Specifying and Verifying Higher-order Rust Iterators

Xavier Denis[0000−0003−2530−8418] and Jacques-Henri Jourdan[0000−0002−9781−7097]

Université Paris-Saclay, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France

Abstract. In Rust, programs are often written using *iterators*, but these pose problems for verification: they are *non-deterministic*, *infinite*, and often *higher-order*, *effectful* and built using *combinators*. We present a general framework for specifying and reasoning with Rust iterators in first-order logic. Our approach is capable of addressing the challenges set out above, which we demonstrate by verifying real Rust iterators, including a higher-order, effectful `Map`. Using the Creusot verification platform, we evaluate our framework on clients of iterators, showing it leads to efficient verification of complex functional properties.

Keywords: Rust · Deductive verification · Iterators · Closures

1 Introduction

The Rust language aims to empower systems software programmers by offering them safe and powerful linguistic abstractions to solve their problems. The most notorious of these abstractions, Rust's *borrowing* mechanism, enables safe usage of pointers without a garbage collector or performance penalty. A close second is perhaps Rust's *iterator* system, through which Rust provides composable mechanisms to express the traversal and modification of collections. Iterators also underlie Rust's `for` loop syntax, and are thus the primary manner Rust developers write loops or interact with data structures. It is therefore essential for a verification tool for Rust to provide good support for iterators.

Rust iterators generate sequences of values. Most importantly, they are objects providing a method `fn next(&mut self) -> Option<Self::Item>`. This method takes a *mutable reference* (`&mut self`) to the iterator, allowing it to change its internal state, and optionally returns a value of type `Self::Item`, the type of the values generated by the iterator. If, instead of returning such a value, the iterator returns `None`, it means iteration has finished for now, though it may resume again later. Rust's `for` loops are just syntactic sugar for repeatedly calling `next` at the beginning of each iteration, until such a call returns `None`. For example, the following two pieces of code present a Rust loop for iterating over integers between 0 (included) and `n` (excluded), using a *range* iterator:

```
                                    let mut iter = 0..n;
                                    loop { match iter.next() {
for i in 0..n { <body> }              None => break,
                                      Some(i) => <body>
                                    } }
```

The piece of code on the left-hand side uses an idiomatic `for` loop, while the other shows its desugared version.

Iterators present unique challenges for verification tools: indeed, because the use of iterators is pervasive in Rust, it is necessary to allow verification of code using iterators with as little interaction as possible. In particular, most common patterns such as iterating over integers in a given range or reading the elements of a vector should not need any annotation other than the loop invariants the user would write if not using iterators. On the other hand, Rust's iterator library is complex, with many features representing as many challenges for verification: iterators can be built from various data structures and modified through *iterator combinators*, which make it possible to create iterators from simpler ones, by, e.g., skipping the first few elements or applying a given function to each of the elements.

Consider the example below:

```
1 let mut cnt = 0;
2 let w = vec![1,2,3].iter().map(|x|{cnt += 1; x + 1}).collect();
3 assert_eq!(w, vec![2,3,4]);  assert_eq!(cnt, 3);
```

On line 2, quite a lot happens at once. First, we produce an iterator over the elements of the vector `vec![1,2,3]` with the syntax `.iter()`, which we transform through a call to `map`. The method `map` is an *iterator combinator*: it returns a new iterator that calls the given closure on each of the elements generated by the underlying iterator, and forwards the value returned by the closure. Interestingly, the closure we pass to `map` *captures mutable state*: it modifies the variable `cnt`. Finally, the method `collect` gathers the elements generated into a new vector `w`.

We aim at requiring only lightweight annotations for verifying this kind of code: the appeal of *iterator chains* like on line 2 are the ergonomics, they are compact and highly-readable. For verification of iterator-based code to be successful, it must preserve this ergonomics. However, despite its apparent simplicity, this piece of code is challenging to verify: it combines higher-order functions and mutable state, uses potentially overflowing integers, and assertions on line 3 check full functional behavior.

More generally, to support iterators, a verification tool for Rust needs to provide a specification scheme that both provides good ergonomics and overcomes the following technical challenges:

- *Strong Automation*: for verification to be used, it must require little to no user interaction and lead to good verification performance.
- *Interruptibility*: iterators can produce infinite sequences of values and can be interrupted before completion, thus specification and verification must happen as the iterator is used, and not at completion.

- *Non-Determinism*: iterators can feature both specification or implementation non-determinism, so the sequence of known values might not be known in advance to the verifier. For example, the order of elements generated by an iterator over a hash table may be left unspecified for a client.
- *Compositionality*: iterators can be consumed by *combinators*, so their specifications need to follow a general pattern which make them *composable*. For example, the specification of a combinator such as `skip(n)`, which skips the first `n` elements of a given iterator, should accept the specification of any iterator, and provide a sound and useful specification for the combined iterator.
- *Higher-Order & Effects*: some iterator combinators, such as `map`, are *higher-order*, they take a closure as parameter. To verify programs using these combinators, a verification tool should overcome the challenges of higher-order functions, which potentially capture mutable state.

