The CREUSOT Environment for the Deductive Verification of Rust Programs

Xavier Denis, Jacques-Henri Jourdan, Claude Marché

The CREUSOT Environment for the Deductive Verification of Rust Programs

Xavier Denis, Jacques-Henri Jourdan, Claude Marché

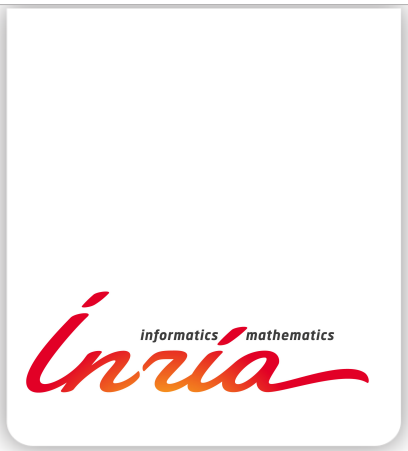
RESEARCH REPORT

N° 9448

December 2021

Project-Team Toccata

ISSN 0249-6399 ISRN INRIA/RR--9448--FR+ENG



The CREUSOT Environment for the Deductive Verification of Rust Programs

Xavier Denis*, Jacques-Henri Jourdan*, Claude Marché*

Project-Team Toccata

Research Report n° 9448 — December 2021 — 28 pages

Abstract: Rust is a fairly recent programming language for system programming, bringing static guarantees of memory safety through a strong *ownership* policy. This feature opens promising advances for *deductive verification* of Rust code, which aims at proving the conformity of some code with respect to a specification of its intended behavior. In this report we present CREUSOT, a tool for the formal specification and deductive verification of Rust programs.
There are two main original features in the approach implemented in CREUSOT. First, CREUSOT's specification language features a notion of *prophecies*, which is central for the specification of behavior of programs performing memory mutation. Prophecies also permit efficient automated reasoning for verifying about such programs.
Rust provides advanced abstraction features based on a notion of *traits*, extensively used in the standard library and in user code. The support for traits is the second main feature of CREUSOT, because it is at the heart of its approach, in particular for providing complex abstraction of the functional behavior of programs.

Key-words: Rust programming language, Formal Specification, Deductive verification, Aliasing and Ownership, Prophecies, Traits

* Université Paris-Saclay, CNRS, Inria, LMF, 91405, Orsay, France

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

L'environnement CREUSOT pour la vérification déductive de programmes Rust

Résumé : Rust est un langage de programmation relativement récent pour la programmation système, apportant des garanties statiques de sûreté des accès mémoire à travers une politique rigoureuse d'*ownership*. Cette approche ouvre une voie prometteuse pour la *vérification déductive* de code Rust, qui vise à prouver la conformité d'un code vis-à-vis d'une spécification de son comportement prévu. Dans ce rapport nous présentons CREUSOT, un outil pour la spécification formelle et la vérification déductive de programmes Rust.

L'approche mise en œuvre dans CREUSOT s'appuie sur deux caractéristiques originales. Premièrement, le langage de spécification de CREUSOT fournit une notion de *prophétie*, qui est centrale pour la spécification du comportement des programmes effectuant des modifications en place de la mémoire. Les prophéties permettent aussi un raisonnement automatisé efficace pour vérifier ces programmes.

Rust fournit des fonctionnalités d'abstraction avancées basées sur une notion de *trait*, largement utilisée dans la bibliothèque standard et dans les codes utilisateur. La prise en charge des traits est la deuxième caractéristique principale de CREUSOT, car elle est au cœur de sa démarche, en particulier pour fournir une abstraction élaborée du comportement fonctionnel des programmes.

Mots-clés : Langage de programmation Rust, Spécification formelle, Vérification Déductive, Alias, *Ownership*, Prophéties, Traits

Contents

List of Figures

1 Introduction

Today, critical services such as transportation, energy or medicine are controlled by software, and are thus one bug away from disaster. Verifying the correctness of these software is highly important. However, these *systems software* are often written in low-level, pointer-manipulating languages such as C/C++ which make verification significantly more 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*. Aliasing is the phenomenon in which two pointers refer to the same region of memory, meaning 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* give a vocabulary to express these challenges. On the other side, language features like the *region typing* of WHY3 [7] or the *ownership typing* of Rust prevent mutable aliasing altogether.

The Rust programming language promises the performance and flexibility of C with none of the memory safety issues. To make good on this promise it employs a powerful ownership type system which guarantees 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 an aliased pointer cannot be used for mutation, 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.

PRUSTI [3] allows the *deductive verification* of a fragment of Rust by translating programs to the Viper verification framework, encoding Rust's ownership types in Viper's separation logic. Programmers can write annotations using a first-order logic (FOL), which are then sent to automated theorem provers. To specify properties of mutable references, PRUSTI uses *pledges*, properties that a reference must uphold when it is released. The specification language of PRUSTI gives users the possibility to express complex functional specifications for their programs. Yet, the complex separation logic encoding powering PRUSTI poses issue: it has difficulty handling certain common Rust borrowing patterns, and leads to high verification times.

RUSTHORN [14] takes an entirely different approach by remarking that Rust's strong type system obviates the need for separation logic at all. Instead, Rust programs are encoded into FOL using a *prophetic* approach developed for this purpose. The elegant, prophetic encoding of mutable borrows allows existing solvers to efficiently reason about pointer programs *without using a memory model*. RUSTHORN demonstrates this by implementing a proof-of-concept targeting Constrained Horn Clauses (CHCs) for a very restricted fragment of Rust. The powerful automation of existing solvers allows can infer specifications and invariants, even in complex pointer programs and verify them rapidly. But this automation is also one of RUSTHORN's key limitations: users have no way of giving their desired specifications or helping provers when

they fail to infer appropriate invariants.

