Specification and Verification of Rust Iterators in Prusti

Master's Thesis Project Description

Jonas Hansen
Supervisors: Aurel Bílý, Prof. Dr. Peter Müller

November 2021

1 Introduction

Rust is a multi-paradigm programming language with a focus on systems programming. Arguably the most unique feature of Rust is its *ownership type system* for memory management. Rust also embraces features from functional programming by supporting first-class functions and closures, whose combination with *iterators* allows programming in a functional style. Even though closures and iterators are orthogonal Rust features, they are often used in combination to produce concise and functional-style code.

In 2019, Astrauskas et al. [AMPS19] have introduced *Prusti*, a tool for formally verifying Rust programs. In their recent work [WBM+21], Wolff and Bílý have leveraged Prusti's capabilities to support the verification of closures and higher-order functions.

This work aims to extend Prusti's verification capabilities for iterators. Iterators in Rust, as in other programming languages, are used to traverse elements of a collection, such as processing each element of a vector in a for loop. Since iterating over a collection is a common task many programs, the need to formally verify iterators arises naturally.

2 Introduction to iterators in Rust

Like in other programming languages, an iterator in Rust is used to traverse elements of a collection. As an example, to traverse all elements in a vector, a programmer can turn a vector into an iterator via the `iter()` method and then traverse the elements of that iterator using a loop as shown in Listing 1.

```
let vec = vec![1, 2, 3, 4, 5];
let iter = vec.iter();
for el in iter {
  println!("{}", el);
}
```

Listing 1: Using an iterator in a for loop

An iterator in Rust is a type which implements the `Iterator` trait. A trait in Rust, like interfaces in other programming languages, defines methods to be implemented on a type. Additionally, a trait might provide default implementations for methods which can be overriden by the implementing type. Listing 2 provides a simplified definition of the `Iterator` trait.

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

Listing 2: The Iterator trait

A struct implementing this trait must define an *associated type*, which specifies the type of the elements to be traversed and the `next` method, which yields the elements. By returning an `Option::None` in `next`, the iterator signals that no more elements are available. Client code which uses an iterator can either use the `next` method directly or use for loops.

Listing 3 demonstrates the implementation of an iterator, which yields increasing numbers up to a bound.

```rust
struct Counter {
    count: u32,
}

impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<u32> {
        if self.count < 3 {
            let res = Some(self.count);
            self.count += 1;
            return res;
        }
        None
    }
}
```

Listing 3: Custom iterator which yields increasing numbers up to a bound

2.1 Turning types into iterators

Rust's standard library offers a variety of iterators and methods to turn types into iterators. It provides the `IntoIterator` trait which types can implement to convert themselves via the `into_iter()` method into an iterator. The implementation of this trait then allows the type to be used in a for loop directly. `IntoIterator` usually consumes the value, i.e. the resulting iterator is the new owner of the value. Therefore, many types also provide the `iter()` and `iter_mut()` methods, which return iterators iterating over immutable and mutable borrows of the value, respectively.

As an example, the `Vec<T>` type implements the `IntoIterator` trait, which consumes the vector and returns an `IntoIter` iterator which iterates over the vector's values. Additionally, since `Vec<T>` coerces into a slice, one can also use `iter()` and `iter_mut()` on a vector to obtain an `Iter` and `IterMut` iterator of the corresponding slice.

Having generated an iterator, the `Iterator` trait provides default methods which nest the corresponding iterator inside other iterators. Such iterators are commonly called *iterator adapters*.

2.2 Iterator adapters

The Rust standard library includes many iterators which are generated from other iterators, such as `Map`, `Filter`, `Zip` and `Take`, which have similar semantics as their counterpart-operations in functional programming. Some of these iterators are used with closures, e.g. the `Filter` iterator which takes a predicate to filter elements from the underlying iterator.

This pattern allows for arbitrary chaining of iterators. It is important to note that iterator adapters are *lazy*, i.e. constructing the iterator does not mean that the computation is performed. Instead, the actual iteration is deferred until the `next` method on the resulting iterator is called.