1.1 Contributions

In order to reach this goal, we propose a new specification scheme for iterators in Rust. Our contributions can be summarized as follows:

- In Section 2, we provide a general specification scheme for Rust iterators in first-order logic. It supports possibly non-deterministic, infinite and interruptible iterators. It is inspired by Filliâtre and Pereira's specification of iterators in Why3 [4], but it is adapted to our style of specification using a *prophetic mutable value semantics* [12] for Rust. This style of specification is particularly well suited to handle mutable values (of which iterators are an instance), by leveraging the non-aliasing guarantees provided by Rust's type system.
- In Section 3.1, we show that this scheme can be trivially instantiated for basic iterators such as range of integers.
- In Section 3.2, we show how this scheme can be instantiated to give full functional specification to *mutating iterators*. These iterators allow to mutate the content of a data structure by iterating over mutable references *pointing to the content of the data structure*.
- In Section 3.3, we show that our specification scheme is *composable*, so that it can be used to specify iterator combinators transforming arbitrary iterators into more complex ones. We give two examples: `take`, which truncates an iterator to at most a given number of elements, and `skip`, which skips a given number of elements at the beginning of iteration.
- To support higher-order iterator combinators, we provide a specification mechanism for *closures* in Section 4. This mechanism distinguishes the three kinds of closures of Rust (`Fn`, `FnMut` and `FnOnce`), and allows specifying the side effects a closure may have on its environment by making explicit the effect of a call on the state of the closure. It allows reducing the verification conditions for closures to *first-order logic*, enabling usage of off-the-shelf automation.

- In Section 5, we explain how we can combine the techniques presented in previous sections to specify higher-order iterator combinators, by taking `map` as an example. This provides a way to verify the functional correctness of programs using higher-order iterators, while requiring lightweight annotations.
- We provide a freely available[1] implementation of our proposal in Creusot [3]. This tool is a state-of-the-art verification platform for safe Rust code, allowing users to verify programs by adding *contracts* to their functions. This implementation extends Creusot's handling of `for` loops to benefit from *structural invariants* provided by the specification of iterators. We evaluate it in Section 6 on several benchmarks.

2 Reasoning on Iteration

In this section, we present the general mechanism we use to specify iterators (Section 2.2), and how this kind of specifications are used in a `for` loop (Section 2.3). Before going in depth into these definitions, we give an introduction to the style of specification we use in this paper.

2.1 Specifications in Rust Programs

One important aspect of specifications of programs written in an imperative programming language such as Rust is their *memory model*, that is the way they handle pointers and mutations performed through them. Following previous work [6, 7, 3], we choose to leverage non-aliasing guarantees of Rust's type system. Because of the non-aliasing guarantees, a given memory location can be mutated through at most one reference at a given point in time, excluding all "spooky actions at a distance" that are customary with pointer aliasing. Therefore, it is possible to give a *mutable value semantics* [12] to Rust programs, meaning that, even though Rust programs can perform mutation of memory, they can be reasoned about in a purely applicative manner. As a result, the Rust type `Box<T>` of heap-allocated pointers, and the Rust type `&T` of read-only references are simply modeled by wrappers over values of type `T` in our specifications. As shown in previous work [3, 6, 7], this interpretation of Rust programs is key to verifying complex Rust programs, because it avoids the use of any kind of separation logic or dynamic frames, which are challenging to automate.

The handling of mutable references `&mut T` requires caution. Such references represent the temporary *borrow* of ownership of a memory location, so that mutations through such a reference will be observed by the initial owner once the borrow ends. To correctly model the propagation of mutations from the mutable reference to the borrowed variable, this style of specification models a mutable reference `r: &mut T` as a *pair* of a current value `*r` of type `T` (representing the current value pointed to by the reference) and of a *prophecy* `^r`, representing the value the reference *will point to when the borrow ends*.

[1] See supplementary material. The final version will link to the web page of the project.

This prophetic interpretation makes it possible to give precise specifications to functions that manipulate mutable references. For example, the function `push` adding a new element at the end of a vector in place can be specified as follows:

```
#[ensures(@^self == (@*self).concat(Seq::singleton(v)))]
fn push(&mut self, v: T);
```

Here, we use the operator `@` to refer to the *model* of a vector, i.e., the mathematical sequence of its elements. The postcondition thus ensures that the content of the final vector pointed to by `self`, denoted by `^self`, is modeled by the sequence of elements of the initial vector `*self`, concatenated with the new element `v`.

We sometimes use purely mathematical functions and predicates, annotated with the `#[logic]` and `#[predicate]` attributes.

We use Rust *traits* to give composable specifications to iterators. They are analogous to Haskell's typeclasses, enabling *ad-hoc polymorphism*. For example, an order relation can be specified as a trait containing both a mathematical order relation with its *laws* (reflexivity, antisymmetry and transitivity), and a program function specified as returning the value prescribed by the logical predicate.

To aid in specification and verification of code, we use *ghost code*, code which exists only during verification and has no influence on runtime behavior.

2.2 Specifying Iterators

In Rust, the mechanism of iterators is captured by a *trait* named `Iterator`, whose simplified definition can be given as:

```
trait Iterator { type Item;  fn next(&mut self) -> Option<Self::Item>;  }
```

This trait describes the *interface* an iterator should implement: an iterator should give a type `Item` of generated elements, and should implement a method `next` which optionally returns the next generated element, and possibly mutates in place the internal state of the iterator through the mutable reference `&mut self`.

As can be seen in Figure 1, we extend[2] the iterator trait with the purely logical *predicates* `produces` and `completed`. We require that any implementation of this trait satisfies the *laws* `produces_refl` and `produces_trans`: such laws are lemmas stated as specifications of purely logical functions (i.e., the preconditions should imply the postconditions). The `next` method is then specified thanks to the two predicates. Any implementation of the `Iterator` trait needs to give a logical definition of `produces` and `completed` predicates, prove the laws, give a program definition for `next` and finally prove that it satisfies its specification.

