Creusot: a Foundry for the Deductive Verification of Rust Programs

Xavier Denis, Jacques-Henri Jourdan, Claude Marché

Creusot: a Foundry for the Deductive Verification of Rust Programs

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché

Université Paris-Saclay, CNRS, ENS Paris-Saclay, INRIA, Laboratoire Méthodes Formelles, F-91405 Gif-sur-Yvette

Abstract. Rust is a fairly recent programming language for system programming, bringing static guarantees of memory safety through a strict *ownership* policy. The strong guarantees brought by this feature opens promising progress for *deductive verification*, which aims at proving the conformity of Rust code with respect to a specification of its intended behavior. We present the foundations of Creusot, a tool for the formal specification and deductive verification of Rust code. A first originality comes from Creusot's specification language, which features a notion of *prophecy* to reason about memory mutation, working in harmony with Rust's ownership system. A second originality is how Creusot builds upon Rust *trait* system to provide several advanced abstraction features.

Keywords: Rust programming language · Deductive program verification · Aliasing and Ownership · Prophecies · Traits.

1 Introduction

Critical services like transportation, energy or medicine are nowadays controlled by software, and thus verifying the correctness of such software is highly important. However, *systems software* is often written in low-level, pointer-manipulating languages such as C/C++ which make verification challenging. The pitfalls and traps of C-family languages are well-known; a common thread among them is the unrestricted usage of *mutable pointers*, which allow *aliasing*. When two pointers are aliased, a write through one will silently change the value pointed by the other, wreaking havoc on the programmer and verification tool's understanding of the program. Much effort has been spent trying to control and reason about aliasing. Specialized logics like *separation logic* or *dynamic frames* [20] give a vocabulary to express these challenges. On the other side, language features like the *region typing* of Why3 [9] or the *ownership typing* of the Rust programming language prevent mutable aliasing altogether. Indeed, Rust promises the performance and flexibility of C with none of the memory safety issues. To make good on this promise, its ownership type system guarantees that pointers are always valid and that mutable ones are unique. The ownership discipline of Rust is enforced by a static analysis called *borrow checking*, which infers the *lifetime* of every borrow (temporary pointer), and ensures that, when mutating, only one pointer can be used to access the data, which is essential for

memory safety. Once a variable is mutably borrowed, a second borrow cannot be created until the lifetime expires. Alternatively, immutable borrows can be duplicated, as one cannot modify the memory they point to. This combination of features, memory safety and low-level control, has led to Rust's exploding popularity. As Rust finds usage in key systems like Firefox and the Linux kernel, it becomes important to move beyond the safety guarantees of the language.

In 2020, Matsushita *et al.* [16] proposed a notion of *prophecies* to reason about the functional behavior of mutable borrows of Rust. Roughly speaking, a prophecy denotes the future value a borrow will have when its lifetime ends. Matsushita *et al.* developed a proof-of-concept tool RustHorn translating Rust code to *Constrained Horn Clauses*, making it possible to check the validity of code assertions using automated solvers. Our tool Creusot[1] allows for auto-active deductive verification of Rust code. Creusot uses a prophetic translation in the lineage of RustHorn, but aims to verify real-world programs, pushing the size, scope and features of programs far beyond RustHorn. Unlike RustHorn, Creusot has a specification language, Pearlite[2], which allows users to write function contracts and loop invariants, where logic formulas can make use of a novel operator ^ (pronounced *final*) to denote prophecies. Creusot has the ambition of going beyond a proof-of-concept verification tool. To support a large subset of Rust, including its standard library, it is mandatory to support the notion of *traits* which are a key mechanism for *abstraction* in Rust. Creusot not only supports the verification of Rust code involving traits, but it also builds upon the trait system to provide important features: first, the concept of *resolution* of prophecies is expressed as a `Resolve` trait; second, logical abstraction of concrete data is provided through a `Model` trait which is used pervasively in our case studies.

1.1 Example: a Polymorphic Sorting Function

As a motivating example, let's consider a simple *generic* sorting routine. We use the "Gnome Sort" algorithm, using a single loop that *swaps* out of order elements successively. We provide an implementation in Fig. 1. To verify this we face two primary challenges. The first is *genericity*: to compare values of a generic type in Rust we need to use the *trait* `Ord`. The traits of Rust are like the *typeclasses* of Haskell: here they allow us to constrain `T` to have an *ordering relation*. In Creusot, we can add the necessary logical specifications to the `Ord` trait, formalizing its documentation [21] stating that it *must* implement a total order. The second challenge is the need to handle the library `Vec` for vectors, providing a *safe* interface to resizeable arrays, despite being implemented using *unsafe code*. Creusot does not permit the verification of unsafe code but allows the specification of *safe abstractions* like `Vec`: we choose a *model* for such types, representing vectors as *mathematical sequences*.