Listing 4 demonstrates the expressiveness of the combined use of closures and iterators. In line 2, we create a closure `is_prime` which checks whether a number n is prime. The implementation is naive: it simply checks whether all numbers between 2 and $n-1$ do not divide n. In line 5-8, we chain several iterators together to fetch the first n prime numbers. Crucially, the actual computation happens in line 8 and not during the intermediate chaining of the iterators. If this were not the case, the `primes` method would not terminate, because it would first generate all numbers from 2 to `u32::MAX`, then filter the primes and then take the first n primes.

```rust
fn primes(n: usize) -> Vec<u32> {
    let is_prime = |num: &u32| (2..*num).into_iter()
                                        .all(|x| num % x != 0);

    (2..).into_iter()   // Range into iterator
      .filter(is_prime) // Create Filter iterator
      .take(n)          // Create Take iterator
      .collect()        // Computation starts
```

Listing 4: Combined usage of iterator adapters and closures

2.3 Collecting values from an iterator

Usually, iterators are used to transform values from one collection to another as shown in Listing 4. In the example listing, the `collect()` method was used to obtain a new vector with the transformed values. This method uses an indirection via the `FromIterator` trait. Because the return type of `primes` is `Vec<T>`, `collect` calls the `from_iter` method (defined in the trait) on `Vec<T>` (which implements the trait). The implementation then exhausts the iterator and returns the transformed elements in a new vector.

3 Approach

Whenever one uses iterators in Rust, one or multiple of the following functionalities are involved:

1. Turn something into an iterator (`IntoIterator`)
2. Traverse the iterator (using `Iterator::next` or for loops)
3. Collect the values from an iterator (`FromIterator`)
4. Application of adapters on iterators (as in Listing 4)

Ideally, Prusti would support all of these use cases. This section describes our approach, using prototype verification code which is subject to change during the implementation.

3.1 Obtaining an iterator from a vector

To start with, a `Vec<T>` can be turned into an `std::vec::IntoIter`, `std::slice::Iter` or `std::slice::IterMut`, respectively, via its corresponding methods (`into_iter()`, `iter()`, `iter_mut()`) available on any vector type. The latter of the two are, however, an indirection via the slice type because `Vec<T>` coerces into the slice type `&[T]` via deref coercion. A specification for any iterator constructed from a vector involves a specification for this indirection as shown in Listing 5.

```
#[extern_spec]
impl<T> std::ops::Deref<Target = [T]> for std::vec::Vec<T> {
    #[ensures(forall(|i: usize| i < self.len() ==> result[idx] == self[idx]))]
    fn deref(&self) -> &'a [T];
}
```

Listing 5: Sample specification for vector coercion into slice

Once we have a specification stating the relation between vectors and slices, we can start to actually specify `Iter` (or `IterMut`). To decouple implementation details from specifications, we suggest to introduce *models* for these iterators. A model represents an abstraction over some type `T`, which can only be used in verification code. This has the advantages that the actual specification gets simpler to read and more concise. Furthermore, the model ensures that the specification does not need to be changed when the underlying implementation changes. Lastly, the model might be a necessity when the implementation contains non-public internals which cannot be used in a specification. The following Listing 6 outlines the definition of a model of the slice iterator `Iter` (the model for `IterMut` and `IntoIter` looks similar).

```
#[model]
struct std::slice::Iter<'a, T> where T: 'a {
    position: usize,
    data: &'a [T],
}

#[extern_spec]
impl<T> [T] {
    #[ensures(model(result).position == 0)]
    #[ensures(model(result).data == self)]
    fn iter(&self) -> std::slice::Iter<'_, T>;
}
```

Listing 6: Using an abstract model for slice iterators

Having a model for the `Iter` iterator, we can finally provide specifications for the implementation of its `Iterator` trait as shown in Listing 7.

```
#[extern_spec]
impl Iterator<'a, T> for std::slice::Iter<'a, T> {
    #[ensures(model(self).data == old(model(self).data))]
    #[ensures(model(self).position >= model(self).data.len() ==> result.is_none())]
    #[ensures(old(model(self).position) < model(self).data.len() ==>
                    model(self).position == old(model(self).position) + 1 &&
                    result.is_some() &&
                    result.unwrap() == model(self).data[old(model(self).position)])]
    fn next(&mut self) -> Option<&'a T>;
}
```