In this report, we present our tool CREUSOT[1], a new automated tool for deductive verification of Rust code. CREUSOT is based on the *prophetic* approach of RUSTHORN, and extends it to a much larger subset of the Rust language. In particular, we permit the verification of programs using *polymorphism* and *traits*. Unlike RUSTHORN, CREUSOT has a specification language, PEARLITE[2], which allows users to write function contracts and loop invariants. PEARLITE introduces a novel operator `^` (pronounced *final*), which allows for *prophetic* reasoning: this replaces the concept of *pledges* used in PRUSTI.

1.1 Motivating Example: a Polymorphic Sorting Function

To write a polymorphic sorting function in Rust, we use a *trait* to provide the order relation. Traits in Rust work like the typeclasses of Haskell: they allow types to provide implementations of specific operations, and use-sites to *constrain* their type parameters to require specific trait implementations. To write a sorting function for a type `T`, we need an order relation on `T`: this is provided by Rust's **`Ord`** trait which requires a type to implement comparison operators: `<`, `<=`, `>`, and `>=`.

Using this trait, we provide in Fig. 1 a simple implementation of Gnome Sort. This code has a lot going on at once. During the rest of this document we will explain each part in detail, but first we give high-level intuition for this function.

Starting from the signature of `gnome_sort`, we can see that as input we accept a mutable borrow to a vector `v`. Types like **`Vec`** are *safe abstractions around unsafe code*: they encapsulate, behind a *safe* interface, code written in a fragment which does not guarantee memory safety. This means no undefined behavior can be observed from a client. While CREUSOT doesn't support verifying the *implementation* of **`Vec`** which makes use of unsafe code, we can verify its clients because of this abstraction barrier. Doing so requires us to provide a *model* for the vector, which in this example has been defined as a mathematical sequence of values of type `T`.

The ability to provide models for types is not built-in to CREUSOT: we've provided it as a *library* built on-top of the support for traits and logic functions in CREUSOT[3]. We discuss the `Model` trait in Section 4. We believe the ability to define key verification concepts such as models in libraries is a major feature of CREUSOT.

To specify the behavior of `gnome_sort` in CREUSOT we use standard `requires` and `ensures` clauses. The body of these clauses is written in PEARLITE, the specification language of CREUSOT. PEARLITE terms have a strongly Rust-inspired syntax but extend it with the key operators required of a specification language. Besides standard logical operators like quantification (`forall<..>`, `exists<..>`) and implication (`==>`), we use two PEARLITE specific 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; using this we can express the effect of mutations performed to a mutable borrow.

[1] *Le Creusot* is an industrial town in the eastern France, whose economy is dominated by well-known 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`

[3] The only builtin support for models in CREUSOT is the `@` notation which is just syntactic sugar.

```
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 === ^@old_v)]
8      #[invariant((@v).permutation_of(@@old_v))]
9      while i < v.len() {
10         if i == 0 || v[i - 1] <= v[i] {
11             i += 1;
12         } else {
13             v.swap(i - 1, i);
14             i -= 1;
15         }
16     }
17 }
```

Figure 1: Gnome Sort and its specification.

The clause on line 1 states that the *model* of the *final* value of `v` is sorted. It does this with the help of a *logical predicate* `sorted_range` which expresses the property that a *sequence* is sorted between two bounds:

```
#[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] }
}
```

The predicate takes as input a *sequence* `s`, this type is non-executable: it only exists in the logic of Creusot, similarly the **`Int`** type refers to the mathematical set of integers $\mathbb{Z}$. The second clause on line 2 uses a second predicate `permutation_of` to express that the final value of `v` should be a permutation of the input value.

With the specification in mind, we can consider the body of `gnome_sort`. Successive pairs of elements in `v` are compared using a counter `i`. When two values are found to be out of order, they are swapped and `i` is decremented. The next iteration will then check whether the swapped value is out of order with the previous cell. This process will move values into their correct position in a manner similar to insertion sort. To formalize this property, we use a *loop invariant*. The key invariant, on line 6, ensures that the fragment of `v` to the left of `i` is sorted. Notice the usage of `@i` to refer to the model of the machine integer `i`, which is a **`Int`**. This invariant is expressed on `@v`, the model of the *current* value of `v`. At the end of our loop we will *resolve* `v`, obtaining that the final value of `v` is the value at the end of the loop, proving the postcondition. The second

invariant, on line 8, proves the second postcondition of the function in a similar fashion. Finally, the invariant on line 7 ensures that the prophecy of `v` doesn't change.

The annotated program is fed through CREUSOT, which translates it to WHY3. In turn, WHY3 generates and discharges the verification conditions using automated provers including Z3 [11], CVC4 [5] and Alt-Ergo [10].

1.2 Contributions

In this report, we present the following contributions:

- We present the design of PEARLITE, a specification language for the Rust programming language, in Section 2. A key, novel feature is the use of *prophecies* for specification and reasoning with mutable borrows.
- We provide a translation of Rust safe code into an ML-like language without the use of a mutable heap, in Section 3. It goes through the medium-level intermediate representation of Rust, and handles mutable borrows using non-determinism and **`assume`** clauses.
- We show how type traits can be used when writing generic specifications in PEARLITE, and explain how Rust support for polymorphism and traits can be translated to WHY3's module system (see Section 4). In particular, this support permits extensions needed for specifying complex code: definition of logic models, and design of appropriate specifications for parts of the Rust standard library.
- We provide and evaluate the tool implementation in Section 5.

