Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Specification and Verification of Iterators in a Rust Verifier

Master's Thesis

Jonas Hansen

June 18, 2022

Advisors: Aurel Bílý, Prof. Dr. Peter Müller
Department of Computer Science, ETH Zürich

Abstract

Iteration is a fundamental part in every programming language. Rust is no exception. The combination of functional programming idioms and iterators allows for writing concise and expressive Rust code.

In this work, we investigate the specification and verification of Rust iterators and implement our solution for Prusti, a deductive verifier for Rust. We aim for modular verification: The specification and verification of an iterator should be independent of its usage in clients.

We demonstrate our implemented solution by specifying a custom iterator implementation and two iterators from the standard library of Rust. Additionally, we show that these specifications can be used to verify looping and non-looping clients.

Acknowledgements

At this point I would like to express my sincere thanks to my supervisor Aurel Bílý. He always supported me tirelessly during my work and always had an open ear for my many questions. Our weekly meetings were an enrichment for me in all aspects and supported me optimally during my work. I learned a lot from Aurel during the last months and enjoyed working with him.

Furthermore, I would like to thank Peter Müller and all members of the *programming methodology* group. The feedback I received from the group during the work turned out to be meaningful and useful and was taken into account in the developed solution. I would also like to take the opportunity to thank the group for their excellent, enriching and instructive lectures held at ETH Zürich.

Contents

Chapter 1

Introduction

The concept of iteration is present in every programming language. In most mainstream imperative languages iterators unfold their presence in loops to traverse the elements of a container, for example a list or an array.

The Rust programming language is no exception. The strength of Rust iterators lies in the iterator API of the standard library. This API heavily relies on functional programming idioms which allows for writing concise and expressive code. Often the use of iterators in Rust is not obvious at first glance. Instead, iteration appears as part of a sequence of chained function calls (`map`, `filter`, ...) which describe the iteration. Under the hood, Rust creates a *chain of iterators*. This chain can then be used as part of a loop or can be processed using aggregation functions (`sum`, `max`, ...).

The rise of deductive program verification in recent years demands for techniques and tools to formally verify iteration in Rust. Prusti [1] is a front-end verifier for Rust built on top of the Viper verification infrastructure [9]. By default, Prusti proves the absence of panics of a Rust program. Due to Rust's type system, this can be achieved with a high degree of automation. Additionally, users can attach contracts to Rust code. Prusti then tries to prove the validity of these contracts.

Prusti does not yet support the verification of code that uses iterators. This work aims to make a first step towards that direction.

1.1 Goals

With this work we pursue the following goals:

Derive a framework for iterator specification We want to develop a framework which can be used to specify Rust iterators. This framework should be applicable to custom-written iterators and the iterators from the standard library.

Modularity Specification and verification of iterators should be modular. That is, iterators are specified and verified independently of their clients. Likewise, the verification of clients should not rely on the actual implementations of the used iterators but only on their specifications.

Traversal of iterators with loops Provide tools and techniques to verify the usage of iterators in loops.

Allow the specification of non-local types Many iterators that we consider are part of the standard library of Rust. We do not want to copy the code of these iterators in order to specify them.

1.2 Outline

The structure of this report is as follows:

- Chapter 2 introduces the needed background knowledge. Especially, Rust iterators and the specification of Rust closures in Prusti are introduced in detail.
- Chapter 3 describes our approach for iterator specification. We build up these techniques gradually to motivate them.
- Chapter 4 contains implementation-specific information about the concepts introduced in Chapter 3.
- Chapter 5 shows our specified iterators and the clients we verified.
- Chapter 6 concludes this work and discusses future work.

Chapter 2

Background

This chapter presents the necessary background information for this thesis. We start by providing an introduction to the Rust programming language in Section 2.1, continued by an in-depth explanation of Rust iterators in Section 2.2. A short description about the deductive front-end verifier Prusti follows in Section 2.3.

2.1 Rust

Rust is a multi-paradigm, general-purpose programming language which puts a strong emphasis on performance via zero-cost abstractions and security due to the aid of a strong, static *ownership type system*.

The ownership type system (whose core part is the *borrow checker*) is Rust's most distinguishing feature. The borrow checker imposes rules on memory management and can be seen as an alternative to garbage collection (Java, C#), reference counting (smart pointers in C++), or the RAII (resource acquisition is initialization) idiom commonly used in C++. Borrow checking is executed at compile time and induces no additional runtime overhead.

The borrow checker assigns a unique *owner* to every value in Rust which usually is a variable. As soon as the owner goes out of scope the corresponding memory location is freed. Ownership of a variable is transferred when the variable is reassigned or passed to a function. In Rust, one can create references (also called *borrows* in this context) to a variable. Creating a reference does not transfer ownership – instead, it allows for readable or writable access to the corresponding memory location. The borrow checker ensures that at any point during the execution of the program there is either at most one mutable borrow or any amount of immutable borrows of a value. With these rules, Rust can statically prove the absence of aliasing, dangling pointers and even the absence of data races in a multithreaded program.

```
1  trait Incrementable {
2      fn increment_by_one(&mut self); // Required method
3
4      fn increment_by_two(&mut self) { // Provided method
5          self.increment_by_one();
6          self.increment_by_one();
7      }
8  }
9
10 struct Counter {
11     value: i32
12 }
13
14 impl Incrementable for Counter {
15     fn increment_by_one(&mut self) {
16         self.value += 1;
17     }
18 }
```

Listing 1: Defining shared behavior with traits.

2.1.1 Traits, generics, and bounds

Since Rust is not an object-oriented programming language, it does not support inheritance on types. Instead, sharing behavior is achieved via *traits*. Traits are comparable to interfaces in Java or C#, as they define methods which a type must implement.

Traits may declare *required methods* which must be implemented on the implementing type and *provided methods* with a body which can be overridden.

Listing 1 shows the declaration of a trait on Line 1. On Line 10, a user-typed type `Counter` is defined which has a field `value` of type `i32`. Finally, the trait is implemented on Line 14 on `Counter`.

One can then write a generic function, whose generic type parameter has a *bound* on the trait as shown in Listing 2: The function `fn inc` is generic over any type parameter `T`. On Line 1 we impose that types must implement the `Incrementable` trait. Trying to invoke the function with a type that does not implement `Incrementable` results in a compile error.

Type parameter bounds can be either written as a `where` clause at the end of the function signature (as in Listing 2) or in-place where the type parameter is declared (i.e. `T: Incrementable`)

```rust
fn inc<T>(incrementable: &mut T) where T: Incrementable {
    incrementable.increment_by_one()
}
```

Listing 2: Trait bounds for generic type parameters.

```rust
trait Incrementable<T> {
    fn increment_by(&mut self, val: T);
}

use std::ops::Add;
impl<T: Add<i32, Output=i32>> Incrementable<T> for Counter {
    fn increment_by(&mut self, val: T) {
        self.value = val + self.value;
    }
}
```

Listing 3: Generic traits.

Polymorphic traits

Traits can be *generic* and thus support parametric polymorphism [13]. In Listing 3 we declare a trait `Incrementable<T>` which is generic over some type `T`. The implementor `Counter` then can implement the trait for any type on which we can perform `i32` addition.

Note that `Counter` could also simply just implement `Incrementable<i32>` and provide a *non-polymorphic implementation* of the trait.

Associated types

Traits can declare *associated types* which must be specified by implementing types as shown in Listing 4. We have already seen in Listing 3 that the `std::ops::Add` trait from the standard library has an associated type `Output` which we used as a bound in the generic implementation.

Monomorphization

Generics in Rust are fully resolved at compile time and do not induce a runtime penalty. During compilation, the compiler analyzes every call to a generic function and creates a copy of that function with the concrete used type parameters. This process is called *monomorphization* and is important for runtime speed and allows for further optimizations throughout the com-

```
1  trait Incrementable {
2      type Result;
3      fn increment_by_one(&mut self) -> Self::Result;
4  }
5
6  impl Incrementable for Counter {
7      type Result = i32;
8      fn increment_by_one(&mut self) -> Self::Result {
9          self.value += 1;
10         self.value
11     }
12 }
```

Listing 4: Associated types in traits.

pilation stage, albeit slowing down compilation in the presence of (many) generics.

2.1.2 Closures

Rust embraces idioms from functional programming by treating functions and *closures* as first-class citizens. "Closures are anonymous functions which, unlike ordinary functions, can capture the environment where they are defined in" [7]. Additionally, they can be passed to higher-order functions.

The `Fn`, `FnMut` and `FnOnce` traits can be used in higher-order functions to impose a bound on the expected closure type. Closures which do not mutate any state automatically derive the `Fn` trait, whereas mutating closures derive `FnMut` and closures which consume values derive `FnOnce`.

Listing 5 shows a closure on Line 9 which first increments the captured variable `count` by one and then returns the sum of the new `count` and the passed argument to the caller. The higher-order function `fn hof` on Line 1 accepts any closure which has one parameter of type `i32` and returns an `i32` value. `fn hof` calls the passed closure with the passed argument `arg` and returns the result to the caller, as we see on Line 13 and onwards.

2.2 Iterators in Rust

Iterators in Rust, as in other programming languages, are used to traverse the elements of some source. A common use case is the traversal of a Rust vector `Vec<T>` which is an array-like container of items of type `T`:

```
1  let vec: Vec<i32> = vec![1,2,3];
2  for el in vec { // prints 1,2,3
```

```rust
fn hof<F>(cl: &mut F, arg: i32) -> i32
    where F: FnMut(i32) -> i32 {
    cl(arg)
}

fn main() {
    let mut count = 0;
    let mut cl = |x: i32| {
        count += 1;
        x + count
    };

    assert_eq!(hof(&mut cl, 1), 1 + 1);
    assert_eq!(hof(&mut cl, 5), 5 + 2);
    assert_eq!(hof(&mut cl, 10), 10 + 3);
}
```

Listing 5: Demonstration of a closure and a higher-order function. The closure captures and mutates state from its context. The high-order function takes a reference to the closure and executes it.

```rust
    print!("{el},");
}
```

Iteration is not limited to this use case: The unbounded range iterator `RangeFrom`[1], introduced with `(start..)` syntax, is an iterator which generates elements on the fly:

```rust
for el in (5..) { // prints 5,6,7,... indefinitely
    print!("{el},");
}
```

Iterators are often used in combination with closures which allows for writing code in a very concise and functional way as demonstrated in Listing 6. The example starts by creating an unbounded range of integers on Line 1, followed by a statement to double its entries on Line 2. We then filter the result on Line 3 by only taking even values and dropping odd ones. The `take` command on Line 4 then ensures that the infinite sequence of mapped and filtered values is limited to only five elements. Finally, all five mapped and filtered values from the original range are collected into a fresh vector on Line 5. This resulting vector then contains the values `6,12,18,24,30`.

[1] `RangeFrom`: https://doc.rust-lang.org/stable/std/ops/struct.RangeFrom.html

```
1  let values = (1..)
2      .map(|x| 3*x)
3      .filter(|x| x % 2 == 0)
4      .take(5)
5      .collect::<Vec<i32>>();
```

Listing 6: Illustrative example of Rust iterator usage.

Line 1 up to Line 4 each introduce a new iterator:

- Line 1 introduces the `RangeFrom<i32>` iterator.
- Line 2 introduces a `Map<RangeFrom<i32>, M>` iterator where `M: FnMut(i32) -> i32` is the type of the map closure.
- Line 3 introduces an `Filter<Map<..., M>, F>` iterator where `F: FnMut(i32) -> bool` is the type of the filter closure.
- Line 4 introduces the terminal `Take<Filter<..., F>>` iterator.

Instantiating these iterators does not mean that any computation happens, for otherwise creating an unbounded range iterator would never terminate. Instead, computation is deferred until the `collect` method is called on Line 5.

2.2.1 The `Iterator` trait

The heart of Rust iterators is the `Iterator`[2] trait. Implementing this trait on a type allows the type to be used in loops and automatically enables chaining as seen in Listing 6.

An abridged version is shown in Listing 7. The trait declares an associated type which defines the type of the returned elements of the iterator. The `fn next` method is a required method and when called returns the immediate next element of the iterator. Furthermore, the trait has numerous provided methods which nest the iterator inside another one as outlined in Section 2.2.3.

