Summer Internship Project
Report

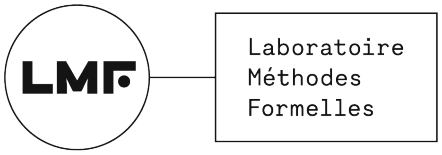
Formal verification of programs with pointers

Submitted by

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

Under the guidance of

Arnaud Goulfouse
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

Chapter 1

Introduction

1.1 The Laboratoire Méthodes Formelles (LMF)

LMF is a joint research centre of University Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, and CentraleSupélec. it's divided to multiple departments interested in various topics such as type systems, topology and quantum computing. 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-Cristophe 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 Le langage de preuve Creusot

1.2.1 Introduction

Creusot is a formal verification language used to verify Rust code. It checks the safety of code against compile-time errors and runtime panics, integer overflows, and, more importantly, the logical correctness of the code, ensuring it adheres to its specifications.

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 features.

Below is a simple example of Creusot code that verifies the correctness of the `SUM_FIRST_N` function.

```
1  extern crate creusot_contracts;
2  use creusot_contracts::*;
3
4  #[requires(n@ * (n@ + 1) / 2 < u32::MAX@)]
5  #[ensures(result@ == n@ * (n@ + 1) / 2)]
6  pub fn sum_first_n(n: u32) -> u32 {
7      let mut sum = 0;
8      #[invariant(sum@ * 2 == produced.len() * (produced.len() + 1))]
9      for i in 1..=n {
10         sum += i;
11     }
12     sum
13 }
```

Figure 1.1: Creusot verification example: sum of first n natural numbers

Explanation:

The function SUM_FIRST_N shown above computes the sum of the first `n` natural numbers. The expressions highlighted in yellow 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 unlimited, 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. 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 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. The subtlety lies in the fact that ghost code is separate from the original code. Therefore, it can be safely erased at compile time, allowing only the original code and the corresponding logical formulas to be executed. This results in faster and safer proof verification. Below are some examples of how to write ghost code in CREUSOT.

Chapter 2

State of art

2.1 Reynolds article

Reynolds' article introduced a new concept in formal program verification: *Separation Logic*, an extension of Hoare 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 Logic, such as concurrency, memory management, and aliasing.

Reynolds is particularly interested in the `in_place_reversal`† algorithm. He frequently discusses it in his research, 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. It's better if the logical model is inductive, because provers generally work better with inductive data structures, for instance, the intuitive logical modeling of lists are sequences, therefore we can write the following predicate to represent a list:

$$\texttt{list}\ \epsilon\ \texttt{i} \stackrel{\text{def}}{=} \texttt{i = nil} \tag{2.1}$$

$$\texttt{list}\ (a.\alpha)\ \texttt{i} \stackrel{\text{def}}{=} \exists\ \texttt{j},\ \texttt{list}\ \alpha\ \texttt{j} \wedge\ \texttt{i} \hookrightarrow \texttt{a} \tag{2.2}$$

$\texttt{i} \hookrightarrow \texttt{a}$ means `i` points to `a`

We can generalize the definition of a list to list segments by passing a tuple of pointers instead of a single one. In this case, the first pointer represents the head of the sub-list, and the second pointer represents the queue of the sub-list.
One of key uses of separation logic can be show in case if we prohibit using `in_place_reversal`† on shared data structures, in this context, given the following precondition for in-place reversal: $\texttt{list p}\ \alpha$, the post-condition $\texttt{list p}\ \overline{\alpha}$ is not sufficient to ensure the correctness of the algorithm. consider the example in the figure below.

