code that is erased at compilation time) that is typechecked according to Rust ownership-based typing rules. *Linear ghost values* carry the ownership and logical model of a data structure during the proof, while the program shares and mutate the actual data. Essentially, this encodes separation logic-style reasoning principles in the Rust type system. Additionally, Verus provides *specification ghost code* that does not follow Rust's ownership discipline and can refer to specification-level objects and assertions. Many case studies leveraging these techniques have been verified using Verus, originating mainly from the systems community.

This raises the question of whether the techniques pioneered by Verus can be applied to other verification tools. As of today, Verus does not have full support for some Rust features—like mutable borrows. One might thus want to combine Verus' ghost code with *e.g.*, Creusot's extensive support for mutable borrows (including functions returning borrows, nested borrows, borrows in structures, etc.) to get the "best of both worlds". Recent work [10] suggests that soundly extending Creusot with *specification ghost code* is possible although nontrivial. It remains to be seen whether *linear ghost code* can be used with mutable borrows, and whether mutable borrows can help writing code and specifications involving ghost code. Finally, we believe that this powerful technique could be better known and could benefit from more case studies related to classic algorithms and data structures.

1.3 Contributions

In this paper, we extend Creusot with linear ghost code.

Unlike Verus, which extends the surface syntax with ghost keywords, we show that this feature can be implemented (§3) in a lightweight fashion, without extending Rust's syntax. We show that mutable borrows have no problematic interaction with this ghost code, as it does not refer to specification expressions. Our implementation leverages Creusot's support for mutable borrows for a more idiomatic API for updating interior mutable memory.

We show how to use this mechanism to prove two classic data structures: union-find and persistent arrays. The problems solved by these case studies and their formal specifications are presented in §2.

The proof of union-find (§4) is inspired by a proof in Why3 [32]. Importantly, the general mechanism of linear ghost code allowed us to replace the ad-hoc axiomatic memory model used in the Why3 proof.

The proof of persistent arrays (§6) is original, and shows that these techniques can be used to prove correct data structures that are observationally purely functional, but which internally use mutable memory. It uses the concepts of resource algebras and local invariants (§5), inspired by the Iris separation logic [18]. These concepts are available in Verus, but we provide a new API which we believe makes them easier to use directly.

Our extension has been integrated in the main Creusot repository [31], and our case studies are part of its official test suite.

2 Description of the Case Studies

In this section, we give an overview of our two case studies: we explain the problem solved by these two data structures, and we show the formal specifications we gave to them. As we shall see, ghost variables are needed *in these specifications*: union-find features hidden global state, and persistent arrays require careful tracking of overlapping accesses.

Our results are summarized in Table 1. "Total" is the line total, without comments or empty lines. "Spec" and "Ghost" are the number of lines dedicated to the specification and

Table 1. Proof statistics of the case studies.

Case study	Total LOC	Spec LOC	Ghost LOC	Time (replay)	Time (full)
Union-find	246	131	48	7.507s	27.999s
Persistent arrays	273	77	92	2.872s	5.500s

ghost code respectively. Timings were measured on a Core i7-13800H with 14 cores and Hyper-Threading disabled, on an average of 10 runs. The first timing column is the time to replay the proof, aided with a file describing the proof tree. It matches typical usage of Creusot, as this file is reused when possible. The second column is the time to rebuild this proof tree as well as the proof.

2.1 Union-Find

```
#[ensures(result.domain().is_empty())]
fn new() -> Ghost<UF<T>>;

#[ensures(!uf.in_domain(result))]
#[ensures((^uf).domain() == uf.domain().insert(result))]
#[ensures((^uf).roots_map() ==
          uf.roots_map().set(result, result))]
#[ensures((^uf).payloads_map() ==
          uf.payloads_map().set(result, payload))]
pub fn elem<T>(uf: Ghost<&mut UF<T>>, payload: T)
    -> Elem<T>;

#[requires(uf.in_domain(elem))]
#[ensures(uf.unchanged() && result == uf.root(elem))]
pub fn find<T>(uf: Ghost<&mut UF<T>>, elem: Elem<T>)
    -> Elem<T>;

#[requires(uf.in_domain(elem) && uf.root(elem) == elem)]
#[ensures(*result == uf.payload(elem))]
pub fn get<T>(uf: Ghost<&UF<T>>, elem: Elem<T>) -> &T;

#[requires(uf.in_domain(x) && uf.in_domain(y))]
#[ensures(uf.domain_unchanged())]
#[ensures(uf.payloads_unchanged())]
#[ensures(result == uf.root(x) || result == uf.root(y))]
#[ensures(forall<z> uf.in_domain(z) ==> (^uf).root(z) ==
      if uf.root(z) == uf.root(x) ||
         uf.root(z) == uf.root(y) {
        result
      } else { uf.root(z) })]
pub fn union<T>(mut uf: Ghost<&mut UF<T>>,
                x: Elem<T>, y: Elem<T>) -> Elem<T>;
```

Figure 1. Executable functions of the union-find data structure and their formal specifications.

Union-Find is a data structure which maintains a partition of a collection of *elements*. It allows the user to *create* new elements, in a new singleton class; to *merge* the classes of two given elements (*via* the *union* operation); and to *find* the canonical representative of a class (called its *root*) from any of its elements. In addition, in our implementation, each class is associated with a user-defined payload of type `T` useful, for example, to identify classes; the *get* operation returns the payload associated with the class of a given element.

We implement union-find with a pointer graph in memory; in that case, operations only take as arguments the elements they operate on, but they have an impact on a global mutable state: the in-memory graph, which we need to model in the interface. To see why, assume that we have given correct specifications to `union` and `find`, without talking about a global mutable state. In this case, consider elements `x1`, `x2`, `y1`, `y2`, initially in disjoint classes, and the following sequence of operations:

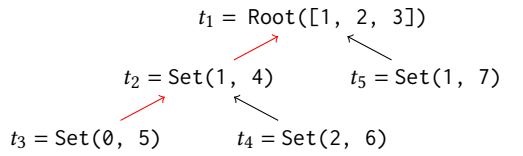
```
1 union(x1, x2); union(y1, y2);
2 let (rx, ry) = (find(x1), find(y1));
3 union(x2, y2);
4 assert!((rx, ry) != (find(x1), find(y1)));
```

At line 2, the two classes of elements are `{x1, x2}` and `{y1, y2}`, meaning that `rx` and `ry` are different roots. However, after the call `union(x2, y2)`, these classes are merged, so that the value returned by `find(x1)` and `find(y1)` are now equal, even though their parameters `x1` and `y1` have not been directly involved in the merge operation. This means that the value returned by `find` not only depends on its parameters, but also depends on a global state which is mutated by `union`. However, in Creusot, the specification of a function can only depend on the values of its parameters, so it will incorrectly conclude that the assertion fails.

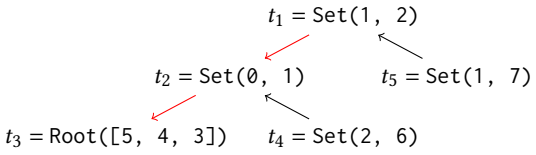
So our previous assumption was wrong: these functions cannot be specified as-is by Creusot. In order to solve this problem, we add a ghost parameter to each of the functions accessing the union-find data structure. In our extension of Creusot, such a parameter has type `Ghost<X>` for some Rust type `X`. At runtime, thanks to the `Ghost` modality, these parameters carry no data and have zero size: manipulating a value of such a type incurs no runtime overhead and cannot influence runtime behavior. However, during verification, a variable of type `Ghost<X>` is modeled by the same logical values as the underlying type `X`, and owns the same resources.

In our specifications, these additional ghost parameters are used to track the mutable state of the union-find data structure, of type `UF<T>`. Because it represents the ownership of mutable state, it is not duplicable (this would be a form of aliased mutable state).

