Compositional Reasoning about Advanced Iterator Patterns in Rust

Conference Paper

Author(s):
Bílý, Aurel; Hansen, Jonas; Müller, Peter; Summers, Alexander J.

Publication date:
2023-10-24

Permanent link:
https://doi.org/10.3929/ethz-b-000643457

Compositional Reasoning about Advanced Iterator Patterns in Rust

Aurel Bílý
Department of Computer Science
ETH Zurich, Switzerland
aurel.bily@inf.ethz.ch

Jonas Hansen
Department of Computer Science
ETH Zurich, Switzerland

Peter Müller
Department of Computer Science
ETH Zurich, Switzerland
peter.mueller@inf.ethz.ch

Alexander J. Summers
University of British Columbia
Canada
alex.summers@ubc.ca

Abstract

Iteration is a control-flow mechanism that consists of repeating statements. Iterators provide an object-oriented abstraction to iteration. Simple iterators confer access to elements of a data structure, but modern languages such as Rust, Java, and C# generalise iteration far beyond this simple use case, allowing iterators to be parameterised with closures (which can modify their captured state as a side effect) and supporting the composition of iterators to form iterator *chains*, where each iterator in the chain consumes values from its predecessor and produces values for its successor. Such generalisations pose four major challenges for modular specification and verification of iterators and the client code using them: (1) How can parameterised iterators be specified modularly and their (accumulated) side effects reasoned about? (2) How can the behaviour of an iterator chain be derived from the specifications of its component iterators? (3) How can proofs about such iterators be automated? (4) How to integrate a concrete methodology into the standard library, without requiring the client code to change?

We present a methodology for the modular specification and verification of advanced iteration idioms with computations affecting captured state as a side effect. It addresses the four challenges above using a combination of inductive two-state invariants, higher-order closure contracts, separation logic-like ownership, and a novel type of out-of-band contracts. We implement our methodology in a state-of-the-art SMT-based Rust verifier. Our evaluation shows that our methodology is sufficiently expressive to handle advanced, idiomatic iteration patterns and requires modest annotation overhead.

1 Introduction

Iterators are a ubiquitous programming idiom used to abstractly and simply enumerate the elements of a collection. Verification of such simple use cases comes down to proving that an iterator yields all elements of a collection in a specified linear order and showing that the used iterator does not get invalidated e.g. by concurrent modifications of the underlying collection [12].

Modern programming languages such as C#, Java, Python, and Rust support iteration patterns that go far beyond the traditional use case. For instance, iterators can perform computations over streams of values, such as computing a moving average; the computation itself is typically *parameterisable* by custom code using closures. In general-purpose languages, both the iterators and the closures may perform side effects, e.g. to modify the data structure in-place or to accumulate results based on the values seen so far; code which uses such iterators depends on properties of this modified state.

Moreover, it is increasingly prevalent for languages to support the *composition* of iterators, where an iterator processes values produced by another iterator (rather than obtained directly from a collection). Following Rust terminology, we call this kind of iterator an *iterator adapter.* Adapters may be composed into *iterator chains* that act as composite iterators.

Modern imperative languages provide a variety of iterator adapters, e.g. Java's Streams API, C#'s LINQ, Python's iterables, and Rust's iterator adapters all support potentially-side-effectful variants of common functional programming operations such as `filter`, `fold`, and `map`. These languages also allow creating custom iterators and iterator adapters.

The following *Rust* example illustrates an iterator chain:

```
let mut s = 0;
some_vector.iter()      // iterate over a Vec<i32>
  .map(|x| {
    s += x; s })        // running totals of seen values
  .filter(|x| x < 10)   // keep totals smaller than 10
  .collect::<Vec<_>>(); // collect results into a vector
```

The function `iter` yields a traditional iterator for the underlying vector, which is the input for the iterator chain. The `map` adapter is parameterised with a closure, which mutates the captured variable `s`. The subsequent `filter` adapter, also parameterised with a filter criterion, removes elements, and finally `collect` stores the produced elements in a new vector.

Such advanced iteration patterns lead to concise and readable code, but they also pose several challenges for modular specification and verification. Modularity is important to

give correctness guarantees for libraries, to make verification scale, and to reduce the effort for re-verification when parts of a codebase change. We identify four key challenges:
(1) Modularly specifying parameterised iterators. The behaviour of iterators such as `map` depends on their argument closures, which vary for different calls. Modularity requires a specification for `map` which is somehow parameterisable by the behaviour of its argument closure, which may itself have side effects (cf. line 4 of our example). Such a specification must capture not only the values produced by the iterator but also the *accumulated side effects* of all calls to the iterator and its closure parameter. Variables mutably captured by such a closure, such as `s` in our example, make these side effects visible to client code when the iteration is over. For example, if `some_vector` contained values 6, 2, 9 then `map`'s specification should imply that the iterator produces the partial sums 6, 8, 17, and that the final value of `s` is 17.
(2) Modular reasoning about iterator chains. Modular reasoning also requires that we can deduce the behaviour of an iterator chain from the specifications of its component iterators. Verification of a chain must not require re-verification of its component iterators (whose code might not even be available); the specification of each iterator should be agnostic as to the source of values it consumes and to the downstream processing of the produced values. In our example, we want to prove that the result is [6, 8] from general-purpose specifications of each of the intermediate adapters.
(3) Automating proofs about iterator chains. The behaviour of iterator chains depends on the behaviour of their component iterators, which in turn depend on their argument closures. This hints at the need for higher-order logics (specifications which depend on specifications), which can greatly complicate proof automation; we instead aim for proofs which can be automated with first-order SMT solvers.
(4) Supporting standard library and existing code. All existing Rust code with custom iterators uses the `Iterator` standard library trait. For iterator specifications to work with existing code, we want to equip types implementing this trait with these specifications, but not necessarily *all* such types in a given codebase.