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

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

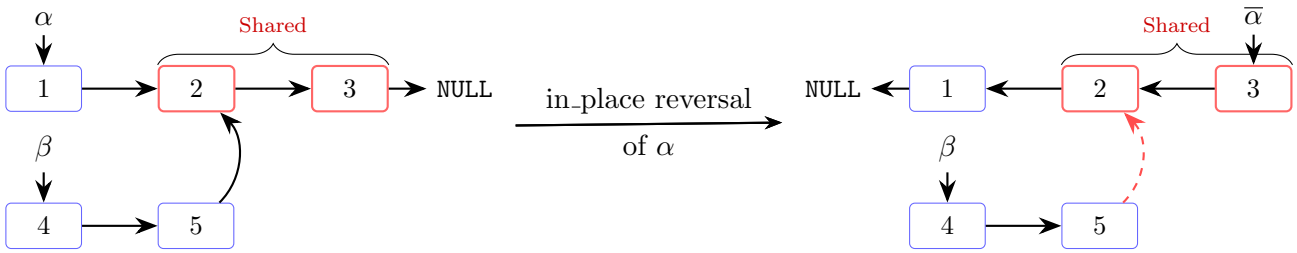


Figure 2.1: in_place reversal on shared lists

where `conflicting (p: Pointer) (seq: Sequence) (p': Pointer) (seq': Sequence)` is a predicate that returns `True` if the two provided lists share any nodes. It is defined as follows:

$$\texttt{conflicting p emp p'}\ \alpha' \stackrel{\text{def}}{=} (\texttt{p} = \texttt{nil}) \tag{2.3}$$

$$\texttt{conflicting p}\ \alpha\ \texttt{p' emp} \stackrel{\text{def}}{=} (\texttt{p'} = \texttt{nil}) \tag{2.4}$$

$$\begin{aligned}\texttt{conflicting p}\ (a.\alpha)\ \texttt{p'}\ (a'.\alpha') \stackrel{\text{def}}{=}\ & (\texttt{p} = \texttt{p'}) \vee \\ & \texttt{conflicting p}\ \alpha\ \texttt{[p'+1]}\ \alpha' \vee \\ & \texttt{conflicting [p+1]}\ \alpha\ \texttt{p'}\ \alpha'\end{aligned} \tag{2.5}$$

Here, `[e]` denotes the content of the address `e`.
We can clearly notice that the precondition, but also the invariant, and the post-condition become extremely complicated, even more, when the program runs in a concurrent context. This is where separation logic proves invaluable. By leveraging heap separation, the specifications become significantly clearer and simpler.

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.

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

Figure 2.2: `PtrOwn<T>::disjoint_lemma` interface

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 were logical, it would not hold, because we quantify universally over `own1` and `own2`; 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).

There are, however, more specialized tools designed specifically for reasoning with Separation Logic. One such tool is Viper, a Rust verifier that supports Separation Logic. It 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 illustration below clarifies the relationships between these languages and their connection to Separation Logic.

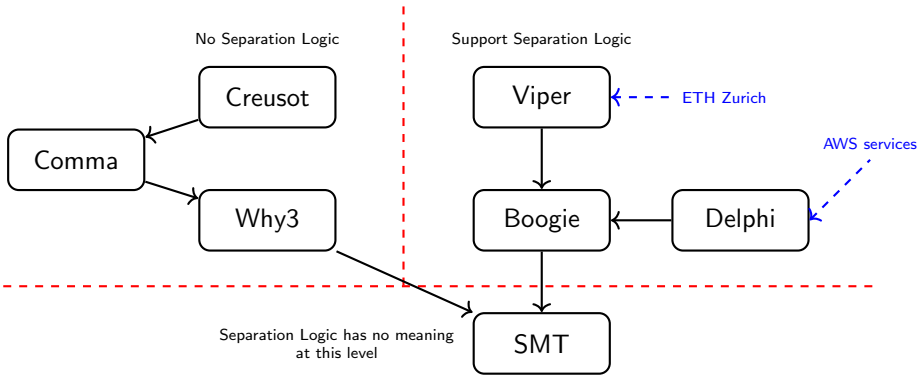


Figure 2.3: Illustration of the relationship between verification tools and intermediate languages.

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.

