Summer Internship Project
Report

Formal verification of programs with pointers

Submitted by

Oualid CHABANE
Third year of bachelor's double degree in
Computer Science, Magistère d'informatique track.
Faculty of sciences of Orsay, Paris-Saclay University.

Under the guidance of

Arnaud Golfouse
Paul Patault
Jean-Christophe Filliâtre

Department of Formal Methods
The Laboratoire Méthodes Formelles (LMF).
4, avenue des Sciences, 91190 Gif-sur-Yvette

Summer Internship 2025

Contents

```rust
extern crate creusot_contracts;
use creusot_contracts::*;

#[requires(n@ * (n@ + 1) / 2 < u32::MAX@)]
#[ensures(result@ == n@ * (n@ + 1) / 2)]
pub fn sum_first_n(n: u32) -> u32 {
    let mut sum = 0;
    #[invariant(sum@ * 2 == produced.len() * (produced.len() + 1))]
    for i in 1..=n {
        sum += i;
    }
    sum
}
```

Figure 1: CREUSOT verification example: sum of first `n` natural numbers

1 Introduction

1.1 The Laboratoire Méthodes Formelles (LMF)

LMF is a joint research center of Paris-Saclay University, CNRS, ENS Paris-Saclay, Inria, and CentraleSupélec. It's divided in multiple departments interested in various topics such as type systems, quantum computing and topology. My research project took place within Toccata's team of the formal methods department. I worked on verification of programs with pointers under the supervision of Jean-Christophe Filliâtre, Arnaud Golfouse and Paul Patault.

There are multiple verification tools widely used and developed at the LMF research center. Among these tools, we find CREUSOT and WHY3.

1.2 Creusot tool

1.2.1 Introduction

CREUSOT [1] is a formal verification tool designed for RUST code. It guarantees safety by detecting compile-time errors, runtime panics, and integer overflows, while also ensuring that the code respects its formal specification.

CREUSOT operates on top of WHY3 indirectly by translating RUST code into an intermediate verification language known as COMA. It facilitates the verification, and it also gives CREUSOT access to the full access to WHY3's backend.

In Figure 1 is a simple example of CREUSOT code that verifies the correctness of the `sum_first_n` function.

Explanation:

The function `sum_first_n` in Figure 1 computes the sum of the first `n` natural numbers. The special comments denoted with `#[...]` represent its specification:

- **Precondition:** It asserts that the sum of the first `n` natural numbers where `n` is provided as a parameter, does not overflow the capacity of the result type. This ensures that the final sum, which will be stored in the return variable will not exceed the capacity of the return type. The operator `@` converts machine integers into mathematical integers, which are unbounded, allowing us to use arithmetic theory.
- **Postcondition:** It states that the returned result is equal to the expected mathematical value: `n * (n + 1) / 2`.
- **Loop invariant:** The expression `produced.len()` corresponds to the number of iterations performed so far. It effectively can play the role of the loop index by applying a transformation.

```
let mut g = ghost!(50);
ghost! {
    *g *= 2;
};
proof_assert!(g@ == 100);
```

Figure 2: An example of ghost state manipulation in Creusot.

The reason we cannot refer directly to the loop index is due to scoping limitations.

1.2.2 Ghost code in Creusot

Rust's ownership and borrowing principles make it difficult to use pointers in proofs, where the need for a notion of ghost code in such a language. Ghost code [2] allows us to perform proofs that are not possible using only raw code and the logical world, especially when dealing with existential quantification, if we know how to construct the value we are seeking for the existential quantifier.

Recently, ghost code has been introduced in Creusot. Ghost code is separate from normal code via syntax and typing. Therefore, it can be safely erased at compile time, allowing only the original code to be executed. This results in more expressive proof verification. In Figure 2 are some examples of how to write ghost code in Creusot.

2 State of art

2.1 Reynolds' separation logic

Reynolds [4] introduces a new concept in formal program verification, *Separation Logic* is an extension of Hoare's Logic that facilitates automatic proofs on low-level imperative programs that use shared mutable data structures, by reasoning about disjoint parts of the heap. It greatly simplifies the formalization and verification of many problems that are otherwise difficult to handle using traditional Hoare's Logic, such as aliasing, memory management, and concurrency.

Reynolds is particularly interested in the `in_place_reversal`† algorithm. He frequently discusses it in this research paper, as it is a very interesting algorithm for exploring formal proofs on mutable data structures with pointers.