1.1 Related language features

In this paper, we use the terms *iterator*, *iterator adapter*, and *iterator chain* consistently with established Rust terminology. The Rust `Iterator` trait corresponds to Java's `Iterator` interface or C#'s `IEnumerator` interface. However, in Rust, the same `Iterator` trait is also implemented by various operations applied on top of a previous iterator. These are called iterator `adapters`, although the adaptation in question is from `Iterator` (and additional parameters) again to `Iterator`, potentially with a different output type. Higher-order functions such as `map` and `filter`, prevalent in functional programming, exist in Rust as iterator adapters, and are comparable to Java's streams API. The behaviour of a chain composed of multiple adapters is *not* equivalent to that of a composition of higher-order functions, however, due to the lazy nature of Rust iterators.

1.2 State of the art

Prior work on iterator verification in Rust includes recent work by Denis et al. [7] for the Creusot verifier [8]; their iterator support extends earlier work by Pereira [16]. Our approach extends Pereira's in a different way, and was developed concurrently [6, 11] with the Creusot-based technique.

Compared with the Creusot-based approach ([7]), our work provides alternative specification primitives and techniques, proof automation and debugging at the level of the Rust language, and integration with the Rust standard library `Iterator` trait and its implementations: existing Rust code can benefit from our work without changing its dependencies. We provide more-detailed comparisons in Sec. 7.

Verification of Rust higher-order functions such as `fold` in Rust was partially addressed by Wolff et al. [20], who provide modular reasoning for side-effectful *closures*, such as the argument to `map` in our example. This prior work does not provide specification or verification support for iterators and adapters that *use* closures in well-known but functionally complex ways (e.g. as an argument to a `fold`).

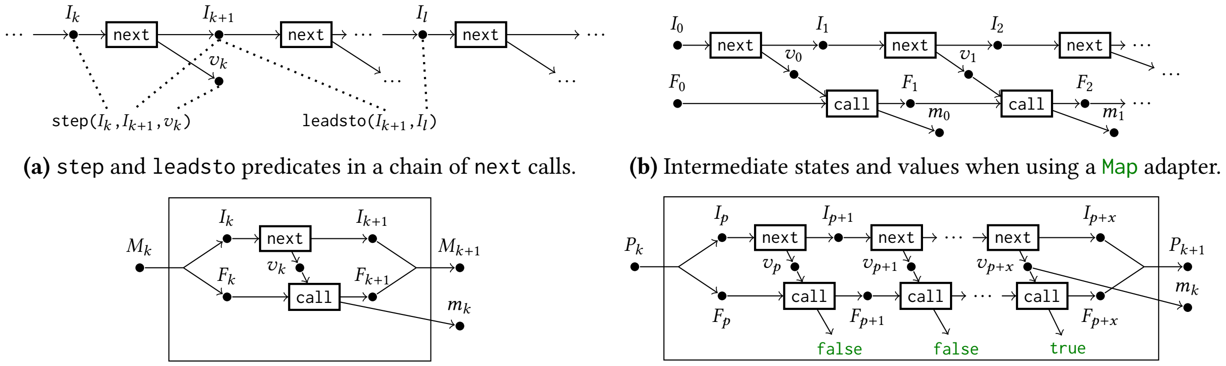
1.3 This work

We present a modular specification and verification technique for advanced iteration patterns that addresses the four challenges above. We present our technique in the context of Rust, whose ownership type system complements our methodology by preventing concurrent modifications of an iterated-over data structure, or undesirable interference between iterators. However, our technique would apply equally to other languages if augmented with an alternative ownership-like technique such as separation logic [17]. Indeed, the underlying implementation encodes our methodology into the Viper verification language.

The main contributions of our paper are:

- We present a specification and verification methodology for general side-effectful iterators (→ Sec. 2, 3).
- We demonstrate how to use this methodology to reason about the effects and resulting values of complex iterator chains (→ Sec. 4).
- We show how to express these specifications modularly for *existing* iterator hierarchies (→ Sec. 5).
- We implement our work as a prototype extension of the Prusti verifier [3] and demonstrate its expressiveness on several challenging examples (→ Sec. 6).

Additional background and details on the soundness of our approach can be found in the extended technical report [6].



(a) step and leadsto predicates in a chain of next calls.

(b) Intermediate states and values when using a Map adapter.

(c) State diagram for a call to Map::next.

(d) State diagram for a call to Filter::next.

Figure 1. State diagrams for iterators.

2 Methodology

In this section we introduce a general-purpose methodology for reasoning about iterators. We describe a model of iterators (→ Sec. 2.1), introduce four predicates that allow one to specify the behaviour of iterators (→ Sec. 2.2), and present the proof obligations needed to verify that an iterator implementation satisfies this specification (→ Sec. 2.3). Finally, we summarise how our methodology addresses the challenges outlined in the introduction (→ Sec. 2.4).