Chapter 3

Problem definition and proposed solution

3.1 Problem definition

3.2 Proposed solution

3.2.1 `PtrOwn` and `RawPtr`

Formally called linear algebraic types, they are 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 as follows:

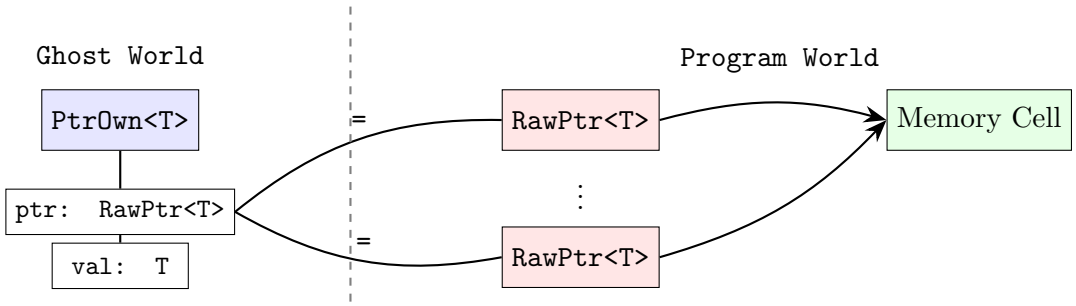


Figure 3.1: Internal structure of `PtrOwn<T>`.

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:

```
1 struct Node<T> {
2     elem: T,
3     pub next: RawPtr<Node<T>>,
4 }
```

3.2.3 Predicates

List Predicate

code†
`list` predicate 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 progam world, and that the list ends with `nil`.

$$\texttt{list p nil} \overset{\text{def}}{=} \texttt{p = null\_ptr()} \tag{3.1}$$

$$\texttt{list p (a.}\alpha\texttt{)} \overset{\text{def}}{=} \texttt{p = a.ptr()} \wedge \texttt{list [p+1]} \ \alpha \tag{3.2}$$

inverse Predicate

code†
the predicate inverse takes two sequences of `PtrOwn` objects and tells if the elements of the first one are set in reverse order in the second one, on can notice that we could have written the predicated in a simpler way using the built-in function `rev`, but it is not the case, before explaining the diffrence, we show a figure of a representation of `Seq<PtrOwn<Node<T>>>` and it's corresponding list in the program world.