To prove properties of algorithms on data structures, it's usually not enough to rely only on the program's representation. We often build a logical model of the data and connect it to the program state using predicates. For instance, the intuitive logical modeling of lists are sequences, therefore we can write the following predicate to represent a list:

$$\texttt{list}\ \epsilon\ \texttt{i} \stackrel{\text{def}}{=} \texttt{i = nil}$$

$$\texttt{list}\ (a.\alpha)\ \texttt{i} \stackrel{\text{def}}{=} \exists\ \texttt{j},\ \texttt{list}\ \alpha\ \texttt{j} \wedge\ \texttt{i} \hookrightarrow \texttt{a, j}$$

where $\texttt{i} \hookrightarrow \texttt{a}$ means "i points to a"
. The predicate `list` allows us to link the code pointer `i` into it's corresponding sequence α.
If we extend the proof to lassos, it would be better to define a list in a more general way, so that a list is identified by a pair of pointers: the first pointer represents the head of the sub-list, and the second pointer represents the tail of the sub-list. This way, we can separate the lasso into its two main parts, the cycle and the entry list.

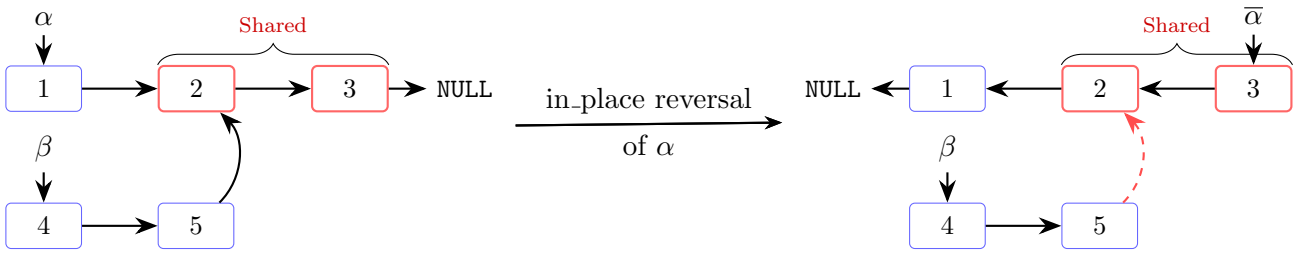

Figure 3: `in_place reversal` on shared lists

One of key uses of separation logic can be seen when trying to use `in_place_reversal`† on shared data structures, in this context, given the following precondition for in-place reversal: `list p` α, the post-condition `list p` $\overline{\alpha}$ is not sufficient to ensure the correctness of the algorithm. Consider the example in Figure 3.

So we need to provide a stronger precondition that prevents such cases. One possible way is:

$$\texttt{list p}\ \alpha\ \wedge\ \forall \texttt{x}, \alpha'.\ \texttt{list x}\ \alpha' \rightarrow \texttt{conflicting(x,}\ \alpha',\ \texttt{p,}\ \alpha) \rightarrow \texttt{x} = \texttt{nil}$$

where `conflicting: Pointer -> Pointer -> bool` is a predicate that returns `True` if the two provided lists share any nodes. It is defined as follows:

$$\begin{aligned}
\texttt{conflicting nil p'} &\stackrel{\text{def}}{=} \texttt{false} \\
\texttt{conflicting p nil} &\stackrel{\text{def}}{=} \texttt{false} \\
\texttt{conflicting p p'} &\stackrel{\text{def}}{=} \texttt{p} = \texttt{p'} \vee \\
&\quad \exists \texttt{a, q, p'} \hookrightarrow \texttt{a, q}\ \wedge\ \texttt{conflicting p q} \vee \\
&\quad \exists \texttt{a, q, p} \hookrightarrow \texttt{a, q}\ \wedge\ \texttt{conflicting q p'}
\end{aligned}$$

In Creusot, we will need to provide the sequences associated with the pointers to obtain the permissions required to access their successors.
We can clearly notice that the precondition, but also the invariant, and the post-condition become extremely complicated. This is where separation logic proves invaluable. By leveraging heap separation, the specifications become significantly clearer and simpler.

$$\begin{aligned}
\texttt{list}\ \epsilon\ \texttt{p} &\stackrel{def}{=} \texttt{p = nil} \\
\texttt{list}\ a.\alpha\ \texttt{p} &\stackrel{def}{=} \texttt{p} \hookrightarrow \texttt{a, p' * list}\ \alpha\ \texttt{p'}
\end{aligned}$$

Although Creusot does not support Separation Logic, its principles can be emulated through the use of the Rust type system and ghost code. The latter can be used to carry over separation logic principles that are implicitly guaranteed by the Rust type system into the logical world. To be more precise, let us consider the interface of `PtrOwn<T>::disjoint_lemma` as an illustrative example.