In contrast to iterators in other programming languages, there is no such method as `has_next` which indicates whether iteration is completed. Instead, `fn next` returns a `Option<Self::Item>`. The `Option`[3] enum is idiomatic in Rust to indicate the absence of values (as opposed to a `null` type in other languages). Returning `Option::None` indicates that iteration is over, whereas returning `Option::Some(...)` indicates that there might be more elements which will be yielded in future invocations. This implies, that an iterator

[2] `Iterator`: https://doc.rust-lang.org/std/iter/trait.Iterator.html
[3] `Option`: https://doc.rust-lang.org/std/option/enum.Option.html

```rust
trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // methods with default implementations
}
```

Listing 7: Outline of the `Iterator` trait.

should never panic when it is depleted – instead when there are no more elements, the iterator will simply always return `None`.

An illustration of an `Iterator` implementation can be found in Appendix A.1.

2.2.2 Fused iterators

Iterators can return interleaved sequences of `Some` and `None` values arbitrarily. It is technically possible but also explicitly allowed in the documentation of the `Iterator` trait. Fig. 2.1 illustrates the different behavior of a fused iterator and a non-fused iterator when calling their `fn next` method six times. The fused iterator always returns `None` after its first occurrence (and keeps doing so), whereas the non-fused iterator does not need to follow this pattern.

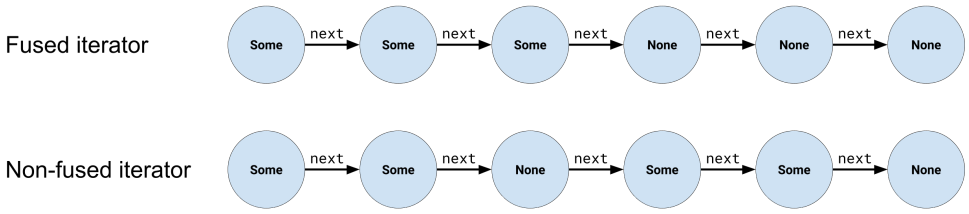


Figure 2.1: Illustration of the behavior of a fused and a non-fused iterator.

Rust's standard library has a marker trait `FusedIterator` to indicate that an iterator keeps returning `None` after the first returned `None`. Any iterator can be fused via the `Iterator::fuse` method.

Any iterator that we consider relevant for this work is a fused iterator. We thus restrict our work to these only.

2.2.3 Iterator adapters

Iterators which contain other iterators are called *iterator adapters*. Examples from the standard library are the `Map`[4] or `Filter`[5] iterators illustrated in Listing 6. Instantiating these adapters can conveniently be achieved via the provided methods on the `Iterator` trait.

Consider the following code snippet, where a `Map` iterator adapts an `Iter` iterator created from a vector of two elements:

```
1 let vec = vec![1,2];
2 let chain = vec.iter().map(|x| 2*x);
3 chain.next(); chain.next(); chain.next();
```

We visualize what happens for each `fn next` call of this iterator chain in Fig. 2.2 on a timeline. When the caller calls `Map::next`, then `Map` simply queries its nested iterator for more elements via a call to `Iter::next`. If the nested iterator yields an element, say `Some(el)`, then `Map` applies the passed closure to the returned `el` and returns that result to the caller. Likewise, if the nested iterator has no more elements, indicated by a returned `None`, then `Map` returns a `None` as well.

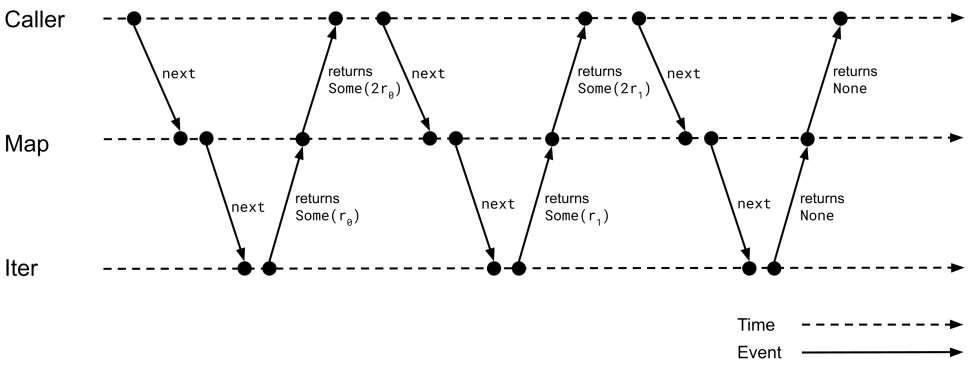


Figure 2.2: Timeline for an iterator adapter when calling its `fn next` method.

An example implementation of `Map` can be found in Appendix A.2.

2.2.4 From collections to iterators

The Rust standard library provides the `IntoIterator`[6] trait which can be used to convert a type into an iterator. The main use for this trait is that

[4] `Map`: https://doc.rust-lang.org/stable/std/iter/struct.Map.html
[5] `Filter`: https://doc.rust-lang.org/stable/std/iter/struct.Filter.html
[6] `IntoIterator`: https://doc.rust-lang.org/std/iter/trait.IntoIterator.html

any type implementing `IntoIterator` can be used in a for loop. For convenience, when a type implements `Iterator` it also automatically implements `IntoIterator`.

A value of a type which implements `IntoIterator` is moved after a call to `into_iter` and thus cannot be used afterwards. This is inconvenient in scenarios where one only wants to traverse the elements of a collection without moving it. Many types in the standard library thus provide `iter()` and `iter_mut()` methods which allow for the iteration over shared references and mutable references of that collection. Conversely, these types, say `T`, also implement `IntoIterator` for `&T` and `&mut T`, respectively.

An example are vector types `Vec<T>`. A vector in Rust can coerce to a slice of type `&[T]` via a process called *deref coercion*[7]. The slice itself provides `iter()` and `iter_mut()` methods which return `Iter`[8] and `IterMut`[9] iterators for immutable and mutable iteration respectively.

2.2.5 From iterators to collections

Having turned a collection into an iterator, it is also possible to go back from the iterator to a collection. This is achieved via the `FromIterator`[10] trait. Often, this is used in combination with the `collect` method, provided by the `Iterator` trait. An application was shown in Listing 6 on Line 5.

2.3 Prusti

Prusti [1] is a deductive verification tool for Rust built on top of the Viper intermediate verification infrastructure. Prusti by default proves the absence of panics during runtime and memory safety of the Rust program by encoding a core proof into the Viper language. Viper then tries to prove the encoding.

As an example, Listing 8 has a main function which panics because $1 \neq 2$ and a `div` function which panics when called with 0. Prusti is able to statically show that these panics might happen when the program is executed and thus returns an error to the user.

Prusti is also able to prove functional specifications provided by the user. Functional specifications can be attached to functions as shown in Listing 9. In that particular example, Prusti *modularly* verifies that the result of the function

[7] Slices can be seen as a contiguous sub-view to the underlying data of the vector. Since a `Vec<T>` coerces to a slice `&[T]`, we can invoke any method on a vector type that is also available on the corresponding slice type. This process is automatically handled by the compiler, and implemented for various standard library types.

[8] `Iter`: https://doc.rust-lang.org/std/slice/struct.Iter.html

[9] `IterMut`: https://doc.rust-lang.org/std/slice/struct.IterMut.html

[10] `FromIterator`: https://doc.rust-lang.org/std/iter/trait.FromIterator.html

```rust
fn divide(num: i32, by: i32) -> i32 {
    num / by
}

fn main() {
    assert!(1 == 2);
}
```

Listing 8: Rust code which panics during runtime.

`increment` indeed is $num + 1$ after assuming that $num >= 10$. *Modular* in this context means that functions are verified independently, i.e., without looking at the surrounding context. Likewise, the call in the `main` function is verified *modularly* by not looking at the method body of `increment`, but only its functional specification. Prusti therefore first *asserts* the precondition to check whether the call is valid and then *assumes* the postcondition. This modular reasoning about function calls is valid because Prusti separately proves that the postcondition holds *if* the precondition is satisfied.

```rust
#[requires( num >= 10 )]
#[ensures( result == num + 1 )]
fn increment(num: i32) -> i32 {
    num + 1
}

fn main() {
    let num = increment(41);
    assert!(num == 5); // Verification fails
}
```

Listing 9: A simple functional specification.

2.3.1 Closures

In a recent work [14, 15], Wolff and Bílý have demonstrated modular verification of Rust closures and implemented it as an extension to Prusti. They provide a way to modularly specify the behavior of a closure including possible side effects due to captured state.

A major challenge in specifying closures comes from the fact that closures are most often passed to higher-order functions, and it is not immediately obvious how a higher-order function specification should look like. They

thus introduced two novel specification concepts: *Specification entailments* and *call descriptions*. Specification entailments are a way to describe the expected behavior of a closure in the context of a higher-order function, whereas call descriptions state that calls to a closure in a higher-order function have happened.

We will now turn our attention to an example in Listing 10 to get an intuition of these two concepts which is taken from the paper [14]. An in-depth description can be found in the paper. Here, we will just summarize the relevant parts which we will need for iterator verification. Therefore we omit some details, such as history invariants to describe the evolution of the captured state of the closure.

```
1  #[requires( cl |= |x: i32| {
2      requires( x > 0 ),
3      ensures( result > 0 )
4  })]
5  #[ensures(
6      exists(|rnd: i32| rnd > 0 &&
7          cl ~~> |arg: i32| -> i32
8              { arg == rnd }
9              { result == outer(result) }
10     ))]
11 #[ensures( result > 0 )]
12 fn hof<F: FnMut(i32) -> i32>(cl: &mut F) -> i32 {
13     let arg = random(); // whose result is > 0
14     cl(arg)
15 }
16
17 fn main() {
18     let dbl =
19         #[requires(true)]
20         #[ensures(result == 2 * x)]
21         |x: i32| -> i32 { 2 * x }
22     let res = hof(&mut dbl);
23     assert!(res % 2 == 0);
24 }
```

Listing 10: Closure specifications.

In the function `fn main`, we define a new closure `dbl` which doubles its argument. Additionally, the closure has a precondition on Line 19 and a postcondition on Line 20, i.e. the closure is always callable and ensures that the result is double the passed-in argument. The higher-order function `fn hof` takes a closure `cl` and calls it with a random argument which we

assume to be always strictly larger than zero.

On Line 1 we see a declaration of a specification entailment. The specification entailment describes the *expected behavior* that any closure passed to `fn hof` must adhere to. In particular, such a closure must have a stronger specification than this expected behavior with respect to behavioral subtyping rules [8]. The requirements in this particular case are indeed fulfilled, as $x > 0 \implies \text{true}$ and $x > 0 \land \text{result} = 2x \implies \text{result} > 0$ are valid.

In general, specification entailments for a closure have the form:

$$\text{cl} \models |a_1, \ldots, a_n| \left\{ \text{requires} \left(P_{exp} \right), \text{ensures} \left(Q_{exp} \right) \right\}$$

where $a_1, \ldots, a_n$ are the parameters of the closure and P_{exp}, Q_{exp} are pre- and postconditions, respectively. The general rules for subtyping against concrete closure specifications P_{cl} and Q_{cl} are [14]:

$$P_{exp} \implies P_{cl} \quad \text{and} \quad \text{old}(P_{exp}) \implies (Q_{cl} \implies Q_{exp})$$

The `old` statement is used to account for state changes during a closure call.

The call description on Line 7 is a way to specify that a call to the closure has happened during the execution of `fn hof`. Note that we put the call description inside an existential quantifier which represents the statically unknown argument of the closure call (which is randomly selected). A call description can describe the state before the closure call and the state after the closure call in curly braces via prestate and poststate assertions. In the example, the prestate assertion `{ arg == rnd }` expresses that the argument of the closure is equal to this existentially quantified random value and the poststate assertion `{ result == outer(result) }` denotes that the result of the closure (`result`) is equal to the result of the higher-order function (`outer(result)`).

Generally, call descriptions have the form

$$\text{cl} \rightsquigarrow |a_1, \ldots, a_n| \, \{P\} \, \{Q\}$$

where $a_1, \ldots, a_n$ are the type-annotated call arguments, and P and Q are pre- and poststate assertions, respectively.

Putting everything together, the verifier gains the following knowledge:

1. `fn hof`'s postcondition on Line 11 is valid due to the specification entailment on Line 1.
2. The call to `fn hof` on Line 22 is valid because the actual specification of `dbl` is stronger than the expected specification of `fn hof`'s closure parameter `cl`.

3. The assertion on Line 23 is valid: We know from the call description of `fn hof` that the result of `fn hof` is the result of a call to the closure with a positive argument. Combined with the specification of `dbl` this implies that `res` must be even.

2.3.2 Subtyping and Refinements

We now turn our attention to functional specifications in the context of traits and their corresponding implementations.