The specifications of the executable functions of our union-find data structure are shown in Fig. 1. They make use of `#[requires(...)]` and `#[ensures(...)]` annotations to specify preconditions and postconditions of functions. These annotations can refer to the function parameters to denote their value at function entry, and postconditions can use `result` to refer to the returned value. Specifications make



(a) Before rerooting



(b) After rerooting

Figure 2. Rerooting of node t_3 in a tree of persistent arrays.

use of specification-level methods such as `root` or `roots_map`; we detail these further in §4.

The function `new` initializes a ghost state `UF<T>`. Operationally, this function does nothing, and the compiler optimizes it away. Its specification uses the `domain()` method that returns the set of elements of the data structure. It states that the new ghost state does not manage any elements.

Functions `elem`, `union` and `find` all take a parameter of type `Ghost<&mut UF<T>>`, a ghost mutable borrow of the state. This indicates that these functions may mutate the ghost state. Following the prophetic encoding used by Creusot [11], we can specify the changes made to the ghost state `uf` by referring to the final value of the ghost borrow using `^uf`. On the other hand, `*uf` refers to the value at function entry, even if used in a postcondition. This notation is often omitted thanks to Rust's *autoderef* mechanism: for example, `uf.domain()` refers to the domain at function entry.

The function `elem` can be used to create a new element. Its specification states that the returned element is fresh, and describes the new global state after addition.

The function `find` returns the root of the class of an element, and the function `get` returns the payload of the root of a class. Interestingly, `find` takes a mutable ghost borrow, because it may perform hidden mutations (the path compression optimization), which modify the internal structure of the data structure. On the other hand, `get` only needs a shared ghost borrow. Their specifications are straightforward.

Finally, `union` merges the classes of elements `x` and `y`.

2.2 Persistent Arrays

A persistent array is a data structure providing array operations (`make`, `get` and `set`), where the `set` operation returns a new persistent array and leaves the previous version unchanged. They are cheaply duplicable.

Baker [5] describes how to implement this structure efficiently. Similar to union-find, persistent arrays are inverted rooted trees. The root of the tree stores a *buffer*: a contiguous sequence of values in memory (in other words, a regular array). It corresponds to the `Root` constructor in Fig. 2. Calling `get` on the root results in direct access to the buffer.

A non-root persistent array is a pointer to its parent and a diff: a pair of index and value describing the difference with the parent. It corresponds to the `Set` constructor in Fig. 2.

At first glance, this structure can be implemented without mutability: `get` only follows pointers, and `set` creates a new tree node without changing others.

However, to optimize subsequent accesses, Baker introduces a *rerooting* operation that selects another node to become the root of the tree when calling `get`. This requires using mutability to rearrange pointers, and to modify the buffer. An example of rerooting is shown in Fig. 2b.

To do so, we first travel from the node that we want to be the new root to the current root node. Then, when going back, we invert the pointers. This means that we always invert a `Root` with a `Set`: we swap the two and invert the value in the `Set` with the value in the array at the given index.

Because of rerooting, the values stored in the persistent array may move from a `Root` node to a `Set` node, or conversely. Hence, we have to be careful when returning references to values in the array: after a rerooting, the reference may be invalidated. In particular, the following function prototype for accessing an element of a persistent array is unsound:

```
fn get<T>(a: &PArray<T>, i: usize) -> &T;
```

Indeed, while holding the returned reference, one may call `get` again on a different persistent array with the same root, which may reroot it and invalidate the returned reference. Techniques using solely the type system of Rust (such as using a mutable reference to the persistent array or carefully choosing lifetimes) cannot solve this problem.

2.2.1 Tokens.

In order to solve this problem in our specifications for persistent arrays, we use *ghost tokens*. The idea is to associate a non-fungible, unique token to the library of persistent arrays, and to require the user to leave it in deposit when calling functions accessing persistent arrays, up to the point where the user no longer holds any reference in a persistent array. Since the token is unique, this prevents overlapping accesses that would invalidate references.

We provide a notion of tokens that is independent of their use cases. An excerpt of this API is given here:

```
impl<'a> Tokens<'a> {
  #[logic]
  pub fn namespaces(self) -> Set<Namespace>;

  #[ensures(result == *self && ^self == *self)]
  #[check(ghost)]
  pub fn reborrow<'b>(&'b mut self) -> Tokens<'b>;
}
```

Since we may want to use several kinds of tokens for different libraries, tokens are associated with a set of *namespaces*,

```
declare_namespace! { PARRAY }

#[ensures(result@ == v@)]
fn make<T>(v: Vec<T>) -> PArray<T>;

#[requires(tokens.contains(PARRAY()))]
#[requires(i@ < a@.len())]
#[ensures(*result == a@[i@])]
fn get<'a, T>(a: &'a PArray<T>, i: usize,
              tokens: Ghost<Tokens<'a>) -> &'a T;

#[requires(tokens.contains(PARRAY()))]
#[requires(i@ < a@.len())]
#[ensures(result@ == a@.set(i@, t))]
fn set<'a, T>(a: &'a PArray<T>, i: usize, t: T,
              tokens: Ghost<Tokens<'a>>) -> PArray<T>;
```

Figure 3. Specifications for persistent arrays.

which the logical method `namespaces` returns. Tokens can thus be split into several tokens with disjoint sets of namespaces, using a `split` function not shown here.

Tokens are parameterized by a lifetime `'a`, which represents the duration of their validity. That way, APIs never need to give tokens back: instead, they ask for tokens of a limited duration of validity. The `reborrow` function allows creating a new token with a shorter lifetime `'b`, while keeping the ability to use the original token once the lifetime `'b` ends. This function manipulates tokens, which are ghost resources: thus, it is meant to be called in ghost code, which the attribute `#[check(ghost)]` allows. Finally, tokens can be created once in `main`, using a dedicated function.

2.2.2 Specifications.

We can now present the specifications of the persistent array API, shown in Fig. 3. They make use of a fresh namespace for tokens `PARRAY`, declared using the `declare_namespace!` macro. Both `get` and `set` require a token containing the `PARRAY` namespace, behind a `Ghost` modality to guarantee the absence of runtime impact.

The rest of the interface corresponds to what we would expect from persistent arrays. The `make` function creates a persistent array from a regular array, and its specification asserts that their logical models (accessed *via* Creusot's view operator `@`) are the same. The function `set` returns a new persistent array changed at index `i`; and `get` returns a reference to the element at index `i`[1]. Importantly, the reference returned by `get` has the same lifetime `'a` as the token, which prevents other calls to `get` or `set` as long as the reference is still living.

[1]Note that `i@` denotes the mathematical unbounded integer corresponding to `i`, an unsigned machine integer.

3 Retrofitting Ghost Code With Ownership in Creusot

In the previous section, we saw how union-find and persistent arrays, libraries using raw pointers or interior mutability, can be specified with ghost resources. In this section, we present our extension of Creusot for ghost code: we describe the primitive mechanisms implemented in the tool, and some resource-aware ghost types that make it possible to reason about programs with mutable aliased state.

3.1 Syntax and Typing of Ghost Code

In our extension of Creusot, ghost code is realized by two primitive mechanisms: a `Ghost<T>` type, that describes a *ghost value*; and a `ghost!{expr}` macro, which delimits a block of *ghost code* `expr` within a piece of potentially executable code. Additionally, the construct `ghost_let!(x = expr)` binds a variable `x` to the return value of the ghost code `expr`, while letting the expression live *in the enclosing scope*. This is sometimes important in conjunction with Creusot's type invariants, which need to be re-established at the end of the block.

In the verification conditions generated by our extension of Creusot, the `Ghost<T>` type is equivalent to `T`; it also carries the same resources as `T`. However, during compilation, this type is replaced by a unit type, so that using variables, parameters or return values of type `Ghost<T>` incurs no runtime overhead. Similarly, ghost blocks delimited by the `ghost!` macro contain normal Rust code, and are interpreted as such by the borrow checker and our verification condition generator. They are replaced by no-ops during compilation.