Finally, we discuss related work and future work in Section 6.

2 The PEARLITE Specification Language

PEARLITE is a behavioral specification language inspired by JML for Java [9], ACSL for C [6], Spark for Ada [15], or the specification language of PRUSTI. Functions are given *contracts* which specify pre and postconditions, 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` clauses of 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*. We detail this notion in the second subsection below, on a toy example illustrating the basic ideas.

2.1 Background Logic

PEARLITE is based on a classical, first-order, multi-sorted logic. Each Rust type corresponds to a sort in this logic. The formulas are constructed from *atomic predicates* and *connectives*. The 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 `===`, defined on any sort. This logical equality is the symbol interpreted to the set-theoretic equality in a set-based semantics. It must be distinguished from the program equality of Rust, `==`, which is just sugar for `PartialEq::eq`.

The set of sorts can be augmented with extra logic sorts and functions through PEARLITE's support for mapping to external definitions already present in the Why3 standard library. An important example is the sort `Int` for unbounded mathematical integers. It comes with the standard arithmetic operations and comparison predicates. A syntactically valid formula is thus for example:

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

Another example of a logic sort is the sort of polymorphic sequences, which is used in our introductory example of Gnome Sort as a model of vectors.

The constructions for formulas and terms are augmented with let-bindings, match and if expressions. Logic predicates and functions can be defined in PEARLITE using the `#[logic]` or `#[predicate]` attributes respectively. While these definitions reuse the syntax of Rust function declarations, they can only have pure bodies and cannot be called from Rust code: they only exist within the logic of CREUSOT. Here are some basic examples.

```
#[logic]
fn sqr(x:Int) -> Int { x * x }

#[predicate]
fn is_square(y:Int) -> bool {
  pearlite!{ exists<z: Int> y === sqr(z) } }
```

As seen above, when a PEARLITE definition uses constructs (such as `exists`) which are not part of Rust syntax, the use of the `pearlite!` macro is required.

Logic functions can even be defined recursively, provided they are provably terminating, which is done by providing a `variant` clause. For example:

```
#[logic]
#[variant(x)]
fn sum_of_odd(x:Int) -> Int {
  if x <= 0 { 0 } else { sum_of_odd(x-1) + 2*x - 1 }
}
```

PEARLITE formulas are type-checked using the front-end of the Rust compiler, but they are not borrow-checked. In particular, this means that ownership of values is not taken into account in PEARLITE: a variable can be used several times in a PEARLITE formula of in a logic function or predicate even though it a type which does not allow copying.

PEARLITE formulas are translated into WHY3's logic in a one-to-one manner, including for predicate and logic functions. Finally, one can introduce lemmas in the proof context using

```
1  #[ensures(if toggle { result === x && ^y === *y }
2                 else { result === y && ^x === *x }   )]
3  fn toggle_borrow<'a>(toggle: bool, x: &'a mut i32,
4                       y: &'a mut i32) -> &'a mut i32 {
5    if toggle { x } else { y }
6  }
7
8  #[ensures(result === 42)]
9  fn toggle_borrow_test() {
10     let mut a = 4;
11     let mut b = 7;
12     let x = toggle_borrow(true, &mut a, &mut b);
13     *x += 2;
14     return a * b;
15 }
```

Figure 2: A toy example illustrating prophecies in the specification language.