This lemma is admitted as an axiom in Creusot, and what it does is verify that the two permissions correspond to two distinct pointers on the heap. If this lemma was logical, it would not hold, because we quantify universally over `own1` and `own2` and we can't guarantee separation in Creusot's logic. Therefore, one could be equal to the other, and the first post-condition would not be satisfied. However, if we admit it as a ghost lemma, the Rust type system, within the context of ghost code prohibits having a mutable borrow and an immutable borrow to the same value. Therefore, `own1` and `own2` are necessarily distinct, and the validity of the axiom follows (the second post-condition is not essential for conveying the idea).

```rust
/// Ensures two PtrOwns reference different memory locations
#[ghost]
#[ensures(own1.ptr().addr_logic() != own2.ptr().addr_logic())]
#[ensures(*own1 == ^own1)]
pub fn disjoint_lemma(own1: &mut PtrOwn<T>, own2: &PtrOwn<T>)
```

Figure 4: `PtrOwn<T>::disjoint_lemma` interface

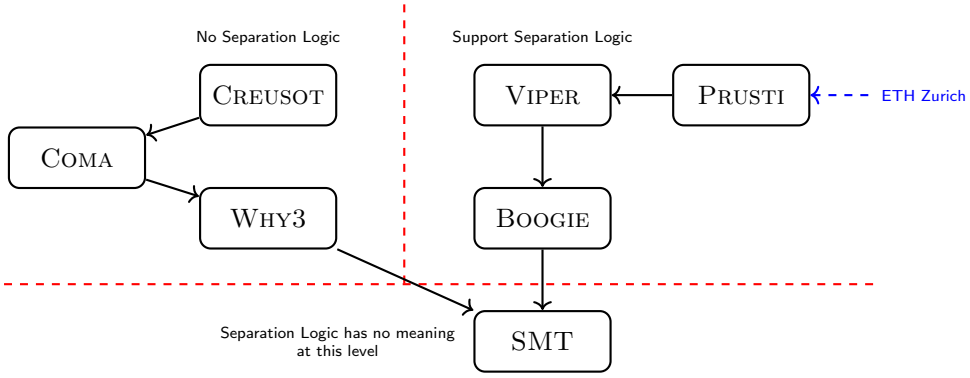

Figure 5: Illustration of the relationship between verification tools and intermediate languages.

There are, however, more specialized tools designed specifically for reasoning with Separation Logic. One such tool is Prusti, a Rust verifier that supports Separation Logic. It is built on top of Viper which is built on top of Boogie, an intermediate verification language (IVL) developed by Microsoft Research, and is widely used in the field of formal verification. The Figure 5 clarifies the relationships between these languages and their connection to Separation Logic.

2.2 Different ways of implementing the problem

The list reversal problem has already been proven in two different ways in Creusot, but both methods have certain limitations:

- **BOX method**: This approach models lists using Rust `Box` type. However, it imposes strong restrictions on the memory model by prohibiting any form of sharing or aliasing, since `Box` does not support multiple references to the same memory location. As a result, the specification is simple and the proof goes through easily.
- **Memory model method**: This approach relies on modular reasoning over the memory. In other words, it involves passing an object that models the entire memory as a parameter to each method to be verified. As a result, verification requires reasoning about the complete memory state. This can quickly lead to complex proofs even for simple algorithms. For example, suppose we have two disjoint lists in the heap, each verified using a `list` predicate. If we reverse one of them, the `list` predicate on the other is not preserved automatically, and we must explicitly include it in the proof. This makes the verification process tedious. This is where separation logic becomes useful, and the solution we propose implicitly uses its principles, thanks to Rust type system.