2.1 Iteration model

We provide informal diagrams illustrating key aspects of our model of iterators in Fig. 1. An *iterator* can be queried repeatedly for values using a next method, producing a sequence of values ($v_0, v_1, \ldots$ in Fig. 1a). Calls to the next method change the internal state of the iterator ($I_0, I_1, \ldots$ in Fig. 1a).

The sequence of produced values can be finite or infinite. In the former case, next yields a designated value $\bot$ to signal that the iteration has completed[1]. We assume that once an iterator has completed, it will remain completed, which is the case in many programming languages[2].

Iterator adapters are iterators which use the results of a previous iterator to produce their values. For instance, the Map adapter applies a function f to the outputs $v_0, v_1, \ldots$ of the previous iterator to produce its result values $m_0, m_1, \ldots$. The function f may itself be a closure with potentially mutable captured variables, which means it has its own state ($F_0, F_1, \ldots$), as illustrated in Fig. 1b.

The overall state of the Map adapter M_i comprises (and encapsulates) the state of the previous iterator I_i and that of its argument closure F_i. Map's next method internally manipulates both state components as illustrated in Fig. 1c.

Each call to Map's next method triggers exactly one call to the predecessor's next method. Other adapters have more complex behaviours: the Filter (Fig. 1d) adapter calls its predecessor's next method until the provided value satisfies the filter criterion determined by an argument closure (or the predecessor has completed). A specification and verification methodology for advanced iterators must therefore be able to capture the evolution of such composite iterator states.

2.2 Specification components

Our methodology uses four specification components to specify the states of and values returned by iterators. We introduce them here using a mathematical notation and show a concrete syntax in Rust later, in Sec. 3.1.

As in prior work [16], we associate each iterator with two specification-only functions to specify the *returned values*. Function produced yields the sequence of elements returned *so far*: produced(I_k) $= [v_0, v_1, \ldots, v_{k-1}]$. Function done yields true iff the iterator has completed. In other words, a call to next with the state I_k returns $\bot$ iff done(I_k) holds.

The evolution of an iterator's *state* across a single call to next is characterised using the third component of our methodology: a two-state predicate step. For a call to next with the initial iterator state I_k, the updated iterator state I_{k+1} and the returned value v_k, step(I_k, I_{k+1}, v_k) must hold.

In general, reasoning about the *accumulated* effects of an iterator concerns an *unbounded* number of next calls. This motivates our fourth key component: we associate each iterator with a predicate leadsto that represents an *inductive*, *two-state* invariant. This invariant relates the current iterator state to *any previous* iterator state. It represents the reflexive, transitive closure of the two-state postcondition step (which relates *consecutive* iterator states): leadsto(I_k, I_l) $\Leftrightarrow (\forall i \cdot k \leq i < l \Rightarrow$ step(I_i, I_{i+1}, _)) for any $0 \leq k \leq l$.

When an iterator is used in a loop, we can naturally make use of leadsto in the loop invariant: leadsto(I_0, I_k) must hold at every iteration, where I_0 is the state of the iterator before the loop, and I_k is the current state.

[1] In Rust, this corresponds to an Option::None.

[2] In Rust, this kind of iterator is a FusedIterator, a trait implemented by the vast majority of Rust's standard library iterators.

2.3 Proof obligations

To use our methodology, programmers need to define the four specification components described in the previous subsection for each concrete iterator implementation. Our methodology then imposes the following proof obligations. First, we check that leadsto includes the reflexive, transitive closure of step. If this well-formedness check fails, the program is rejected. Second, we check whether the definitions of the four components correctly reflect the behaviour of the iterator implementation. This is done by verifying that the implementation of the next method satisfies the following five postconditions. In these conditions, I_k, I_{k+1}, and v_k denote the prestate, poststate, and result value of next, respectively. For simplicity we assume that the iterator has no methods that modify its state other than next.

- $Q_1 \equiv \texttt{step}(I_k, I_{k+1}, v_k)$: step reflects the behaviour of a single call to next.
- $Q_2 \equiv \texttt{done}(I_k) \Leftrightarrow v_k = \bot$: done reflects correctly whether the iterator returns a value.
- $Q_3 \equiv \texttt{done}(I_k) \Rightarrow \texttt{produced}(I_{k+1}) = \texttt{produced}(I_k)$: if the iterator had already completed, the produced sequence is left unchanged.
- $Q_4 \equiv \neg\texttt{done}(I_k) \Rightarrow \texttt{produced}(I_{k+1}) = \texttt{produced}(I_k) +\!\!+ [v_k]$: if the iterator had *not* completed, the produced sequence is extended by the returned value.
- $Q_5 \equiv \texttt{done}(I_k) \Rightarrow \texttt{done}(I_{k+1})$: monotonic completion.

Importantly, these postconditions also allow *client code* to reason about the results and effects of calls to an iterator.

2.4 Challenges revisited

In this subsection we summarise how our methodology addresses the first three challenges presented in Sec. 1.

(1) Modularly specifying parameterised iterators. We capture the result and effects of a single call to next using the two-state step predicate. When the iterator is parameterised with a closure (as in the Map example), we use a generalisation of *call descriptions* [20] to abstractly and generically describe the results and effects of calls to the closure. A call description, written F ⇝ |args| { pre } { post } expresses that a call to F *has happened*, the assertion pre held in the prestate of that call, the assertion post held in the poststate; the call's arguments are bound to the names args and the return value to a reserved name result in the assertions pre and post.