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.

```
#[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 model mutable borrows in PEARLITE on the example of Fig. 2. The function `toggle_borrow` returns, depending on its first Boolean parameter, either its second or third argument, which are *mutable borrows*. The `toggle_borrow_test` function then exemplifies a call to that function. The `ensures` clause on line 8 states a postcondition

that we intend to prove about the second function. For that purpose, the auxiliary function `toggle_borrow` must be given an appropriate contract, namely the postcondition of lines 1 and 2.

First, the postcondition states that the result of `toggle_borrow` (denoted by the identifier `result`) is either the borrow `x` or the borrow `y`, depending on the value of `toggle`. Importantly, when we say `result === x` (*i.e.* when `toggle === true`), we are stating that the *borrow* `result` (a pointer) is equal to the *borrow* `x`, not merely the values being pointed. This means that mutating the borrow returned by the function will have the effect of mutating the variable pointed to by `x`. In the case of the call in `toggle_borrow_test`, this makes it possible to prove that the value of `a` at line 14 is 6.

Second, the specification needs to state that the other borrow parameter (`y` if `toggle === true`, `x` otherwise) is dropped by the function and will not be mutated anywhere, so that the caller knows that the value it points to will no longer change until the borrow's lifetime ends. This is specified using a *prophecy*, a concept introduced by RUSTHORN [14]. For any mutable borrow `b`, one can write `^b` in PEARLITE to denote its *final value*, the value that the variable it points to will have when the lifetime will expire. Prophecies act like a bridge between the lender and borrower, when a borrow is dropped we recover information which allows us to update the value of the lender. Dropping a mutable borrow is equivalent to stating that the current value pointed to by the borrow, `*b`, equals its final value, `^b`. In the case of the call in `toggle_borrow_test`, this makes it possible to prove that the value of `b` at line 14 is the value it has at the point of call to `toggle_borrow`, which is 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 [14, 2] that it is sound to *prophesize* it in the logic.

The first equality (*i.e.* `result === x` when `toggle === true`) actually implies `^result === ^x`, as we are asserting that `result` and `x` are *mathematically* equal (they are completely indistinguishable) even prophetically. This explains why the first equality is enough to specify that a mutation through the borrow `result` causes a mutation of the variable pointed by the argument borrow `x` (when `toggle === true`). Since they both have the same prophecy, modifications to the memory pointed by `result` will affect the lender of `x`. The approach is explained in more detail and validated in RUSTHORN [14], and RUSTHORNBELT [2].

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 languages 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, and as such, there are rich APIs to access, extract and manipulate the MIR of Rust programs. 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 RustBelt [13] to prove the soundness of the Rust type system, and RUSTHORNBELT [2] reuses this formalization to formally verify the correctness of

its prophetic translation, which we claim lends credibility to CREUSOT's own prophetic translation.

3.1 Verifying MIR

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 by reconstructing the structured control flow from the MIR graph, and then use Why3's carefully designed WP computation algorithms.

Restructuring Programs Böhm and Jacopini [8] show that any CFG can be transformed into a structured program using only loops and conditional statements so long as we can introduce auxiliary variables. Baker [4] refines this algorithm, avoiding all auxiliary variables when the graph is *reducible* given that multi-level `break` statements are also allowed. The notion of reducibility is defined on control flow graphs and is equivalent to asking that the program which generates the graph has no `goto` statements. As Rust has no `goto` construct, every MIR graph generated by the Rust compiler front-end is thus reducible. We implemented a version of this algorithm translating MIR into WhyML, Why3's programming language. This algorithm is packaged into a front-end language for Why3 called *MLCFG*, an unstructured, ML-like language.

3.2 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 `x` is the only way to access that memory location: mutating memory through that pointer can only be observed by a read using variable `x`.

Therefore (as in RUSTHORN), if `t` is the translation of type `T`, then we 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.

3.3 Translating Borrows to Prophecies in WHY3

The translation of borrows using prophetic values in WHY3 is inspired from the verification condition generation in RUSTHORN [14] and in RUSTHORNBELT [2]: it is designed such that the verification conditions generated by WHY3 from the program generated by CREUSOT are equivalent to those generated directly by RUSTHORN and RUSTHORNBELT. Hence, we claim that the soundness proofs given for RUSTHORN and RUSTHORNBELT also apply to our translation.

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

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

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

The first important case of the translation scheme occurs at the creation of a borrow. A statement

```
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 a 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`. This assignment "in advance" is admissible because until the lifetime of `y` is ended, Rust's ownership typing guarantees that `x` is not accessible. The second important case occurs when a borrow is *dropped*, where we insert a *resolution statement*:

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

The contents of an `assume` clause states a fact as a trusted hypothesis for subsequent statements. Consequently, the clause added above corresponds to the fact that at this point the value pointed to by the borrow will not change, and therefore the prophecy must have been realized. Of course, in Rust, borrows can be contained within other types, so in practice we use a generalized notion of resolution to handle borrows nested within other types. We discuss this mechanism in Section 4.

The simplified translation of the `toggle_borrow` example, from Fig. 2, is thus as given in Fig. 3. The verification of the postcondition of `toggle_borrow_test` can be proven automatically, with the context providing the proper equalities to show that the returned value is 42. It is crucial that we assign the future value of a borrowed variable at the moment of lending: in the presence of arbitrarily complex control flow, it would be impossible to restore a value at the end of the lifetime. By mentioning prophecies in specifications we can preserve the relevant information for a lender to recover a concrete value.

4 Support for Polymorphism and Traits

To implement ad-hoc polymorphism, Rust makes use of a *trait system* like the typeclasses of Haskell. Each trait consists of zero or more associated functions, types and constants. Traits can be given *implementations* for specific types, at which point a concrete value, type or constant must be provided for each of the trait's items. When the type parameters of a function are *constrained* to implement a specific trait, it becomes possible to use those associated items in the function's body or type signature. A notable feature of typeclass-style systems is *instance resolution*: the language will automatically identify the appropriate instance to use if one exists, and will otherwise identify which constraint was missing to select an instance. To achieve this, resolution relies on the property of *coherence* which states that at most one instance of a trait

```
let toggle_borrow (toggle: bool) (a: borrow int32)
                  (b: borrow int32) : borrow int32
  ensures { if toggle then result = a /\ b.final = b.current
    else result = b /\ a.final = a.current }
= if toggle then a else b

let cfg toggle_borrow_test ()
  ensures { result = 42 }
=
  var a, b : int;
  var bor_a, bor_b, x : borrow int;
  a <- 4; a <- 7;

  (* let x = toggle_borrow(true, &mut a, &mut b); *)
  bor_a <- { current = a ; final = any int32 };
  a <- bor_a.final;
  bor_b <- { current = b ; final = any int32 };
  b <- bor_b.final;
  x <- toggle_borrow true bor_a bor_b;

  (* *x += 2; *)
  x <- { current = 2; final = x.final };
  assume { x.final = x.current };

  (* return a * b *)
  a * b
```

Figure 3: Simplified translation of the projection toggle example in MLCFG.

exists for a type. Coherence enables ecosystem-wide modularity and many common operations have been pushed into 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, as shown below, CREUSOT not only allows verifying programs using traits, it also *uses* traits for some of its core features: the `Resolve` and `Model` traits.