[1] *Le Creusot* is an industrial town in the eastern France, whose economy is dominated by metallurgical companies, *cf.* https://en.wikipedia.org/wiki/Le_Creusot

[2] Pearlite is a structure occurring in common grades of steels, cf https://en.wikipedia.org/wiki/Pearlite

```
1  #[ensures(sorted_range(@^v, 0, (@^v).len()))]
2  #[ensures((@^v).permutation_of(@*v))]
3  fn gnome_sort<T: Ord>(v: &mut Vec<T>) {
4      let old_v = Ghost::record(&v);
5      let mut i = 0;
6      #[invariant(sorted_range(@*v, 0, @i))]
7      #[invariant((@*v).permutation_of(@*@old_v))]
8      while i < v.len() {
9          if i == 0 || v[i - 1] <= v[i] {
10             i += 1;
11         } else {
12             v.swap(i - 1, i);
13             i -= 1;
14         }
15     }
16 }
```

Fig. 1. Gnome Sort and its specification.

Now, we can finally specify what it means to *sort* a vector. We do this using two `ensures` clauses which establish the postconditions of the function. The first uses a helper *logical predicate* `sorted_range` to define what it means for a sequence to be sorted according to a generic order on `T`.

```
#[predicate]
fn sorted_range<T: Ord>(s: Seq<T>, l: Int, u: Int) -> bool {
    pearlite! { forall<i: Int, j : Int>
      l <= i && i < j && j < u ==> s[i] <= s[j] }
}
```

This definition is written using PEARLITE, the specification language of CREUSOT. While PEARLITE has a Rust inspired syntax, it adds several constructs like quantification `forall<..>`, or implication `==>`. Using these we can say a sequence is sorted if for any ordered pair of indices the values at those indices respect the order on `T`. On line 1 of Fig. 1, we use this definition with two more PEARLITE operators: the model operator (`@`) (syntactic sugar for the `Model` trait), and the prophetic operator *final* (`^`). The final operator provides access to the value of a mutable borrow at the end of its lifetime; here we use it to talk about the value the vector pointed by `v` has after the function call. The second postcondition on line 2 makes use of a similar helper predicate `permutation_of` to require that the final value of `v` is a permutation of its initial value. To prove these postconditions we provide two *loop invariants* (lines 6-7). The first states that the segment of the vector before `i` is sorted, while the second states that the vector at each iteration is a permutation of the input. To state this invariant about permutations we make use of the `Ghost` type, to record a *ghost value* which does not exist at runtime but does during proof, allowing us to remember the original value of `v`. The annotated program is fed through CREUSOT, which

translates it to Why3. In turn, Why3 generates and discharges the verification conditions in under 2 seconds using automated provers including Z3 [19], CVC4 [2] and Alt-Ergo [6].

1.2 Contributions

Our contributions, and the structure of this paper, are summarized as follows. In §2 we give an introduction through the Pearlite specification language and the verification process of Creusot. This section illustrates how prophecies are integrated in Pearlite. In §3, we explain how Creusot translate Rust functions into Why3 functions, this section explains how we generate the verification conditions for mutable borrows. In §4 we first present in deeper details how Rust traits are interpreted by Creusot, and in a second step we discuss the specific Creusot traits used in borrow resolution and in data abstraction. In §5 we present an overview of the implementation of Creusot and an experimental evaluation of the tool on a set of benchmarks. We discuss related work in §6. Notice that due to space limitations, we focus here on the original features of Creusot and refer to an extended research report [8] for more details.

2 Specifying and Proving Programs using Prophecies

To formally specify the intended behavior of programs, one can use so-called *Behavioral Interface Specification Languages* [10], such as JML for Java [5], ACSL for C [3] or Spark for Ada [17]. Creusot follows this tradition by introducing the Pearlite specification language, where Rust functions are given *contracts* which specify *pre-* and *postconditions*: logic formulas which respectively hold at the entrance and exit of a function call.

Traditionally, specification languages introduce specific contract clauses to specify the behavior of pointers such as the `assignable` or `assigns` in ACSL and JML. Pearlite has no equivalent clause. Instead specifications can refer not only to the value of a borrow at function entry but also to its value at the end of its lifetime: this is the notion of *prophecies* that we detail in §2.2.

2.1 Background Logic

The background logic of Pearlite is a classical, first-order, multi-sorted logic. Each Rust type corresponds to a sort in this logic. The logic connectives denoted by `&&`, `||` and `!` mirror their Rust counterparts, but Pearlite also introduces `==>` for implication, and the quantifiers `forall<v:t> formula` and `exists<v:t> formula`. Atomic predicates can be built using custom *logic functions and predicates*, constant literals, variables and built-in symbols, a central case being the logical equality denoted by a tripled equal sign (`===`) and defined on any sort. This logical equality is the symbol interpreted to the set-theoretic equality in a set-based semantics. This distinguishes it from the *program equality* of Rust, `==`, which is sugar for `PartialEq::eq`. Finally, Pearlite has support

for logical sorts which do not exist in Rust, like `Int`, the unbounded mathematical integers, or like `Seq<T>` the generic sort of mathematical sequences. A syntactically valid formula is thus for example:

```
forall<x: Int> x >= 0 ==> exists<y: Int> y >= 0 && x * x === x + 2 * y
```

See [8] for more technical details on the background logic. PEARLITE formulas are type-checked by the front-end of the Rust compiler, but they are not borrow-checked. Hence values can be used in logic functions or predicates, even if the Rust ownership rules would forbid copying them.