Iterators are specified as *state machines*: a value of an iterator type is seen as a state, the predicate `produces` defines the transition relation (noted $a \overset{s}{\rightsquigarrow} b$),

[2] In our implementation, to keep better compatibility with existing Rust code, we choose to define the iterator specification as a sub-trait of the `Iterator` trait from Rust's standard library, and to give the specification of `next` using Creusot's `extern_spec!` mechanism. For simplicity, we present it here as a unique trait: the main idea of the specification is the same.

```
trait Iterator {
  type Item;
  #[predicate] fn completed(&mut self) -> bool;
  #[predicate] fn produces(self, visited: Seq<Self::Item>, _: Self)
                    -> bool;
  #[law]    // I.e., ∀ a, a ⇝(ε) a
  #[ensures(a.produces(Seq::EMPTY, a))]
  fn produces_refl(a: Self);

  #[law]   // I.e., ∀ a b c, a ⇝(v) b ∧ b ⇝(w) c ⇒ a ⇝(v·w) c
  #[requires(a.produces(ab, b) && b.produces(bc, c))]
  #[ensures(a.produces(ab.concat(bc), c))]
  fn produces_trans(a: Self, ab: Seq<Self::Item>,
                    b: Self, bc: Seq<Self::Item>, c: Self);

  #[ensures(match result {
      None => self.completed(),
      Some(v) => (*self).produces(Seq::singleton(v), ^self)})]
  fn next(&mut self) -> Option<Self::Item>;
}
```

Fig. 1. Iterator trait extended with specification.

and the predicate `completed` (noted $completed(r)$) give the set of final states. The `completed` predicate takes a mutable reference `&mut self`, which allows us to specify mutations that happen when an iterator returns `None`. This added expressivity in the specification allows us to express properties of *unfused* iterators which may intermittently produce `None` during iteration. The `produces` transition relation is annotated with *sequences* of generated values rather than with unique values so that a user can reason about interesting properties of *sequences* as a whole rather than directly reasoning about the notion of transitive closure, which the automated solvers don't handle well. The price to pay is the laws of reflexivity and transitivity which the implementers have to prove.

2.3 Structural Invariant of `for` Loops

Part of the appeal of `for` loops is the *structure* they provide over the looping process. When a programmer sees a `for`, they can conclude that the body will be executed once for each element in the iterator. Unlike with `while` loops, it is not possible to decrement the loop index or otherwise perform unpredictable looping patterns. This informal reasoning can be formalized as a loop invariant, provided *structurally* by the `for` loop itself. The iterator at the i-th iteration is the result of calling `next` exactly i times on some initial state. In our formalism, given an initial iterator state `initial` and a current iterator state `iter`, we can state this invariant as $\exists p$, `initial` $\overset{p}{\rightsquigarrow}$ `iter`. This invariant holds for *any* `for` loop over *any* iterator: it can be derived from the laws `produces_refl` and `produces_trans`.

When using our extension to Creusot, every `for` loop benefits from this structural invariant: we change the way these loops are desugared into the more primitive `loop` construct, by adding ghost variables `init_iter` and `produced` and the new structural invariant `init_iter.produces(produced, iter)`. More precisely, a simple for loop `for x in iter {<body>}` is desugared into:

```
let init_iter = ghost! { iter };
let mut produced = ghost! { Seq::EMPTY };
#[invariant(structural, init_iter.produces(produced, iter))]
loop { match iter.next() {
  None => break,
  Some(x) => {
    produced = ghost! { produced.concat(Seq::singleton(x)) };
    <body> },
} }
```

Interestingly, the ghost variable `produced` can be referred to in a user invariant to relate the state of the loop with the iteration state. In the piece of code in Figure 2, we use a variable `count` to count the number of elements generated by an iterator, and use such an invariant to verify its intended meaning.

```
let mut count = 0;
#[invariant(count_is_n, @count == produced.len())]
for i in 0..n { count += 1; assert!(0 <= i && i < n); }
assert!(n < 0 || count == n);
```

Fig. 2. A simple for loop using ranges.

3 Examples of Specifications of Simple Iterators

In Section 2, we have presented a general framework to specify iterators and use them in `for` loops. In this section, we present several simple examples of iterators defined in this framework.

3.1 The `Range` Iterator

We start with a simple `Range` iterator, whose purpose is to iterate over the integers in a given range. The notation `a..b` used idiomatically in Rust is a syntactic sugar for this kind of iterators. The original definition from the Rust standard library is generic over the type of integers used, but, for the sake of simplicity, we use a monomorphic version here:

```
struct Range { start: usize, end: usize }
```

If `self.start` $\geq$ `self.end`, the `next` method returns `None`. Otherwise, it increments `self.start` and returns the initial value of `Some(self.start)`. Note that the upper bound of the range, `end`, is *excluded* in the iteration.