Consider the following code outline, and assume the existence of a trait method f_T with preconditions P_T and postconditions Q_T and an implementation of that method f_I with pre- and postconditions P_I and Q_I, respectively.

```
1  trait SomeTrait {
2      // requires P_T
3      // ensures Q_T
4      fn f(&self, p: i32) -> i32;
5  }
6
7  struct SomeStruct;
8  impl SomeTrait for SomeStruct {
9      // requires P_I
10     // ensures Q_I
11     fn f(&self, p: i32) -> i32 {
12         // ...
13     }
14 }
```

The standard rules of behavioral subtyping [8] apply in this scenario:

$$P_T \implies P_I \quad \text{and} \quad Q_I \implies Q_T$$

For the sake of the example, assume $P_T \equiv 0 \leq p$ and $P_I \equiv p \leq 0$ which violates the behavioral subtyping rules. Consider the following code:

```
1  fn client<T: SomeTrait>(x: T) {
2      x.foo(5)
3  }
```

Modularly reasoning about `client` reveals no problem: We call `x.foo` with the argument `5` which clearly satisfies the precondition of the trait. However, if we call `client` with our wrongly subtyped implementation, the precondition 5 <= 0 is clearly not satisfied and the method call `x.foo(5)` might panic.

Prusti imposes[11] behavioral subtyping rules on traits and their implementations. The process of collecting, checking and selecting the effective specifications which hold for a method call is called *specification refinement*.

Collection Whenever the specification of an implementation has been identified, the specification of the trait has to be identified as well.

Checking For every identified pair of trait / implementation specifications, Prusti has to check whether the behavioral subtyping rules are satisfied.

Selection Computation of the *effective* specification that holds for a call to a method. Modularly verifying `fn client` from above yields the effective specification to be the one from the trait (since we do not know the concrete type of `x`). A call to `fn client` however might depend on another effective specification, since we know the trait's specification *and* the implementation's specification. There are various strategies to compute the effective specification in this scenario. Prusti uses a strategy called *selective replacement*. Selective replacement always picks the most specific specification which is available. A more detailed discussion about this and the alternatives can be found in [4].

Providing no preconditions for a trait method is equivalent to annotating it with `#[requires(true)]`. This has the advantage that one does not always have to provide a specific weaker precondition when the method is called. If no precondition on a trait method would by default implicitly mean that the precondition is `#[requires(false)]`, then one would always need to explicitly mark the implementation of such a trait method with `#[requires(true)]` to make it callable.

Likewise, if no postcondition is defined on a trait method, Prusti treats this as if there were an `#[ensures(true)]`. This is a reasonable assumption, because it is the weakest postcondition possible and implementations may refine it as they wish.

2.3.3 Loops

Prusti supports verification of loops with user-provided invariants. Invariants can directly be declared inside the loop body via the `body_invariant!(.)`

[11] Throughout this thesis, we identified some problems in that matter, especially when generics are involved. It was however agreed upon that fixing them is out of scope for the thesis: https://github.com/viperproject/prusti-dev/issues/1022

macro.

Prusti verifies the loop invariant inductively, i.e. given a `while` loop of the form:

```
while {
    g = G;
    g
} {
    body_invariant!(I);
    B;
}
```

Prusti checks that [11]:

- I holds initially upon entering the loop body, only when g evaluates to `true`
- I is preserved throughout the execution of the body and the guard. That is, after assuming that I is true and executing `B; G`, if g evaluates to true again, then I must hold too.

These checks are needed according to the standard rules of Hoare logic. Especially, in order to validate the *inductive step* (second point above), Prusti forgets any prior knowledge about any variable that is written to in the guard or the body. This process is called *havocking* of the loop targets.

Prusti is not able to handle borrows which cross the loop boundary. As a consequence, it is in currently impossible to write a `for` loop over mutable or immutable borrows such as:

```
let mut v = vec![1,2,3];
let mut iter = v.iter_mut();
for el in iter { /* ... */ }
```

2.3.4 Pure functions and predicates

Often, we want to use functions as part of specifications. This is problematic when the function has side effects (e.g. mutates state) or is non-deterministic. Functions in Prusti thus can be annotated with `#[pure]` to mark it as *deterministic* and *side effect free*. Prusti checks and enforces these rules via various checks, for example the function must not have a parameter which takes a mutable borrow of a type.

Additionally, users can create predicate functions in Prusti with the `predicate!` macro. Predicates can be considered a subset of all pure functions. The difference is that predicates support a superset of Rust syntax, such as quantifiers

or the arrow notation `==>` for implications inside the predicate body. A consequence is, that predicates can only be used in specification code, as the semantics of the additional syntax is not defined Rust.

2.3.5 Trusted functions

Functions can be annotated with `#[trusted]` which makes Prusti completely ignore the body of the function. This is for example needed when the function contains code which is not yet supported by Prusti. It is also convenient if users want to write *ghost functions*; functions which only exists for the sole purpose of verification.

2.3.6 Snapshots

Prusti has two strategies for encoding types: Heap-based encoding [10] and snapshot-based encoding [12]. A *snapshot* can be thought of as a *deep* copy of a value. Snapshots of a value can be compared in specifications using the *snapshot equality* operator `===`.

As an example, consider Listing 11 with a type `Struct` and a function `foo` which uses the snapshot equality operator in a precondition on the two parameters. The precondition states that the snapshot of the parameter `s1` must be equal to the snapshot of the parameter `s2` which in turn means that the snapshots of their fields are equal, i.e., `s1.0 === s2.0`. Ultimately, this will be encoded as an integer comparison in Viper. Thus calling `foo(Struct::new(1), Struct::new(1))` successfully verifies, whereas calling `foo(Struct::new(1), Struct::new(2))` yields a verification error.

Snapshot comparison can be confused with Rust's partial equality operator `==` which is syntactic sugar for the method `PartialEq::eq`. It is generally not the same: `PartialEq::eq` can be arbitrarily implemented, whereas the snapshot equality operator always compares *snapshots* of a value.

```
1  struct Struct(i32);
2
3  impl Struct {
4      #[ensures(result.0 == val)]
5      fn new(val: i32) -> Self { Struct(val) }
6  }
7
8  #[requires(s1 === s2)]
9  fn foo(s1: Struct, s2: Struct) { /* ... */ }
```

Listing 11: Demonstration of the snapshot equality operator `===` in Prusti.

Chapter 3

Methodology

In this chapter we present our approach to specify Rust iterators and introduce the necessary concepts to support them.

We will start by motivating our approach in Section 3.1 with a simple custom iterator-like type and outline the challenge to leverage this specification to *real* iterators implementing the `Iterator` trait.

In Section 3.2, we then address this challenge by introducing *specification extensions* and a novel concept called *type-dependent contracts* in order to assemble a unified approach for Rust iterator specification.

We continue by showing how iterator adapters (see Section 2.2.3) can be specified with respect to their nested iterators in Section 3.3.

Finally, we discuss the specification of converting a data structure into an iterator and converting an iterator into a data structure (see Section 2.2.4 and Section 2.2.5) in Section 3.4.

3.1 Specifying iteration

3.1.1 A general framework for iterator specification

In their 2016 paper [5], Filliâtre and Pereira have showed a general framework for specifying iteration. This framework is not limited to iteration over array-like containers, but supports also iteration that is non-finite, non-deterministic or the result from an algorithm (e.g. when we generate prime numbers on the fly).

The core parts of their framework is a (ghost) sequence *visited* and the two predicates *enumerated* and *completed*. These two predicates "act as an *abstraction barrier*"[5] between iterators and their clients to enable modular specification.

The ghost sequence *visited* Represents the finite sequence of all elements that have been visited so far throughout the iteration. Iterating clients can use *visited* to refer to previously observed values, which is especially useful when defining loop invariants as we will see later.

The *completed* **predicate** Describes whether the iteration is completed or is still ongoing.

For example, if we iterate over an array-like container a (such as a `Vec` in Rust), *completed* can be defined as [5]:

$$completed(v, a) \triangleq \|visited\| = a.len()$$

Here, *completed* has two parameters: v is the ghost sequence of already visited values and a is the array that we are iterating over. The iteration is finished as soon as the length of v is equal to the length a.

A similar definition follows for sets [5], where the termination criterion is that the length of *visited* must be equal to the cardinality of that set.

For infinite iteration, this predicate simply states that the iteration never terminates [5], i.e.

$$completed(v, a) \triangleq false$$

The *enumerated* **predicate** Describes the ghost sequence of already visited values.

Again, for array-like containers, this predicate simply states that the already visited values are a prefix of the whole container [5]:

$$enumerated(v, a) \triangleq \forall i.\, 0 \leq i < \|v\| \implies v[i] = a[i]$$

For iteration over a set s, the enumerated predicate states that every visited value is part of the set and that the visited values are all distinct [5]:

$$enumerated(v, s) \triangleq distinct(v) \wedge \forall x.\, x \in v \implies x \in s$$

3.1.2 First adaption to Rust iterators

In this section, we will conduct a first attempt to link the concepts introduced in Section 3.1.1 to Rust and Prusti. We will not look at a real iterator which implements the `Iterator` trait yet. Instead, we will look at a fake type which exposes the same method as a real Rust iterator would and treat it as an iterator anyway (i.e. pretend that it is usable in a for loop).

Listing 12 shows an iterator-like type `VecIter` with two fields `pos: usize` and `vec: Vec<i32>`, a constructor method `fn new` and a method `fn fake_next` which mimics `Iterator::next`. The behavior of `VecIter` is to iterate over the vector, like Rust's `Iter` iterator.

```
struct VecIter {
    vec: Vec<i32>,
    pos: usize,
}

impl VecIter {
    fn new(vec: Vec<i32>) -> Self {
        Self { pos: 0, vec }
    }

    fn fake_next(&mut self) -> Option<i32> {
        if self.pos < self.vec.len() {
            let val = self.vec[self.pos];
            self.pos += 1;
            return Some(val);
        }
        None
    }
}
```

Listing 12: A fake `Iterator` implementation.

Requirements We start by stating how we would like to use `VecIter` in clients.

First, a client might use `VecIter` *directly*, i.e., without a loop:

```
let mut iter = VecIter::new(vec![13, 37]);
let el = iter.fake_next(); assert!(el.unwrap() == 13);
let el = iter.fake_next(); assert!(el.unwrap() == 37);
let el = iter.fake_next(); assert!(el.is_none());
```

A specification of `VecIter` should be strong enough that all assertions can be proven, as well as that `fn unwrap` does not panic. To prove this, the verifier needs to know whether the immediate returned result from `fn fake_next` is an `Option::Some` or an `Option::None`. Furthermore, in the former case, the verifier needs to know what the value of a returned `Option::Some` is.

Another use case is to use the iterator in a loop. For example, we would like to compute the sum of every element yielded by the iterator and the maximum value as follows:

```
let mut iter = VecIter::new(vec![13, 42, 37]);
let first = iter.next().unwrap();
let mut sum = first;
let mut max = first;
```

```
for el in iter {
    sum += el;
    if el >= max { max = el; }
}
assert!(sum == 13 + 42 + 37);
assert!(max == 42);
```

Specification for non-looping clients A specification for non-looping clients can be simple: We need to add postconditions to `fn fake_next` which describe how the state evolves as we call the method and what the result actually is:

```
#[ensures(self.vec === old(self.vec))]
#[ensures(old(self.pos < self.vec.len()) ==> (
            self.pos == old(self.pos) + 1 &&
            result == Some( self.vec[old(self.pos) ])
))]
#[ensures(old(self.pos >= self.vec.len()) ==> (
            self.pos == old(self.pos) &&
            result == None
))]
fn fake_next(&mut self) -> Option<u32> { /* ... */ }
```

The specification states that `vec` never changes throughout a call to `fn fake_next` and that `pos` increases by one if and only if the iterator is not depleted. Furthermore, the returned result is specified in both cases, such that Prusti safely can assume that unwrapping the `Option` does not panic.

The specification about the non-changing `vec` is essential for verification: `fn fake_next` has a mutable receiver, and since methods are verified modularly in Prusti, a client using this method does not know that the vector remains the same. Dropping this postcondition would cause the verifier to reject the second assertion in the non-looping client.

Specification for looping clients In order to verify the looping clients mentioned earlier, we need suitable loop invariants. For the "compute the maximum" example, we need two invariants to state that the maximum is computed as part of the loop body:

```
for el in iter {
    body_invariant!(exists(|i: usize| i < self.pos ==>
                           max == iter.vec[i] ));
    body_invariant!(forall(|i: usize| i < self.pos ==>
                           max >= iter.vec[i] ));
```

```
    /* ... */
}
```

However, there are two problems with these invariants: First, they rely on iterator-internal fields (such as `pos`). Second, even if we had a specification such as in the non-looping client, these invariants would be too weak.