A useful feature of PEARLITE is the introduction of user lemmas using the so-called *lemma function* construction. To achieve this, one provides a contract to a logical function returning `()`. By proving the contract valid, one obtains a lemma stating that for all values of arguments, the preconditions imply the postconditions. This construction is even able to prove lemmas by induction. Here is an example (detailed in [8]):

```
#[logic]
#[requires(x >= 0)]
#[variant(x)]
#[ensures(sum_of_odd(x) === sqr(x))]
fn sum_of_odd_is_sqr(x:Int) { if x > 0 { sum_of_odd_is_sqr(x-1) } }
```

This code is automatically proved conforming to its contract. Any call to this function would then add the hypothesis $\forall x, x \geq 0 \Rightarrow \texttt{sum\_of\_odd}(x) = x^2$ in the current proof context.

2.2 Borrows and Prophecies

We illustrate the use of prophecies to specify mutable borrows in Fig. 2. The function `pick` returns, depending on its first boolean parameter, either its second or third argument. We wish to show that the client `pick_test` returns `42`. For that purpose, `pick` must be given an appropriate contract: namely the postcondition of lines 1 and 2. The first part of the postcondition states that the result of `pick` (denoted by the identifier `result`) is either `x` or `y`, depending on the value of `t`. Importantly, when we say `result === x` (*i.e.,* when `t === true`), we are stating that the *borrow* `result`, which is a pointer, is equal to the *borrow* `x`, not merely the values being pointed. In particular, this captures that writing through the returned pointer affect the variable pointed to by `x`. In `pick_test`, this entails that the final value of `a` is 6. The second part of the post-condition is needed to state that the other borrow parameter (`y` when `t === true`) is released so the caller knows the value it points to can no longer change until the lifetime `'a` ends. This is specified using a *prophecy*. For any mutable borrow `b`, one can write `^b` in PEARLITE to denote its *final value*, the value the variable it points to will have when the lifetime expires. Prophecies act like a bridge between the lender and borrower, when a borrow is released we recover information which allows us to update the value of the lender. Releasing a mutable borrow is equivalent to stating that the current value of the borrow, `*b`, equals its final value, `^b`. We

```
1  #[ensures(if t { result === x && ^y === *y }
2            else { result === y && ^x === *x })]
3  fn pick<'a>(t: bool, x: &'a mut i32, y: &'a mut i32) -> &'a mut i32 {
4    if t { x } else { y }
5  }
6
7  #[ensures(result === 42)]
8  fn pick_test() {
9      let (mut a, mut b) = (4, 7);
10     let x = pick(true, &mut a, &mut b);
11     *x += 2; return a * b;
12 }
```

Fig. 2. A toy example illustrating prophecies in the specification language.

refer to this process as the *resolution* of the borrow. Thanks to the resolution, we can prove the postcondition of `pick` and deduce in `pick_test` that the value of `b` at line 11 is its original value `7`. The final value of a borrow is a logical artifact: it is not necessarily known at runtime when the specification mentions it, but one can prove [16,15] that it is sound to *prophesize* it in the logic. Note that the first equality (*i.e.,* `result === x` when `t === true`) actually implies the equality `^result === ^x`, as we are asserting that `result` and `x` are completely indistinguishable. This explains why the first equality is enough to specify that a mutation through the borrow `result` causes a mutation of the variable pointed to by the argument `x` (when `t === true`). Since they both have the same prophecy, modifications to the memory pointed to by `result` will affect the lender of `x`. The approach is detailed and validated in prior work [16,15].

3 Handling Rust Function Bodies

Creusot translates Rust programs into WhyML, the programming language of Why3. Rather than starting from source level Rust, translation begins from the Mid-Level Intermediate Representation (MIR) of the Rust compiler. MIR is a key language in the compilation of Rust and is the final result of desugaring and type checking Rust code. Many tools that wish to consume Rust code target MIR. There are rich APIs to access, extract and manipulate MIR code. Furthermore, MIR is created *after* type checking and is the representation on which Rust's flagship static analysis *borrow checking* is formulated. MIR is also the language modeled in RustHornBelt [15] to prove the correctness of prophetic translation. Thus, we want to design our translation from MIR to Why3 so the generated verification conditions are as close as possible to those proved sound in RustHornBelt. MIR programs are unstructured: they are represented as a control-flow graph (CFG) whose nodes are basic blocks composed of atomic instructions (borrowing, arithmetic, dereferencing, etc.) each terminated by a function call, a goto, a switch, an abort or a return. To verify a MIR program, we

find ourselves needing to calculate the *weakest-precondition (WP) of a function represented as a CFG*. We achieve this using a dedicated Why3 input front-end called MLCFG, that reconstructs a structured WhyML code from a control flow graph, and then use Why3's carefully designed WP computation algorithms.