4.1 Specifying Trait Behavior

Rust uses a trait *hierarchy* to represent comparison operations. The trait **`PartialOrd`** implements a *heterogenous partial order*, instances must provide implementations for all of `lt`, `gt`, `le`, `ge`,

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

    #[ensures(result === (self.le_log(o)))]
    fn le(&self, o: &Self) -> bool;

    #[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);
    ...
}

#[logic]
fn le_log<T: Ord>(a: T, b: T) -> bool {
    ! (a.cmp_log(b) === Greater)
}
```

Figure 4: A simplified **Ord** trait with specifications

and `partial_cmp`. Additionally, the official documentation [18] 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 Section 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. A law cannot mention program symbols like `le`, since it is not a logic function (it could potentially have side effects or diverge). Instead, we require **PartialOrd** to have a logical counterpart `le_log`, which is tied to `le` through a postcondition.

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*. 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**. We require a definition of `cmp_log` and a proof of transitivity. Then, a generic definition of `le_log` is provided, which makes it possible to specify the *program* method `le`. Note that transitivity in terms of `le_log` is deduced easily from that of `cmp_log`. 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.

Contract Refinement Every implementation of a trait for a specific type must *refine* the contract of the trait. It must weaken pre-conditions and strengthen postconditions. This possibil-

ity 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 attempt to resolve it to a specific implementation of the trait, and if so, will use that instead, thus using the stronger implementation contract where possible.

4.2 The `Resolve` Trait

As shown with **`Ord`**, traits are not limited to having exclusively program functions. We can define traits which contain logical functions, and use those within specifications. We use this to express the notion of *resolution* discussed in Section 2.

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

Much like how the **`Drop`** trait is provided by Rust for types to define their behavior when dropping, we use the `Resolve` trait to define the knowledge gained from resolving a specific type. When a type implements `Resolve`, CREUSOT will use its specific implementation, thus refining `resolve` to a concrete predicate.

As an example, 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 }
    }
}
```

That is, when a mutable borrow is resolved (*i.e.* when it becomes dead), CREUSOT will gain the knowledge provided by the implementation of `resolve` above, which coincides with the resolution we described in Section 2.

However, using the `Resolve` traits makes it possible to generalize modularly the resolution mechanism to data structures containing mutable borrows. We also wish to learn resolution formulas for, *e.g.*, 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 an implementation of the `Resolve` trait for data structures. Consider the case of pairs: we have the following implementation:

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

The resolution of a pair is no more than the resolutions of its components. If this pair contained mutable borrows we would learn the formula

```
*self.0 === ^self.0 && *self.1 === ^self.1
```

The `Resolve` trait is declared **`unsafe`** because the implementations of `resolve` are trusted, *i.e.*, assumed correct. Note that for user-defined types, `resolve` can be *safely* automatically derived.

Unlike normal traits, CREUSOT guarantees that a type always has an implementation of `Resolve`. If no explicit implementation is found, we use a *default* implementation which says nothing about the contents of the resolved type.

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 typically 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.

By using a trait, we can provide helper instances, for example, we define the model of a borrow to be the model of the borrowed object:

```
impl<T : Model> Model for & T {
    type ModelTy = T::ModelTY;
    #[logic]
    fn model(self) -> Self::ModelTy {
        @(*self)
    }
}
```

Then, in specifications we can use `@x` instead of `@*x`.

4.4 Abstraction and Genericity in WHY3

Just like any other Rust feature, CREUSOT needs to translate Rust traits into WHY3. WHY3 supports methods of abstraction: first a form of *parametric polymorphism à la* ML, and second a fairly original module system where a module can include a mixture of concrete definitions

and abstract declarations, the latter forming a set of module parameters. The classical notion of module *functor* is replaced by a notion of module *cloning* [12], an operation that copies the content of a module into another, substituting an arbitrary subset of the parameters by concrete definitions or new abstract ones.

This notion of cloning is *generative* in the sense that it produces a fresh, partially instantiated, copy of an existing module. This generativity aspect makes the module system significantly different from a type class system. To understand this problem, consider, for example, a logic function `max`, which is polymorphic over a type `T` implementing the trait **`Ord`**:

```
#[logic] fn max<T: Ord>(x: T, y: T) -> T {
    pearlite!{ if x <= y { y } else { x } }
}
```

Following WHY3's pattern for genericity, it is idiomatic to translate this function to a module `Max` containing abstract definitions for the type `T` and a clone of a module **`Ord`** providing access to abstract symbols for the **`Ord`** trait.

```
module Max
  type t
  clone Ord with type t = t
  function max (a b : t) : t =
    if Ord.le_log a b then a else b
end
```

We would translate analogously another logic function `min`, which has a similar definition. Note, these two functions could have been defined in different modules (or crates) in Rust, so that we cannot use the same WHY3 module in our translation or we would loose compositionality.

Now, imagine we want to make use of `min` and `max` in the same definition. For example, it would be convenient to prove a lemma which *relates* them:

```
#[logic]
#[ensures(min(x, y) <= max(x, y))]
fn min_le_max<T: Ord>(x: T, y: T) -> T {}
```

Still following WHY3 design patterns, in addition to cloning the module for **`Ord`**, the module containing `min_le_max` in WHY3 needs to clone both the modules defining `min` and `max`:

```
module MinLeMax
  type t
  clone Ord with type t = t
  clone Min with type t = t
  clone Max with type t = t
  let function min_le_max (a b : t) : unit
    ensures { Ord.le_log (Min.min a b) (Max.max a b) }
  = ()
end
```

But now, we have a problem: this is not provable. Because of the generativity of cloning, the symbols `Ord.cmp_log`, `Max.Ord.cmp_log` and `Min.Ord.cmp_log` (from which the logical comparison operator `Ord.le_log` is defined) are three *different* abstract symbols, and there is no reason the logic could relate them. However, in Rust, the *coherence* property of type traits guarantees that there is only one instance for each trait, so that these three symbols, in Rust, are necessarily the same semantically.