We capture the *accumulated* results and effects of an iteration using the produced sequence and the reflexive, transitive leadsto predicate. When the iterator state includes closures, we use two-state invariants on the *closure* state (again following [20]) to express how they evolve during an iteration.

Postconditions for the next method tie together these predicates with concrete iterator implementations. They allow clients to determine the behaviour of a single call to next (using step) as well as of a full iteration (using leadsto). In particular, clients aware of the concrete argument closure passed to the iterator can use these postconditions to determine both the sequence of returned values and all relevant accumulated side-effects (e.g. on closure-captured state).

(2) Modular reasoning about iterator chains. To handle chaining, the specification of iterator adapters may refer to the step and leadsto predicates of the previous iterator in the chain, making the adapter's specification *parametric* in that of the input iterator. In particular, neither the input iterator nor the adapter need to be re-verified when forming an iterator chain.

(3) Automating proofs about iterator chains. We handle this challenge with an encoding of our methodology into first-order logic components suitable for verification with SMT-based verifiers. The four predicates defined above are encoded as uninterpreted functions; their argument states are encoded via a mathematical abstraction of heap-dependent values known as *snapshots* in Prusti [20]; similar abstractions are used in the Creusot verifier, as well as commonly in symbolic execution tools for separation logics [13, 18, 19]. A suitable first-order encoding of call descriptions is provided in prior work [20]. We discuss more details of our first-order encoding in our extended technical report [6].

3 Specifying individual iterators

In this section, we instantiate the methodology introduced in Sec. 2 in Rust (→ Sec. 3.1) and illustrate its use on a simple, unchained iterator (→ Sec. 3.2). The specification language used corresponds to the Prusti specification language.

3.1 Defining the specification components

In Rust, iterators are types which implement the Iterator trait. This is achieved with an explicit syntactic declaration (**impl** Iterator **for** SomeType) which also declares the element type and provides the definition of one required method: next. The implementation of this method can modify the current state of the iterator and must output an Option which contains the next element, or None to signal completion. The Iterator trait also defines a large number of other methods with default (rarely overridden) implementations.

For now, let's imagine adding the four key components of our specifications (cf. Sec. 2.2) as ghost functions of the Iterator trait (see below). In practice this would force that *each and every* concrete iterator type in a codebase would need to provide concrete definitions for these components. We explain our more-flexible opt-in approach in Sec. 5.

```rust
trait Iterator {
  fn produced(&self) -> GhostSeq<Self::Item>;
  fn step(p: &Self, c: &Self,
          r: &Option<Self::Item>) -> bool;
  fn leadsto(p: &Self, c: &Self) -> bool;
  fn done(&self) -> bool; }
```

The `produced` method is a getter[3] for a ghost sequence of values. The two-state predicates `step` and `leadsto` take references to two copies (a previous version and a current version) of the iterator type. `step` additionally takes a reference to the returned value, represented in Rust as an `Option` to allow for `None` to signal the end of iteration. Finally, `done` defines the stopping condition. Since these methods are ghost code, their definitions may use non-executable constructs, such as quantifiers, in implementations of the `Iterator` trait.

Finally, we express the proof obligations from Sec. 2 as postconditions of the `next` method, which must be satisfied by each implementation of `next`. Here, `&result` denotes a shared reference to the value returned by the method.

```
trait Iterator {
(Q1) #[ensures(old(&self).step(&self, &result))]
(Q2) #[ensures(old(self.done()) == (result === None))]
(Q3) #[ensures(old(self.done()) ==> self.produced() ===
        old(self.produced()))]
(Q4) #[ensures(!old(self.done()) ==> self.produced() ===
        old(self.produced()).append(result))]
(Q5) #[ensures(old(self.done()) ==> self.done())]
     fn next(&mut self) -> Option<Self::Item>; }
```

This specification uses logical equality ===, which Prusti provides to mean equality on snapshots; similar logical equalities are used in other Rust verifiers; Rust doesn't require that all types implement the language's equality ==.

3.2 Example

We illustrate our methodology using a simple iterator returning a range of consecutive numbers, written as follows:

```
struct Counter(i32, i32); Ⓐ
impl Counter {
  #[ensures(Iterator::leadsto(&result, &result))]
  fn new(end: i32) -> Self { ... } Ⓑ }
impl Iterator for Counter { Ⓒ
  type Item = i32; Ⓓ
  fn next(&mut self) -> Option<Self::Item> { Ⓔ
    if self.0 > self.1 { return None; }
    self.0 += 1;
    Some(self.0 - 1) } }
```

The code above first Ⓐ defines a type which represents our iterator. Its state contains two `i32`-typed variables: its current position and its stopping point. Ⓑ defines a convenience constructor for the counter, which must ensure that the two-state invariant `leadsto` is (reflexively) satisfied to begin with. Ⓒ is the declaration that marks `Counter` to be an iterator, consisting of a declaration of the type of elements emitted by this iterator Ⓓ, and an implementation of the `next` method Ⓔ. The latter checks if the limit has been reached yet: if so, no more items are emitted (`None` is returned), otherwise the internal position is updated and its old value is returned (wrapped in `Some`).