To see why, we need to look at a different, but semantically equivalent version of the loop, using while syntax with an explicit loop guard:

```
let mut next = None;
while {
    next = iter.fake_next();
    next.is_some()
} {
    // ... invariants ...
    let el = next.unwrap()
}
```

Recall from Section 2.3.3, that Prusti proves the two invariants *inductively*. In the inductive step, Prusti havocs all loop targets, which includes `iter`, as we make a mutable call to one of its methods. By havocking any knowledge about `iter`, we also lose any knowledge about its state, namely the values of `vec` and `pos`, even if they are part of the specification of `fn fake_next`.

We could try to manually frame this knowledge as part of loop invariants:

```
body_invariant!(self.vec == original_vec);
body_invariant!(self.pos == i);
```

where `original_vec` is a copy of the original vector and `i` is a running variable that we manually manage in the loop.

Again, these additional invariants rely on the internals of the iterator. This is undesirable because we want our specification be modular. Thus, to arrive at a better solution, we will now adapt the technique from Section 3.1.1.

A ghost sequence of visited values As a start, we need a ghost sequence of already visited values, ideally for all iterators. Prusti does not provide such a ghost sequence yet. We will circumvent this by creating our own `GhostSeq<T>` type[1] which is to be considered an abstract, specification-only type.

[1] A detailed explanation and implementation of `GhostSeq<T>` can be found in Appendix A.3. For now, we treat this type as if it were a *real* ghost type which can be created, indexed, extended and compared to other ghost sequences.

We provide `GhostSeq<T>` via a function `fn visited` defined on a trait which is implemented via a blanket implementation on all iterators:

```
trait IteratorSpecVisited {
    type Item;

    #[pure] #[trusted]
    fn visited(&self) -> GhostSeq<Self::Item> {
        unimplemented!()
    }
}

impl<I, T> IteratorSpecVisited for I
    where I: Iterator<Item = T> {
    type Item = T;
}
```

Predicate implementations The predicates `completed` and `enumerated` can be implemented for `VecIter` as Prusti predicates. Recall from Section 2.3.4 that Prusti predicates support a superset of Rust syntax, allowing us to use quantifiers inside `enumerated` to achieve the desired specification.

The `completed` predicate simply states that the iteration is over as soon as `pos` is equal to the length of `vec`:

```
predicate! {
    fn completed(iter: &VecIter) -> bool {
        iter.pos == iter.vec.len()
    }
}
```

The `enumerated` predicate first and foremost has to describe the ghost sequence of already visited values. However, we ultimately want it to be used as part of loop invariants, meaning it ideally also describes the evolution of the iterator's state throughout an unknown amount of calls to the iterators `fn fake_next` method. This means, that we want to establish a *type invariant* with this predicate which is strong enough such that Prusti can prove a loop invariant that uses the predicate *inductively*.

To achieve that, `enumerated` should be *transitive*, i.e. for an observed sequence of iterator states $I_0, I_1, \ldots, I_n$ and any three states of that sequence I_l, I_m, I_o with $l \leq m \leq o$, if $enumerated(I_l, I_m)$ and $enumerated(I_m, I_o)$ holds, then $enumerated(I_l, I_o)$ must hold too.

`enumerated` thus takes two parameters: The *current* state of an iterator and *some previous* state of that iterator. We then can describe the evolution of the iterator's state with respect to an older version of itself.

```
predicate! {
    fn enumerated(prev: &VecIter, iter: &VecIter) -> bool {
        // Visited sequence
        iter.pos == iter.visited().len() &&
        forall( |i: usize| i < iter.visited().len() ==>
            iter.visited()[i] == iter.vec[i] ) &&

        // State evolution
        iter.vec === prev.vec &&
        iter.pos >= prev.pos
    }
}
```

Notice that we require `enumerated` to be *reflexive* as well. We see why when we add it to the specification of `fn fake_next`.

Adding specifications for VecIter Having set up the predicates and the visited ghost sequence, we can provide a specification for `fn fake_next`, shown in Listing 13. Note that we also updated the body of the method to update the sequence of visited values on Line 26. The specification is divided in three different blocks:

General specification Consists of specifications which are agnostic of the actual iterator. That is, they solely rely on the defined predicates and should be applicable to all iterators. Thus, this specification could be part of the specification of the trait `Iterator::next`. Line 2 and Line 3 establish the `enumerated` invariant as a pre- and postcondition. Since we can not use an `old(.)` expression as part of a precondition, we simply pass the current state of the iterator twice into the predicate. This is the reason for our reflexivity requirement for `enumerated`. On Line 4 up to Line 8, we specify whether the returned result is an `Option::None` or an `Option::Some`, as well as the condition that a completed iterator remains completed[2].

State update Contains iterator-specific specifications which specify the post-state after a call to `fn fake_next`. Generally, we would state here that `vec` does not change and `pos` increases by one if the iterator is not completed. However, the former is already part of the `enumerated` invariant, so we only strengthen the knowledge about `pos` on Line 11 and Line 14.

Result Specifies the returned value by concretizing the contained value of the returned `Option` and linking it to the ghost sequence.

[2]See Section 2.2 about fused iterators: We generally assume that our iterators are fused.

```
1  // General specification
2  #[requires( enumerated(self, self) )]
3  #[ensures(  enumerated(old(self), self) )]
4  #[ensures(
5      ( !old(completed(self)) == result.is_some() ) &&
6      ( old(completed(self)) == result.is_none() ) &&
7      ( old(completed(self)) ==> completed(self) )
8  )]
9
10 // State update
11 #[ensures(!old(completed(self)) ==> (
12     self.pos == old(self.pos) + 1
13 ))]
14 #[ensures(old(completed(self)) ==> (
15     self.pos == old(self.pos)
16 ))]
17
18 // Result
19 #[ensures(result.is_some() ==> (
20     result == Some(self.vec[old(self.pos)])
21 ))]
22 fn fake_next(&mut self) -> Option<u32> {
23     if self.pos < self.vec.len() {
24         let val = self.vec[self.pos];
25         self.pos += 1;
26         self.visited.push(val);
27         return Some(val);
28     }
29     None
30 }
```

Listing 13: Specification for the fake `VecIter` iterator.

Verification of the method As one can easily check, the specification describes what `fn fake_next` does and thus Prusti successfully verifies the method body against this specification. For simplicity, we omitted some details. For example, Prusti would need more information about the visited sequence update on Line 26.

Verification of non-looping clients Going back to our initially proposed clients, one can also immediately see that the specification shown in Listing 13 is strong enough to verify the non-looping client as this specification is just a generalization of the initial proposed simpler specification. In particular, Prusti also knows that `Option::unwrap` does not panic: We added a precondition `#[requires(self.is_some())]` to this method (omitted in the

listing), which is checked automatically when we call this method.

Verification of looping clients The invariants of the looping client can be rewritten to take advantage of the enumerated predicate *and* the ghost sequence of already visited values. For example, computation of the maximum can be achieved with the following code:

```
1  let mut iter = VecIter::new(vec![13, 42, 37]);
2  let snap_iter = snap(iter);
3  let mut next = iter.fake_next();
4  let mut max = next.unwrap();
5  while {
6      next = iter.fake_next();
7      next.is_some()
8  } {
9      body_invariant!(enumerated(snap_iter, iter));
10     let el = next.unwrap();
11     if el >= max { max = el; }
12     body_invariant!(exists(|i: usize| i < iter.visited().len()
13                                 ==> max == iter.visited[i]));
14     body_invariant!(forall(|i: usize| i < iter.visited.len()
15                                 ==> max >= iter.visited[i]));
16 }
17 assert!(max == 42);
```

Here, we manually keep track of a *snapshot* of the iterator, which stores the state prior to the call to `fn fake_next` to be used as part of the invariant `enumerated`.

We now informally reason why the specification is enough for the assert on Line 17 to verify. We denote with I_i the state of `iter` after i calls to its `fn fake_next` method. For example, after Line 1, `iter` has state I_0 with $pos(I_0) = 0$. From the code, we also observe that `snap_iter` always refers to state I_0.

First, we assume that `fn new` has a suitable specification, such that we know that after the construction of a `VecIter` $\neg completed(I_0)$ and $enumerated(I_0)$ holds. Consequently, the call on Line 3 is valid. Furthermore, by the postcondition of `fn fake_next`, we know that the returned value of this call is `Some(13)`, and thus can be unwrapped in Line 4 and assigned to the initial maximum value.

Second, the invariants hold upon entering the loop: $enumerated(I_0, I_1)$ holds by the postcondition of `fn fake_next`. From the actual definition of `enumerated`, we know that `iter.visited().len()` is equal to 1 and the value of this element is 13. This implies that the other invariants hold too.

Third, the invariants are preserved throughout the loop iteration. Assume we are in the k-th iteration for some $k > 0$. From the induction base, we know that $enumerated(I_0, I_{k-1})$ holds as well as that `max` is the maximum of `iter.visited()` and contained in it. We furthermore can assume everything that happens as part of the loop guard, namely:

- There exists a new iterator state I_k, such that $enumerated(I_{k-1}, I_k)$ holds.
- `next.is_some()` is true.
- The value *val* of `next` is the last element of `iter.visited()` (where `iter` has state I_k).

From the loop body, we see that the first invariant is preserved because we do not change any internals of the iterator *and* enumerated is transitive. For the latter two invariants, we can perform a case distinction: If $val < max$, then max remains the same and the second two invariants hold by the induction hypothesis. If $val >= max$, we update `max` to be equal to val. Since val is contained in `visited` at position $pos(I_{k-1})$, we deduce that `max` is also contained in `visited` at position $pos(I_{k-1})$. It follows that all invariants are preserved throughout a loop iteration.

Lastly, after the loop we know that `next.is_none()` holds and thus $completed(I_n)$ holds for some n. Combined with the transitive guarantees of *enumerated*, we deduce that `iter.visited()` is equal to `iter.vec`. Most importantly, we know as part of *enumerated* that `iter.vec === vec![13, 42, 37]`. Combining this with the invariants, we know that $max == 42$ and the assertion verifies.

3.1.3 Towards a generalized Rust iterator specification

In Section 3.1.1, we have introduced a general framework for specifying iteration and in Section 3.1.2 we came up with a first adaption to Rust.

The adoption, however, is not enough: We did not really implement the `Iterator` trait for `VecIter`, it just happened to be the case that `VecIter` has a method with the same signature as `Iterator::next`. Instead, our iterator should implement the trait itself with the same body.

To arrive at a real implementation of the `Iterator` trait for `VecIter`, we first observe that we need a specification for `Iterator::next`: Recall from Section 2.3.2 that Prusti enforces behavioral subtyping rules. `VecIter::fake_next` has a precondition, namely `#[requires(enumerated(...))]`. Not annotating `Iterator::next` with a suitable precondition would immediately lead to an invalid refinement of `VecIter::next`, as $true \implies enumerated(\dots)$ in general is *not* valid for non-trivial definitions of *enumerated*.

One option is to annotate `Iterator::next` with `#[requires(false)]` which solves the refinement problem. We can do more though: Recall from List-

```
trait IteratorSpec: Iterator {
    predicate! { fn completed(&self) -> bool; }
    predicate! { fn enumerated(&self, prev: &Self) -> bool; }
    predicate! { fn post(old_self: &Self,
                         new_self: &Self,
                         res: &Option<Self::Item>) -> bool; }
}
```

Listing 14: The trait `IteratorSpec` which introduces a specification-only abstraction between the `Iterator` and its implementations.

ing 13 that we have divided the functional specification of `VecIter::fake_next` into three different groups. The first group will look the same for every iterator modulo the definitions of the predicates. If we treat the two predicates as abstract placeholder predicates which will be defined for concrete iterators, this specification could be generalized and attached to `Iterator::next`.

Another related open point is how we even specify `Iterator` at all, since this trait is part of the standard library and can not be directly annotated with specifications. For the remainder of Chapter 3, we treat these standard library traits and types as if we could directly add specifications to them. We present our solution to handle non-local code in Section 4.2.

3.2 Specification extensions

In Section 3.1.3, we came to the conclusion that we *need* a specification for `Iterator`, for otherwise this will lead to invalid refinements on concrete iterators.

The two predicates *enumerated* and *completed* already provide a suitable abstraction between iterators and their clients. We can leverage this idea and introduce an abstraction between the specification of the `Iterator` trait and the specification of iterator implementations. Abstraction in Rust is achieved via traits and we thus introduce the `IteratorSpec` trait, shown in Listing 14. We call this trait a *specification extension*: The implementation of the trait *is* specification code. This trait defines three *abstract predicates*, `fn completed`, `fn enumerated` and `fn post`. An *abstract predicate* means that the implementing type must implement this method as a Prusti predicate. The first two predicates were already explained in Section 3.1.2. The third predicate, `fn post`, may be used to specify the side effects of *one* call to `fn next`, including a specification for the returned result of that call.