3.1 Translating Owned Pointers

The ownership discipline of Rust makes a simple translation of (mutable) owned pointers possible. Consider the case of a local variable `x` containing an owned pointer to the heap (*i.e.,* of type `Box<T>` for some type `T`). Then, we know that mutating memory through `x` can only be observed by a read using variable `x`. Therefore, if `t` is the translation of type `T`, then we can translate type `Box<T>` to type `t` as well. An assignment through the pointer `*x = e` is simply translated to the assignment: `x = e'`, where e' is the translation of e.

3.2 Translating Borrows to Prophecies in WHY3

Just like in RUSTHORNBELT, borrows are translated into pairs of a current value and a final value. Hence, we introduce a new polymorphic record type in WHY3:

```
type borrow 'a = { current : 'a ; final : 'a }
```

A Rust variable of type `&mut T` is translated by CREUSOT as an object of type `borrow t`, where `t` is the translation of `T`. The PEARLITE notations `*x` and `^x` are translated into `x.current` and `x.final`.

Using the pattern of prophecies for encoding mutable borrows into a functional language poses a difficulty: when we create a borrow, we obtain a new *prophetized* value which comes out of thin air. We won't know anything concrete about this prophecy until the borrow is *resolved* at the moment it is dropped, and then at the lifetime's end the lender will have the value of this same prophecy. In between these points there can be arbitrary control flow, ownership changes, or *reborrowing*. We may no longer know who the lender of a borrow was at its lifetime's end, and therefore have no way to propagate the prophecy to the lender. Our solution may be surprising: we update the lender with the prophecy at the moment of the borrow's creation, foreseeing all the mutations that will occur. This is valid because the value of the lender cannot be observed before the lifetime's end. As a result, the creation of a borrow

```
let y : &mut T = &mut x;
```

is translated into

```
y : borrow t <- { current = x ; final = any t };
x : t <- y.final;
```

where `any t` is the WhyML non-deterministic construct which returns an arbitrary value of type `t`. It encodes the fact that the final value is not yet known, it is thus *prophetized*. The second line gives to x the final future value of `y`.

An important other case occurs when a borrow is *dropped*, where we insert a *resolution statement*:

```
1  val pick (t: bool) (a: borrow int32) (b: borrow int32) : borrow int32
2    ensures { if t then result = a /\ b.final = b.current
3      else result = b /\ a.final = a.current }
4
5  let cfg pick_test () =
6    ensures { result = 42 }
7    var a, b : int; var bor_a, bor_b, x : borrow int;
8    { a <- 4; a <- 7;
9    (* let x = pick(true, &mut a, &mut b); *)
10   bor_a <- { current = a ; final = any int32 }; a <- bor_a.final;
11   bor_b <- { current = b ; final = any int32 }; b <- bor_b.final;
12   x <- pick true bor_a bor_b;
13   (* *x += 2; *)
14   x <- { current = x.current + 2; final = x.final };
15   assume { x.final = x.current };
16   (* return a * b *)
17   return a * b }
```

Fig. 3. Simplified translation of the projection pick example in MLCFG.

```
assume { y.final = y.current };
```

The contents of a WhyML `assume` clause states a fact as a trusted hypothesis for subsequent statements. Resolution corresponds to the fact that at this point the value pointed to by the borrow will not change, and therefore its prophecy has been fulfilled.

The simplified translation of the `pick` example, from Fig. 2, is given in Fig. 3. See [8] for more details. The postcondition of `pick_test` and `pick` are proven using Why3 and SMT solvers.

4 Support for Rust Traits

Rust makes heavy use of a *trait system* to implement abstractions. Like typeclasses in Haskell, traits allow functions, types or constants to be associated to specific types, and can automatically select the correct instance at each call-site. The trait system enables ecosystem-wide modularity and many common operations are expressed using traits, such as equality in the `PartialEq` and `Eq` traits, order relations in `PartialOrd` and `Ord`, and accessing collections in the `Index` and `IndexMut` traits. Supporting Rust's trait system is necessary for a verification tool, they manifest themselves in even the most basic programs, like Fig. 1. In this section, we explain how traits can be used in Creusot to modularly verify programs. But Creusot not only verifies programs using traits, it also *uses* traits for some of its core features: the `Resolve` and `Model` traits.

```
trait Ord {
    #[logic] fn cmp_log(self, o: Self) -> Ordering;

    #[ensures(result === self.cmp_log(*o))]
    fn cmp(&self, o: &Self) -> Ordering;

    #[law]
    #[requires(a.cmp_log(*b) === o && b.cmp_log(*c) === o)]
    #[ensures(a.cmp_log(*c) === o)]
    fn trans(a: &Self, b: &Self, c: &Self, o: Ordering);
    ...
}
```