3 Problem definition and proposed solution

3.1 Problem definition

Formal verification of mutable data structures with pointers is one of the most challenging aspects of formal methods. The in-place list reversal algorithm is a canonical example of this challenge, as it can easily lead to memory leaks, dangling pointers, or logical errors.

```rust
pub fn reverse_in_place(mut p: RawPtr<Self>) -> RawPtr<Self> {
    let mut q: *const Node<T> = ptr::null();
    while !p.is_null() {
        let p2 = unsafe { &mut *p };
        let next = p2.next;
        p2.next = q;
        q = p;
        p = next;
    }
    q
}
```

Figure 6: Rust code of `in_place_reversal`.

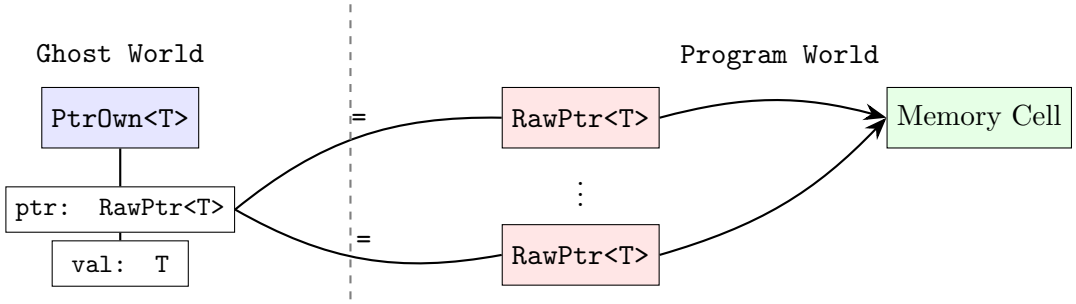

Figure 7: Internal structure of `PtrOwn<T>`.

The core problem is that, for a given list implementation in Rust using raw pointers, we need to formally verify the correctness of an in-place reversal algorithm. Not only that, but also the preservation of memory safety and the proper handling of shared lists. The rust code that we have proposed is in Figure 6

3.2 Proposed solution

3.2.1 PtrOwn and RawPtr

Used by Creusot to manipulate pointers in proofs. `PtrOwn` models ownership of memory cells in the ghost world and can be used in parallel with `RawPtr`, which represents the corresponding address of the cell represented by `PtrOwn`. The internal representation of `PtrOwn` is shown in Figure 7.

Our solution relies on the use of linear algebraic types in Creusot, specifically, `PtrOwn` and `RawPtr`. This allows us to prove the correctness of in-place reversal even in the presence of shared data structures, making it better than the BOX method†. Moreover, it also outperforms the memory model† approach, our method requires to reason locally on memory. In other words, we only need to verify the parts of the heap we are manipulating, without having to reason about independent regions. This provides a form of separation logic, implicitly enforced by the rust type system. In the following we will present our solutions in steps.

3.2.2 Data Structure

The list type is represented by `RawPtr<Node<T>>` where `Node<T>` is defined like the following:

```rust
struct Node<T> {
    elem: T,
    pub next: RawPtr<Node<T>>,
}
```