The WHY3 way of dealing with this issue is to specify *substitutions* when cloning the modules `Min` and `Max` to tell WHY3 that these three symbols are the same:

```
clone Min with type t = t, predicate Ord.cmp_log = Ord.cmp_log
clone Max with type t = t, predicate Ord.cmp_log = Ord.cmp_log
```

However, this only works because `cmp_log` is an abstract symbol: if it were *defined*, e.g., from the comparison operator `cmp_log`, then this substitution would be forbidden by WHY3, and we would end up with three different *definitions* of `cmp_log`. This is for example the case of the logical function `le_log`, which is *defined* from `cmp_log`. Of course, if a substitution is applied to `cmp_log`, we can prove that the three versions of `le_log` are in fact extensionally equal. But this proof of extensional equality is an additional proof step for provers, which may make them fail, and, more importantly, if the definition of `le_log` was significantly more complex (e.g., if it used recursion), then the provers would not be able to prove their equivalence automatically.

4.5 Working Around Generativity

To overcome this difficulty, CREUSOT adopts a different translation strategy. First, because any function can be polymorphic and have trait bounds, CREUSOT generates separate WHY3 modules for each function. This also applies to trait methods: in particular, in the example above, the trait **`Ord`** would be translated into *several* WHY3 modules, separating each of its methods.

Second, for any program function or logic function, CREUSOT generates several modules:

- An *interface module* only exposes the type and specification of the function, together with a clone of the interface modules of the functions its interface depends on directly.

- For logic functions and predicates, a *definition module* refines the interface module with the body of the function, and clones the interface modules of dependencies which are required for the well-formedness of the definition.

- A *proof module* aims at generating the VCs of functions which require proofs, such as program functions and lemma functions. It refines the interface module by providing the body of the function, but additionally clones the definition modules of all its transitive dependencies. Importantly, these clones use substitutions to refine the interface module of its transitive dependencies to their corresponding cloned definition modules. Because it gives concrete definitions for all the dependencies, this module cannot be cloned without encountering the problem described in Section 4.4: it is only used to generate VCs.

Using this scheme, we make sure that, in proof modules, transitive dependencies (i.e., the `le_log` logic function for the `min_le_max` lemma function) have only one definition: all their uses in dependencies are substituted by a unique definition.

In our example, CREUSOT generates two modules for the `max` function:

```
module Max_Intf
  type t
  function max (a b : t) : t
end
module Max_Def
  type t
  clone Le_log_Intf with type t = t
  function max (a b : t) : t = (* ... *)
end Max_Def
```

The first is the *interface module*, while the second is the *definition module* for the function `max`. Note that there is no *proof module* for `max`, since this logic function has no contract to prove.

Then, the `min_le_max` lemma function gives rise to three WHY3 modules:

```
module Min_le_max_Intf
  type t
  function min_le_max (a b : t) : unit
end
module Min_le_max_Def
  type t
  clone Le_log_Intf with type t = t
  clone Min_Intf with type t = t
  clone Max_Intf with type t = t
  function min_le_max (a b : t) : unit = ()
  axiom min_le_max_spec : forall a b,
    Le_log_Intf.le_log (Min_Intf.min a b) (Max_Intf.max a b)
end
module Min_le_max_proof
  type t
  clone Cmp_log_Def with type t = t
  clone Le_log_Def with type t = t
  clone Min_Def with type t = t,
    function Le_log_Intf.le_log = Le_log_Def.le_log,
    function Le_log_Intf.Cmp_log_Intf.cmp_log = (* ... *)
  clone Max_Def with (* ... *)
  function min_le_max (a b : t) : unit
    ensures { (* ... *) } = ()
end
```

Notice how the interface module contains only what is necessary to type-check a function which depends on `min_le_max`; the definition module contains the specification of the function, but

with abstract interface modules for its dependencies, and the proof module substitutes all abstract symbols in the dependencies with concrete definitions. Moreover, as desired, the function `le_log` only has one definition.

There are two subtleties we have not yet discussed: first, when a function depends on several different instantiations of the same function, the corresponding module is cloned several times, once with each different type substitution. Second, when referring to a trait method of a concrete type, the corresponding instance is cloned instead of the generic abstract definition, so that the possibly more precise specification is available.

5 Experimentation and Evaluation

We evaluated the performance of CREUSOT on a wide range of benchmarks, including several from PRUSTI and RUSTHORN. These benchmarks cover a wide range of Rust features making heavy use of polymorphism and traits. Additionally, in several cases 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.

5.1 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 and available in the supplementary material coming with this submission. During execution CREUSOT takes over after type checking is performed, translating functions into MLCFG and outputting the result to a file. The resulting file can then be loaded in WHY3 and verified using either its IDE or command line.

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. Libraries like *Vec*, which internally make use of unsafe Rust features behind a safe interface may be axiomatized so that their safe clients can still be verified.

By integrating with Rust's build tool *Cargo*, CREUSOT also has full support for verified libraries. Functions of a library may be annotated with contracts which can then be used in downstream projects. We use this functionality to provide an annotated subset of Rust's standard library which ships with CREUSOT.