In order to instantiate our iterator specification scheme with `Range`, we use the `produces` and `completed` predicates defined by:

$$\begin{aligned} r \overset{v}{\rightsquigarrow} r' \quad &\triangleq \quad |v| = r'.\texttt{start} - r.\texttt{start} \land r.\texttt{end} = r'.\texttt{end} \\ &\quad \land\ |v| > 0 \Rightarrow r'.\texttt{start} \leq r'.\texttt{end} \\ &\quad \land\ \forall\, i \in [0, |v| - 1],\ v[i] = r.\texttt{start} + i \\ \mathit{completed}(r) \quad &\triangleq \quad *r = \hat{}\, r \land (*r).\texttt{end} \geq (*r).\texttt{start} \end{aligned}$$

Transitivity and reflexivity are easily verified.

Rust's standard library also contains ranges whose upper bound is included rather than excluded, and ranges without an upper bound. They can all be specified using similar techniques.

Note that with these definitions, the structural invariant of `for` loops directly implies that the loop index (the last produced value) is in the range. In addition, if the range is non-empty, one can deduce that the last iterated value is `end` $- 1$. These two properties usually require an additional invariant if the loop is encoded using the `while` construct. For an illustration consider Figure 2.

3.2 `IterMut`: Mutating Iteration Over a Vector

Our approach to iterators can be used to iterate over elements of a vector. But instead of presenting the simple case of a *read-only* vector iterator, we study a more general iterator, `IterMut`, permitting to both *read and write* vector elements while iterating; the simpler case of the read-only iterator uses the same ideas.

This iterator produces mutable references for each element of a vector in turn. The state of this iterator is a mutable reference to the *slice* (i.e., a fragment of a vector) of elements that remain to be iterated:

```
struct IterMut<'a, T> { inner: &'a mut [T] }
```

To define the production relation of `IterMut`, we use a helper function tr, which transposes a mutable reference to a slice into a *sequence* of mutable references to its elements. It defining property is:

$$|tr(s)| = |s| \ \land\ \forall\, i \in [0, |s| - 1],\ tr(*s)[i] = *s[i] \ \land\ tr(\hat{}\, s)[i] = \hat{}\, s[i]$$

With the help of tr, the `produces` and `completed` relations of `IterMut` are simple to express:

$$\begin{aligned} it \overset{v}{\rightsquigarrow} it' \quad &\triangleq \quad tr(it.\texttt{inner}) = v \cdot tr(it'.\texttt{inner}) \\ \mathit{completed}(it) \quad &\triangleq \quad *r = \hat{}\, r \land |*r| = 0 \end{aligned}$$

It means that the iterator it produces a sequence of mutable references, which must be the initial segment of $tr(it.\texttt{inner})$, into a final state it' such that

$tr(it.\texttt{inner})$ is the sequence of mutable references that are left to be generated. Such an iterator is completed when the inner slice is empty.

This compact specification is enough to reason about mutating through the returned pointers as in the following example:

```
#[invariant(all_zero, forall<i: Int> 0 <= i && i < produced.len()
                                    ==> @^produced[i] == 0)]
for x in v.iter_mut() { *x = 0; }
assert!{ forall<i: Int> 0 <= i && i < (@v).len() ==> @(@^v)[i] == 0 }
```

That is, we are able to prove with a simple loop invariant that this loop sets to 0 all the elements of the vector.

The reasoning that occurs to prove this program is as follows. First, at the end of a loop iteration, we know that the final value of the borrow `x` is equal to 0 since we have just written 0 and this value will not change since `x` goes out of scope. Together with the invariant of the preceding iteration, this is enough to prove that the invariant is maintained. Second, after the loop has executed, the final iterator state is empty, so we know `produced` contains the complete sequence of borrows to elements of v. But, thanks to the loop invariant, the prophetic value of each of these borrows is 0. So we can deduce that the final content of v is a sequence of zeros.

3.3 Iterator Transformers

Because all iterators implement the same trait `Iterator` which gives them a specification, we can easily build combinators which wrap and transform the behavior of an iterator.

It is important to note that, following Rust's standard library, these transformers are generic over the *type* of the underlying iterator; individual values of a type cannot have different predicates. While the verification tool cannot know the concrete definitions of `produces` or `completed` for the wrapped iterator, it knows it must satisfy the `Iterator` trait interface.

The simplest example is `Take<I>` (where `I` is another iterator), which truncates an iterator to produce at *most* n elements. The state of `Take<I>` is a record with two fields: a counter `n` for the remaining elements to take and an iterator `iter` to take from. The specification predicates of `Take<I>` are defined as follows:

$$
\begin{aligned}
it \overset{v}{\rightsquigarrow} it' \quad &\triangleq \quad it.\texttt{iter} \overset{v}{\rightsquigarrow} it'.\texttt{iter} \wedge it.\texttt{n} = it'.\texttt{n} + |v| \\
completed(it) \quad &\triangleq \quad (*it).\texttt{n} = 0 \wedge *it = \texttt{\^{}}it \\
&\quad \vee (*it).\texttt{n} > 0 \wedge (*it).\texttt{n} = (\texttt{\^{}}it).\texttt{n} + 1 \wedge completed(it.\texttt{iter})
\end{aligned}
$$

The subtle definition here is that of $completed(it)$: if the counter is 0, then `next` does nothing. But, following Rust's implementation, if the counter is not 0, then it is first decremented even if the call to the underlying iterator returns `None`.

Again, when instantiated to a specific underlying iterator *type*, we can substitute the definitions of $(\rightsquigarrow)$ and $completed(-)$ for the underlying iterator, to get

a concrete definition of these predicates for `Take<I>`, which are easier to handle by automated solvers.

Another common transformer is `Skip<I>`, whose goal is to *skip* the first n elements of an iterator. Similarly to `Take<I>`, the state is a record with two fields: a number `n` of elements to skip and an underlying iterator `iter`.