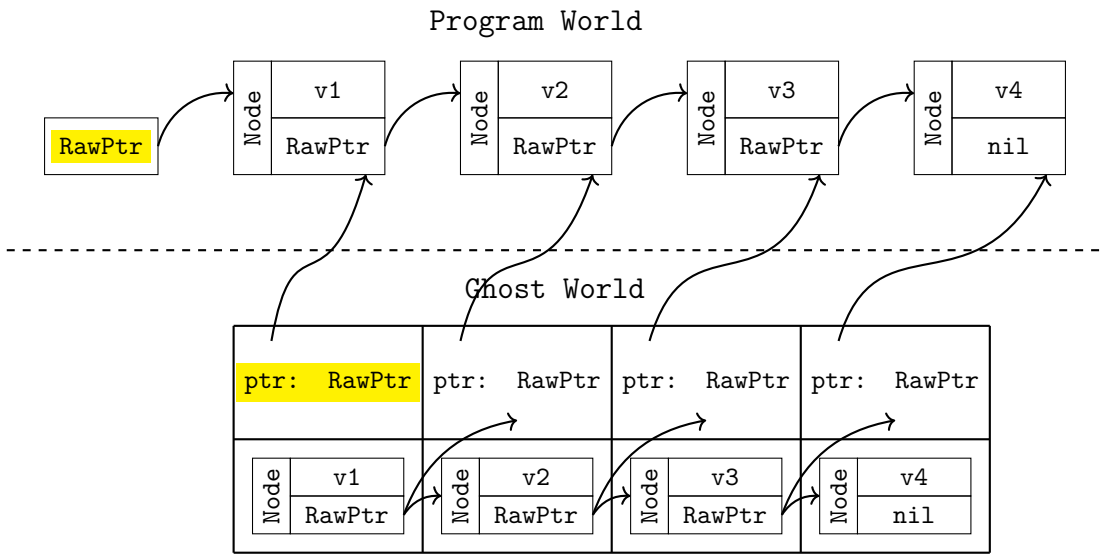

Figure 8: Memory representation of a list with 3 elements with raw pointers and Creusot sequences.

3.2.3 Predicates

- **The `list` Predicate** †
  We defined a predicate `list` that takes a pointer of type `RawPtr` and the abstract ghost sequence of permission `PtrOwn` that represents the list algebraically `seq`. It checks recursively that the pointers inside the permissions in the sequence correspond to the pointers in the program world, and that the list ends with `nil`.

```
#[predicate]
#[variant(perm_seq.len())]
fn list(l: RawPtr<Self>, perm_seq: Seq<PtrOwn<Node<T>>>) -> bool {
    if l.is_null_logic() {
        perm_seq.len() == 0
    } else {
         if perm_seq.len() > 0 {
            let ptr = perm_seq[0].ptr();
            l == ptr && Self::list(perm_seq[0].val().next, perm_seq.tail())
        } else {
            false
        }
    }
}
```

  A pointer and a sequence that verify this predicate, can be seen on the memory level as in Figure 8.

- **The `inverse` Predicate** †
  The predicate `inverse` takes two sequences of `PtrOwn` objects and determines whether the elements of the first sequence appear in reverse order in the second. One might think that this predicate could be more simply expressed using the built-in `rev` function, as in `inverse(s1, s2) := s1.rev() == s2`. However, this is not the case due to our choice of algebraic representation for the list and the invariants enforced by the algorithm.

  At some point in the specification, we need to express that the elements of a certain sequence in the program appear in reverse order compared to their version at the beginning. But we

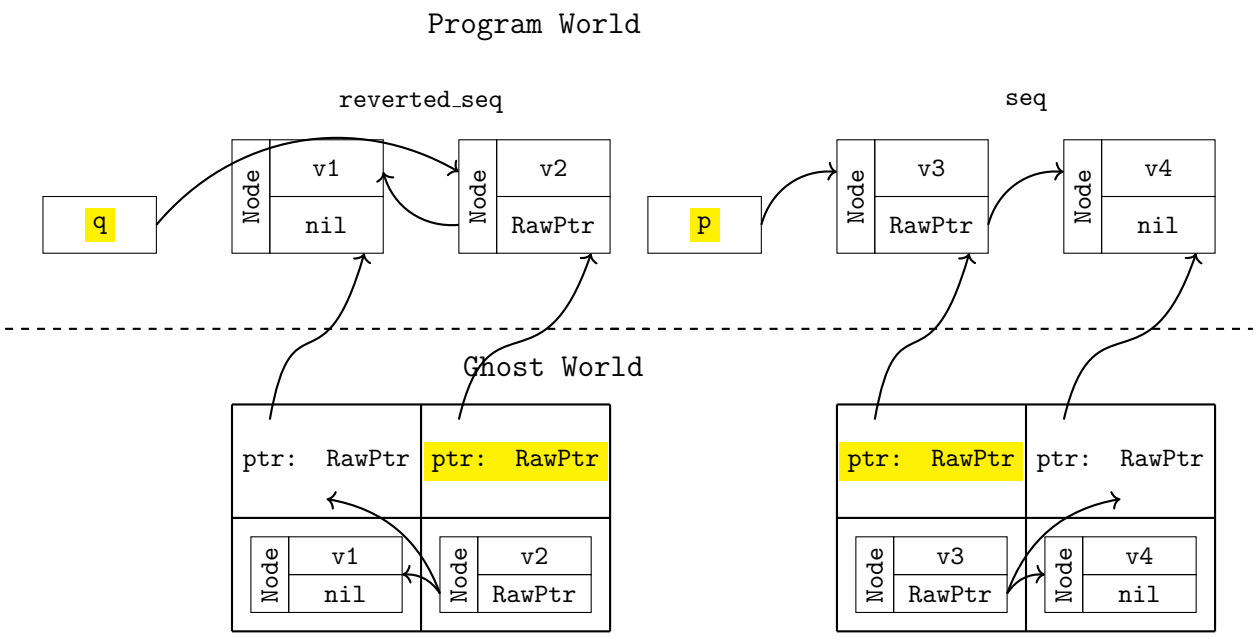

Figure 9: Memory representation of data mid-execution.