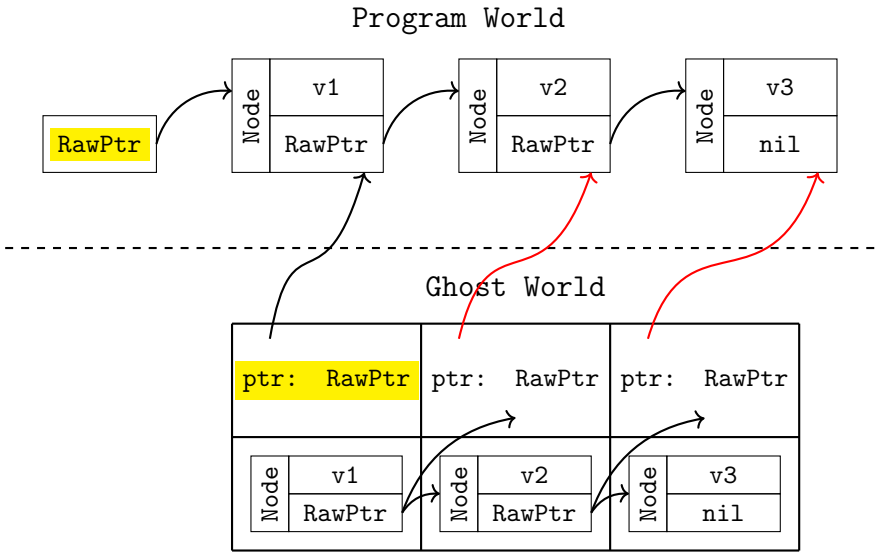


Figure 3.2: Program representation: Raw pointer-based list

Chapter 4

Conclusion and prespectives

Chapter 5

Appendix

5.1 in_place_reversal algorithm

alalal

5.2 Code

```
#![allow(dead_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 {
        //On n'aura pas vraiment besoin de l puisque on suppose que perm_seq ne peut pas etre
        //des permissions de l
        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)]
//
//#[ensures(^seq == seq.push_front())]
pub fn cons(e: T, l: RawPtr<Self>, seq: &mut Ghost<Seq<PtrOwn<Node<T>>>>) -> RawPtr<Self>
    // let ee = snapshot!(e);
    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 {
    //requires nth >= 0
    let mut i = 0;
    //let mut seq_taililng = snapshot!(**seq);
    proof_assert!(**seq == seq.subsequence(0, seq.len()));
    #[invariant(0 <= i@ && i@ <= nth@)]
    #[invariant(Self::list(p, seq.subsequence(i@, seq.len())))]
    loop {
        //je ne comprends pas pourquoi il n'arirve pas à prouver les deux assertions en b
        // hypothèse: snapshot! n'est bon pour tracker la valeur de seq meme si c'est une
        //proof_assert!(seq_taililng[0] == seq[i@]);
        //proof_assert!(Self::list(p, *seq_taililng));
        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.
        i += 1;
        //seq_taililng = snapshot!((*seq_taililng).tail());
    }
}

#[predicate]
```

```
pub fn contains(s: Seq<PtrOwn<Node<T>>>, x: T) -> bool
where
    T: Sized, // TODO: don't require this (problem: uses index)
{
    pearlite! { exists<i: Int> 0 <= i &&  i < s.len() && s[i].val().elem == x }
}

#[predicate]
pub fn reverse(seq: Seq<PtrOwn<Node<T>>>, other: Seq<PtrOwn<Node<T>>>, lb: Int, lh: Int)
where
    T: Sized, // TODO: don't require this (problem: uses index)
{
    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() && Self::reverse(**seq, *^seq, 0, seq.len()))]
//stabilité par inversion
//#[ensures(forall<i: Int> 0 <= i && i < seq.len() ==> exists<j: Int> 0 <= j && j < (^seq

// #[ensures(seq.len() == (^seq).len())]
// #[ensures(forall<e: T> Self::contains(**seq, e) ==> Self::contains(*^seq, e))]
pub fn reverse_in_place(
    mut p: RawPtr<Self>,
    seq: &mut Ghost<Seq<PtrOwn<Node<T>>>>,
) -> RawPtr<Self> {
    //requires p n'est pas un lasso
    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::reverse(_seq0.subsequence(0, reverted_seq.len()), *reverted_seq, 0,
    //Question!!!!!!!!! Ican either keep this invariant which proves everything, or remo
    //which makes the code cleaner.
    #[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() {
        //snapshot!(Self::disjunciton_lemma(&*_seq0, &**seq, &*reverted_seq));
        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 se
            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
            //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 th
            //a7070
            proof_assert!((*_sloop_exit).tail() == (*_sloop_entry).tail());

            //this should be enough to prove #[invariant(Self::list (p, **seq))], whith using
            //proof_assert!(Self::list(p, (*snap2).tail()));
            //a1313
            proof_assert!(Self::list(p, (*_sloop_exit).tail()));
            // ==> invariant #1 checks for iteration n+1
        }
        //snapshot!(Self::disjunciton_lemma(&*_seq0, &**seq, &*reverted_seq));
        //Pour montrer ensures#1 (ensures(seq.len() == (^seq).len() && Self::reverse(**seq, *
        //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.
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().e
    #[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
    #[invariant(inv(seq))]
    loop {
        //let vec1 = snapshot!(vec);
        //let seq1 = snapshot!(seq);
```

```
        //proof_assert!(vec1.view().len() > 0 ==> vec1.view()[vec1.view().len()-1] == vec0.vi

        if let Some(v) = vec.pop() {
            //let vv = snapshot!(v);
            l = Node::cons(v, l, &mut seq);
            //proof_assert!(vec1.view().len() == vec.view().len() + 1);
            //proof_assert!(seq[0].val().elem == *vv && seq.tail() == **seq1);
            //proof_assert!(forall<i: Int>
            //  vec1.view().len() <= i && i < vec0.view().len() ==> seq1[i - vec1.view().len(
            //proof_assert!(vec1.view() == vec.view().push_back(*vv));
            //proof_assert!(vec1.view()[vec.view().len()] == vec0.view()[vec.view().len()]);
            //proof_assert!(seq[0].val().elem == vec0.view()[vec.view().len()]);
        } else {
            break;
        }
    }
    (l, seq)
}

// impl<T> Drop for RawPtr<Node<T>> {
//     fn drop(&mut self) {
//         while !self.is_null() {
//             unsafe {
//                 let next = self.next;
//                 drop(p);
//                 p = next;
//             }
//         }
//     }
// }
```