Fig. 4. A simplified `Ord` trait with specifications

4.1 Specifying Trait Behavior

The trait `PartialOrd` implements a *heterogenous partial order*: instances must provide implementations for all of `lt`, `gt`, `le`, `ge`, and `partial_cmp`. Additionally, the official documentation [21] requires that these definitions are mutually compatible, for example: if `a.le(b)` then `a.lt(b) || a.eq(b)`. This is an example of a *law* for `PartialOrd`. In CREUSOT, laws can be included in traits using the `#[law]` annotation and written in the style of *lemma functions* (see §2.1). A particularity of trait laws is their *auto-loading*: whenever we use *any* associated item of a trait or implementation, we will bring into scope any laws from that trait.

Traits can be arranged into a hierarchy, with *sub-traits* refining or expanding upon their super-traits. The *sub-trait* `Ord` strengthens the specification of `PartialOrd`, requiring the order to be *total* and *homogeneous*. The laws of `Ord` constrain the behavior of functions defined in the super-trait `PartialOrd`. In Fig. 4 we present a simplified version of our specifications for `Ord` (see [8] for more details). We require a definition of `cmp_log` and a proof of transitivity. Each time a user makes use of a comparison operation, CREUSOT will load the laws of `Ord`, allowing us to leverage the transitivity of our order.

Every implementation of a trait for a specific type must *refine* the contract of the trait. It must weaken preconditions and strengthen postconditions. This possibility of refinement allows implementations to provide stronger contracts which leverage specific knowledge of the type the trait is being implemented for. Whenever a trait method is used, CREUSOT will use the most specific contract possible. Performing the translation to Why3 of all such different usage of traits is indeed highly non-trivial: it relies on algorithms for construction of specific dependency graphs which are detailed in our research report [8].

4.2 The `Resolve` Trait

We use traits to generalize the notion of *resolution* discussed in §3.2, as follows.

```
#[trusted] trait Resolve {
    #[predicate] fn resolve(self) -> bool;
}
```

Much like how Rust's `Drop` trait allows types to customize their program destructors, we use the `Resolve` trait to define the knowledge gained from resolving a specific type. Following discussion of §3.2, the `Resolve` trait is given the following implementation for mutable borrows:

```
unsafe impl<T> Resolve for &mut T {
    #[predicate]
    fn resolve(self) -> bool { pearlite! { ^self === *self } }
}
```

so indeed, when a mutable borrow `r` is resolved, instead of assuming `^r === *r`, Creusot will assume the equivalent assertion `r.resolve()`.

Because `Resolve` represents information that is *assumed* about a type, an incorrect implementation can introduce unsoundness to Creusot, for this reason we mark the trait as `#[trusted]`, and require all implementations to do the same. This mirrors the notion of `unsafe trait` in Rust for those traits where a malicious implementation could introduce undefined behavior in safe code.

The `Resolve` trait makes it possible to generalize the resolution mechanism to data structures containing mutable borrows, like vectors of borrows, pairs of pairs of borrows, etc. For example, when we resolve a pair `p` of mutable borrows, we wish to learn that both components of `p` are resolved, that is, `*p.0 === ^p.0 && *p.1 === ^p.1`. To achieve this goal, we give the following implementation of the `Resolve` trait for pairs:

```
unsafe impl<T1: Resolve, T2: Resolve> Resolve for (T1, T2) {
    #[predicate]
    fn resolve(self) -> bool {
        pearlite! { self.0.resolve() && self.1.resolve() }
    }
}
```

Then, resolving `x: (&mut T, &mut T)` would expand into the resolution of each component of the pair.

Like `Drop`, we need to be able to resolve a value of *any* type, but we don't have the benefit of being a first-class language feature of Rust. We solve this using a cutting-edge feature of Rust, *specialization*. This allows us to provide a generic implementation for every type `T`, and then provide more specific instances which specialize resolution. In practice, this means users can write `x.resolve()` for any value, and never need to constrain generic parameters to implement `Resolve`.

4.3 Specifying with Models: the `Model` Trait

Traits provide a convenient mechanism for abstracting specifications, just like in programs. When working with complex data structures we wish to treat their specifications in terms of a *model* which abstracts away implementation details.

For example, we may wish to view a `HashMap` as a mathematical map between two types, or a `Vec` as a sequence of values. To do this, we provide a function which shows how to interpret concrete values as members of the *model*. In certain cases (like `Vec` in CREUSOT), we may even take the existence of this function as an axiom. This *design pattern* is common enough that we can capture it in a trait.

```
trait Model {
    type ModelTy;
    #[logic] fn model(self) -> Self::ModelTy;
}
```

Each implementation of the `Model` trait specifies the type of the model and a function to interpret itself as a value of that type. By making this a trait, we can provide convenience instances that improve ergonomics. CREUSOT goes further and provides syntactic sugar for this trait. Rather than using `x.model()`, users can write `@x` where appropriate. Apart from this small sugar, models purely are a *library concern*, CREUSOT as a tool has no specific awareness of them.