cannot use the `inverse` predicate as defined above, because even though the elements in the two sequences are logically reversed, `inverse` will return `false` since the internal pointers are reversed as well, and pointer equality does not hold.

3.2.4 Code specification of `in_place_reversal`

Reference to the code in Figure 6

- **Pre-condition**: As a precondition we require that the pointer `p` and the provided sequence `seq` form a list, in other means, it obeys to the structure presented in Figure 8.
- **Post-conditions**: As a post-condition we need to ensure that, the result pointer `result` and the final sequence `seq` form a list and obeys to the structure presented in Figure 8. Also we have to precise that the elements in the final list are set in reverse order of the primary list.

```
#[ensures(Self::list(result, *^seq))]
#[ensures(Self::inverse(**seq, *^seq, 0, (*^seq).len()))]
```

  where `^` denotes the object at the end of the program.

- **Invariants**: In the loop, we maintain two sequences: `reverted_seq` and `seq`. At each iteration, we redirect the pointer in the correct direction, pop the head of the `seq` sequence, and push it into `reverted_seq`. As a result, the memory representation of the data during execution resembles the diagram shown in Figure 9.

  In order to ensure the post-conditions, we need to maintain during the loop that the pointer `q` and the sequence `reverted_seq` form a valid list, the latter will hold, at the end of the program, the sequence corresponding to the reversed list. This invariant is crucial to guarantee the first post-condition at the end of the iteration.

```
#[invariant(Self::list(q, *reverted_seq))]
```

  We must also ensure that the remaining part of the list, represented by the pointer `p` and the sequence `seq`, continues to form a valid list.

```
#[invariant(Self::list(p, **seq))]
```

  More importantly, we use the predicate `inverse` to check that the elements in `reverted_seq`

are pushed in the reverse order. Otherwise, if we relied on a built-in reversal function, the correctness would not be guaranteed, as illustrated in Figure 9.

```
#[invariant(Self::inverse(_seq0.subsequence(0, reverted_seq.len()),
↪  *reverted_seq, 0, reverted_seq.len()))]
```

4 Conclusion and perspectives

4.1 Conclusion

This work successfully addresses the challenge of formally verifying in-place linked list reversal in the presence of shared data structures using Creusot. I have learned many concepts through this research work, not only about formal verification but also about program semantics and notions about the provers' compilers.

4.2 Perspectives

we are looking forward to extending the proof to lasso structures and, more importantly, to proving the Morris [3] tree traversal algorithm, which was my initial subject. Unfortunately, due to the complexity of the proof compared to the given time, We decided to work on a simpler but very important algorithm, which is the list reversal.

References

[1] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. "Creusot: a Foundry for the Deductive Verification of Rust Programs". In: *Lecture Notes in Computer Science*. Lecture Notes in Computer Science. Madrid, Spain: Springer Verlag, Oct. 2022. URL: https://inria.hal.science/hal-03737878.

[2] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. "The Spirit of Ghost Code". In: *Computer Aided Verification (CAV 2014)*. Vol. 8559. Lecture Notes in Computer Science. 26th International Conference, Vienna Summer of Logic, Austria. Springer, 2014, pp. 1–16. DOI: 10.1007/978-3-319-08867-9_1. URL: https://hal.science/hal-00873187v3.

[3] Joseph M. Morris. "Traversing binary trees simply and cheaply". In: *Information Processing Letters* 9.5 (1979), pp. 197–200. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(79)90068-1. URL: https://www.sciencedirect.com/science/article/pii/0020019079900681.