The idea is then the following: The specification of `Iterator` is abstractly defined via the indirection of `IteratorSpec`. We then implement the specification extension for our desired iterator. On the callsite, for a call to

```
trait Iterator {
    #[type_contract(Self: IteratorSpec, [
        requires( self.enumerated(self) ),
        ensures ( self.enumerated(old(self)) ),
        ensures ( !old(self.completed()) == result.is_some() ),
        ensures ( old(self.completed()) == result.is_some() ),
        ensures ( !old(self.completed())
                                ==> old(self.completed()) ),
        ensures ( IteratorSpec::post(old(self), self, &result) ),
    ])]
    fn next(&mut self) -> Option<Self::Item>;
}
```

Listing 15: A type-dependent contract to conditionally provide a specification for `Iterator` which is only active when an implementor of `Iterator` implements `IteratorSpec`.

`Iterator::next`, this *actual* implementation is used to encode the method call.

3.2.1 Type-dependent contracts for iterators

Having described the specification-indirection via the `IteratorSpec` trait, we need to attach it to `Iterator` somehow. Not every iterator implements `IteratorSpec`, but *if* it does, we want to strengthen the specification of `Iterator::next` with this additional knowledge. To this end, we propose to use a novel technique called *type-dependent contracts* to attach contracts to methods *conditionally*.

Listing 15 shows a type-dependent contract for the `Iterator` trait. It consists of two parts: A constraint and an unbounded amount of `requires` and `ensures` clauses.

The constraint describes a condition for which types the nested specifications should be active. It is syntactically equivalent to a Rust `where` clause (without the keyword) and can access its surrounding context (e.g. generics of the method or the trait). In the example we constrain the specification to hold only if `Self: IteratorSpec`, i.e. for implementations of `Iterator` which also implement `IteratorSpec`.

The `requires` and `ensures` clauses are just ordinary Prusti specifications.

Type safety

The specifications themselves are type-checked like every specification in Prusti. That is, specifications inside a type-dependent contract are aware

of the fact that they are only active if `Self: IteratorSpec`, and thus, the specification can be successfully type-checked.

Subtyping

If an iterator `SomeIter` implements `IteratorSpec` and does *not* have any separate annotations on `SomeIter::next`, then Prusti will automatically inherit the specifications of `Iterator::next`. Since the condition of the type-dependent contract is satisfied, these specifications will effectively be all specifications which are part of the type-dependent contract. Subtyping is then trivially satisfied.

Notice that *if* we had custom specifications on `SomeIter::next` this would not automatically work: The refinement strategy of Prusti inherits the specifications of the trait only if the implementation has no specifications. In this case, the specifications of the implementation would need to satisfy the subtyping check.

On the other hand, if `SomeIter` does not implement `IteratorSpec`, then there is nothing to check.

Encoding method calls

Encoding of method calls in the presence of type-dependent contracts works analogously to encoding *ordinary* method calls. The only difference is that we first statically determine whether the constraint of the type-dependent contract is active or not. If it is active, we take the additional specifications of this contract into account. It is important to mention though that the predicates are not abstract anymore: At the callsite, we have precise knowledge of the receiver of a call to `Iterator::next`, and thus we also know to which concrete predicates the contract refers to.

3.2.2 Putting everything together: Specification of `VecIter`

With the given techniques, we are able to provide a specification for `VecIter` in the new setting. The specification of the `Iterator` remains the same as in Listing 15, and we only need to implement the specification extension as shown in Listing 16.

The general procedure for a specification is then always the same: Implement the specification extension which describes the already visited values as part of the *enumerated* predicate and describe the termination criterion via the *completed* predicate. Additionally, we add a specification to the iterator's `fn next` method which describes state changes as well as the value of the returned `Option` result.

```
impl IteratorSpec for VecIter {
    predicate! {
        fn completed(&self) -> bool {
            iter.pos == iter.vec.len()
        }
    }

    predicate! {
        fn enumerated(&self, prev: &Self) -> bool {
            // Visited sequence
            iter.pos == iter.visited.len() &&
            forall( |i: usize| i < iter.visited.len() ==>
                iter.visited[i] == iter.vec[i] ) &&

            // State evolution
            iter.vec == prev.vec &&
            iter.pos >= prev.pos
        }
    }

    predicate! {
        fn post(old_self: &Self,
                new_self: &Self,
                res: Option<Self::Item>) -> bool {
            !old(self.completed()) ==>
                ( self.pos == old(self.pos) + 1 ) &&
            old(self.completed()) ==>
                ( self.pos == old(self.pos) ) &&
            res.is_some() ==>
                ( res == Some(self.vec[old(self.pos)]) )
        }
    }
}
```

Listing 16: Implementation of `IteratorSpec` for `VecIter`.

3.3 Describing effects of nested iteration

Iterator adapters (see Section 2.2.3) are heavily used in Rust code. In fact, many of the standard library iterators are iterator adapters which take the result of their nested iterator and refine it before returning it.

We want our specifications for iterator adapters to be modular, i.e. the specification should be independent of the concrete nested iterator (and its respective specification). To that end, we need a way to specify the result of an adapter in *relation to the* result of the nested iterator.

Throughout this section, we will exemplify iterator adapter specification on the `Map` iterator. An implementation is shown in Appendix A.2.

3.3.1 Introduction

As outlined in Section 3.1.1, modular specification of iterators is achieved via the predicates *enumerated* and *completed*, which describe the already visited elements of an iterator thus far. This remains the same for adapters: We want to describe every element visited so far, but we now rely on the already visited elements of the nested iterator.

To get a general idea about how this works, let V_k^A be the visited values after k calls to the adapter and V_k^N the visited values after k calls to the nested iterator. Consider iteration over a vector `vec![1,2,3,4]` using the `Iter` iterator, nested inside a `Map` iterator which simply doubles all the elements. The following table illustrates the evolution of the `visited` sequences after a certain amount of calls to `fn Map::next`:

k	0	1	2	3	4
V_k^{Iter}	$\varnothing$	1	1, 2	1, 2, 3	1, 2, 3, 4
V_k^{Map}	$\varnothing$	2	2, 4	2, 4, 6	2, 4, 6, 8

After the k-th call to `Map`'s `next` method, it holds that

$$\forall i.\, 0 \leq i < \|V_k^{\text{Map}}\| \implies V_k^{\text{Map}}[i] = 2 \cdot V_k^{\text{Iter}}[i]$$

That is, `Map`'s i-th visited element is simply `Iter`'s i-th element, doubled.

`Map` might arguably be the simplest form of an iterator adapter because there is a one-to-one relation of visited values: The amount of visited values always matches the amount of visited values of the nested iterator, i.e. $\|V_k^{\text{Map}}\| = \|V_k^N\|$.

```rust
let mut count = 0;
let f = |x: i32| {
    count += 1;
    2 * x + count;
}
let res = vec![1,2,3,4].iter().map(f);
assert_eq!(res, vec![3, 6, 9, 12]);
```

Listing 17: Using `Map` with a closure that captures and mutates its surrounding context by increasing `count` with every invocation.

There are more complicated scenarios though: For example, a `Filter` or a `Skip` iterator yields at most as many elements as their nested iterator:

$$\|V_k^A\| \leq \|V_k^N\|$$

In contrast, `Repeat` increases the amount of elements:

$$\|V_k^A\| \geq \|V_k^N\|$$

3.3.2 State evolution of closures

Recall from Section 2.1.2 that closures can capture and mutate the surrounding context as shown in Listing 17 where the `Map` iterator does not only double the passed value, but also adds `count` to it which is increased with every invocation.

A specification of already visited values thus has to always take any captured state of a closure into account.

k	0	1	2	3	4
V_k^{Iter}	$\varnothing$	1	1, 2	1, 2, 3	1, 2, 3, 4
V_k^{Map}	$\varnothing$	3	3, 6	3, 6, 9	3, 6, 9, 12

To this end, let f_k be the closure in its k-th state. Again, after k calls to `Map`'s `fn next` method, we obtain the following description for the k already visited elements:

$$\forall i.\, 0 \leq i < \|V_k^{\text{Map}}\| \implies V_k^{\text{Map}}[i] = f_k\left(V_i^{\text{Iter}}[i]\right)$$
$$\text{where} \quad f_k(x) = 2 \cdot x + k$$

```
1  impl<I: IteratorSpec, ...> IteratorSpec for Map<I, ...> {
2      predicate! {
3      fn completed(&self) -> bool {
4          self.iter.completed()
5      } }
6
7      predicate! {
8      fn enumerated(&self, prev: &Self) -> bool {
9          // ...
10         self.iter.enumerated(&prev.iter) &&
11         self.visited().len() == self.iter.visited().len() &&
12         forall(|i: usize| i < self.visited().len() ==>
13             self.visited()[i] == 2 * self.iter.visited()[i]
14         )
15         // ...
16     } }
17
18     // ...
19 }
```

Listing 18: Specification for a simplified `Map` iterator.

3.3.3 A first specification for the `Map` iterator

We are now able to put together a first specification for the `Map` iterator. We will focus on the definition of *enumerated* and *completed*.

First we look at a simplified version of `Map` where we assume that the used closure simply doubles its value (and has no captured state). Furthermore, we assume that `Map` has a field `iter`, the nested iterator. As a last assumption, we assume that the nested iterator also implements the `IteratorSpec` trait.

A specification is shown in Listing 18 as part of an `IteratorSpec` trait implementation of `Map`. This example, albeit not terribly exciting, shows how we can specify V_k^{Map} in the presence of a nested iterators visited sequence V_k^N. Generally, the specification can be seen as part of the k-th call to `Map::next`, and our main goal is to describe V_k^{Map} in the postcondition of that call.

First, on Line 3, we specify that `Map` is completed when the nested iterator is completed. Next, in order that `Map` is enumerated, the nested iterator must be enumerated as well, i.e. V_k^N is fully determined according to the specification of the nested iterator. Having that knowledge, we can then proceed by connecting V_k^{Map} and V_k^N via a universal quantification over all visited values.

```
fn enumerated(&self, prev: &Self) -> bool {
    // ...
    self.iter.enumerated(&prev.iter)
    self.cl_states.len() == self.visited().len() + 1 &&
    forall(|i: usize| i < self.visited().len() ==>
        F::call_mut ~~> |cl: F, arg: I::Item| -> B
            { cl === self.cl_states[i] &&
              arg == self.iter.visited()[i] }
            { cl === self.cl_states[i+1] &&
              self.visited()[i] == result }
    )
    // ...
}
```

Listing 19: Specification for the `Map` iterator with a closure.

Taking closures into account

We now turn our attention towards a more exciting example, where `Map` has a *real* closure, potentially with captured state. As outlined in Section 3.3.1, V_k^{Map} is now dependent on the result of the closure in a certain state. In particular, we need to connect every $V_k^{\text{Map}}[i]$ with the result of a call to the closure in its i-th state (i.e., it has already been called $i-1$ times), whose argument was the result of the nested iterator.

We outline a specification in Listing 19. Notice that we require the existence of a ghost sequence of closure states `cl_states` which is initialized properly.

First, we again universally quantify over all visited values of `Map`. What follows inside the universal quantifier is a call description for the closure. In the prestate assertion, we link the closure state to a sequence of *observed* closure states. Furthermore, we constrain the argument to be $V_k^N[i]$, the i-th visited value of the nested iterator. In the poststate assertion, we again link the possibly changed closure state to the next position in the sequence of observed closure states as well as linking the closure's result to $V_k^{\text{Map}}[i]$.

This specification is not yet strong enough to be used in any clients. The missing part is how the state of the nested iterator evolves for one call to `Map::next`. This knowledge is needed: For example, if we used `VecIter` as part of `Map`, we need to know that the `pos` field of `VecIter` increases by one if we call `Map::next` (and there are more elements available). We can encode this as part of the `fn post` specification function of `Map`:

```
fn post(old_self: &Self,
        new_self: &Self, res: Option<Self::Item>) -> bool {
    // ...
```

```
    exists(|nested_res: Option<I::Item>| {
        IteratorSpec::post(
            old_self.iter,
            new_self.iter,
            &nested_res
        )
    })
    // ...
}
```

The specifications shown before, albeit syntactically simple, are very powerful: First, they are modular as they do not depend on the nested iterator and the concrete closure. The nested iterator is hidden behind the specification extension `IteratorSpec` and `Map`'s specification depends solely on this extension, meaning we can use *any* nested iterator which provides an implementation of that trait.