5 Experimentation and Evaluation

We evaluated the performance of CREUSOT on a wide range of benchmarks. These benchmarks make heavy use of polymorphism and traits. Additionally, we improved on the benchmarks of other tools by proving additional functional properties. The evaluation shows that CREUSOT's approach scales well, with verification times remaining low even in complex examples. Furthermore, it provides evidence that our prophetic specifications are well-suited and concise.

Implementation Like many other Rust verification tools, CREUSOT is implemented as an extension of the Rust compiler, and integrates easily into standard Rust workflows. The total implementation including the 'verification standard library' of Creusot totals 14k lines of code, published under an LGPL license, available at https://github.com/xldenis/creusot/. During execution CREUSOT translates Rust libraries into MLCFG and outputs the result to a file. The resulting file can then be loaded in WHY3 and verified using either its IDE or command line[3].

Language Support CREUSOT supports a large subset of *safe* Rust, including structs and enums, all forms of borrowing, loops and recursions. As we discussed in this paper, we also support polymorphism and traits, including associated types and functions, and super traits. Furthermore, we extend Rust with both logic functions and predicates, which can be used in the specifications of functions and traits. CREUSOT also allows types like `Vec` to be axiomatized so their safe clients can still be verified.

[3] Note to reviewers: the page at https://www.lri.fr/~xldenis/icfem2022/ provides detailed instructions on how to reproduce the experiments below.

Name	Has generics?	Has traits?	LOC	Spec. LOC	# of VCs	Time (s)	Additional Properties
Inc Some List	✗	✗	25	22	4	0.98	Func. correctness
Inc Max	✗	✗	12	3	2	0.53	Func. correctness
Inc Max Many	✗	✗	13	3	2	0.74	Func. correctness
Binary Search	✓	✓	21	20	31	2.15	Func. correctness
Knapsack 0/1	✓	✗	32	52	81	3.94	—
Knapsack 0/1	✓	✗	32	106	113	5.96	Func. correctness
Knuth Shuffle	✓	✗	9	11	1	0.30	Permutation
100 doors	✗	✗	18	6	3	1.08	—
Heap Sort	✓	✓	30	71	125	14.6	Func. correctness
Selection Sort	✓	✓	15	27	30	2.14	Func. correctness
Gnome Sort	✓	✓	11	17	31	2.06	Func. correctness
Filter Vector	✓	✗	21	39	6	0.98	—
Sparse Array†	✓	✗	47	75	37	4.86	Func. behavior
In place List Rev.	✓	✗	12	10	1	0.55	Func. correctness
All Zero List	✓	✗	11	10	1	0.64	Func. correctness
Swap Pair	✓	✗	9	3	2	0.48	—
HashMap	✓	✓	50	111	71	5.43	Func. correctness

Table 1. Selected results of our evaluation. The column "LOC" indicates the lines of program code (excluding blank lines) we verify. The column "Spec. LOC" measures the lines of specifications (excluding blank) used. "# of VCs" measures the number of verification conditions that are sent as proof tasks to CVC4 or Z3. "Time (s)" measures the time Why3 takes to run the provers. The "Has traits?" measures whether the test case has a function with a generic parameter constrained by a trait. Tests marked with † required a few manual proof steps in Why3 IDE [7].

Evaluation We measure the verification performance for programs translated with Creusot. We adapted and generalized programs from the Prusti [1] benchmark suite, additionally strengthening the verified properties. Other examples were inspired from the Why3 gallery [4], Rosetta Code [18] or RustHorn [16].

Note that Why3 has support for a wide range of manual proof tactics that allow users to setup proof structure before handing off obligations to provers. As these can dramatically help verification, we avoid them in our evaluation and instead apply a standard proof strategy to all examples. Each example is proved using Why3's "Auto Level 2" strategy, a common first step when verifying programs with Why3. One benchmark required a small number of additional manual proof steps, "Sparse Array", to prove a complex lemma about injections between sequences.

Our evaluation was performed using a 2016 Macbook Pro running macOS 11.6 installation with a Intel Core i7-7920HQ CPU and 16 GB of RAM. We relied on a combination of Alt-Ergo 2.4.1, Z3 4.8.17 and CVC4 1.8 as back-ends to Why3.

Discussion The selected results are presented in Table 1, where benchmarks are grouped by origin. The first group come RUSTHORN's evaluation [16, §4.3], where we added specifications of the intended functional behavior. The second group of benchmarks are adapted from PRUSTI's evaluation [1, §7.2]. The third group are novel examples contributed as part of CREUSOT's test suite. "Filter Vector" is a challenging example regarding reasoning on memory separation [12]. "Sparse Array" is an example from the VACID-0 benchmarks [14]. The proof involves a mathematical lemma with a few steps of manual proof [7] before sending the sub-goals to SMT solvers. "In Place List Rev." is the in-place linked-list reversal procedure, classically used as an illustration of reasoning in separation logic. It is remarkable that the Rust code for that can be verified without the need for separation logic.