For the verification tool to be correct, this *erasure* transformation should not change the observable behavior of the program: non-ghost values and executable control flow should not depend on ghost values, for example. This is guaranteed through an *erasure property*, which depends on syntactic criteria on the source code.

To begin with, executable code should not be able to read the content of a `Ghost` value. Instead, accessing such a value is limited to code in `ghost!` blocks. In these blocks, the inner value can be accessed using one of the methods `into_inner` (to get full ownership), `deref` (to get a shared reference), or `deref_mut` (to get a mutable reference), or by using the dereferencing operator `*`, which is syntactic sugar for `deref` or `deref_mut`:

```
let mut g: Ghost<T> = ghost!(value);
ghost! {
  let by_ref: &T     = g.deref();     // or &*g
  let by_mut: &mut T = g.deref_mut(); // or &mut *g
  let own   : T      = g.into_inner();
};
```

Using these functions outside a `ghost!` block results in a compilation error.

To ensure that data does not escape from a `ghost!` block into executable code, the return value of these blocks is

wrapped in the `Ghost` type. For the same reason, it is not allowed to write or create a mutable borrow to a regular variable through ghost block boundaries.

Runtime variables should not be uninitialized by ghost code: this would alter runtime behavior by preventing destructors to be run. Thus, only values that are either `Copy` or wrapped in type `Ghost` can be moved inside a ghost block. Since shared references are `Copy`, this means regular program values may be freely read in the block. Because mutable references are not `Copy`, a ghost block cannot use a mutable reference created outside it, which prevents ghost code from mutating regular variables.

Note that ghost values are first-class objects: to pass a ghost value to a function or to store it in a structure, one simply has to pass it as a value of type `Ghost<T>`. Then, one can open a `ghost!` block and retrieve a value of type `T` from the `Ghost<T>` by using `deref`, `deref_mut` or `into_inner`.

3.2 Preventing Failures and Divergence in Ghost Code

Unfortunately, not all Rust code can be run in a `ghost!` block. First, ghost code must be terminating to preserve the semantics of the program after erasure. Since Creusot does not support checking termination of loops and of recursive calls, our extension simply forbids such constructs.

Perhaps more surprisingly, not all terminating Rust functions are safe to call in ghost code. Indeed, to make verification practical, absence of out-of-memory errors is not part of the guarantees provided by Creusot. For instance, there is no precondition for adding an element to a vector when calling `Vec::push`. Now, consider the following piece of code:

```
let v = ghost!{vec![1,2]};
for i in 0..usize::MAX { ghost!{ v.push(0); }; }
ghost!{ assert!(v.len() > usize::MAX); }
```

The assertion should fail, because the return type of `v.len()` is `usize`. But the assertion should also clearly succeed, because the length of the vector is `usize::MAX + 1`.

A solution would be to add a precondition `v@.len() < usize::MAX` to `Vec::push`. Such a condition is not only annoying to verify in practice, it is also useless for executable code since a program would run out of memory long before overflowing the length. Thus, Creusot does not require it.

We instead annotate functions that are safe to call in ghost code with an attribute `#[check(ghost)]`. This excludes functions that may raise out-of-memory errors. Then, ghost code and functions marked with `#[check(ghost)]` can only call other functions marked with `#[check(ghost)]`.

3.3 Ghost Permissions for Raw Pointers

When reasoning on a program with raw pointers, program variables containing these pointers cannot carry the ownership of the corresponding memory because, contrarily to `Box<T>` or `&mut T`, raw pointers are duplicable. This is an issue for tools like Creusot that rely on Rust typing to tie the logical contents of program variables with the data they own.

```
impl<T> PtrOwn<T> {
  #[ensures(result.1.ptr() == result.0)]
  #[ensures(*result.1.val() == v)]
  pub fn new(v: T) -> (*const T, Ghost<PtrOwn<T>>);

  #[requires(ptr == own.ptr())]
  #[ensures(*result == *own.val())]
  pub unsafe fn as_ref(ptr: *const T,
                       own: Ghost<&PtrOwn<T>>) -> &T;

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

Figure 4. Axiomatized interface for raw pointers.

Verus developers show [22] that ghost code gives a solution to this problem by making *ghost data* carry the ownership (and thus logical model) of *aliased program data*. The idea is to separate the runtime pointer value (of type `*const T`[2]) from its ownership, which we store in a ghost variable of type `PtrOwn`. An excerpt of the corresponding Creusot API for `PtrOwn` appears in Fig. 4.

The type `PtrOwn<T>` can be seen as a "points-to" predicate of separation logic: it is a *permission* to access a memory location. The logical method `ptr()` can be used in specifications to retrieve the pointer value, while the logical method `val()` returns the value stored in the memory location.

Pointers can be allocated with `PtrOwn::new`, which returns both the pointer and the `PtrOwn` wrapped in `Ghost` to indicate that it can be used in ghost code.

The pointer can be accessed mutably or not, using the executable functions `as_mut` and `as_ref`. At runtime, they simply return the raw pointer passed as their first parameter, cast to a borrow type. In the proof, they use the permission passed as a ghost borrow in the second parameter to guarantee that the memory access is safe. Their precondition states that the permission indeed corresponds to the pointer, and their first postcondition states that the content of the returned borrow corresponds to the value predicted by the permission.

The two other postconditions of `as_mut` specify the state of the permission after the returned borrow ends. They use Creusot's full support for mutable borrows via the *prophetic*

[2]Raw pointers in Rust can be of type `*mut T` and `*const T`; they only differ in variance and `*const T` pointers can also be used to mutate memory.

encoding described by Denis et al. [11]. Indeed, `(^own).ptr() == own.ptr()` states that the pointer associated with the permission does not change, and `^result == *(^own).val()` states that the final value of the permission is the final value of the returned borrow: in other words, the last write done on the returned borrow gets reflected in the permission.

These functions are "unsafe" in the sense that the *type system* does not guarantee their safety, but Creusot does. Thus, following the usual Rust conventions, they are marked as **`unsafe`**, which requires the **`unsafe`** keyword at call points.

3.4 Ghost Containers

Typically, permissions are stored in a global ghost structure that manages all the raw pointers used in a program data structure. Thus, we need ghost containers to organize the permissions in a single ghost value. §3.2 demonstrated that usual data structures are not suitable for ghost code. We thus provide idealized containers for ghost code: `Seq<T>`, for finite sequences; `FSet<T>`, for finite sets; and `FMap<K, V>`, for finite mappings. Those containers have infinite capacity: calling `len` on them returns an unbounded integer `Int`. Similarly to runtime data structures like `Vec` or `HashMap`, a value of these types owns its components: the resource contained in a value of type `Seq<T>` is the separating conjunction of the resources contained by the value at each slot.

An excerpt of the API of the ghost type `FMap` of finite mappings follows:

```
impl<K, V> FMap<K, V> {
  pub fn new() -> Ghost<Self>;
  pub fn get_ghost(&self, key: &K) -> Option<&V>;
  pub fn get_mut_ghost(&mut self, key: &K)
      -> Option<&mut V>;
  pub fn split_mut_ghost(&mut self, key: &K)
      -> (&mut V, &mut Self);
}
```

For brevity, we removed their formal specifications and the `#[check(ghost)]` attributes.

The function `new` builds a new empty map, and wraps its result in `Ghost`, to make sure a value of type `FMap` is never used in executable code. This is important because we do not provide an executable implementation for ghost containers. All other functions take the map directly as parameter (via *e.g.*, `&self`), which force them to be called from ghost code.

The functions `get_ghost` and `get_mut_ghost` provide immutable and mutable access to the content of the map, similarly to other idiomatic APIs in Rust (*e.g.*, `HashMap::get` or `HashMap::get_mut`). The function `split_mut_ghost` is similar to Rust's `<&mut [T]>::split_first_mut`: it provides mutable access to the value associated with a key (provided that this key is bound by the map), and mutable access to the rest of the map. This is useful for mutably accessing several values in the same map simultaneously. This function would be impossible to write for Rust's `HashMap`, as it requires restoring the binding in the map when the returned borrows expire.