Second, the specification is powerful because after a certain amount of calls to `Map::next`, we know the precise state of the closure and both visited sequences of the two involved iterators. This is illustrated as part of the following listing[3]:

```
let mut count = 0;
let f =
    #[ensures(result = 2*x + count)]
    |x: i32| -> i32 {
        count += 1;
        2 * x + count
};
let map = [0..].iter().map(f);
// We know as part of initialization of Iter and Map:
// cl_states = f_0 where f_0.count = 0
// V_0^Iter = ∅
// V_0^Map = ∅

map.next(); ...; map.next(); // k times

// We know from Map's specification:
// cl_states = f_0, ..., f_{k+1} where f_k.count = k
// V_k^Map = f_0(V_k^Iter[0]), ..., f_k(V_k^Iter[k])

// and from Iter's specification:
// V_k^Iter[i] = i
//     ⟹ V_k^Map = f_0(0), ..., f_k(k)

```

[3]We hereby assume that `Iter` is specified.

```
// and the specification of the closure:
// c_k(x) = 2 · x + k
//     ⟹ V_k^Map = 0, ..., 2 · k + k
```

3.3.4 Offsetting

In the previous sections, we have assumed that the i-th element of V_k^{Map} is always at the i-th position of V_k^N as well as $\|V_k^{\text{Map}}\| = \|V_k^N\|$. This is not always the case, as illustrated in the following snippet:

```
let v = vec![1,2,3,4];
let mut iter = v.iter();
iter.next();
// V_1^Iter = 1

let mut map = iter.map(|x| x);
map.next();
// V_2^Iter = 1, 2
// V_1^Iter = 2
```

If the nested iterator has already visited j elements, then using it as part of an iterator adapter means that the adapter's visited sequence is *off* by j elements.

We can fix this by keeping an *offset* flag as part of `Map`. When `Map` is created, we initialize this offset to be equal to the length of the visited sequence of the nested iterator. Furthermore, the `enumerated` implementation of `Map` needs to account for this additional knowledge:

$$\|V_k^{\text{Map}}\| = \|V_k^{\text{Iter}}\| - \text{offset} \wedge$$
$$\forall i.\, 0 \leq i < \|V_k^{\text{Map}}\| \implies V_k^{\text{Map}}[i] = f_k\left(V_i^{\text{Iter}}[i + \text{offset}]\right)$$

3.3.5 Specification in the absence of `IteratorSpec`

In section Section 3.3.3, we have shown how we can specify an iterator adapter by linking its ghost sequence of visited values to the nested iterators ghost sequence of visited values. Concretely, our definition of `Map::enumerated` relies on `I::enumerated` where `I` is the type of the nested iterator. We thus implemented `IteratorSpec` for `Map` *only* if `I` also implements `IteratorSpec`, indicated by a trait bound on the generic type parameter of the nested iterator. With this additional information, we were able to ensure that the

visited sequence of the nested iterator is properly specified as part of the specification of the adapter.

In an ideal scenario, every iterator of the standard library would implement this specification extension and all iterator adapter specifications can then rely on this additional knowledge. This is highly idealized, and users might come up with their own iterator implementations, *forcing* them to implement `IteratorSpec` in order that they get a specification when it is nested inside one of the (conditionally) specified adapters. This is too restrictive, and we might need a better solution.

To this end, we get rid of the type parameter bound on the nested iterator (e.g. `I: IteratorSpec` in Listing 18) for any iterator adapter specification extension implementation, say `Map` as our running example. This effectively means that we have less information available for our iterator adapter specifications, as we have no explicit knowledge whether the nested iterator implements `IteratorSpec` or not. As a consequence, the specification of the adapter can not enforce that the visited sequence of the nested iterator is fully specified.

Ultimately, we still want to specify the visited ghost sequence of `Map` as part of its `enumerated` predicate implementation. We observe that given an iterator is in some state I_k, and we call its `fn next` method, this will lead to the following result: Either we get a `None`, which marks completion of the iteration[4], or we get `Some` value r_k with a potential new state of the iterator I_{k+1}. We denote this behavior as a *configuration* of the iterator $C_k = \langle I_k, r_k \rangle$, which can be read as: If the iterator is in some state I_k, it will return result r_k. We denote the absence of a returned value with $\varnothing$. Given a configuration C_k and a call to `Iterator::next(C_k)`, this induces a new configuration C_{k+1}, denoted as $C_k \overset{next}{\rightsquigarrow} C_{k+1}$.

As an example, consider iteration over a vector with k elements using the slice iterator `Iter`. We observe the following infinite *trace* of configurations:

$$\langle I_0, r_0 \rangle \overset{next}{\rightsquigarrow} \langle I_1, r_1 \rangle \overset{next}{\rightsquigarrow} \ldots \overset{next}{\rightsquigarrow} \langle I_k, r_k \rangle \overset{next}{\rightsquigarrow} \langle I_{k+1}, \varnothing \rangle \overset{next}{\rightsquigarrow} \langle I_{k+2}, \varnothing \rangle \overset{next}{\rightsquigarrow} \ldots$$

Using `Iter` as a nested iterator inside a `Map` iterator, we observe the following: After we have called `Map::next` k times without observing any `None` value, we can assume a trace T_{k+1} of the nested iterator with length $\|T_{k+1}\| = k + 1$ and arrive the following formula[5] for `Maps` ghost sequence of visited values V_k^{Map}:

$$\begin{aligned} &\|V_k^{\text{Map}}\| = \|T_{k+1}\| \wedge \\ &\forall i.\, 0 \leq i < \|V_k^{\text{Map}}\| \implies V_k^{\text{Map}}[i] = f_k(r_i) \quad \text{where } \langle I_i, r_i \rangle \in T_{k+1} \end{aligned}$$

[4] Recall that we assume that iterators are fused.

[5] We omit offsetting for the sake of simplicity.

Notice at this point, that this formula is independent of V_k^N; the ghost sequence of visited values of the nested iterator. It is purely based on the *observation* of state changes induced by method calls, as well as their returned result.

To derive a specification for this mathematical description, we need to describe the trace T_{k+1} as part of the specification of `Map`. Concretely, we need a way to describe a call $\overset{next}{\rightsquigarrow}$ with its induced state changes as well as the returned result. This is exactly what we can achieve with call descriptions for closures: Recall from Section 2.3.1 that a call description $cl \rightsquigarrow (args)\ \{P\}\ \{Q\}$ describes that a call to cl happened, where P is an assertion which held prior to the call and Q is an assertion which held after the call.

We can leverage closure call descriptions to generalized call descriptions, which describe the call to any function, including trait methods. In our running example, we can *link* two configurations $\langle I_k, r_k \rangle \overset{next}{\rightsquigarrow} \langle I_{k+1}, r_{k+1} \rangle$ with a call description:

$$\texttt{Iterator::next}\ (iter) \rightsquigarrow \{iter = I_k\}\ \{iter = I_{k+1} \land result = r_k\}$$

This call description describes that some call to the method `Iterator::next` happened on some iterator instance *iter*. Prior to the call, this instance of the iterator was in some state I_k, and after the call, the instance changed its state to a new state I_{k+1}[6]. Additionally, in the poststate assertion, we have access to the returned value, and thus can link it with r_k. Notice that some details are omitted for simplicity, as *result* is actually an `Option` and not a concrete value. We deal with such details later.

Generalized call descriptions

Generalized call descriptions are an extension of closure call descriptions [14, 15] for *any* function call as opposed to closure calls. This includes ordinary functions, associated functions or methods[7].

Syntactically, there is nothing we need to change. A call to a method `foo.bar(arg1, arg2)` where the type of the receiver `foo` is `Foo` can be equivalently written as `Foo::bar(foo, arg1, arg2)`. A generalized call description thus syntactically has the form:

$$M \rightsquigarrow |recv*, a_1, ..., a_n|\ \{P\}\ \{Q\}$$

[6] The equality in $iter = I_k$ can be interpreted as *comparing* a snapshot of the state of *iter* to the (snapshot) state I_k.

[7] A method is an associated function whose first parameter is `self`, an associated function is a function associated with a type (i.e. a function inside an implementation block), see https://doc.rust-lang.org/reference/items/associated-items.html#associated-functions-and-methods

where M is the identifier of the function (e.g., `Type::method`), $recv*$ is the receiver (only relevant if M is a method) and $a_1, \ldots, a_n$ are the additional arguments that M has been called with. As in closure call descriptions, P is a prestate assertion that is evaluated prior to the call and Q a poststate assertion that is evaluated after the call.

The encoding works as for closure call descriptions[8]. Assuming we have a call description like before, this will be encoded like the following first-order formula:

$$\begin{aligned}&\exists\, recv, \overline{recv}, a_1, \overline{a_1}, a_2, \overline{a_2}, \ldots, r\,.\\ &\quad pre(\overline{recv}, \overline{a_1}, \ldots) \wedge P(\overline{recv}, \overline{a_1}, \ldots) \wedge\\ &\quad post(recv, \overline{recv}, a_1, \overline{a_1}, \ldots, r) \wedge Q(recv, \overline{recv}, a_1, \overline{a_1}, \ldots, r)\end{aligned}$$

where $recv$ denotes the state of the receiver of the method call, $a_1, \ldots, a_n$ are the arguments, r is the result of the call and pre and $post$ are the refined[9] pre- and postconditions of the function. A bar ($\overline{recv}$) indicates that this is a pre-call value.

3.3.6 Specification of `Map` with call descriptions

We now are able to provide a specification for `enumerated` for `Map` using generalized call descriptions.

The annotated specification is shown in Listing 20, where we again assume the existence of a ghost sequence of closure states `cl_states` on the iterator, as well as a ghost sequence of iterator states `iter_states` of the nested iterator and a ghost flag `completed` which signals completion of the iterator.

Callsite monomorphization

Notice that the specification in Listing 20 is completely unaware of the fact whether the nested iterator `I` implements `IteratorSpec` or not.

Assume we use two iterators as nested iterators for `Map` as shown in Listing 21.

Furthermore, assume that `Iter1` does not implement `IteratorSpec` but has a simple postcondition on its `fn next` method which states that all yielded elements are positive. On the other hand, we assume that `Iter2` implements `IteratorSpec`. What do we know for the two calls on Line 6 and Line 7?

[8] Closures in Prusti can have invariants. Invariants on types do not yet exist in Prusti, so we left that detail out in the encoding.

[9] For a plain function these are just the declared specifications. In case of a trait method implementation, we pick the specification according to the refinement strategy, see Section 2.3.2

```
fn enumerated(&self, prev: &Self) -> bool {
  // ...
  self.iter_states.len() == self.visited().len() + 1 &&
  self.cl_states.len() == self.visited().len() + 1 &&
  self.visited().len() == self.iter.visited().len() &&

  // Describe a trace T_k of configurations for
  // the k so far observed values: ⟨I_0, r_0⟩ ~next~> ... ~next~> ⟨I_k, r_k⟩
  forall(|i: usize| i < self.visited().len() ==> (
    Iterator::next ~~> |iter: I| -> Option<I::Item>
      { iter === self.iter_states[i] }
      { iter === self.iter_states[i+1] &&
        result.is_some() &&
        // A call to the closure happened with the argument r_i,
        // leading to the i-th visited value of this iterator.
        F::call_mut ~~> |cl: F, arg: I::Item| -> B
          { cl === self.cl_states[i]
            Some(arg) == outer(result) }
          { cl === self.cl_states[i+1] &&
            self.visited()[i] == result }
      }
  )) &&

  // Iteration is completed after k calls iff the k+1-th
  // call led to a returned None:
  // ... ~next~> ⟨I_{k+1}, ∅⟩
  self.completed == (
    Iterator::next ~~> |iter: I| -> Option<I::Item>
      { iter === self.iter_states[self.iter_states.len() - 1] }
      { result.is_none() }
  )
}

fn completed(&self, prev: &Self) -> bool {
  self.completed
}
```

Listing 20: A specification of `Map` using generalized call descriptions.

```
let mut iter1 = Iter1::new();
let mut iter2 = Iter2::new();
let mut map1 = iter1.map(|x| x);
let mut map2 = iter2.map(|x| x);

map1.next();
map2.next();
```

Listing 21: We nest two iterators `Iter1` and `Iter2` inside a `Map` iterator. The former nested iterator does not implement `IteratorSpec` while the latter does. Since `Map` uses call descriptions to describe the effects of its nested iterator, we get different knowledge for both `fn next` calls.

```
let v = vec![1,2,3,4];
let mut iter = v.iter();
let mut map = iter.map(|x| 2*x);
let v: Vec<i32> = map.collect();
assert!(v == vec![2,4,6,8])
```

Listing 22: Conversion from a vector into an `Iter`, then a `Map` and collecting the resulting values in a new vector.