Listing 7: Sample specification for the implementation of the slice iterator

3.2 Specification of iterators

In subsection 3.1 we have outlined how to provide specifications for iterators via a model which is an abstraction over the actual implementation. In many scenarios, however, we would like to provide a specification for an iterator which is verified against real code. We illustrate our approach with Listing 8, which includes a user-typed iterator adapter for the `Counter` iterator introduced in Listing 3.

```
struct DoublingIter<I> {
    iter: I,
}

impl<I> Iterator for DoublingIter<I> where I: Iterator<Item = u32> {
    type Item = u32;

    fn next(&mut self) -> Option<u32> {
        match self.iter.next() {
            Some(val) => Some(2 * val),
            None => None,
        }
    }
}

fn test_iter_adapter() {
    let mut iter = DoublingIter {
        iter: Counter { count: 0 },
    };
    assert!(iter.next().unwrap() == 0);
    assert!(iter.next().unwrap() == 2);
    assert!(iter.next().unwrap() == 4);
    assert!(iter.next().is_none());
}
```

Listing 8: Example for a custom iterator adapter

A specification of `DoublingIter` depends on the specification of its nested iterator `self.iter`. To specify the result of `DoublingIter`, we propose to use *call descriptions*. The concept of call descriptions was introduced in [WBM+21] in the context of closures and higher-order functions. Using a call description in a higher-order function involving some closure `cl` expresses that `cl` was called in some prestate P and some poststate Q. Ultimately, call descriptions allow specifications to tie the result of a closure with the result of the higher-order function.

Analogously, an outer iterator can be thought of as a higher-order function and the inner iterator as the closure. Our intended usage of call descriptions for outer iterators is shown in Listing 9. This specification expresses that `self.iter.next` was called (with no specific pre-call state) and in the poststate of the call, if the result `self.iter.next` is `Some(val)`, then the outer iterator's result is `Some(2*val)`. Likewise, if `self.iter.next` is `None`, the outer iterator's result is `None`.

With both specifications, the verifier should be able to prove all assert statements in the method `test_iter_adapter()`. Even though methods are verified modularly in Prusti and the specification of `DoublingIter` itself is not enough to prove the assert statements, the verifier can combine both specifications from `Counter` and `DoublingIter` because the type of `iter` is `DoublingIter<Counter>`.

```
// ...
impl Iterator for Counter {
    type Item = u32;

    #[ensures(result.is_some()     ==> self.count == old(self.count) + 1)]
    #[ensures(result.is_none()     ==> self.count == old(self.count))]
    #[ensures(old(self.count) < 3  ==> result.is_some() &&
                                       result.unwrap() == old(self.count))]
    #[ensures(old(self.count) >= 3 ==> result.is_none())]
    fn next(&mut self) -> Option<u32> {
        // ...
    }
}

impl<I> Iterator for DoublingIter<I> where I: Iterator<Item = u32> {
    type Item = u32;

    #[ensures(
        old(self.iter.next) ~> || -> I::Item
            { true }
            {
                result.is_some() ==  outer(result).is_some() &&
                result.is_some() ==> outer(result).unwrap() == 2*result.unwrap()
            }
    )]
    fn next(&mut self) -> Option<u32> {
        // ...
    }
}
// ...
```

Listing 9: Sample specification for iterators

3.3 Collecting values from iterators

In the previous chapter, we verified properties of iterator elements by iterating via the `next` method of the iterator. In many scenarios, however, an iterator's values are collected in a collection prior to processing as outlined in subsection 2.3. The following Listing 10 collects the values of `DoublingIter` into a `Vec<u32>` before making assertions about the content of the vector.

```
fn test_iter_collect() {
    let mut iter = DoublingIter {
        iter: Counter { count: 0 },
    };
    let vec: Vec<u32> = iter.collect();
    assert!(vec == vec![0, 2, 4]);
}
```