```
pub struct Elem<T>(*const Node<T>);
enum Node<T> {
    Root { rank: PeanoInt, payload: T },
    Link(Elem<T>),
}

pub struct UF<T>(UFInner<T>);
struct UFInner<T> {
    domain: Snapshot<FSet<Elem<T>>>,
    perms: FMap<Elem<T>, PtrOwn<Node<T>>>,
    payloads: Snapshot<Mapping<Elem<T>, T>>,
    roots: Snapshot<Mapping<Elem<T>, Elem<T>>>,
}
```

Figure 5. Type definitions for union-find.

3.5 Snapshots

We also make use of existing support in Creusot for *specification ghost code* [10, 11] called *snapshots*. This feature is comparable to Verus' "spec" mode, and similarly complements the use of linear ghost code (Verus' "tracked" mode). Snapshots can remember the *logical* value of an expression, and can later be used in specifications:

```
let mut x = 1;
let s: Snapshot<i32> = snapshot!(x);
x = 2;
proof_assert!(*s == 1);
```

Like `Ghost` values, `Snapshot` values are zero-sized, and `snapshot!` is fully erased. This is the reason we use the `proof_assert!` macro instead of the usual macro `assert!`: it makes Creusot check the validity of a logical assertion, without enforcing it at runtime, which would be impossible since snapshots are erased by compilation.

Snapshots can also be used as first-class values: they can be stored in structures, or passed as argument. They do not carry ownership: they are freely duplicable (`Snapshot` is `Copy`). This makes them unsuitable for tracking the ownership and value of data targeted by a raw pointer. Nevertheless, snapshots are very useful to track the logical properties of ghost values! We use them in §4 and §6 to specify the invariant of our union-find and persistent arrays implementations.

4 Case Study: Union-Find

We are now ready to use ghost code to prove the implementation of the union-find data structure described in §2. This proof is adapted from a proof in Why3 [32], but their proof handles aliasing in a less principled way; we refer to §8 for a discussion of this point.

4.1 Tracking Ownership with `Ghost`

The types used in our implementation of union-find are defined in Fig. 5. The type `Elem` of elements is a raw pointer to the `Node` type, which is mutually recursive with `Elem`.

```
1  impl<T> Invariant for UF<T> {
2    #[logic]
3    fn invariant(self) -> bool {
4      ∀<e> self.0.domain.contains(e)  ⟹
5        self.0.perms.contains(e) &&
6        self.0.perms[e].ptr() == e.0 &&
7        self.0.domain.contains(self.0.roots[e]) &&
8        self.0.roots[self.0.roots[e]]
9                          == self.0.roots[e] &&
10       match *self.0.perms[e].val() {
11         Node::Link(e2) =>
12           self.0.roots[e] != e &&
13           self.0.domain.contains(e2) &&
14           self.0.roots[e] == self.0.roots[e2],
15         Node::Root { payload, .. } =>
16           self.0.roots[e] == e &&
17           self.0.payloads[e] == payload,
18       }
19   }
20 }
```

Figure 6. Invariant for union-find.

```
1  pub fn find<T>(mut uf: Ghost<&mut UF<T>>, elem: Elem<T>)
2      -> Elem<T> {
3    ghost_let!(perm =
4      uf.0.perms.get_ghost(&elem).unwrap());
5    match unsafe { PtrOwn::as_ref(elem.0, perm) } {
6      &Node::Root { .. } => elem,
7      &Node::Link(e) => {
8        let root = find(ghost! {&mut **uf}, e);
9        ghost_let!(mut uf = &mut uf.0);
10       ghost_let!(perm =
11         uf.perms.get_mut_ghost(&elem).unwrap());
12       unsafe {
13         *PtrOwn::as_mut(elem.0, perm) = Node::Link(root)
14       };
15       root
16     }
17   }
18 }
```

Figure 7. Implementation of `find`.

`Node` has two variants: `Link` is used for non-canonical elements, and contains a pointer to an element higher in the tree. The `Root` variant is for canonical representatives. It contains the payload of the class and its rank, used for the *union by rank* optimization. Ranks use the type of so-called *Peano integers*, invented by Clochard et al. [8]: they render reasoning about overflows of ranks unnecessary.

The ghost state used to track the mutable state of the data structure is stored in the `UFInner` record, which is wrapped in `UF` for reasons that will become clear in §4.2. This record contains four fields: `domain` is the finite set of all the elements of the data structure; `perms` maps each element to the ghost permissions used to access nodes; `payloads` is a mapping from elements to the payload they have been assigned at their creation; and `roots` is a mapping from elements to their canonical representative.

4.2 Invariant

The field `perms` is the only one that owns a resource: for each element, it owns the ghost permission required to dereference the corresponding raw pointer, and carries the values stored in the memory. The other fields are snapshots: they are pure data used to track the logical state of the data structure. Snapshot fields are tied to ghost permissions through a *type invariant* on the type `UF`. This type invariant is added as pre- and postconditions to every function that uses `UF`: this ensures that the functions preserve the consistency of the data structure. We wrap `UFInner` in `UF` so that the type `UFInner` can be used to manipulate a temporarily inconsistent state: our functions can break the invariant while they are executing, as long as they restore it before returning.

The type invariant specifying the well-formedness properties for each element `e` in the domain is shown in Fig. 6. Note that **`self`** has type `UF<T>`, so we access the fields of the inner record of type `UFInner` through **`self`**`.0`.

The clauses at lines 5 and 6 ensure that we hold, at position `e` in the finite map `perms`, the permission to access the node of element `e` (in particular, line 6 ensures that this is indeed the right permission).

The clauses at lines 7 and 9 guarantee well-formedness of the `roots` mapping: roots are elements of the domain, and the `roots` mapping is idempotent. The rest of the invariant ensures that the values stored in memory are consistent with the logical state: if the value is a `Link`, then it is not a root, and it points to an element in the domain with the same root. If the value is a `Root`, then, well, it is a root, and the payload stored in memory corresponds to the `payloads` mapping.

The rank is not mentioned. It is only used in the union by rank optimization, which has no functional impact.

4.3 Proofs of the Specification

Once the invariant is stated, we can prove the correctness of the main operations of the union-find data structure, such as `find`, `union` and `elem`.

We illustrate this process by detailing the proof of the function `find` in Fig. 7. It finds the canonical representative of the class of a given element by following link nodes until the root of the class is reached. Once the root is found, the path is "compressed" by changing all the link nodes traversed so that they point directly to the root.

Line 4 retrieves a ghost shared borrow of the permission corresponding to the element passed in parameter. The fact that it is indeed in the finite mapping `uf.0.perms` is guaranteed by the type invariant. Then, line 5 uses this permission

to read the node and pattern match on its contents: if it is a `Root` node, then we can return `elem` directly (line 6).

If it is a `Link` node, we perform a recursive call at line 8. Then, we perform path compression: in order to open the type invariant, we create a ghost mutable borrow to the `UFInner` structure (line 9), then we get a ghost mutable borrow to the permission to the node stored in memory at line 11, and finally write the new link node at line 13.

For lack of space, we do not detail the proof for the other functions, but we mention an interesting step in the proof of the `elem` function: indeed, this function has a postcondition stating that the returned element is fresh. In order to prove this postcondition, we need to prove that the returned element is not already in the domain.

To that end, we make use of the following ghost function, part of the `PtrOwn` library:

```
#[check(ghost)]
#[ensures(*own1 == ^own1)]
#[ensures(own1.ptr() != own2.ptr())]
pub fn disjoint_lemma(own1: &mut PtrOwn<T>,
                      own2: &PtrOwn<T>);
```

This function does not compute anything, nor does it mutate any state. Instead, it *proves* something: its postcondition states that the pointers of the two permissions are different.

In general, the verification effort boils down to writing the ghost code which retrieves the ghost permissions needed to access the nodes, and to update the fields of `UFInner`. Once this ghost code is written, Creusot generates the required verification conditions, and uses SMT solvers to discharge them. In particular, it automatically generates the assertion that the type invariant holds for `uf` at the end of the function.

5 Resource Algebras, Local Invariants

In §2.2, we presented an interface for persistent arrays suitable for verification. But persistent arrays use mutation internally: we need to be able to track a global mutable state again. At first, it seems like we have nothing to track this state in our interface: persistent arrays are duplicable; and the `Tokens` object is intentionally stateless.

Iris [18, 19] provides a solution for this problem, with *invariants*, accompanied by *resource algebras*. Verus [14, 22] adapts these to a deductive program verification tool. Creusot, similarly to Verus, introduces a *non-atomic* version of invariants, which we call *local invariants*: they allow us to execute several operations while the invariant is open, but they do not support concurrency (*i.e.*, the `Sync` trait). We also implement resource algebras, and provide original mechanisms to manipulate them in a deductive verification setting.

5.1 Local Invariants

A local invariant is a container for ghost data and ownership, always satisfying some predicate. The formal specifications of local invariants are long and technical, so we only give an overview here and refer to Appendix A for the full details.

The type of local invariants is `LocalInv<T>`. The inner type `T` must implement a trait `Protocol` that describes the predicate that the inner data of type `T` satisfies via a predicate `fn protocol(self, public: Public) -> bool`, where the type `Public` is given in the trait instance and allows the invariant to expose unchangeable data to its users.

It is unsound to open local invariants twice simultaneously. To prevent these reentrant uses, we associate a *namespace* to each invariant, and require a `Token` containing this namespace for opening them. In fact, the tokens appearing in Fig. 3 are used internally for that purpose.

Creating a `LocalInv<T>` requires giving its namespace, the value of its public data, and a value of type `Ghost<T>` to initialize the ownership it contains. As a precondition, it is required to prove that the protocol holds initially.

To open a local invariant, one can use the `open` function:

```
pub fn open<'a, T: Protocol, A>(
  l: Ghost<&'a LocalInv<T>>,
  tk: Ghost<Tokens<'a>>,
  f: impl FnOnce(Ghost<&'a mut T>) -> A) -> A;
```

The closure `f` is called while the invariant is open. The closure gains access to the ghost data contained in the invariant, and returns data that is forwarded by `open`.

The lifetime `'a` annotates all the parameters: even though the protocol is restored when the closure returns, borrows pointing inside the invariant may escape through the return value; but these borrows cannot outlive `'a`, making it impossible to drop or open the invariant while they are active.

Being a higher-order function, the specification of `open` is particularly difficult to read (see Appendix A). Rather than reproduce it here, we describe how it translates in user code:

```
let l: Ghost<&LocalInv<T>> = ...;
let tk: Ghost<Tokens> = ...;
let res = LocalInv::open(l, tk, |inner: &mut T| {
  // precondition: (*inner).protocol(l.public())
  ...
  // postcondition: (^inner).protocol(l.public()) && P
});
proof_assert!(P);
```

The precondition of `open` requires that the namespace of `l` be contained in `tk`. As its precondition, the closure knows that `inner` satisfies the protocol. As its postcondition, we are required to prove that the last value written into `inner` also satisfies the protocol. The closure may have other postconditions (or preconditions), represented here by the assertion P: for example, we may have written into mutable variables the closure captures. Those facts are available after `open` returns. Thanks to Creusot's automatic inference of closure specifications, we do not need to annotate the closure.

```
trait RA {
  #[logic]
  fn op(self, other: Self) -> Option<Self>;

  #[logic]
  #[ensures(match result {
    Some(c) => factor.op(c) == Some(self),
    None => ∀<c: Self> factor.op(c) ≠ Some(self) }
  )]
  fn factor(self, f: Self) -> Option<Self>;

  #[logic(sealed)]
  fn incl(self, other: Self) -> bool {
    other.factor(self) ≠ None
  }

  #[logic(sealed)]
  fn update(self, x: Self) -> bool {
    ∀<y> self.op(y) ≠ None ⟹ x.op(y) ≠ None
  }
  ...
}
```

Figure 8. Excerpt of the trait for resource algebra.

5.2 Resource Algebras

Users of an invariant need a mechanism to get information about the shared state in the invariant. The public data of the invariant provides such a mechanism, but it is fixed when creating the invariant, and cannot evolve.

Resource algebras are an essential tool, which solve this problem and extend the expressiveness of invariants greatly. The idea is to allow the user to define herself the kind of resources both the invariant and its clients can own, and to let her define rules to compose and update them, provided these rules are sound.

In Creusot, the trait `RA` allows defining a resource algebra. Then, the type `Resource<A>` uses a resource algebra `A` to provide the custom resource the user wants.

5.2.1 The Trait `RA`. Fig. 8 shows an excerpt of the trait `RA` or resource algebra. It is inspired by Gratzer et al. [13], except for the fact that we do not support step indexing.

The method `op` lets us compose elements. It is partial, meaning some combinations are *invalid*: this allows to restrict how resources owned by several actors are compatible, so that, *e.g.*, a client of an invariant can deduce information about the resources owned by the invariant. Associativity and commutativity are required, though not shown here.

From the definition of `op`, we derive `factor` and `incl`. First, `factor` solves, if possible, the equation $a = x \cdot b$. From `factor`, we define `incl`, which returns `true` if `factor` is `Some`: it asserts that `self` is strictly *included* in `other` in the sense of the composition operation. Because `incl` is `sealed`, its definition cannot be overridden by an implementor.

```
struct Resource<A: RA>(...);
impl<A: RA> Resource<A> {
  #[ensures(result@ == *r)]
  #[check(ghost)]
  pub fn alloc(r: Snapshot<R>) -> Ghost<Self>;

  // specification omitted
  pub fn split_mut(&mut self,
                   a: Snapshot<R>, b: Snapshot<R>)
      -> (&mut Self, &mut Self);

  #[requires(self.id() == other.id())]
  #[ensures(result.id() == self.id())]
  #[ensures(Some(result@) == self@.op(other@))]
  #[check(ghost)]
  pub fn join(self, other: Self) -> Self;