Our RUSTHORN tests show that we maintain the verification performance of RUSTHORN, as these examples are rapidly verified by our provers. While some manual annotation is required, even for safety, the overhead is low, and mostly consists of stating the properties we wanted to prove in the first place.

The PRUSTI examples listed here are derived from their introductory paper in 2019 [1]. In their paper they provide two versions for their functions, the first proving only safety while the second proves portions of functional correctness.

The difference in verification performance is made evident by the "Knapsack 0/1" example of PRUSTI. This example solves the 0/1-Knapsack problem using the traditional dynamic programming approach. PRUSTI takes over 2 minutes to verify the safety of the problem, whereas our proof of safety passes in approximately 4 seconds. This difference in performance helps us go further, being able to rapidly check proofs allows for faster iteration, which enabled us to extend this example with a complete proof of functional correctness. Our version of the Knapsack Problem with functional correctness takes longer to verify, with the proof passing in approximately 6 seconds.

6 Related Work

RUSTHORN [16] laid the foundations for CREUSOT by developing a prophetic encoding of mutable borrows and applying it to Rust. It translates MIR programs directly to Constrained Horn Clauses where existing dedicated automated solvers can be thrown at the task. CREUSOT on the other hand introduces an intermediate step: we translate first to an intermediate language which is then lowered to first-order logic (FOL) by calculating weakest-preconditions. As a tool RUSTHORN remains a proof-of-concept, it supports a core fragment of Rust: algebraic data types, borrows, simple loops and arithmetic and polymorphism. There is no support for unsafe types like `Vec` or for traits like `Eq`. Moreover, RUSTHORN has no specification language, it is limited to the verification of program assertions, which are by essence limited to executable boolean expressions on program variables, without any way to relate them with an abstract model. It relies entirely on automation to infer both function postconditions and loop invariants, meaning a seemingly small change can cause verification times

to spiral out of control or fail unpredictably. While not an automated verifier, RustHornBelt mechanizes a proof of soundness for prophetic verification of Rust [15], by extending the prior RustBelt proof. The proof shows that the uniqueness and lifetimes of mutable borrows enables prophetizing their final values, placing Creusot's approach on solid theoretical grounds. However, there remains a gap between an implementation like Creusot and the mechanization. In particular, the language of RustHornBelt λRust makes a number of simplifying assumptions when compared to MIR, like boxing function parameters or using a CPS structure for the programs. Furthermore, RustHornBelt establishes the soundness of the final verification conditions directly but Creusot introduces an intermediate step by targeting a functional language.

Prusti [1] is another deductive verifier for Rust, based on the Viper separation logic platform. It does not use a prophetic encoding, instead modeling ownership using *permissions*. Like Creusot, Prusti has a specification language which can be used to give contracts and invariants. Because Prusti has no notion of prophecy, it does not use the *final* operator ($^\wedge$) to specify mutable borrows, instead using *pledges*. A pledge is an assertion that is guaranteed to hold at the time when the borrow expires, which is not necessarily in the body of the function. In contrast, the *final* operator of Creusot brings prophecies as first-class objects in the specification language, to specify the future values of borrows. The semantics of Prusti specifications were designed to preserve the behavior of program assertions when lifted into pure contracts. In particular, arithmetic in Prusti's specifications is machine arithmetic and has to be checked for overflow. Creusot takes a different approach by using a more abstract specifications language (Pearlite), which is usually easier to reason with. A consequence of this difference is that Pearlite logical functions cannot be executed, while Prusti's pure functions can be used in programs. While Prusti's permission system supports the common borrowing patterns of Rust, it struggles with patterns like *reborrowing in a loop* (*e.g.,* "All Zeros List" 5), with data structures *containing borrows* like pairs of mutable borrows 5, or with *nested borrows*. In contrast, Creusot's translation of Rust types using prophecies for mutable borrows is general and *compositional*: we place no restrictions on the usage of mutable borrows or their position within types. Another noticeable difference with Prusti lies in the choice of the underlying logic. Prusti encodes specifications into separation logic and delegates verification to Viper, whereas Creusot encodes them into FOL and delegates verification to SMT solvers via Why3. Prusti chooses to verify Rust's ownership discipline with Viper, while Creusot depends on Rust's borrow checker for that, which means Creusot relies on the soundness of Rust's type system and of its implementation. We believe this difference explains the significant blow-up in verification times: on simple examples verification takes an order of magnitude more time than with Creusot. The simpler underlying logic in Creusot, allows it to benefit from Why3's mature infrastructure to manage a herd of automated provers and a tactic system to provide guidance when they go astray. Both Prusti and Creusot support traits and polymorphism. However, because mutable borrows

need special care in Prusti, a generic Rust function cannot be instantiated with a mutable borrow, which causes no problem in Creusot. Moreover, properties of traits in Prusti are specified using only pre- and postconditions; we use *laws* for specifying such properties, which we find more flexible.