For the former, we certainly know that a call to `iter1.next()` has happened. We also know that the returned element is positive, since the call description used in `Map` encodes the postcondition of `Iter1::next`.

For the second call, we know more though: Since `Iter2` implements the specification extension `IteratorSpec`, we know that the type-dependent contract on the specification of `Iterator::next` is active. Thus, when we encode the call description of `Map`, we encode the active type-dependent contract on `Iterator::next`, which will ultimately refer to the implementation of `IteratorSpec` on `Iter2`. On the callsite, we thus additionally know everything that is part of the specification `Iter2::enumerated`. Consequently, we get information about the `visited` sequence of `Iter2`.

3.4 From collections to iterators and back

In this section, we discuss how we can turn a `Vec<T>` into an iterator and turn an iterator into `Vec<T>`, as shown in Listing 22.

On Line 2, we create a new `Iter` iterator from a vector. We can specify this `fn iter` function and initialize the resulting `Iter` accordingly. Likewise, we could create a slice iterator via `&v.into_iter()`, to immutably traverse the elements of the vector. We can also consume the vector by directly

creating an `IntoIter` with `v.into_iter()`. For both of these cases we need a specification for the `IntoIterator` trait for `Vec<T>` and `&Vec<T>`, respectively.

On Line 3, we create the iterator adapter `Map`. These iterator adapters are usually created via provided methods on `Iterator` directly, and we must initialize the adapter accordingly. This means that we must provide a specification which establishes the type invariant *enumerated* of `Map`:

```
1 trait Iterator {
2     // ...
3
4     // Specs, which must imply Map::enumerated
5     fn map<B, F>(self, f: F) -> Map<Self, F>
6         where F: FnMut(Self::Item) -> B { ... }
7     // ...
8 }
```

Lastly, we collect the values of the resulting iterator chain on Line 4 which we discuss next.

3.4.1 Collection

The method `Iterator::collect` is polymorphic and implemented via an indirection: It takes anything that turns into an iterator (`IntoIterator`) and returns a type which is known to be constructible from this iterable (`FromIterator`):

```
1 trait Iterator {
2     // ...
3     fn collect<B: FromIterator<Self::Item>>(self) -> B {
4         FromIterator::from_iter(self)
5     }
6     // ...
7 }
```

To derive a full specification for Listing 22 and verify the assertion, we need a specification for `Iterator::collect` and the implementation of `FromIterator` on `Vec<T>`.

We start with the latter. A simplified implementation would iterate over the provided iterator and store every returned element in the resulting vector. We want to capture this behavior in our specification:

```
1 impl<T> FromIterator<T> for Vec<T> {
2     fn from_iter<I: IntoIterator<Item = T>>(iter: I) -> Vec<T> {
3         let mut result = Vec::new();
```

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

Listing 23: A specification for collecting the values of an iterator into a vector.

```
        for el in iter { result.push(el) }
        result
    }
}
```

Again, upon calling `fn collect`, we may assume the existence of a trace T of observed iterator states and results as `fn collect` traverses the iterator:

$$\langle I_0, r_0 \rangle \overset{next}{\rightsquigarrow} \langle I_1, r_1 \rangle \overset{next}{\rightsquigarrow} \dots \overset{next}{\rightsquigarrow} \langle I_k, r_k \rangle$$

We now want to combine this trace with the elements of the vector. We thus propose the specification in Listing 23, which again as in Section 3.3.5 uses call descriptions to link the configurations with the resulting vector.

Note again, that our knowledge of the resulting `Vec<T>` is dependent on the strength of the specification of `I`. If `I` implements `IteratorSpec`, then we have precise knowledge about the values of `Vec<T>`, as the call description takes the activated type-dependent contract into account.

Finally, we need a specification for `Iterator::collect` itself. This can be achieved with a call description again, where we simply link the result of `Iterator::collect` with the result of `FromIterator::from_iter`:

```
trait Iterator {
    // ...
    #[ensures(
        FromIterator::from_iter ~~> |iter: Self| -> B
            { iter == self }
            { result == outer(result) }
    )]
    fn collect<B: FromIterator<Self::Item>>(self) -> B { ... }
    // ...
}
```

Chapter 4

Implementation

This chapter explains needed changes that we introduced to support our methodology. We start by outlining implementation-specific aspects of specification extensions and abstract predicates in Section 4.1. What follows is a discussion how we specify standard library code in Section 4.2. In Section 4.3 we discuss, as an extension to Section 4.2, how we deal with code that is implemented in a way which is impossible to specify. As a last point, in Section 4.4, we discuss implementation-specific details of type-dependent contracts.

4.1 Specification extensions

The support for specifications, as we introduced them in Section 3.2, posed several challenges during the implementation phase of the project. Our iterator specification is *polymorphic*: The specification does not depend on concrete assertions, but mostly refers to trait methods, whose implementation provides the assertions.

In order to support such polymorphic specifications, we refactored and extended how Prusti internally represents user-typed specifications and interacts with them, for example when encoding a method call.

Prusti encodes each to-be-checked Rust method, hereby called *procedure*, into a separate Viper file. This encoding includes all transitive dependencies of the procedure. For example, if the procedure uses some struct (as part of the body or the specification), this struct will be part of the encoded Viper file as well. The logic in Prusti which orchestrates this process is called `ProcedureEncoder`. It internally is structured in different modules. For example, one module is the `TypeEncoder` which is responsible for encoding types.

To better handle our specifications, we created a new `specification` module.

This module acts as a facade for specifications. Each Rust function, internally identified with a unique `DefId`[1], has a specification which consists of the following information[2]:

Preconditions A vector `Vec<DefId>`, where each element identifies a type-checked precondition function.

Postconditions A vector `Vec<DefId>`, where each element identifies a type-checked postcondition function.

Kind Defines whether the method is a predicate, pure or impure (represented as an `enum`). A method is a predicate if it is inside a `predicate!` block. It is pure when annotated with the `#[pure]` attribute, and impure if no annotation is present.

Trustedness A boolean flag indicating whether the procedure is trusted or not.

When the `ProcedureEncoder` needs specifications, for example to encode the pre- and postconditions of a method call, it will invoke a query to the `specification` module. Such a query mainly[3] consists of the `DefId` of an item for which the encoder wants the specification. The `specification` module then assembles the specification for the query. This assembly process most notably returns *refinement information*: If the queried `DefId` is a trait method implementation, the assembled specification will consist of the specification of the trait method *and* the implementation method. The procedure encoder then can use this refinement information to select the *effective* specification for the call (and encode a check that the subtyping is valid).

Additionally, when the `specification` module builds the assembled specification of a trait method and its implementation, it checks whether the *kind* of the implementation specification is valid with respect to the *kind* of the trait specification. This enforces that an abstract predicate[4] defined on a trait is indeed implemented as a predicate in the implementation. Likewise, a pure trait method must not be impurely implemented. If the `specification` module did not enforce this, the modular verification of a generic function depending on some trait method would assume that this trait method has no side effects. Calling the generic method with an implementation of the method that has side effects then could lead to soundness issues. Table 4.1 shows the enforced rules.

[1] A `DefId` is a type of the compiler API which uniquely identifies an *item*, for example a function or an implementation block, throughout the compilation process.

[2] This list is not exhaustive. For example, we also store information for error reporting.

[3] A query also includes a *cause*. For example, the cause for a query can be `MethodCallEncoding`. This cause is mainly relevant for caching and performance.

[4] We talk more about abstract predicates in Section 4.1.1.

		Kind of implementation method		
		Impure	Pure	Predicate
Kind of trait method	Impure	✓	✓	✗
	Pure	✗	✓	✗
	Predicate	✗	✗	✓

Table 4.1: Table of valid (✓) and invalid (✗) procedure refinements of a trait method and its implementation. For example, it is possible to mark an impure trait method as pure in the implementation. The converse is not possible: If the trait method is pure, then the implementation must be pure as well.

As a summary, the `specification` module provides a uniform access to (user-typed) specifications and especially enriches specifications with refinement information to better handle polymorphic specifications. The `ProcedureEncoder` uses this module as part of its own encoding process to encode polymorphic specifications.

4.1.1 Abstract predicates

Abstract predicates are trait methods which implementors must implement as a predicate. They are introduced with a surrounding `predicate! { ... }` macro:

```
trait Trait {
    predicate! {
        fn pred(&self, args...) -> ReturnType;
    }
}
```

It is not possible to use an abstract predicate in non-specification code, because the implementor of the abstract predicate can use a superset of Rust syntax as part of the body of the method. Consequently, Prusti will throw an error when verifying the following code:

```
#[ensures(t.pred(...))] // Valid usage
fn foo<T: Trait>(t: T) {
    t.pred(); // Cannot use abstract predicate here
}
```

Refinement

We disallow that abstract predicates have a method body. Abstract predicates thus are always *required* trait methods, as opposed to *provided* trait methods. We impose this restriction because we can not modularly determine how a predicate is used: It may be used as part of a pre- or postcondition, or a combination of the two. If Prusti *knew* that an abstract predicate is only used in preconditions, we could check that the body of the implementation weakens the body of the trait method; a valid refinement. We could do a similar strengthening check if the predicate appeared only as part of postconditions. However, it would not be clear what refinements we should allow if it were used as part of pre- *and* postconditions. We thus disallow predicates with bodies on traits altogether.

As was already mention, we require that the implementor implements an abstract predicate as a predicate, i.e., the following implementation of the aforementioned abstract predicate is disallowed:

```
impl Trait for Struct {
    // This is an impure method;
    // an invalid refinement of Trait::pred
    fn pred(&self, args...) -> ReturnType { /* body */ }
}
```

The method `Struct::pred` is impure and thus might have side effects. A generic function `foo` like listed before would accept a pre- and postcondition which uses the abstract predicate. However, calling this function with `Struct` is unsound, because specifications must be side-effect free.

Likewise, we disallow implementing an abstract predicate as a pure function. Not ensuring this would not immediately cause soundness issues, but it would be rather inconsistent. An abstract predicate defines a function which is only meant to be used in specifications. An implementor should respect this fact with its implementation.

4.2 Specifying standard library code

This section discusses how specifications for non-local Rust types can be provided. Non-local types are types which come from another crate, e.g. the standard library or from `crates.io`.

This is needed for iterators: We want to specify various iterators from the standard library (`Map`, `Filter`) and the `Iterator` trait itself. Copying the code from the standard library and specify this copy is not an option. Eventually, the standard library iterator specifications could be distributed as a part of Prusti, and users could simply rely on these specifications when verifying

```
struct Foo;
impl Foo {
    fn produce(&self, x: i32) -> i32 {
        x + 41
    }
}

#[extern_spec]
impl Foo {
    #[ensures(result == x + 41)]
    fn produce(&self, x: i32) -> i32;
}
```

Listing 24: External specification for local type.

their iterator-based code (see Section 6.1). Copying the standard library makes this impossible.

It is also a necessity in other cases: Rust code is *rarely* self-contained by having no external dependencies, even if it only uses the standard library. The need for *external specifications* thus not just arises for iterators, but in almost every Rust program that is to be verified by Prusti.

4.2.1 Prior work and external specifications

External specifications are not a novelty in this work. The ideas and foundations were developed by Karen Hong during an internship.

The core idea is that Prusti specification annotations do not need to be part of the implementation. As a simple example, consider Listing 24 which contains a user-written type `Foo` with one method that simply returns a constant integer. Normally, we would add a postcondition directly to `Foo::produce`. It is however possible, to use the `#[extern_spec]` procedural macro from Prusti to declare a specification for `Foo::produce` in a different source code region. The macro `#[extern_spec]` introduces a new external specification. It is immediately followed by an `impl` block containing the to-be-specified method `produce` in a bodyless form with the specification itself. This external specification can be read as "attach this postcondition to the method `produce` which is declared inside some `impl Foo` block".

In this example, there is no difference from declaring the specification directly on the method or as an external specification: The body of `Foo::produce` will be checked against the postcondition and likewise, a client which calls the method will refer to this specification.

Listing 24 seems a bit artificial, because `Foo` is locally defined, and an external specification is not needed. If `Foo` instead were really external type,

```
use foo_crate::Foo;

#[extern_spec]
impl Foo {
    #[ensures(result == x + 41)]
    fn produce(&self, x: i32) -> i32;
}

fn main() {
    let f = Foo;
    let res = f.produce(1);
    assert!(res == 42)
}
```

Listing 25: External specification for external type.

as in Listing 25, the external specification remains the same. The only important difference is that the specification is not checked against body of `Foo::produce`, as it is not locally available. Nevertheless, Prusti uses the defined external specification in `fn main` and can prove the assertion.