To specify this iterator, we give definitions of our four methodology components, starting with `done` and `step`:

```
fn done(&self) -> bool { self.0 > self.1 }
fn step(p: &Self, c: &Self,
        r: &Option<Self::Item>) -> bool {
  !p.done() ==> (c.0 == p.0 + 1) }
```

The two-state postcondition `step` defines how the state is updated. Here, `p` and `p` refer to the iterator snapshot before and after the execution of `next`. The remaining properties are specified in the definition of `leadsto`:

```
fn leadsto(p: &Self, c: &Self) -> bool {
Ⓐ    p.1 == c.1
Ⓑ  && p.0 <= c.0
Ⓒ  && 0 <= c.0 && c.0 <= c.1 + 1
Ⓓ  && c.produced().len() == c.0 as usize
Ⓔ  && forall(|x: i32| 0 <= x && x < c.0 ==>
         c.produced()[x as usize] == x) }
```

In this definition, Ⓐ the upper bound remains constant. Ⓑ the current position is monotonically increasing, and Ⓒ it remains within bounds. Ⓓ the number of produced elements is equal to the current position. Finally, Ⓔ the value of every produced element can be defined by its position in the sequence.

With such a definition, we can prove results about `Counter`, even across unboundedly many calls, for example:

```
let mut counter = Counter::new(90);
let val_pre = counter.next().unwrap();
assume!(n < 80);
for i in 0..n { counter.next().unwrap(); }
assert!(counter.next().unwrap() > val_pre);
```

At this point we have specified a simple iterator. We have used all four of our methodology's components defined in Sec. 2. Although the approach is relatively heavy for `Counter` (the definition of `leadsto` in particular), we will shortly see it pays off when considering more advanced (and idiomatic) cases. A similar specification can be used for iterating over a slice or `Vec`.

4 Specifying iterator chains

In this section, we introduce our technical solutions to the first two challenges described in Sec. 1: modular reasoning about parameterised iterators and iterator chains. We first introduce iterator adapters in Rust (→ Sec. 4.1), then specify the standard library type `Map` (→ Sec. 4.2). In the extended technical report, we build up the specification more gradually, starting from a simple number-doubling iterator adapter.

4.1 Iterator adapters in Rust

In Rust, iterator adapters are types which wrap an instance of the `Iterator` trait while implementing the `Iterator` trait themselves. When the adapter's `next` method is invoked, it calls the previous iterator's `next` method some number of times, adapting the results. As an example, the `Map` adapter

[3] Traits in Rust cannot declare fields or properties, so we must use a method.

applies a closure to every element produced by the previous iterator, while the Filter adapter uses a closure as a logical predicate to decide whether each value from the previous iterator should be returned or not.

The Rust standard library provides many such adapters for a variety of common use cases. The iterator chain in the code example shown in Sec. 1 is actually composed of iterator adapters, albeit hidden behind some convenience syntax (e.g. `x.map(...)` wraps the iterator `x` in a Map adapter).

4.2 Specifying Map

The Map iterator adapter takes values from a previous iterator and applies a user-supplied closure to each of them. To this end, the Map type has two type parameters: one to represent the previous iterator in the chain and one to represent the closure[4]. A Map instance encapsulates (owns) its closure (field `f`) and its previously-chained iterator (field `prev`).

```
struct Map<I, F> {
  prev: I, // the wrapped previous iterator
  f: F, }  // the closure parameter
```

Map's `next` method applies its closure to each element[5]:

```
fn next(&mut self) -> Option<Self::Item> {
  self.prev.next().map(&mut self.f) }
```

To provide a generic specification of Map, we must account for the side effects of the closure call, even when the exact type of the closure is unknown; this property of side effects on the *closure*'s mutable state cannot be captured using the `produced` sequences alone. Instead, we use a call description to connect the effect of an iteration step to the effects of a call to the closure itself (we will refine this definition later):

```
fn step(p: &Self, c: &Self,
        r: &Option<Self::Item>) -> bool {
Ⓐ !p.prev.done() ==>
 Ⓑ FnMut::call_mut ⇝ |cl_self, arg|
   Ⓒ {    p.f === cl_self
   Ⓓ   && arg === c.prev.produced().last() }
   Ⓔ {    c.f === cl_self
   Ⓕ   && r === Some(result) }) }
```

For each step we know (if the previous iterator had not completed Ⓐ) that Ⓑ there was a call to the closure originally stored Ⓒ in field `f` of the Map, and Ⓔ the updated closure ends up stored in field `f` of the Map. For the call, Ⓓ the argument given to the closure was the last element yielded by the previous iterator, and Ⓕ the element yielded by Map (`r`) was the result of the closure (`Some(result)`).

4.3 Transitive side effects

The specification in the previous subsection refers to the values produced by the previous iterator, but does not account for changes to its state. To address *transitive* side effects of `next` calls (i.e. those on closures earlier in the iterator chain), we must directly describe the calls to the previous iterator. Our call description feature can also be used to describe this necessary connection, this time relating each change to the current iterator to the corresponding pre- and poststate of a call to `next` on the *previous* iterator, all the while keeping the specification generic with respect to the previous iterator's type. The following definition of `step` exemplifies this powerful idiom, using two nested call descriptions:

```
fn step(p: &Self, c: &Self,
        map_res: &Option<Self::Item>) -> bool {
Ⓐ Iterator::next ⇝ |it_self|
  Ⓑ { p.prev === it_self }
  Ⓒ { let prev_res = result;
  Ⓓ     c.prev === it_self
  Ⓔ   && (prev_res.is_some() ==>
          Ⓕ FnMut::call_mut ⇝ |cl_self, arg|
              {    p.f === cl_self
                && Some(arg) === prev_res }
              {    c.f === cl_self
                && map_res === Some(result) } )
  Ⓖ   && prev_res.is_none() ==> map_res.is_none() } }
```

In this version of `step`, we begin by Ⓐ describing that a call to the previous iterator *will happen*, where Ⓑ the original state of the previous iterator is stored in the field `prev` of the original state of the Map, and Ⓓ the new state of the previous iterator is as stored in the field `prev` of the resulting state of Map. To avoid confusion, we Ⓒ used a let expression to name the result of the previous iterator `prev_res`; the result returned from the Map is `map_res`. Ⓔ If the previous iterator returned an element, we Ⓕ describe how this relates a call to the closure, as before. Ⓖ Otherwise, no element is returned from the Map iterator either.