Aeneas [11] is a novel verifier for Rust targeting interactive verification of programs in established proof assistants like F* or Coq. To achieve this they also translate Rust programs to functional programs in a State-Error Monad. Instead of using prophecies they use *backwards functions* to reconstruct the value of a lender has after the borrows expiry. This approach appears to have a deep and close link to prophecies as used by Creusot, instead of using non-determinism to pull the value out of thin air, Aeneas constructs the actual *witness* of this value. The constructive approach that Aeneas takes may very well be better suited to interactive provers which traditionally prefer constructive logics. Aeneas also makes the choice of using so called *extrinsic proofs*, all specification and proof work is done in the prover, with no annotations present in Rust. While this allows them to leverage all the existing tools in the underlying prover, the proof engineer must manually sync these proofs and specifications with the Rust code as it evolves. This attests to the different audiences targeted by the tools, Aeneas seeks to enable the users of existing advanced verification tools to perform more ergonomic verification using their traditional toolkits, while Creusot seeks to bring verification to regular engineers. In terms of language support, Aeneas is currently more limited than Creusot, it has no support for *loops*, *nested borrows* or *traits*.

Beyond the Rust ecosystem, Spark/Ada is a tool suite for deductive verification of Ada programs. For a long-time, it was restricted to a subset of Ada without pointers. Support for pointers was added in 2020 [13], based on an ownership policy similar to Rust's. At the start Spark used a notion of pledges similar to Prusti's, but they have now replaced it with prophecies. Similarly to Creusot, Spark/Ada makes use of the ownership information computed by the compiler to encode specifications and code into a first-order logic, instead of relying on a separation logic.

References

1. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust types for modular specification and verification. Proc. ACM Program. Lang. **3**, 147:1–147:30 (2019). https://doi.org/10.1145/3360573
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Computer Aided Verification (2011). https://doi.org/10.1007/978-3-642-22110-1_14
3. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.16 (2020), https://frama-c.com/html/acsl.html
4. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's verify this with Why3. Int. J. on Software Tools for Technology Transfer **17**(6), 709–727 (2015). https://doi.org/10.1007/s10009-014-0314-5

5. Cok, D.R.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: Formal Integrated Development Environment. vol. 149, pp. 79–92 (2014). https://doi.org/10.4204/EPTCS.149.8
6. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-Ergo 2.2. In: Satisfiability Modulo Theories (2018), https://hal.inria.fr/hal-01960203
7. Dailler, S., Marché, C., Moy, Y.: Lightweight interactive proving inside an automatic program verifier. In: Formal Integrated Development Environment (2018). https://doi.org/10.4204/EPTCS.284.1
8. Denis, X., Jourdan, J.H., Marché, C.: The Creusot environment for the deductive verification of Rust programs. Research Report 9448, Inria Saclay - Île de France (2021), https://hal.inria.fr/hal-03526634
9. Filliâtre, J.C., Gondelman, L., Paskevich, A.: A pragmatic type system for deductive verification. Research report, Université Paris Sud (2016), https://hal.archives-ouvertes.fr/hal-01256434v3
10. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.: Behavioral interface specification languages. ACM Comput. Surv. **44**(3) (2012). https://doi.org/10.1145/2187671.2187678
11. Ho, S., Protzenko, J.: Aeneas: Rust verification by functional translation (2022). https://doi.org/10.48550/ARXIV.2206.07185
12. Hubert, T., Marché, C.: Separation analysis for deductive verification. In: Heap Analysis and Verification. pp. 81–93 (2007), https://hal.inria.fr/hal-03630177
13. Jaloyan, G.A., Dross, C., Maalej, M., Moy, Y., Paskevich, A.: Verification of programs with pointers in SPARK. In: Formal Methods and Software Engineering. pp. 55–72 (2020). https://doi.org/10.1007/978-3-030-63406-3_4
14. Leino, K.R.M., Moskal, M.: VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In: Verified Software, Tools, Techniques and Experiments (2010)
15. Matsushita, Y., Denis, X., Jacques-Henri, J., Dreyer, D.: RustHornBelt: A semantic foundation for functional verification of Rust programs with unsafe code. In: Programming Language Design and Implementation (2022). https://doi.org/10.1145/3519939.3523704
16. Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based verification for Rust programs. ACM Trans. Progr. Lang. Syst. **43**(4), 15:1–15:54 (2021). https://doi.org/10.1145/3462205
17. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press (2015)
18. Mol, M., other contributors: The Rosetta Code chrestomathy of programs, http://rosettacode.org
19. de Moura, L., Bjørner, N.: Z3, an efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
20. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: ECOOP 2009 — Object-Oriented Programming. pp. 148–172 (2009). https://doi.org/10.1007/978-3-642-03013-0_8
21. The Rust Community: The `std::cmp::Ord` trait of Rust, https://doc.rust-lang.org/std/cmp/trait.Ord.html