  #[requires(upd.premise(self@))]
  #[ensures((^self).id() == self.id())]
  #[ensures((^self)@ == upd.update(self@, *result))]
  #[check(ghost)]
  pub fn update<U: Update<R>>(&mut self, upd: U)
      -> Snapshot<U::Choice>;
}
```

Figure 9. Excerpt of the interface of ghost resources.

Finally, we derive `update` from `op`. It states that `x` has at least as many valid compositions as `self`. This operation describes how a ghost resource can be mutated, without invalidating the composition with the frame.

5.2.2 The type of Custom Resources. Once a resource algebra is defined, it can be used in ghost resources. The `Resource<A>` type shown in Fig. 9, serves this purpose.

A `Resource` logically contains a value of type `A`, accessed with the view operator `@`, as well as an identifier, accessed with the logical function `id`. Functions on `Resource` require that each input has the same identifier, and return resources with that identifier. Function `alloc` creates a new ghost resource with a fresh identifier.

The `join` exploits the validity of the composition of two resources of the same `id` to combine them into a single one.

Conversely, Creusot's support for mutable borrows allows defining the `split_mut` method, which uses mutable borrows in argument and return positions. It splits a resource into two parts, described by the arguments `a` and `b`, provided their composition is included in `self`. When the lifetime of the returned borrows expires, the composition of the *last* value written in the borrows is written in `self`.

5.2.3 Updates. Because the definition quantifies over all possible frames, it is difficult for automated solvers to prove that an update is valid with the raw definition of `RA::update`.

We propose an original approach to simplify this process. Instead of using `RA::update`, the `Resource::update` method

takes a value of a type implementing the trait `Update` (see Appendix B). For example, we implement this trait to describe the addition of a binding to an `FMap`. This trait defines a logical method `update`, which returns the updated resource value. The implementor can specify a required premise for the update, and has to prove that the update is indeed valid.

6 Case Study: Persistent Arrays

In this section, we use resource algebras and local invariants to track the invariant (and its associated ownership) of the persistent arrays data structure.

In order to store persistent arrays in memory, we use Rust's type `Rc` of reference-counted pointers, which allows easy sharing by incrementing the reference count. Since `Rc` allows sharing, it forbids mutation. However, the "rerooting" operation, essential for efficient implementation of persistent arrays, requires mutating memory. To that end, we use Rust's mechanism of *interior mutability*, which allows mutating memory through shared pointers, bypassing the usual "mutation xor aliasing" rule.

6.1 Tracking State With Interior Mutability

Because interior mutability allows both sharing and mutation, tracking the value contained in an interior mutable cell causes a similar problem to tracking the value pointed to by a raw pointer. Thus, we use a similar solution: an interior mutable cell whose state is tracked by a ghost token.

Taking inspiration from Verus, we added to Creusot the type `PermCell<T>` of interior mutable cells containing a value of type `T`. They are not pointers, but contain a value of type `T` "inline", without indirection. Of course, it is only interesting when it is part of a value pointed to by aliased pointers.

Each `PermCell` is associated with a ghost permission of type `PermCellOwn` (similar to `PtrOwn`), which tracks the internal value of the cell. Similarly to `PtrOwn::as_ref` and `PtrOwn::as_mut`, the type `PermCell` has two methods `borrow` and `borrow_mut` for taking a shared or mutable borrow inside a `PermCell` from a shared reference to it.

Each permission `PermCellOwn` can be used for only one `PermCell` in memory. To guarantee permissions are not confused, both `PermCell` and `PermCellOwn` values are uniquely identified by a ghost *identifier* returned by the logical method `id()`, whose values have to match when calling `borrow` or `borrow_mut`. Identifiers play here the same role as the pointer value returned by the method `ptr` of `PtrOwn`. Specifications of these methods are very similar to their counterparts for `PtrOwn`; we do not give them here but refer to Appendix C.

6.2 Authority and Fragment

As sketched in §5, we use local invariants to manage the internal state of persistent arrays. It thus owns the `PermCellOwn` tokens, and states that the values stored in the cells are as expected. But we need a mechanism to relate the inner state of the invariant (*i.e.*, the set of cells it knows about and the associated array contents) with the state seen by the clients owning the `PArray` instances.

```
impl<K, V> Authority<K, V> {
  #[ensures(result@ == FMap::empty())]
  #[check(ghost)]
  pub fn new() -> Ghost<Self>;

  #[requires(!self@.contains(*k))]
  #[ensures((^self)@ == self@.insert(*k, *v))]
  #[ensures((^self).id() == self.id())]
  #[ensures(result.id() == self.id())]
  #[ensures(result@ == (*k, *v))]
  #[check(ghost)]
  pub fn insert(&mut self, k: Snapshot<K>,
                v: Snapshot<V>) -> Fragment<K, V>;

  #[requires(self.id() == frag.id())]
  #[ensures(self@.get(frag@.0) == Some(frag@.1))]
  #[check(ghost)]
  pub fn contains(&self, frag: &Fragment<K, V>);
}
```

Figure 10. Specifications for authority and fragment.

This is a classic problem in Iris, solved with a resource algebra, which manages an append-only finite map. It provides an exclusive *authoritative* resource: it is non-duplicable, and knows about the whole map. It may be used to perform updates: in particular, new key-value bindings can be added to the authoritative resource. Other resources, called *fragments*, are also available: they are duplicable, and hold one of the key-value bindings known by the authoritative resource.

We implemented this resource algebra in our extension of Creusot, and after adding a thin abstraction layer, we get the API presented in Fig. 10. Unsurprisingly, it uses two ghost types, `Authority` and `Fragment`, and has a logical method `id` which identifies the underlying ghost variable. The method `new` creates a new empty instance, `insert` inserts a new binding to the map, returning the fragment and updating the authority. Finally, `contains` is a *ghost lemma*, which takes an authority and a fragment with matching identifiers and deduces an equality between values stored in the fragment and in the authoritative map for the given key.

6.3 Type Definitions and Invariants

The types used in our implementation of persistent arrays are shown in Fig. 11. The `Authority` part is stored in a local invariant; this local invariant is opened with the `PARRAY` namespace declared in §2.2. The local invariant also contains ghost state similar to what is found in §4.

A value of type `PArray` contains an `Rc` pointer to the cell containing its state of type `Inner`. This inner state is either