The overall structure of this definition of `step` mirrors the Map model presented in Fig. 1c: each call description corresponds to one box in the diagram, and values from the previous iterator flow through the closure.

4.4 Unboundedly many steps

Finally, reasoning about *unboundedly many iteration steps* (e.g. when calling `next` in a loop), simply requires us to suitably define `leadsto` for Map. To store the intermediate states of the closures and the iterators, we add a ghost sequence field `st: GhostSeq<(F, I)>` to Map, where every element is a tuple containing a closure state and an iterator state.

```
fn leadsto(p: &Self, c: &Self) -> bool {
      c.produced().len() == c.prev.produced().len()
  Ⓐ && p.st.is_prefix_of(c.st)
  Ⓑ && c.st.len() == c.produced().len() + 1
  Ⓒ && c.st.last() === (c.f, c.prev)
```

[4] The actual declaration has a third type parameter to represent the return type of the closure, omitted here for brevity.

[5] The `map` method on an Option type (unrelated to iterators) applies the given closure on the value contained in a Some, but leaves None intact.

```
    && forall(|idx: usize| idx < c.produced().len() ==>
    Ⓓ I::next ⇝ |iter_self|
        {    c.st[idx].1 === iter_self }
        {    c.st[idx + 1].1 === iter_self
          && result === Some(c.prev.produced()[idx])
          && FnMut::call_mut ⇝ |cl_self, arg|
              {    c.st[idx].0 === cl_self
                && arg === c.prev.produced()[idx] }
              {    c.st[idx + 1].0 === cl_self
                && result === c.produced()[idx] }}) }
```

In this definition of leadsto, we establish a connection between the ghost sequence st and the concrete data stored in the Map struct. In particular, Ⓑ states there are as many intermediate states as there are yielded elements, plus one for the current state. Ⓒ The last tuple in the st sequence corresponds to the current data stored in Map. For each yielded element Ⓓ we re-use our nested call descriptions to establish the connection between consecutive intermediate states. We also Ⓐ state that the state sequence is expanded monotonically, i.e. the intermediate states in any previous version are a prefix of the intermediate states of any newer version.

With this specification, we can verify code which uses a Map parameterised by a closure with captured state, applied to an unbounded number of elements, such as this summation:

```
let vec: Vec<i32> = ...;
let vals = GhostSeq::of_vec(&vec); // ghost
let mut pos = 0usize; let mut s = 0;
let cl = #[requires(pos < vals.len() && x == vals[pos])]
         #[ensures(pos == old(pos) + 1 && result == s)]
         #[invariant(   0 <= pos && pos <= vals.len()
                     && s == vals[0..pos].sum())]
         |x| { s += x; pos += 1; s };
// iterate values of vector
let mut map_iter = vec.into_iter().map(cl);
#[invariant(pos == map.prev.pos)] for el in map_iter {}
assert!(s == GhostSeq::of_vec(&vec).sum());
```

In this example, leadsto is maintained for map_iter throughout the loop, which also implies the closure history invariant. Once the loop exits, it is known that the position of the vector iterator reached the end of the vector, and so s == vals[0..pos].sum() collapses to s == vals.sum().

5 Type-dependent contracts

So far we considered adding methods to Rust's Iterator trait, but (a) this is standard library code that cannot be modified, and (b) this forces *all* iterators in a codebase to implement the features of our methodology. Several verification tools allow the declaration of specifications in separate files from source code (an idea we believe to originate from Spec# [4]); in Prusti the extern_spec feature [2] allows the declaration of a specification for a function/type from a different file. This solves (a), but (b) requires a more-subtle technique; we explain our solution to this (4th main challenge) here.

Firstly, we declare a new trait with the (specification-only) methods of our methodology (leaving Iterator unchanged):

```
trait IteratorSpec : Iterator { Ⓐ
  fn produced(&self) -> GhostSeq<Self::Item Ⓑ>;
  fn leadsto(p: &Self, c: &Self) -> bool;
  fn step(p: &Self, c: &Self,
          r: &Option<Self::Item Ⓒ>) -> bool;
  fn done(&self) -> bool; }
```

In the above, we Ⓐ declare IteratorSpec and use Iterator as its super-trait. This means that any type that implements IteratorSpec must also implement Iterator. Note that at Ⓑ and Ⓒ we re-use the associated type Item from the Iterator supertrait.