Name	Has generics?	Has traits?	LOC	Spec. LOC	# of VCs	Avg. Time / VC	Safety	Additional Properties
Gnome Sort	✓	✓	14	17	34	0.06	✓	Func. correctness
Filter Vector	✓	✗	40	36	6	0.08	✓	Func. behavior required for safety
Sparse Array†	✓	✗	53	81	79	0.15	✓	Func. behavior, model and invariants
In place List Rev.	✓	✗	21	10	1	0.06	✓	Func. correctness
Inc Some List	✗	✗	39	21	4	0.03	✓	Func. correctness
Inc Max	✗	✗	12	3	2	0.04	✓	Func. correctness
Inc Max Many	✗	✗	13	3	2	0.04	✓	Func. correctness

Table 1: Selected results of our evaluation (part 1). 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. "Avg. Time / VC" measures the average verification time per prover. Tests marked with † required manual proof steps.

5.2 Evaluation

We measure the verification performance for programs translated with CREUSOT. We adapted and generalized programs from the PRUSTI [3] benchmark suite, additionally strengthening the verified properties. Other examples were inspired from the WHY3 gallery [7], Rosetta Code [16] or RUSTHORN [14]. The examples we present focus on traits and polymorphic code, and make usage of types like `Vec`. For axiomatized types like `Vec` we use the interface provided as part of CREUSOT's standard library.

Proof Strategy 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 change the complexity of 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

Name	Has generics?	Has traits?	LOC	Spec. LOC	# of VCs	Avg. Time / VC	Safety	Additional Properties
Binary Search	✓	✓	25	20	51	0.03	✓	Func. correctness
Knapsack 0/1	✓	✗	31	60	81	0.08	✓	—
Knapsack 0/1	✓	✗	32	113	113	0.11	✓	Func. correctness
Knuth Shuffle	✓	✗	19	5	1	0.17	✓	Permutation
100 doors	✗	✗	21	6	47	0.09	✓	—
Heap Sort†	✓	✓	38	75	231	0.13	✓	Func. correctness
Selection Sort	✓	✓	19	27	50	0.08	✓	Func. correcntess

Table 2: Selected results of our evaluation (part 2).

2" strategy: it first attempts to solve the entire verification condition (VC) for a function, and if it fails after 1 second, *splits* the VC and attempts to solve each sub-goal for 1 second. This strategy is a common first step when verifying programs with WHY3. Two benchmarks required a small number of additional manual proof steps, `Sparse Array` to prove a complex lemma about injections between sequences, and `Heap Sort` to hint solvers about non-linear arithmetic.

Our evaluation was performed using an Ubuntu 21.10 installation with a Intel Core i5-10310U CPU with SMT and 16 GB of RAM. We relied on a combination of Z3 4.8.12 and CVC4 1.8 as back-ends to WHY3.

5.3 Discussion

The selected results are presented in Table 1 and Table 2, where benchmarks are grouped by origin. The first group of benchmarks of Table 1 are novel examples contributed as part of CREUSOT's test suite. "Filter Vector" is a challenging example regarding reasoning on memory separation [**?**]. "Sparse Array" is an example from the VACID-0 benchmarks [**?**], originally coming from an exercise in computer algorithmics [1]. Verifying its functional behavior involves the design of a complex model. Moreover one of the VCs is proved thanks to a mathematical lemma that requires a few steps of manual proof steps 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 can be verified without the need for separation logic. The second group of Table 1 come RUSTHORN's evaluation [14, Section 4.3], where we added specifications of the intended functional behavior. The third group, in Table 2, are adapted from PRUSTI's evaluation [3, Section 7.2]. We proved the generic versions of these, whereas PRUSTI's evaluation is done on monomorphized versions.

The "Has traits?" measures whether the test case has a function with a generic parameter constrained by a trait. Given the deep integration of traits into Creusot, every test uses traits through either `Model` or `Resolve`.

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 the safety, the overhead is low, and mostly consists of stating the properties we wanted to prove in the first place. The cost of additionally verifying the functional behavior is also low.

The PRUSTI examples listed here are derived from their introducing paper in 2019 [3]. In their paper they provide two versions for their functions, the first proving only safety while the second proves some portions of functional correctness. Though they claim to prove the safety of a polymorphic binary search and selection sort, we were not able to confirm this result. We prove a fully polymorphic version of binary search, and selection sort using an **`Ord`** trait with specification for comparisons. Compared to the PRUSTI implementation, our full, polymorphic verification is over an order of magnitude faster.

The difference in verification performance is made more 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 7 seconds using a single CPU core or a little over 1 second when using all cores. 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 on a single core in approximately 12 seconds.

We interpret the large performance difference between PRUSTI and CREUSOT as arising from the different logical translation of Rust. PRUSTI encodes MIR programs into the separation logic of Viper, which requires it to carefully manage Viper's view of *permissions*. Despite Viper's support for solving separation logic, the proof obligations contain detailed ownership information which Viper must continuosly track and preserve. Since the prophetic translation of Creusot entirely *eliminates* ownership, the corresponding proof obligations are much simpler.

6 Related Work and Future Work

RUSTHORN [14] 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 code assertions inserted, which are by essence limited to executable Boolean expressions on program variables, without any possibility to relate them to any 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.

The core translation of RUSTHORN was mechanically verified in the Coq proof assistant. The resulting proof RUSTHORNBELT [2] shows that the prophetic encoding of mutable borrows is sound, even in the presence of the complex borrowing patterns of Rust. Because of the similarities in the core translations of CREUSOT and RUSTHORN, we take that as evidence for the soundness of our meta-theory. RUSTHORNBELT shows the soundness of a prophetic encoding straight into FOL. It remains future work for us to extend this formalization with an intermediate step translating MIR to a functional language, as CREUSOT does.