Listing 10: Collection of iterator values

In order to prove the assertions in this method, we need a specification for the `collect` method, defined in `FromIterator` implemented on `Vec<T>`. The main challenge is to specify which element in the resulting vector corresponds to which `next` call of the iterator. Listing 11 contains a prototype specification. The specification assumes a *ghost sequence* of the states of the iterator `I` that was collected. This sequence has the same size as the returned vector, i.e. one element for every call to `I::next`. In line 9, we use a call description to relate the state change induced by a call to `I::next` with the ghost sequence, as well as relating the inner result to the corresponding vector element. Note that this call description makes the state of the iterator explicit, i.e. in Rust `I::next(&mut iter)` is semantically equivalent to `iter.next()`.

```rust
#[extern_spec]
impl<T> FromIterator<T> for Vec<T> {
    #[ensures(
        exists(|iter_states: GhostSeq<I>|
            iter_states[0] == old(iter) &&
            iter_states.len() == result.len() + 1 &&
            iter_states[result.len()] == iter &&
            forall(|i: usize| 0 <= i && i < result.len() ==>
                I::next ~> |state: &mut I|
                    { state == iter_states[i] }
                    {
                        state == iter_states[i + 1] && result == outer(result)[i]
                    }
                )
        )
    )]
    fn from_iter<I: IntoIterator<Item = T>>(iter: I) -> Vec<T>;
}
```

Listing 11: Specification for collecting iterator values

4 Core goals

The following list of core goals are derived from our approach described in section 3 and an analysis of existing Prusti functionality. The implementation of these goals should enable a user to specify self-defined custom iterators in many scenarios.

- **Support for specifications on trait implementations**
  Currently, Prusti does not support the specification of trait method implementations[1], which is required by a reasonable specification of `Iterator` implementations.
  Estimated time: **2 weeks**

- **Traversal of iterators with for loops**
  While the direct traversal of an iteration via its `next` method should not need any specific changes to Prusti, the traversal in for loops as in Listing 1 is currently (explicitly) not supported in Prusti, which is clearly a desired feature when using iterators.
  Estimated time: **2 weeks**

- **Modeling support for standard library iterators**
  This goal aims to provide a way to specify the standard library's `Iter`, `IterMut` and `IntoIter` iterators including needed specifications for vector-to-slice coercion. We try to achieve this by using the high-level modeling technique outlined in section 3. Another part of this goal is to collect and analyse other coercions to the slice type (e.g. for strings) and provide specifications for these types as well, if possible.
  Estimated time: **4 weeks**

- **Implementation of iterator adapter specifications**
  The goal is to implement the support for specifications of iterator adapters. Throughout the implementation, we will evaluate our approach on different use cases imitating the functionality from common iterator implementations (like in the standard library).
  Estimated time: **4 weeks**

- **Evaluation**
  Before writing the final report, we will evaluate Prusti's effectiveness of verifying existing Rust code without user-defined specifications. With this goal, we can test the strength of Prusti's default supplied specifications of iterators. We evaluate this on open-source cargo crates.
  Estimated time: **1 week**

- **Final report**
  Writing of the final report including preparation for the final project presentation.
  Estimated time: **6 weeks**

[1] See https://github.com/viperproject/prusti-dev/issues/625

5 Extension goals

The estimated time spent on extension goals is **5 weeks**.

- **Specifications for standard library iterator adapters**
  As mentioned in the introduction, the standard library offers a large variety of different iterator adapters, such as `Map`, `Zip` or `Filter`. In this extension goal, we aim to provide specifications for a subset of these iterators.

- **Case study**
  We will work on a case study to show the effectiveness of Prusti in verifying Rust code using iterators. This case study will also serve as a tutorial. The actual case study remains to be defined.

- **Specification for comprehensions**
  For some iterators, we can use *comprehensions* such as `max()` or `sum()` to compress the iterator into a single value. In this extension goal, we would like to provide specifications for a subset of these comprehensions.

- **Async iterators**
  Rust supports asynchronous features, such as the `async`/`await` keywords for writing asynchronous functions. In this goal, we evaluate to what extent it is already possible with Prusti to verify asynchronous iteration over collections such as in `Stream`s.

References

[AMPS19] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 3, pages 147:1–147:30. ACM, 2019.

[WBM+21] F. Wolff, A. Bílý, C. Matheja, P. Müller, and A. J. Summers. Modular specification and verification of closures in rust. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2021. To appear.