5.1 Type-dependent contracts

The additional trait IteratorSpec allows us to remove the four function declarations from the Iterator trait, but does not provide a way to express the postconditions of the next method in Iterator (Rust does not allow a sub-trait to redeclare/override methods of a super-trait). To provide a specification for next without changing the Iterator trait, we can use an external specification provided in a separate file. For example, we could (erroneously) specify Iterator::next to always return a result as follows:

```
#[extern_spec] // out-of-band contract
trait Iterator {
  #[ensures(result.is_some())]
  fn next(&mut self) -> Option<Self::Item>; }
```

However, such an extern spec for Iterator cannot use the methods from IteratorSpec, because not every implementation of Iterator also implements IteratorSpec. To solve this issue, we introduce type-dependent contracts, that is, contracts that apply *only* to implementations of a given trait. The following extern spec for Iterator uses this feature to impose postconditions on next *only* for implementations that also implement IteratorSpec:

```
#[extern_spec]
trait Iterator { Ⓐ
  #[type_dependent(Self: IteratorSpec, [ Ⓑ
  (Q1) ensures(old(&self).step(&self, &result)),
  (Q2),(Q3),(Q4),(Q5) ])] // as in Sec. 3.1
  fn next(&mut self) -> Option<Self::Item>; }
```

As before, Ⓐ an extern spec is added to the Iterator trait. The Ⓑ type_dependent attribute introduces postconditions Q_1 - Q_5 only for implementations that also implement IteratorSpec. As a result, it is allowed to refer to methods of IteratorSpec, e.g. to call **self**.done().

Note that the type-dependent contract is defined on the type *being extended*, i.e. it is part of the specification of Iterator::next, not part of the extension trait IteratorSpec. This decision is motivated by Rust's *coherence* rules, which ensure that there is always exactly one implementation to choose for any particular trait method call. Analogously, we require this property of method *contracts*.

To ensure that our type-dependent contracts are sound, we require that any type-dependent contract refines its base contract. This consists of the usual *behavioural subtyping* checks [14]: the precondition must be weakened and the postcondition strengthened. The usage shown in this section can selectively equip iterator types with our methodology without changes to their code or the standard library.

6 Evaluation

We implemented our technique as a prototype extension to the Prusti verifier [3]. Following the design laid out in Sec. 3, our iterator methodology is implemented primarily in user-facing Rust code (which can be packaged into a standard library of specifications), not as an *ad hoc* feature of Prusti.

To enable annotating key standard library types, we added support for declaring external specifications for trait types as well as for our novel type-dependent contracts. Most other extensions were routine; this suggests that layering our methodology onto existing tools is fairly lightweight. Due to this lightweight integration, we don't treat `leadsto` as a built-in type invariant, but rather as a postcondition on `next`; intended properties such as transitivity are therefore not checked by default (but proofs relying on them would of course fail otherwise).

We evaluated our work on a number of challenging test cases, modelling various combinations of idiomatic iterators found in the Rust standard library, as well as custom iterator implementations discussed in this paper. The results of our evaluation are shown in Table 1 (in terms of lines of specification, code and verification times). Generally, the specification overhead is heavier (roughly one-to-one with code) for the generic library functions such as `Map`, but these specifications need only be written once. Importantly, for client code *using* these iterators, the specification overhead is typically lighter. A substantial body (roughly 340 LoC) of common specifications were also necessary as our implementation neither builds in pre-defined support for common types such as `Option`, or our new `GhostSeq` type. These specifications need only be written once and could in principle be added as a "standard library" of specifications. We consider the verification times using our prototype implementation to be generally reasonable, but with some expensive outliers; we suspect that these require some additional effort to control quantifier instantiation in the underlying solver [5].

7 Related work

As discussed in Sec. 1, a simpler version of call descriptions (for closure calls only) was introduced by Wolff et al. [20] for closure verification; that work does not handle any of the four main challenges we identify for iterator verification.

Creusot [8] is Rust verifier that maps a subset of Rust to a functional language for verification in Why3 [10]. Its prophetic encoding of Rust references simplifies the specification of some reborrowing patterns, meaning that the signature of traits such as `IterMut` are supported, unlike the reference support available in Prusti at time of writing. As mentioned earlier, a recent iterator verification methodology was designed by Denis et al. [7] for Creusot in parallel with this work, which supports `IterMut` directly.

Their specifications require that all relevant properties of heap values are representable as value by mathematical abstractions; in particular, no notion of instance/reference identity is available in the model. We use a similar idea for the current *automation* of proofs using our specification technique (via snapshots), but our specifications make explicit how closure/iterator/adapter instances persist and are called across multiple states. This may afford advantages when applying our methodology in other settings where instance identity is important to the properties verified.

In terms of the four challenges in our introduction, [7] handles the first two and partially addresses the third (automation), but some iterator proofs fail without user intervention in the form of custom Why3 tactics; debugging and fixing these proofs must be performed at the level of the intermediate functional language, not in Rust. Our technique provides automation at the level of Rust via extensive use of quantifiers; this can make proofs slower and sometimes require annotating quantifiers with *triggers* (a standard notion for SMT-based tools), but all errors and annotations can be understood in the Rust program via Rust expressions. We believe this degree of automation and consistent level of abstraction is important for users of such a methodology.

The above work and ours are both extensions of work by Pereira [16], which defines a modular verification technique for iterators in a *functional* setting using two predicates per iterator to specify the sequence of produced values and a termination condition. It partially addresses Challenge 1 by providing abstract, implementation-independent specifications for individual iterators. However, although higher-order iterators are discussed, this only refers to simple iterators parameterised by side-effect-free functions, as in a functional `fold` operation. Support for iterators with side effects and iterator composition are not addressed.