[4] J.C. Reynolds. "Separation logic: a logic for shared mutable data structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

5 Appendix

5.1 in_place_reversal algorithm

```
                j := nil; while i <> nil do
                    (k := [i + 1]; [i + 1] := j; j := i; i := k).
```

5.2 Code

```
extern crate creusot_contracts;
use ::std::ptr;
use creusot_contracts::ptr_own::{PtrOwn, RawPtr};
use creusot_contracts::*;
pub struct Node<T> {
    elem: T,
    pub next: RawPtr<Node<T>>,
}

impl<T> Node<T> {
    #[predicate]
    #[variant(perm_seq.len())]
    fn list(l: RawPtr<Self>, perm_seq: Seq<PtrOwn<Node<T>>>) -> bool {
        pearlite! {
            if l.is_null_logic() {
                perm_seq.len() == 0
            } else {
                 if perm_seq.len() > 0 {
                    let ptr = perm_seq[0].ptr();
                    l == ptr && Self::list(perm_seq[0].val().next,
                    ↪  perm_seq.tail())
                } else {
                    false
                }
            }
        }
    }

    #[ensures(Self::list(result.0, *result.1))]
    #[ensures(result.0.is_null_logic())]
    pub fn empty() -> (RawPtr<Self>, Ghost<Seq<PtrOwn<Node<T>>>>) {
        (ptr::null(), Seq::new())
    }

    #[requires(Self::list(l, **seq))]
    #[ensures(Self::list(result,  *^seq))]
    #[ensures(forall<i:Int> 0 <= i && i < (^seq).tail().len() ==> seq[i] ==
    ↪  (^seq).tail()[i])]
    #[ensures((^seq)[0].val().elem == e)]
    #[ensures((^seq)[0].ptr() == result)]
    #[ensures((^seq).len() == seq.len() + 1)]
    pub fn cons(e: T, l: RawPtr<Self>, seq: &mut Ghost<Seq<PtrOwn<Node<T>>>>) ->
    ↪  RawPtr<Self> {
        let (raw, own) = PtrOwn::new(Node { elem: e, next: l });
```

```
    let _seq2 = snapshot!(**seq);
    ghost!(seq.push_front_ghost(own.into_inner()));
    proof_assert!(*_seq2 == seq.tail());

    raw
}

#[requires(Self::list(p, **seq))]
#[requires(0 <= nth@ && nth@ < seq.len() )]
#[ensures(seq[nth@].val().elem == *result)]
pub fn nth(mut p: RawPtr<Self>, nth: i128, seq: &Ghost<Seq<PtrOwn<Node<T>>>>)
↪  -> &T {
    let mut i = 0;
    proof_assert!(**seq == seq.subsequence(0, seq.len()));
    #[invariant(0 <= i@ && i@ <= nth@)]
    #[invariant(Self::list(p, seq.subsequence(i@, seq.len())))]
    loop {
        let rw = unsafe {
            PtrOwn::as_ref(p,
            ↪  ghost!(seq.get_ghost(Int::new(i).into_inner()).unwrap()))
        };

        if i == nth {
            return &rw.elem;
        }

        p = rw.next;
        proof_assert!(seq.subsequence(i@, seq.len()).tail() ==
        ↪  seq.subsequence(i@+1, seq.len()));
        i += 1;
    }
}

#[predicate]
pub fn inverse(seq: Seq<PtrOwn<Node<T>>>, other: Seq<PtrOwn<Node<T>>>, lb:
↪  Int, lh: Int) -> bool
where
    T: Sized,
{
    pearlite! {
         forall<i: Int>
        lb <= i && i < lh
        ==> seq[i].val().elem == other[other.len() - i - 1].val().elem
    }
}

#[requires(Self::list(p, **seq))]
#[ensures(Self::list(result, *^seq))]
#[ensures(seq.len() == (^seq).len())]
#[ensures(Self::inverse(**seq, *^seq, 0, (*^seq).len()))]
pub fn reverse_in_place(
    mut p: RawPtr<Self>,
```

```
    seq: &mut Ghost<Seq<PtrOwn<Node<T>>>>,
) -> RawPtr<Self> {
    snapshot! {
        let _ = Seq::<T>::ext_eq;
    };
    let mut q: *const Node<T> = ptr::null();
    let mut reverted_seq = Seq::new();
    let _seq0 = snapshot!(**seq);

    #[invariant(Self::list(q, *reverted_seq))]
    #[invariant(Self::list(p, **seq))]
    #[invariant(Self::inverse(_seq0.subsequence(0, reverted_seq.len()),
    ↪  *reverted_seq, 0, reverted_seq.len()))]