```
1  enum Inner<T> {
2    Root(Vec<T>),
3    Set { i:usize, val:T, nxt:Rc<PermCell<Inner<T>>> }
4  }
5
6  pub struct PArray<T> {
7    pcell: Rc<PermCell<Inner<T>>>,
8    frag: Ghost<Fragment<Id, Seq<T>>>,
9    inv: Ghost<Rc<LocalInv<PA<T>>>>,
10 }
11
12 impl<T> Invariant for PArray<T> {
13   #[logic]
14   fn invariant(self) -> bool {
15       self.frag@.0 == self.pcell@.id() &&
16       self.frag.id() == self.inv@.public() &&
17       self.inv@.namespace() == PARRAY()
18   }
19 }
20
21 struct PA<T> {
22   perms: FMap<Id, PermCellOwn<Inner<T>>>,
23   auth: Authority<Id, Seq<T>>,
24   depth: Snapshot<Mapping<Id, Int>>,
25 }
26
27 impl<T> Protocol for PA<T> {
28   type Public = Id;
29
30   #[logic]
31   fn protocol(self, resource_id: Id) -> bool {
32     self.auth.id() == resource_id &&
33     ∀<id> self.auth@.contains(id) ⟹
34       self.perms.contains(id) &&
35       self.perms[id].id() == id &&
36       match self.perms[id].val() {
37         Inner::Root(v) => self.auth@[id] == v@,
38         Inner::Set { index, value, next } =>
39           self.auth@.contains(next@.id()) &&
40           self.depth[id] > self.depth[next@.id()] &&
41           index@ < self.auth@[next@.id()].len() &&
42           self.auth@[id] ==
43             self.auth@[next@.id()].set(index@, *value)
44       }
45   }
46 }
```

Figure 11. Definitions for persistent arrays.

`Root`, for a direct representation with a mutable vector, or `Set`, for a patch of another persistent array.

The type `PArray` also contains two ghost fields. The first, `inv`, points to the shared local invariant with a `Rc`. The `Rc` type is only used to enable sharing; there is no reference counting, since it is wrapped in `Ghost` and Creusot does not model the reference count. The second, `frag`, is the fragment resource, as described in §6.2. It allows easy access to the observable value of the array in specifications (with `pa.frag@.1`), which is kept in sync with the state stored in the local invariant thanks to the authoritative part.

A type invariant on `PArray` restricts the values that can be stored in these ghost fields: the identifier of the `PermCell` is the key of the fragment, the public information of the local invariant is the identifier of the authoritative/fragment ghost resource, and the namespace of the invariant is `PARRAY()`.

The local invariant owns the ghost state `PA`: it stores the permissions for the interior mutable cells (in a finite map), the authoritative resource, and *depth*s that increase along links (depths are used to guarantee that there is no loop in the link graph, which would make rerooting incorrect).

The first clause of this invariant, line 32, ensures that the public data of the invariant is indeed the identifier for the `Authority`. The rest of the invariant is similar to the type invariant shown in §4.2, the authoritative map playing the role of the domain. Each cell identifier in the map is required to be well-formed: the `permissions` map must contain the right ghost permission (line 35). From line 36 to 46, the invariant ensures consistency of the logical content of the authority and the program values.

The code and the proof of the functions shown in §2.2 rely on opening the invariant, which is done with the function `LocalInv::open`, performing the required accesses, and mutating the ghost state.

7 Discussion

In this section, we discuss our design choices, the limitations of our work, and possible improvements.

7.1 Expressiveness

Step-Indexing. When compared with Iris, Creusot does not have a notion of step-indexing. This makes usage and implementation of ghost code simpler, as Creusot does not have to have an equivalent of the *later* modality. The flip side is that Creusot cannot encode higher-order ghost state [17], as it would not be able to prove termination of ghost code in the general case. As a concrete consequence, trait objects (denoted by the `dyn` keyword in Rust) are not supported in Creusot, as they can be used to implement recursion via a construction close to the Landin knot [20] using invariants and higher-order ghost state without step-indexing.

***Fractional* Points-To.** A less inherent limitation is the lack of fractional *points-to* in Creusot, which is usually used to express sharing of memory locations. In separation logic, it is common to attach fractions to *points-to* assertions to denote partial ownership of memory locations [6]. When

owning the full fraction (*i.e.*, 1), one has exclusive ownership of the memory location, and can perform both reads and writes. When owning a fraction strictly smaller than 1, one can only perform reads.

In Creusot, we can share a ghost object by using either a shared borrow, or a `Rc` (as seen in §6). The lifetime of a shared borrow is infectious, limiting its use in data structures. On the other hand, `Rc` can be freely shared, but cannot give back full ownership: this would require proving the reference count reaches 1, but Creusot does not track this count. These limitations could be lifted by having `PtrOwn` parameterized by a fraction: we don't foresee any difficulties in doing this.

Poor Support for `std` Types with Interior Mutability. It is not possible to give strong specifications in Creusot to the interior mutable types of Rust's standard library (*e.g.*, `Cell`, `RefCell`, ...). For example, the specification for `Cell` only states that the contained value verifies some fixed predicate. The reason is that the API of these types do not accept ghost tokens such as `PermCellOwn`. Instead, if strong specifications are needed, Creusot-specific types must be used, like `PermCell`, but this may require rewriting code. On the other hand, thanks to formal verification, the dynamic checks required by *e.g.*, `RefCell` are no longer necessary.

Concurrency. Creusot has a prototyped support for parallelism, including sequentially consistent atomic variables, mutexes, thread spawning and atomic invariants that can be shared between threads. This support is mostly unsurprising, and uses the same ideas as exposed in this paper for atomics variables, atomic invariants and resource algebras.

7.2 Soundness

The soundness of the methodology of Creusot has already been studied in the RustHornBelt project [27]. This work misses important recent features of Creusot presented in this paper: ghost ownership and ghost code, type invariants, snapshots. We plan to extend RustHornBelt to support these, and we do not envision any fundamental difficulties: in particular, `PtrOwn` tokens should be modeled by a maps-to predicate, custom resources by Iris' native support for ghost resources, and local invariants by Iris' *non-atomic* invariants. We believe we could prove the soundness of ghost code erasure as well. However, due to the complexity of RustHornBelt, stemming from the subtlety of Rust's type system, this requires a lot of work.

Similar work has been partially done by Verus [14]. However, this work does not model the specificities of Creusot such as mutable borrows.

7.3 Improvements to the Case Studies

In §4, we used raw pointers to implement union-find, instead of a combination of `Rc` and `PermCell` as was done in persistent arrays. This means that each element allocates memory on the heap that will never be freed. This could be improved by either providing a freeing function to union-find, that frees all the pointers at once; or, we could use `Rc` and `PermCell`. As union-find requires comparison of pointers, this would require extending the interface of `Rc` specified by Creusot to allow reading the underlying pointer. We do not envision any difficulties in doing this.

At the end of §2.2, we discussed how the `get` function of persistent arrays locks the token with a lifetime, so that the returned value cannot be invalidated by a rerooting. Although this locking is required when using two arrays that have the same root, it is unnecessary in other cases; in particular, two distinct calls to `make` produce arrays with two separate roots, so we can hold a reference to an element of one while the other gets rerooted. To lift this limitation, namespaces could be made more fine-grained. Each call to `make` would generate a fresh namespace alongside the returned array, allowing more modularity.

8 Related Work

Verus [14, 22] is the first tool using resource-tracking ghost code to prove programs. Our types `PermCell`, `PermCellOwn` and `PtrOwn` are indeed very similar to Verus's types `PermCell`, `pcell::PointsTo` and `raw_ptr::PointsTo`, and the code written in our `ghost!` macro is similar to Verus' "proof mode". Perhaps the most important difference is the incomplete support for mutable borrows in Verus, which makes it impossible to specify functions such as `PtrOwn::as_mut` and `PermCell::borrow_mut`. They are idiomatic: they correspond to, *e.g.*, `<*mut T>::as_mut` and `RefCell::borrow_mut` in Rust. They improve expressiveness, since one cannot write to one field of a record stored in a `PermCell` or accessible from a raw pointer without touching the other fields.

Properly speaking, Verus ghost code is not Rust code: it is a different language, with added keywords and syntax, which can be embedded in Rust code using a dedicated `verus!` macro. On the contrary, Creusot uses Rust's attribute macros in order to annotate syntactically correct Rust code, and the syntax and semantics of our ghost code are that of Rust code.

Verus supports resource algebras, but advocates for a different mechanism, called VerusSync, for customizable ghost resources. VerusSync is tailored to the use of ghost resources of Verus, and it is unclear to us whether it is as expressive as pure resource algebras. As a result, although Verus does model pure resources algebras in its standard library, it does not make them as easy to use. As an example, ghost state updates are done via the equivalent of the `RA::update` function, which is unwieldy as it requires describing the full result of the update. To easily describe those updates, we provide the `Update` trait, for which Verus has no counterpart.

Verus has many case studies, including fairly large verified developments of systems code. Among others, Verus users have verified a distributed key-value store, a concurrent memory allocator, and various libraries useful to develop

low-level system code. To the best of our knowledge, none of these case studies include a data structure close to union-find or persistent arrays.

Poli et al. [30] present the tool Mendel, whose aim is to verify Rust programs using interior mutability. It uses a novel concept of *implicit capabilities* that can be used to annotate programs and prove strong specifications even in the presence of interior mutability. Contrarily to our approach, Mendel does not support unsafe uses of raw pointers, and its encoding to first-order logic uses a *memory model* (*i.e.*, the SMT solvers directly reason on fractions of the heap), which Creusot tries to avoid for performance reasons. On the other hand, Mendel is able to verify Rust code using the idiomatic Rust types from the standard library, while our approach requires using `PermCell`.

The interior mutable type `GhostCell` has been proposed by Yanovski et al. [33]. Accesses to its contents requires using a value of type `GhostToken`, whose purpose is very similar to our `PermCellOwn`. Contrarily to our work, Yanovski et al. focus on type safety and do not prove any functional properties. In addition, they use the technique of *branded lifetimes* to link the token and the cell, while we use preconditions stating that the identifiers agree.

Gillian-Rust [4] is a verification tool for Rust based on the Gillian platform [26]. It is able to formally verify Rust programs with unsafe uses of raw pointers by using separation logic directly. Interestingly, Gillian-Rust is able to use Creusot specifications, allowing hybrid proofs written using both approaches at the same time.

The union-find data structure has been formally verified by several authors: first, Charguéraud and Pottier [7] verified an OCaml implementation using Rocq and the CFML separation logic framework. Their specification includes functional correctness and Tarjan's complexity bound proportional to the inverse Ackermann function. Mével et al. [29] later translated this proof into the Iris separation logic for a program written in HeapLang, Iris's toy programming language. These proofs use separation logic, and are very manual by virtue of being done in Rocq.

An unpublished proof [32] of the union-find data structure in Why3 is available in Why3's repository. Our proof is similar, but we use the generic mechanism of ghost permissions in order to handle aliased mutable memory, whereas they use an ad-hoc memory model, axiomatized with a global map representing values stored in memory.

Conchon and Filliâtre [9] present a persistent version of the union-find data structure, using persistent arrays as a backend. They provide a proof of correctness in Rocq, using an ad-hoc memory model similar to that of the union-find proof in Why3 above. The proof is manual, and the verified version is not directly executable, but rather a Rocq translation of OCaml code.

Allain et al. [1] present the data structure of *snapshottable store*, a generalization of persistent arrays. They provide a manual formalization in Rocq with the Iris separation logic. Contrarily to our library, their formally verified code is written in HeapLang and not directly executable.

References

[1] Clément Allain, Basile Clément, Alexandre Moine, and Gabriel Scherer. 2024. Snapshottable Stores. In *ICFP*. doi:10.1145/3674637

[2] Vytautas Astrauskas, Aurel Bílỳ, Joná š Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NFM*. doi:10.1007/978-3-031-06773-0_5

[3] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. In *OOPSLA*. doi:10.1145/3360573

[4] Sacha-Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. 2025. A Hybrid Approach to Semi-automated Rust Verification. In *PLDI*. doi:10.1145/3729289

[5] Henry G. Baker. 1978. Shallow Binding in Lisp 1.5. *CACM* 21, 7 (1978). doi:10.1145/359545.359566

[6] John Boyland. 2003. Checking interference with fractional permissions. In *SAS*. doi:10.1007/3-540-44898-5_4

[7] Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *JAR* 62, 3 (2019). doi:10.1007/s10817-017-9431-7

[8] Martin Clochard, Jean-Christophe Filliâtre, and Andrei Paskevich. 2015. How to Avoid Proving the Absence of Integer Overflows. In *VSSTE*. doi:10.1007/978-3-319-29613-5_6

[9] Sylvain Conchon and Jean-Christophe Filliâtre. 2007. A Persistent Union-Find Data Structure. In *Workshop on ML*. doi:10.1145/1292535.1292541

[10] Xavier Denis, Arnaud Golfouse, Armaël Guéneau, Jacques-Henri Jourdan, and Dominik Stolz. 2025. Using a Prophecy-Based Encoding of Rust Borrows in a Realistic Verification Tool. (2025). https://inria.hal.science/view/index/docid/5244847

[11] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: a Foundry for the Deductive Verification of Rust Programs. In *ICFEM*. doi:10.1007/978-3-031-17244-1_6

[12] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 2016. A Pragmatic Type System for Deductive Verification. (2016). https://inria.hal.science/hal-01256434

[13] Daniel Gratzer, Mathias Adam Møller, and Lars Birkedal. 2025. Idempotent Resources in Separation Logic: The Heart of Core in Iris. In *FoSSaCS*. doi:10.1007/978-3-031-90897-2_3

[14] Travis Hance. 2024. *Verifying Concurrent Systems Code*. Ph. D. Dissertation. Carnegie Mellon University. doi:10.1184/R1/27203919

[15] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust Verification by Functional Translation. In *ICFP*. doi:10.1145/3547647

[16] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. In *POPL*. doi:10.1145/3158154

[17] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-Order Ghost State. In *ICFP*. doi:10.1145/2951913.2951943

[18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the Ground up: a Modular Foundation for Higher-Order Concurrent Separation Logic. *JFP* (2018). doi:10.1017/S0956796818000151

[19] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. *POPL*. doi:10.1145/2676726.2676980

[20] P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* (1964). doi:10.1093/comjnl/6.4.308

[21] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *SOSP*. doi:10.1145/3694715.3695952

[22] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. In *OOPSLA*. doi:10.1145/3586037

[23] Nico Lehmann, Adam T Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. In *PLDI*. doi:10.1145/3591283

[24] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2022. Linear Types for Large-Scale Systems Verification. In *OOPSLA*. doi:10.1145/3527313

[25] Maroua Maalej, Tucker Taft, and Yannick Moy. 2018. Safe Dynamic Memory Management in Ada and SPARK. In *Ada-Europe*. doi:10.1007/978-3-319-92432-8_3

[26] Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *CAV*. doi:10.1007/978-3-030-81688-9_38

[27] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *PLDI*. doi:10.1145/3519939.3523704

[28] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-Based Verification for Rust Programs. In *TOPLAS*. doi:10.1145/3462205

[29] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and time Receipts in Iris. In *ESOP*. doi:10.1007/978-3-030-17184-1_1

[30] Federico Poli, Xavier Denis, Peter Müller, and Alexander J Summers. 2024. Reasoning about Interior Mutability in Rust using Library-Defined Capabilities. *arXiv preprint arXiv:2405.08372* (2024). doi:10.48550/arXiv.2405.08372

[31] Creusot team. 2020-2025. Creusot source code. https://github.com/creusot-rs/creusot/ Accessed: 2025-12-03.

[32] The VOCaL Project. 2021. Proof of Union-Find with Why3. https://github.com/ocaml-gospel/vocal/blob/0bdaa49/proofs/why3/UnionFind_impl.mlw Accessed: 2025-08-18.

[33] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. GhostCell: Separating Permissions from Data in Rust. In *ICFP*. doi:10.1145/3473597

A Specifications for `LocalInv`