The $\rightsquigarrow$ relation of `Skip<I>` is defined as follows:

$$\begin{aligned} it \overset{v}{\rightsquigarrow} it' \quad &\triangleq \quad v = \varepsilon \wedge it = it' \\ &\quad \vee\ it'.\mathtt{n} = 0 \wedge |v| > 0 \wedge \exists\, w,\, |w| = it.\mathtt{n} \wedge it.\mathtt{iter} \overset{w \cdot v}{\rightsquigarrow} it'.\mathtt{iter} \end{aligned}$$

The first disjunct is needed to ensure reflexivity of ($\rightsquigarrow$). The second disjunct describes what happens after a non-empty sequence of calls. If we produced some sequence of elements v, then we must have been able to skip n elements first, which we existentially quantify over.

If the `Skip<I>` iterator is completed, the underlying iterator has also completed, but potentially after having generated some skipped elements that we existentially quantify over:

$$\begin{aligned} \mathit{completed}(it) \quad &\triangleq \quad \exists\, w\ i,\, (\hat{}it).\mathtt{n} = 0 \wedge |w| \leq (*it).\mathtt{n} \\ &\qquad \wedge (*it).\mathtt{iter} \overset{w}{\rightsquigarrow} *i \wedge \mathit{completed}(i) \wedge \hat{}i = (\hat{}it).\mathtt{iter} \end{aligned}$$

Using `Skip<I>` and `Take<I>` we are able to prove an algebraic property of iterators: if we take n elements and then skip n elements from that iterator, we must necessarily get the empty iterator.

```
assert!(iter.take(n).skip(n).next().is_none())
```

This property is easy to prove from the composition of both production relations.

4 Closures in Rust

Unlike traditional functional languages, Rust has no function type for closures. Two closures, even with identical bodies, are not of the same type: closures are each given a unique, anonymous type representing the captured environment. This design is motivated by the need to fully resolve closures during compilation: the compiler is always able to identify exactly which piece of code is used at every call site. To abstract over closures and write higher-order functions, Rust provides three traits that the closure type may implement: `FnOnce`, `FnMut`, and `Fn`. They describe the different ways a closure's environment can be passed during a call: by ownership, by mutable reference or by immutable reference. The compiler automatically provides the relevant instances when a user writes a closure.

Traditionally, verifying higher-order code with mutable state has needed *seperation logic* or *dynamic frames*, but because of Rust's mutable value semantics we can avoid these tools. Instead, we provide a specification for higher-order functions in first-order logic, which generates simple verification conditions (see code of Section 6). Specifically, we extend `FnOnce`, `FnMut`, and `Fn` with logical predicates that capture the pre- and post- conditions of closures. We begin by considering the simplest case, `FnOnce`:

```
pub trait FnOnce<Args> {
  #[predicate] fn precondition(self, a: Args) -> bool;
  #[predicate] fn postcondition_once(self, a: Args, res: Self::Output)
                  -> bool;
  #[requires(self.precondition(args))]
  #[ensures(self.postcondition_once(args, result))]
  fn call_once(self, args: Args) -> Self::Output;
}
```

The predicates `precondition` and `postcondition_once` refer to the specification added to the `call_once` method used to call the closure.

A call to a `FnOnce` closure *consumes* it. On the other hand, `FnMut` allows a mutable closure to be called multiple times. Here is our extended `FnMut` trait:

```
pub trait FnMut<Args> : FnOnce<Args> {
  #[predicate] fn unnest(self, _: Self) -> bool;
  #[ensures(self.unnest(self))]
  #[law] fn unnest_refl(self);
  #[requires(self.unnest(b) && b.unnest(c))]
  #[ensures(self.unnest(c))]
  #[law] fn unnest_trans(self, b: Self, c: Self);
  #[predicate] fn postcondition_mut(&mut self, _: Args, _: Self::Output)
                  -> bool;
  #[requires((*self).precondition(arg))]
  #[ensures(self.postcondition_mut(arg, result))]
  fn call_mut(&mut self, arg: Args) -> Self::Output;
[...] }
```

Because every `FnMut` closure is also an `FnOnce` closure, we can reuse the precondition predicate to specify `call_mut`. However, we need a new predicate for the richer postconditions that become possible: since the closure is called using a mutable borrow, the postcondition specify changes made to captured variables.

Rust compiles closures via closure conversion, the state of each closure becomes a struct holding references to all captured variables. However, this struct can only be modified in a restricted fashion: we can only mutate the values pointed by the captures, and not the captures themselves. In particular, this means the *prophecies* of captures remain constant. We capture this property in an *unnesting* predicate `F::unnest(a, b)`. It expresses that the prophecies in the state of type `F` have not changed from `a` to `b`. This property is both reflexive and transitive which we capture via laws. The unnesting predicate is essential to link the states of a closure throughout repeated calls. Without it we would lose track of the contained prophecies.

In addition to these predicates, our `FnMut` trait contains laws we elided: `unnest` is implied by `postcondition_mut`, and `postcondition_mut` is linked to the `postcondition` predicate of the `FnOnce` trait.