    #[invariant(reverted_seq.len() + seq.len() == _seq0.len())]
    #[invariant(**seq == _seq0.subsequence(reverted_seq.len(), _seq0.len()))]
    #[invariant(inv(seq))]
    #[invariant(inv(reverted_seq))]
    while !p.is_null() {
        let _sloop_entry = snapshot!(**seq);
        let _revs_loop_entry = snapshot!(*reverted_seq);
        let p2 =
            unsafe { PtrOwn::as_mut(p,
            ↪  ghost!(seq.get_mut_ghost(*ghost!(0int)).unwrap())) };
        let next = p2.next;
        p2.next = q;
        q = p;
        p = next;
        let _sloop_exit = snapshot!(**seq);


        ↪  ghost!((*reverted_seq).push_front_ghost(seq.pop_front_ghost().unwrap()));

        //a0156: Assertion used to prove invariant #1 (we can remove it and
        ↪  use use_th seq.FreeMonoid instead)
        proof_assert!(reverted_seq.tail() == *_revs_loop_entry);

        //Hypothesis: invariant(Self::list (p, **seq))
        // We need to add to the hypothesis the fac that the tail of the
        ↪  previous seq is the new seq
        //a1369
        proof_assert!((*_sloop_exit).tail() == **seq);

        //In order to proof the last assertion, we need the following
        ↪  assertion
        //It esnures that seq.tail() didn't change between the beginig of the
        ↪  loop and the end, what ensures the stability of our invariant
        //a7070
        proof_assert!((*_sloop_exit).tail() == (*_sloop_entry).tail());

        //this should be enough to prove #[invariant(Self::list (p, **seq))],
        ↪  whith using the latter, creusot proves well the remaining
        ↪  invariant about q
        //proof_assert!(Self::list(p, (*snap2).tail()));
```

```
            //a1313
            proof_assert!(Self::list(p, (*_sloop_exit).tail()));
            // ==> invariant #1 checks for iteration n+1
        }
        //Pour montrer ensures#1 (ensures(seq.len() == (^seq).len() &&
        ↪  Self::inverse(**seq, *^seq, 0, seq.len())))
        //a4224
        proof_assert!(_seq0.subsequence(0, reverted_seq.len()) == *_seq0);
        ghost!(**seq = reverted_seq.into_inner());
        q
    }
}

#[ensures(Node::list(result.0, *result.1))]
#[ensures(result.1.len() == vec.view().len())]
#[ensures(forall<i: Int> 0 <= i && i < vec.view().len() ==>
↪  (*result.1)[i].val().elem == vec.view()[i])]
pub fn list_of_vector1<T>(mut vec: Vec<T>) -> (RawPtr<Node<T>>,
↪  Ghost<Seq<PtrOwn<Node<T>>>>) {
    //Takes possession of elements in the vector
    let (mut l, mut seq) = Node::empty();
    let _vec0 = snapshot!(vec);
    #[invariant(forall<i: Int>
        vec.view().len() <= i && i < _vec0.view().len() ==> seq[i -
        ↪  vec.view().len()].val().elem == _vec0.view()[i])]
    #[invariant(Node::list(l, *seq))]
    #[invariant(vec.view().len() + seq.len() == _vec0.view().len())]
    #[invariant(forall<i: Int> 0 <= i && i < vec.view().len() ==> vec.view()[i] ==
    ↪  _vec0.view()[i])]
    #[invariant(inv(seq))]
    loop {
        if let Some(v) = vec.pop() {
            l = Node::cons(v, l, &mut seq);
        } else {
            break;
        }
    }
    (l, seq)
}

pub fn tr() {
    let v1 = creusot_contracts::vec![1, 5, 3];
    let (list1, mut _seq1) = list_of_vector1(v1.clone());
    assert!(*Node::nth(list1, 0, &_seq1) == 1);
    assert!(*Node::nth(list1, 1, &_seq1) == 5);
    assert!(*Node::nth(list1, 2, &_seq1) == 3);
    let l2 = Node::reverse_in_place(list1, &mut _seq1);
    assert!(*Node::nth(l2, 2, &_seq1) == 1);
    assert!(*Node::nth(l2, 1, &_seq1) == 5);
    assert!(*Node::nth(l2, 0, &_seq1) == 3);

    print!("ok");
}
```