PRUSTI [3] is another deductive verifier for Rust, based on the Viper separation logic platform. It does not use a prophetic encoding, instead using its own approach based on *permissions*. Like CREUSOT, PRUSTI has a specification language which can be used to give contracts and invariants. Because PRUSTI has no notion of prophecies, it does not use the *final* operator (^) to discuss mutable borrows, instead introducing a notion of *pledges*. Pledges are specifications which are enforced at the end of a borrow's lifetime, 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. It also has a formal logical interpretation [2]. PRUSTI has support for a form of logic functions in the form of *pure* functions: functions which lie in a pure, terminating subset of Rust. CREUSOT supports a richer notion of *logic* and *predicate* functions : it uses an extended, purely functional language PEARLITE, with the ability to introduce recursive definitions. It also has the ability to refer to existing Why3 standard library, opening support for *mathematical types* like sequences or the integers, which are fundamental building blocks for complex specifications. Moreover, PEARLITE is fully integrated with Rust traits and their support in CREUSOT.

As far as we are aware, PRUSTI's permission system supports many of the common borrowing patterns of Rust, but still struggles with patterns like *reborrowing in a loop*. In contrast, the prophetic approach of CREUSOT handles borrows in their totality.

Another noticeable difference with PRUSTI lies in the choice of underlying logic. PRUSTI encodes into some separation logic and delegates verification to Viper, whereas CREUSOT encodes into FOL and delegates verification to SMT solvers via WHY3. We believe this difference explain 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.

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 [**?**], 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. User

feedback seems to say that using prophecies is simpler to grasp than pledges for the writer of contracts. 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.

The Cogent [17] language was developed to write verified systems software. Programs are written in a functional language which is then efficiently compiled to C, while producing certificates which can be used to prove correctness of compilation and functional behavior in Isabelle/HOL. However, to our knowledge, Cogent does not allow the creation of references to values created within Cogent. Pointers must be provided from surrounding C code and can be manipulated within Cogent but not created. The lack of a borrowing mechanism like Rust enforces a much stricter ownership discipline on Cogent programs.

Future Work Today, CREUSOT allows the verification of complex programs making use of traits, polymorphism and mutable references, but this remains a fragment of Rust. Much work remains on adding support for closures [19] and iterators [**?**] in an efficient manner, as well as exploring other complex features of Rust like *trait objects* (existential types). In addition, while CREUSOT has robust support for logical functions and predicates, including in traits, useful features of mature verification platforms are still missing. In particular, we are referring to *ghost code* and the related notion of *type invariants*. These features can be essential in the verification of complex imperative code, and there remain questions about their interaction with prophetic verification.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] Anonymous. RustHornBelt: A semantic foundation for functional verification of rust programs with unsafe code. *Proceedings of the ACM on Programming Language Design and Implementation*, 2022. Submitted for review.

[3] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging Rust types for modular specification and verification. *Proc. ACM Program. Lang.*, 3(OOPSLA):147:1–147:30, 2019. doi:10.1145/3360573.

[4] Brenda S. Baker. An Algorithm for Structuring Flowgraphs. *Journal of the ACM*, 24(1):98–120, January 1977. doi:10.1145/321992.321999.

[5] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi:10.1007/978-3-642-22110-1_14.

[6] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.16*, 2020. URL: https://frama-c.com/html/acsl.html.

[7] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6):709–727, 2015. See also http://toccata.lri.fr/gallery/fm2012comp.en.html. URL: http://hal.inria.fr/hal-00967132/en, doi:10.1007/s10009-014-0314-5.

[8] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, May 1966. doi:10.1145/355592.365646.

[9] David R. Cok. OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment*, volume 149 of *Electronic Proceedings in Theoretical Computer Science*, pages 79–92, 2014. doi:10.4204/EPTCS.149.8.

[10] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, July 2018. URL: https://hal.inria.fr/hal-01960203.

[11] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

[12] Jean-Christophe Filliâtre and Andrei Paskevich. Abstraction and genericity in Why3. In Tiziana Margaria and Bernhard Steffen, editors, *9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 12476 of *Lecture Notes in Computer Science*, pages 122–142, Rhodes, Greece, October 2020. Springer. See also `http://why3.lri.fr/isola-2020/`. URL: `https://hal.inria.fr/hal-02696246`.

[13] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, January 2018. `doi:10.1145/3158154`.

[14] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. Rusthorn: Chc-based verification for rust programs. In Peter Müller, editor, *Programming Languages and Systems*, pages 484–514, Cham, 2020. Springer International Publishing.

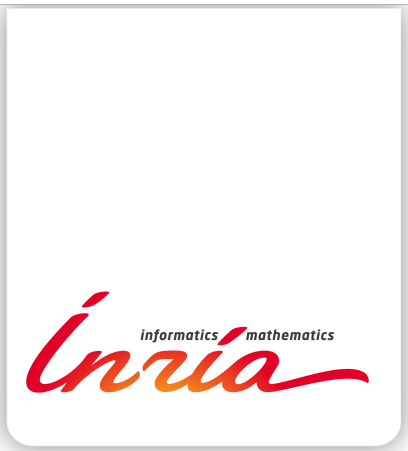
[15] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015.

[16] Michael Mol and other contributors. The Rosetta Code chrestomathy of programs. URL: `http://rosettacode.org`.

[17] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: Bringing down the cost of verification. *ACM SIGPLAN Notices*, 51(9):89–102, 2016.

[18] The Rust Community. The `std::cmp::`**`Ord`** trait of Rust. URL: `https://doc.rust-lang.org/std/cmp/trait.Ord.html`.

[19] Fabian Wolff, Aurel Bílỳ, Christoph Matheja, Peter Müller, and Alexander J Summers. Modular specification and verification of closures in rust. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–29, 2021.



RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399