Finally, `Fn` imposes that the closure is immutable. Each call upholds the postcondition and leaves the state intact. Again, in the following, we elided laws relating `postcondition`, `postcondition_mut` and `postcondition_once`:

```
pub trait Fn<Args> : FnMut<Args> {
```

```
  #[predicate] fn postcondition(&self, _: Args, _: Self::Output) -> bool;

  #[requires((*self).precondition(arg))]
  #[ensures(self.postcondition(arg, result))]
  fn call(&self, arg: Args) -> Self::Output;
[...] }
```

5 A Higher-order Iterator Combinator: `Map`

The challenge with the specification of `Map` is proving the preconditions of the closure being called. `Map` treats the closure opaquely, it cannot tell what the concrete pre- and post- conditions are, the justification for the precondition must come from elsewhere. To help work through this, we use a thought experiment where we see `Map` implemented as a loop with a yield instruction to generate elements, in the style of e.g., Python generators:

```
fn map<I : Iterator, B, F: FnMut(I::Item) -> B>(iter: I, func: F) {
  for a in iter { yield (f)(a) }
}
```

To verify it, we need `f.precondition(a)` to be true at each iteration, so we need an invariant which implies it. This exposes the key property that must be true of our closure: the postcondition at iteration n must be able to establish the precondition for iteration $n+1$. In the vocabulary of iterators:

$$it \overset{s \cdot e_1 : e_2}{\rightsquigarrow} i' \to pre(*f, e_1) \to post(f, e_1, r) \to pre(\hat{}f, e_2)$$

This expresses that if we eventually produce an element e_1 which satisfies the precondition of the initial closure $*f$, then combined with the postcondition of f, we must be able to establish the precondition for the final closure $\hat{}f$ with the following element e_2. Quantifying over a prefix s in the iteration from a known initial state i ensures this property holds for all possible subsequent iterations.

To encode this property in `Map`, we use a *type invariant*, which allows specifying a property that values of a type must uphold. Here, the invariant states that (1) the precondition for the next call will be verified; (2) the preservation property above holds for the current state it; (3) these two invariants are reestablished if the underlying iterator returns `None` (this is usually trivial since the underlying iterator often is fused: it cannot generate new elements once it returns `None`); and (4) the type invariant of the underlying iterator holds.

These invariants are initially required as a precondition of the `map` method used to create the `Map` iterator. In order to be tackled by automated solvers, this verification condition need to be unfolded: it is therefore crucial that closures and their pre- and post- conditions are statically resolved thanks to the unique anonymous closure types in Rust.

The specification predicates for `Map` can now be stated:

$$
\begin{aligned}
it \overset{v}{\rightsquigarrow} it' \triangleq\ & \exists v'\, fs,\ |v'| = |fs| = |v| \wedge it.\texttt{iter} \overset{v'}{\rightsquigarrow} it'.\texttt{iter} \\
& \wedge (it.\texttt{func} = *fs[0] \wedge \hat{}fs[0] = *fs[1] \wedge .. \wedge \hat{}fs[n] = it'.\texttt{func}) \\
& \wedge \forall\, i \in [0, |v| - 1],\ pre(*fs[i], v'[i]) \wedge post(fs[i], v'[i], v[i]) \\
& \wedge unnest(it.\texttt{func}, it'.\texttt{func}) \\
completed(it) \triangleq\ & completed(it.\texttt{iter}) \wedge (*it).\texttt{func} = (\hat{}it).\texttt{func}
\end{aligned}
$$

In $\rightsquigarrow$, we quantify existentially over two pieces of information: the sequence of values v' produced by the underlying iterator and the sequence of *mutable references* of states fs that the closure traverses. We require that fs forms a chain, the final state of each element being the same as the current value of the following one. Finally, we require the closure pre- and post- conditions for every iteration, and that the first and last state are related by the unnesting relation. The definition of $completed(-)$, on the other hand, straightforwardly states that the underlying iterator is completed.

Interestingly, the user of this specification can use the precondition of the closure to encode closure invariants that she wishes to maintain along the iteration (as with loop invariants). This specification for `Map` allows us to specify many use cases, so long as the supplied closure is "history-free": its specification does not depend on the sequence of previously generated values, like in `x.map(|a : u32| a + 5)`. While this is certainly the most common usage of `map`, we sometimes need a more powerful specification.

Extending `Map` With Ghost Information. If we attempt to use the previous specification of `Map` to verify the counter-example of Section 1, we will rapidly encounter an issue: to establish that `cnt` properly counts the number of iterations would require a (manual) induction on the iterated sequence. While the prior specification allows the closure to specify the impact of an immediate call, it has no way of reasoning on the position in the iteration. In our prior thought experiment using a generator, we have no way of writing an invariant which depends on `produced`, as we allowed for usual `for` loops.

To make the verification of this kind of code simpler, we extend the signature of `Map` to the provide to the closure the sequence of elements generated by the underlying iterator since the creation of the mapping iterator object. This information does not change the behavior of the program: we make it *ghost*, so it can only be used in specifications.

The extended version, `MapExt`, is thus given an additional ghost field, `produced`, containing this sequence. The relation ($\rightsquigarrow$) is extended to account for this ghost information, by adding a conjunct stating that $it'.\texttt{produced} = it.\texttt{produced} \cdot v'$ and passing the additional ghost parameter $it.\texttt{produced} \cdot v'[0..i-1]$ to the pre- and post- conditions. The $completed()$ relation is extended by adding the conjunct $(\hat{}it).\texttt{produced} = \varepsilon$ (the produced field is reset when the iterator returns `None`). The type invariants are adapted accordingly.