Iterators and generic algorithms in the C++ Standard Template Library (STL) are concepts closely related to iterators and iterator adapters in Rust. Musser and Wang [15] formalise C++ iterators, then use their specifications to (manually) prove the correctness of some algorithms generic over the input iterator. As noted in Sec. 1.1, Rust iterators are akin to *streams* in other languages, a concept which exists in the

Test (implementation)	LoS	LoC	VT (s)
`counter`	32	30	9.53
`double`	43	25	9.95
`filter.vpr`	90*	109*	18.24*
`map`	67	38	42.12
`option_intoiter`	36	19	7.41
`vec_intoiter`	42	13	7.05
`zip`	79	32	84.46

Test (client code)	LoS	LoC	VT (s)
`counter`	0	14	11.53
`double`	1	6	10.16
`filter.vpr`	10*	27*	5.68*
`map`	14	22	79.78
`option_intoiter`	0	4	6.96
`vec_intoiter`	4	19	16.48
`zip`	2	6	67.12

Table 1. Evaluation. *LoS* and *LoC* represent the line of specifications and lines of code, respectively. *VT* represents the verification time, measured as the wall-clock runtime averaged over 7 runs using an Intel Core i9-10885H 2.40GHz CPU with 16 GiB of RAM, excluding the slowest and fastest runs. Test cases ending in `.vpr` were encoded manually into Viper: our methodology supports these test cases, but issues in the underlying Prusti tool (independent from the contributions of this paper) currently prevent our implementation from supporting the analogous examples in Rust. These Viper encoded examples are more verbose and run through a simpler tool chain; for these reasons we mark the data with *s in the tables here.

Boost C++ libraries [1] and more recently in the C++20 standard as “range adapters” or “views”[6]. To our knowledge, deductive verification of such features was not yet attempted.

8 Conclusion

We have presented a novel methodology for modularly specifying and verifying the complex iterator patterns found in modern programming languages. Our methodology is compatible with standard techniques for reasoning about side-effectful programs, such as Rust’s ownership system and separation logics. We evaluated our methodology in Rust, which has rich iterator support in its standard library, as well as a type system which can be used to automatically take care of these ownership requirements. Applying our methodology in languages without such a type system would require specifications to govern side-effects, but the adaptation of our novel methodology would be straightforward.

To ensure our methodology is usable with pre-existing real-world codebases and integrates well with other verification efforts, we have prioritised modularity throughout the design of our methodology and the ability to selectively apply our methodology per iterator implementation technique in our implemented extension to the state-of-the-art Rust verifier Prusti. To this end, we introduced type-dependent contracts which, along with specification extension traits, allow specifying standard-library iterators and their clients without modifying their source code or dependencies. We have identified other cases where the feature is useful when specifying the Rust standard library [9].

References

[1] 1999. Boost C++ libraries. https://www.boost.org/

[2] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers. 2022. The Prusti Project: Formal Verification for Rust (invited). In *NASA Formal Methods (14th International Symposium).* Springer, 88–108. https://doi.org/10.1007/978-3-031-06773-0_5

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

[4] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. 2004. The Spec# programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices.* Springer, 49–69.

[5] N. Becker, P. Müller, and A. J. Summers. 2019. The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS).* Springer, 99–116.

[6] A. Bílý, J. Hansen, P. Müller, and A. J. Summers. 2022. Compositional Reasoning for Side-effectful Iterators and Iterator Adapters. arXiv:2210.09857 [cs.LO]

[7] Xavier Denis and Jacques-Henri Jourdan. 2023. Specifying and Verifying Higher-order Rust Iterators. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 93–110.

[8] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: a Foundry for the Deductive Verication of Rust Programs. In *International Conference on Formal Engineering Methods (ICFEM) (LNCS).* Springer Verlag, Madrid, Spain. https://hal.inria.fr/hal-03737878

[9] Julian Dunskus. 2022. *Annotating the Rust Standard Library with Specifications for Use in a Rust Verifier.* Master’s thesis. ETH Zürich.

[10] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—where programs meet provers. In *European Symposium on Programming (ESOP).* Springer, 125–128.

[11] Jonas Hansen. 2022. *Specification and Verification of Iterators in a Rust Verifier.* Master’s thesis. ETH Zürich.

[12] Bart Jacobs, David Cok, Bruce Weide, Kevin Bierhoff, Neelakantan Krishnaswami, et al. 2006. *Proceedings of the 2006 Conference on Specification and Verification of Component-based Systems (SAVCBS).* ACM Digital Library.

[13] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium.* Springer, 41–55.

[14] Barbara Liskov and John Guttag. 2000. *Program development in JAVA: abstraction, specification, and object-oriented design.* Pearson Education.

[15] David R Musser and Changqing Wang. 1995. *A basis for formal specification and verification of generic algorithms in the C++ standard*

[6] https://en.cppreference.com/w/cpp/ranges

template library. Technical Report. Citeseer.

[16] Mário José Parreira Pereira. 2018. *Tools and Techniques for the Verification of Modular Stateful Code*. Theses. Université Paris Saclay. https://tel.archives-ouvertes.fr/tel-01980343

[17] John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*. IEEE, 55–74.

[18] M. Schwerhoff. 2016. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. Ph. D. Dissertation. ETH Zurich.

[19] Jan Smans, Bart Jacobs, and Frank Piessens. 2010. Heap-dependent expressions in separation logic. In *Formal Techniques for Distributed Systems*. Springer, 170–185.

[20] Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. 2021. Modular specification and verification of closures in Rust. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct. 2021), 1–29. https://doi.org/10.1145/3485522