```
#[requires(value.protocol(*public))]
#[ensures(result.public() == *public)]
#[ensures(result.namespace() == *namespace)]
#[check(ghost)]
pub fn new<T>(
  value: Ghost<T>,
  public: Snapshot<T::Public>,
  namespace: Snapshot<Namespace>,
) -> Ghost<LocalInv<T>>;

#[requires(tokens.contains(l.namespace()))]
#[requires(∀<t: Ghost<&mut T>>
    (**t).invariant_with_data(l.public()) && inv(t) ⟹
    f.precondition((t,)) &&
    ∀<res: A>
      f.postcondition_once((t,), res) ⟹
      (^*t).invariant_with_data(l.public()))]
#[ensures(∃<t: Ghost<&mut T>>
    (**t).invariant_with_data(l.public()) &&
    f.postcondition_once((t,), result))]
#[check(ghost)]
pub fn open<'a, T: Protocol, A>(
  l: Ghost<&'a LocalInv<T>>,
  tokens: Ghost<Tokens<'a>>,
  f: impl FnOnce(Ghost<&'a mut T>) -> A) -> A;
```

B Trait `Update` for Justifying Ghost Resource Updates

```
pub trait Update<R: RA> {
  type Choice;

  #[logic]
  fn premise(self, from: R) -> bool;

  #[logic]
  #[requires(self.premise(from))]
  fn update(self, from: R, ch: Self::Choice) -> R;

  #[logic]
  #[requires(self.premise(from))]
  #[requires(from.op(frame) != None)]
  #[ensures(self.update(from, result).op(frame) != None)]
  fn frame_preserving(self, from: R, frame: R)
      -> Self::Choice;
}
```

C Specifications for `PermCell`

```
impl<T> PermCell<T> {
  #[ensures(result.0.id() == result.1.id())]
  #[ensures((*result.1)@ == value)]
  pub fn new(value: T) -> (Self, Ghost<PermCellOwn<T>>);

  #[requires(self.id() == perm.id())]
  #[ensures(*result == perm@)]
  pub unsafe fn borrow<'a>(&'a self,
        perm: Ghost<&'a PermCellOwn<T>>) -> &'a T;

  #[requires(self.id() == perm.id())]
  #[ensures(self.id() == (^perm).id())]
  #[ensures(*result == perm@)]
  #[ensures(^result == (^perm)@)]
  pub unsafe fn borrow_mut<'a>(&'a self,
        perm: Ghost<&'a mut PermCellOwn<T>>)
      -> &'a mut T;
}
```

Received 2025-09-12; accepted 2025-11-13