Having this extra information avoids the need for an explicit induction after the fact to establish that we've properly counted the number of iterations. It suffices to look at the postcondition of the last call to `next`. While this example only makes use of the length of the sequence, this ghost information is useful in a wide variety of situations.

6 Evaluation

In this section we measure the performance of both the proofs of iterators and their clients, using the Creusot [3] tool for verification of Rust programs. It allows for verification of Rust programs, and require some annotations to verify the functional correctness of Rust programs. Verification is performed by translating annotated Rust code into a pure, first-order functional program. Then, Creusot uses Why3 [14] to generate verification conditions, which are discharged using automated solvers such as CVC5, Z3 or Alt-Ergo.

The results in Figure 3, were gathered using a Macbook Pro with an M1 Pro CPU and 32 GB of RAM, running macOS 12.2. Why3 was limited to using four provers simultaneously among Z3 4.11.2, CVC5 1.0.2, and Alt-Ergo 2.4.1.

Why3 supports *proof transformations*: manual tactics which can be used in combination with automated solvers. Because we wish to obtain ergonomic specifications which work well with automation, we minimize their use. Nevertheless, certain complex proofs required minor manual work, which we clearly indicate.

Iterator	LOC	Spec	Time	Fully auto.
`Range`	13	39	0.40	✓
`IterMut`	12	34	0.61	✓
`Map`	23	46	0.89	✗
`MapExt`	42	115	1.06	✗
`Skip<I>`	20	53	0.51	✗
`Take<I>`	17	43	0.40	✓
`Fuse`	29	51	0.52	✗

Benchmark	LOC	Spec	Time	Fully auto.
`all_zero`	5	3	0.43	✓
`skip_take`	3	2	0.40	✓
`counter`	12	4	0.55	✓
`concat_vec`	3	3	0.41	✓
`decuple_range`	9	3	0.64	✓
`hillel`	89	109	0.86	✓
`knights_tour`	89	55	1.15	✓

Fig. 3. Selected evaluation results. "LOC" counts the lines of program code, while "Spec" counts specification code and assertions. "Time" measures in seconds the time taken to solve the proofs. "Fully auto." determines whether manual tactics were used.

The left table in Figure 3 contains a selection of the iterators and combinators we have verified. The `Range`, `IterMut`, `Skip` and `Take` iterators are implementations of the iterators described in Sections 3.1 to 3.3. The `Fuse` combinator is responsible for transforming any iterator into a *fused* one, which will always return `None` after the first, never resuming iteration. Two versions of `Map` are provided, the first is the standard library `Map`, which is restricted to closures whose preconditions are 'history-free', the version in `MapExt` is provided with ghost information about previous calls as explained in Section 5.

Some manual proof steps were required to prove several iterators. For `Skip<I>` and `Fuse`, the manual tactics consist only of telling Why3 to access lemmas about sequences. For `Map` and `MapExt`, tactics were used to instantiate the existential quantifiers within the production relation.

We also verified several clients of iterators, in particular featuring combinations of several iterators. The example `decuple_range` maps a `Range`, multiplying elements by 10, collecting the results into a vector and verifying functional correctness; `counter` is an annotated version of the example in the introduction, verifying we can use mutable state to count the elements of an iterator; `concat_vec` uses `extend` to append an iterator to the end of a vector; `all_zero` is uses `IterMut` to zero every cell of a vector; `take_skip` checks that if we take n elements and then skip the n next elements of the resulting iterator, we must get `None`. `hillel` is a port of a prior Creusot solution to Hillel Wayne's verification challenges [15]. `knights_tour` is a port of a prior solution to the Knight's Tour problem. In both of these cases, updating the code to use `for`-loops and iterators actually *reduced* the number of lines of specification.

Because our lines of specification include the assertions which test functional properties, we believe the resulting overhead is reasonable, especially in our client examples. Additionally, our specifications for iterators seem to have low impact on verification times. We compared `hillel` and `knights_tour` with alternative versions that only differ by using traditional `while` loops instead of iterators, verification times are 0.91 and 1.14 respectively. This provides evidence that integrating our iterators does not cause prohibitive increases in verification time.

7 Related and Future Work

RustHorn [6] and RustHornBelt [7] show how the non-aliasing guarantees of Rust can be used for reducing the verification of Rust programs into the proof of first-order logic formulas. These works serve as theoretical foundations for Creusot [3], which we use to evaluate our specification scheme for iterators.

Prusti [1] is a semi-automatic verifier for Rust built on the Viper [9] separation logic verification platform. Prusti models mutable borrowing and ownership using separation logic permissions, unlike our choice of using a prophetic mutable value semantics. This leads to differences in the specification languages: whereas Creusot uses the `^` operator to reason about borrows, Prusti uses a notion called *pledges*. Pledges are assertions which must be true at the end of a specific lifetime. At the time of writing, pledges are not fully first-class in Prusti's specification logic: they are used through a kind of postcondition. In particular a ghost predicate like `produces` cannot contain a pledge. The `^` operator can be used anywhere in specifications, which allows us to give a natural specification to mutating iterators like `IterMut` (Section 3.2).

The verification of higher-order programs is a well-studied problem, Régis-Gianas and Pottier [13] show how to verify them using higher-order logic. Prusti supports closures by modeling them in Viper's separation logic [16]. Like our approach, Prusti transforms specifications of higher-order programs into first-

order verification conditions, but in separation logic. They introduce several constructs to specify closures: *history invariants*, *specification entailment*, and *call descriptions*. We instead enable users to refer to pre- and post- conditions of closures via a *trait*. While we not have the constructs Prusti provides primitively for closures, we believe these constructs can be encoded using our primitives, at the cost of lower ergonomics. Our approach is more expressive: unlike Prusti's call descriptions, we can distinguish the *order* of calls (see Section 5). Also, Prusti's approach for borrows makes it difficult to handle iterators such as IterMut.

Like us, Aeneas [5] verifies Rust programs by translation to a functional language, and targets traditional proof assistants such as Coq, or F*. They use a technique called *backward functions* to interpret mutable borrows. To our knowledge, Aeneas supports neither closures nor iterators.

The formalization of iterators is a well-studied subject with implementations in a variety of imperative and functional languages: WhyML [4], Eiffel [10], Java [8], and OCaml [11]. Of particular relevance is the approach developed by Filliâtre and Pereira [4], which specifies iterators in WhyML using a ghost field `visited : seq 'a` and two predicates `permitted : cursor 'a -> bool` and `completed : cursor 'a -> bool` where `cursor 'a` is an iterator for values of type `'a`. This work leverages Why3's regions system to distinguish individual cursors over time. In contrast, in our context, we lose *object identity*: there is no way to identify that two iterator values are two successive states of the same iterator. We thus generalize this approach to our setting by explicitly providing pre- and post- states in `produces`. Our work is also more expressive: we specify and verify higher-order iterators using potentially mutable closures, which are ruled out by Why3's region system. The framework of iteration described by Polikarpova, Tschannen, and Furia [10] is limited to finite, deterministic iteration: the user must provide up front the sequence of abstract values the iterator will produce. Pottier [11] presents an implementation of iterators for a hash map written in OCaml. They do this by working in the separation logic CFML [2], utilizing Coq's powerful but manual reasoning mechanisms for theorem proving. While Pottier does not provide a general specification of iterators (*cascades*) with mutable state, CFML should permit it, though usage may require a challenging proof.

Future Work. While we have specified and proved key iterators, many more remain. The `filter` transformer is interesting as each call to `next` may make an unbounded number of steps with the underlying iterator using the provided *mutable* closure. Rust provides a hierarchy of traits that further refine iterators like `DoubleEndedIterator`, and `ExactSizeIterator`. The recent integration of *generic associated types* enables new, more flexible forms of iteration like *lending iterators*. We believe these would naturally integrate into our framework, but remain to be done. Finally, while we believe we have developed a correct, and simple approach to specify closures, the ergonomics leave much room for improvement. Improving this will help make our specifications more concise and user-friendly. In particular, we would like to explore automatic inference of pre- and post-conditions of simple closures.

References

[1] Vytautas Astrauskas et al. "The Prusti Project: Formal Verification for Rust". In: *NASA Formal Methods*. Vol. 13260. LNCS. 2022. DOI: 10.1007/978-3-031-06773-0_5.

[2] Arthur Charguéraud. "Characteristic formulae for the verification of imperative programs". In: *ICFP*. 2011. DOI: 10.1145/2034773.2034828.

[3] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. "Creusot: A Foundry for the Deductive Verication of Rust Programs". In: *ICFEM*. Vol. 13478. LNCS. 2022. DOI: 10.1007/978-3-031-17244-1_6.

[4] Jean-Christophe Filliâtre and Mário Pereira. "A Modular Way to Reason About Iteration". In: *NASA Formal Methods*. Vol. 9690. LNCS. 2016. DOI: 10.1007/978-3-319-40648-0_24.

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

[6] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. "RustHorn: CHC-based verification for Rust programs". In: *TOPLAS* 43.4 (2021), pp. 1–54. DOI: 10.1145/3462205.

[7] Yusuke Matsushita et al. "RustHornBelt: A Semantic Foundation for Functional Verification of Rust Programs with Unsafe Code". In: *PLDI*. 2022. DOI: 10.1145/3519939.3523704.

[8] João Mota, Marco Giunti, and António Ravara. *On Using VeriFast, VerCors, Plural, and KeY to Check Object Usage*. 2022. URL: http://arxiv.org/abs/2209.05136.

[9] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *VMCAI*. Vol. 9583. LNCS. 2016. DOI: 10.1007/978-3-662-49122-5_2.

[10] Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. "A Fully Verified Container Library". In: *Formal Aspects of Computing* 30.5 (2018). DOI: 10.1007/s00165-017-0435-1.

[11] François Pottier. "Verifying a Hash Table and Its Iterators in Higher-Order Separation Logic". In: *CPP*. 2017. DOI: 10.1145/3018610.3018624.

[12] Dimitri Racordon et al. "Implementation Strategies for Mutable Value Semantics." In: *J. Object Technol.* 21.2 (2022), pp. 2–1.

[13] Yann Régis-Gianas and François Pottier. "A Hoare Logic for Call-by-Value Functional Programs". In: *MPC*. Vol. 5133. LNCS. 2008.

[14] The Why3 development team. *The Why3 verification platform*. URL: https://why3.lri.fr/.

[15] *The Great Theorem Prover Showdown*. Hillel Wayne. Apr. 25, 2018. URL: https://www.hillelwayne.com/post/theorem-prover-showdown/ (visited on 10/14/2022).

[16] Fabian Wolff et al. "Modular Specification and Verification of Closures in Rust". In: *OOPSLA*. 2021. DOI: 10.1145/3485522.