Prior to this thesis it was possible to provide external specifications for inherent[5] implementation blocks and for modules. While technically possible to attach an external specification to a trait implementation, it failed in certain scenarios. To properly support iterators, we need support for external specifications in these scenarios as well.

4.2.2 Definitions, requirements, and usage scenarios

The block below the `#[extern_spec]` including the macro in Listing 24 is called an *external specification* which *attaches* to the method `Foo::produce`. This method is said to be *externally specified*. The `impl Foo { ... }` block is called the *context* of the external specification.

Requirements

To further develop the already existing technique for external specifications, we propose the following general requirements for external specifications:

[5]There are two kinds of implementation blocks in Rust: *Inherent implementations* and *trait implementations*. Inherent implementation blocks are of the form `impl Foo { }`, whereas trait implementations are of the form `impl SomeTrait for Foo { }`. More information can be found in the Rust reference: https://doc.rust-lang.org/reference/items/implementations.html

Well-definedness An external specification is well-defined if it attaches to an explicitly typed[6] method (either local or non-local). It should not be possible to declare external specifications for non-existing methods.

Coherency An external specification should clearly express whether it belongs to a trait method, an implementation of that method or a method inside an inherent implementation block.

Uniqueness It should not be allowed that multiple external specifications can be attached to the same method, as it then would not be clear which specification should be active for a method call.

An external specification is said to be *valid* if all the above conditions are met. If an external specification is not valid, this should be reported appropriately to the user.

Usage scenarios and non-usage scenarios

To properly support iterators, we at least need to be able to attach external specifications to the following items:

- Trait methods to specify the `Iterator` trait.
- Methods which appear as part of trait implementations to specify iterator implementations.
- Methods which appear as part of inherent implementations to specify other non-local types (e.g. `Option<T>`, the return type of any iterator).

We do not need to attach external specifications to free functions or Rust modules, the latter of which is already supported in Prusti. Additionally, versioning[7] might be an interesting addition in the future.

4.2.3 Syntax

As discussed in Section 4.2.1, external specifications are introduced with the procedural macro `#[extern_spec]` followed by the context of the external specification. Rust procedural macros can only parse and interpret Rust syntax[8], so the context ideally is Rust syntax itself.

In Listing 26, we outline external specifications in all scenarios that were mentioned earlier. We observe, that the context is the same code block that

[6] That is, it attaches to a method that is syntactically present in the source code. Consider a generic method `Foo<T>::foo`. After monomorphization, there exist an arbitrary amount of copies of this method, namely one for each monomorphized version. We do not consider these as *explicitly typed*.

[7] External crates can be imported with a specific version into a Rust project when using Rust's package manager Cargo. The need for version-dependent external specifications might be needed in these scenarios.

[8] More precisely, everything that is supported by a `proc_macro::TokenStream`

```
#[extern_spec]
trait Trait {
    // Specs S_Trait
    fn foo(&self, x: i32) -> i32;
}
```

```
#[extern_spec]
impl Trait for Struct {
    // Specs S_TraitImpl
    fn foo(&self, x: i32) -> i32;
}
```

```
#[extern_spec]
impl Struct {
    // Specs S_Impl
    fn foo(&self, x: i32) -> i32;
}
```

Listing 26: Syntax for declaring external specifications.

the externally specified method was defined in. That is, from the context, we can uniquely identify the externally specified method:

- Specs S_{Trait} are attached to `Trait::foo`.
- Specs $S_{\text{TraitImpl}}$ are attached to `<Struct as Trait>::foo`.
- Specs S_{Impl} are attached to `Struct::foo`.

4.2.4 External specifications and generics

As we have seen Section 2.1.1, traits, their implementations and even types can be generic, and consequently, external specifications have to work in a generic setting as well.

Type parameters in inherent implementations

First, we look at generic types. Assume a generic `struct Foo<T>` with two implementation blocks `impl<T> Foo<T> {}` and `impl for Foo<i32> {}` as shown in Listing 27. It must be possible to externally specify both `Foo<T>::foo` and `Foo<i32>::bar`. In the same listing, we indeed provide two external specifications for them which contain the same generics as the implementation blocks themselves. This again allows us to identify the correct to-be-specified methods.

Naturally, the question arises whether we should be allowed to provide an external specification for `Foo<T>::foo` inside an `impl Foo<i32> {}` block as shown in Listing 28. This specification might seem to be well-defined, as it is attached to an existing method `Foo<i32>::foo`. However, this is not true, because `Foo<i32>::foo` is actually a monomorphized version of `Foo<T>::foo`. We disallow this, because the specification should describe the behavior of the method for all monomorphized versions of it, not just a specific one.

```
impl<T> Foo<T> {
    fn foo(&self) {}
}

impl Foo<i32> {
    fn bar(&self) {}
}
```

```
#[extern_spec]
impl<T> Foo<T> {
    // Specs S^T_foo
    fn foo(&self);
}

#[extern_spec]
impl Foo<i32> {
    // Specs S^i32_bar
    fn bar(&self);
}
```

Listing 27: Syntax for externally specifying generic types.

```
#[extern_spec]
impl Foo<i32> {
    // Specs S^i32_foo
    fn foo(&self);
}
```

Listing 28: External specification which does not match the generic signature.

One could argue that the specification S^{i32}_{foo} for `Foo<i32>::foo` could be more specific, because the type of `T` is known. It is questionable though why this would be even needed. Even though S^{i32}_{foo} could syntactically account for the known type, we would still require that it is semantically equivalent in the presence of S^{T}_{foo}. If the latter was not present, this could be a possibility, but it would violate the idea why the generic was introduced in the first place: Defining a method, parametric over types which exposes some semantical behavior that we would like to specify.

Type parameters appearing in trait implementations

We impose the same restrictions for generics appearing in trait implementations as for inherent implementations: The context of the external specification must match the generic signature of the to-be-specified method.

Notice that it is not necessary, and thus disallowed, to specify the associated type in such an external specification. Providing the same associated type in the external specification as in the *actual* implementation is redundant. Providing a different associated type is wrong, as the external specification would not be well-defined anymore.

```
1 impl<T: A, U: B> Trait<T> for Struct<U> { ... }
2
3 #[extern_spec] impl<T: A, U: B> Trait<T> for Struct<U>
4 #[extern_spec] impl<X: B, Y: A> Trait<Y> for Struct<X>
5 #[extern_spec] impl<T, U> Trait<T> for Struct<U>
6 #[extern_spec] impl<T: A+C, U: B+D> Trait<T> for Struct<U>
7 #[extern_spec] impl<T: A, U: B> Trait<U> for Struct<T>
```

Listing 29: External specification with bounds on the generic parameter.

Polymorphic traits

Once again, we uphold the same requirements as in external trait/inherent implementations.

As for trait implementations, associated types need not be provided in the external specification, as they provide no useful relevant information.

Type parameter bounds

What is left for generics in the context of external specifications are type parameter bounds.

Consider Listing 29 which has a generic trait implementation. Importantly, the generic type parameters have trait bounds. In the following lines, we have provided various scenarios how a user might attach external specification to this trait implementation.

The following behavior is expected and matched against our rules for external specification:

- Line 3 is valid, because it precisely and uniquely identifies the correct method.
- Line 4 is valid, because the name and the order of the type parameters do not matter.
- Line 5 is invalid, because `Trait<T>` is implemented for `Struct<U>` only if `T: A` and `U: B`.
- Line 6 is valid, even though the bounds are tighter.
- Line 7 is invalid for the same reason as Line 5 (the generics are swapped and thus the trait bounds not satisfied).
- Any two-combination of valid lines is invalid, because the external specification would then not be unique (e.g. Line 3 and Line 4).

4.2.5 Ensuring well-definedness

How can we ensure that an external specification is well-defined, i.e., that it really attaches to an existing method? Technically, the compiler can help us here. When Prusti verifies a file or crate, all macros including `#[extern_spec]` are processed first, then the resulting code is compiled.

To ensure that an external specification is well-defined, we thus can insert a *fake* call to the method that we think the external specification should be attached to.

Consider again Listing 26 with different external specifications. For the inherent implementation block, we generate code as follows:

```
struct Wrapper_UUID;
impl Wrapper_UUID {

    // Specs S_Impl
    fn bar(_self: &Struct, x: i32) -> i32 {
        <Struct>::foo(_self, x: i32);
        unimplemented!()
    }
}
```

The function inside the wrapper invokes a "call" to the externally specified method. If this method does not exist, the compiler throws an error which Prusti can intercept and report appropriately to the user.

The same happens in the desugaring of the trait implementation: The only difference is that the method call explicitly refers to the trait method via `<Struct as Trait>::foo`.

External trait specifications

Generating the wrapper for traits is a bit more involved. Notice that we add a `_self` parameter to the wrapper function which represents the receiver of the externally specified method. This is not possible for traits, as the *implementing* type is not yet determined. To overcome this problem, we attach a special type parameter to the wrapper:

```
struct<Prusti_Self> Wrapper_UUID(
    std::marker::PhantomData<Prusti_Self>
);
impl<Prusti_Self> Wrapper_UUID<Prusti_Self>
    where Prusti_Self: Trait {

    // Specs S_Trait
    fn foo(_self: &Prusti_Self, x: i32) -> i32 {
```

```
9           <Prusti_Self as Trait>::foo(_self, x);
10          unimplemented!()
11      }
12  }
```

This approach allows us to check well-definedness even in the absence of a concrete type. Furthermore, with this, we make sure that the specifications S_{Trait} can be type-checked.

Generics

Generics appearing as part of a type or a trait block are also added to the wrapper struct to ensure that the compiler can successfully type-check the external specification.

4.2.6 Ensuring coherency

The aforementioned solution to check for well-definedness has a shadowing issue. If `Struct` had no inherent implementation block with a `foo` method, the external specification for `Struct::foo` would still be accepted by the compiler. The compiler would assume that the fake call `<Struct>::foo(_self, x)` refers to the method of the trait implementation.

This can be problematic in different ways. In the best case, this is just confusing for the user and may result in non-unique external specifications. There is a more important problem though: The inherent implementation could be deleted from the external library. This would then mean that the previously correctly attached external specification gets attached to the wrong method (the trait implementation), leading to unexpected and probably undesired verification results.

As a consequence, Prusti requires user-annotations to be very explicit and ensures that the user-annotations are coherent with what Prusti understood.

To address this, we attach additional information to the wrapper method about the *source* of the external specification. Concretely, we add an annotation `#[prusti::extern_spec = "X"]` where X is either `"inherent_impl"`, `"trait_impl"` or `"trait"`. We then analyze whether the fake method call really refers to a method of this kind and report an error otherwise.

Additionally, we require that the generics match with the target method. For example, the `Vec::len` method is defined inside the following block:

```
1  impl<T, A: Allocator> Vec<T, A> { ... }
```

```
#[extern_spec]
impl Struct {
    // Specs S_foo
    fn foo(&self);
}
```

```
#[extern_spec]
impl Struct {
    // Specs S*_foo
    fn foo(&self);
}
```

Listing 30: Duplicate external specifications.

If the user tried to externally specify this method without the generics, Prusti will report an error.

4.2.7 Ensuring uniqueness

As already mentioned, we propose that external specifications must be unique. It is thus not allowed to externally specify the same method twice as in Listing 30 which would only cause no trouble if $S_{\text{foo}} \equiv S^*_{\text{foo}}$. Since it generally does not seem useful to have two syntactical different specifications which are semantically equivalent, we simply disallow this.

After the desugaring of all the external specifications, we thus check whether there are any duplicated externally specified methods and report an error if this is the case.

4.3 Type models

We now turn our attention towards the actual specifications for external code. Consider the `Iter` iterator for which we want to specify its `Iterator` implementation. A specification of that implementation certainly depends on the internal state of `Iter`. Inspecting the struct definition and its fields reveals hard-to-specify raw pointers. Even worse, the fields are not public and can not be accessed by an external specification[9].

To this end, we propose to use *type models* to introduce an abstraction between the specification and the implementation (Fig. 4.1). This technique is not a novelty, and for example also known in the Java Modeling Language [3, 2]. An abstraction between the specification and the to-be-specified type comes with several advantages:

- Non-public internals are not a problem anymore, since the specification depends on the model, defined locally.
- Complicated internals can be simplified.
- Changes of the implementation do not affect the specification.

[9] Remember that specifications are type-checked, which certainly will fail if we access a non-`pub` field from an external crate in a local crate.

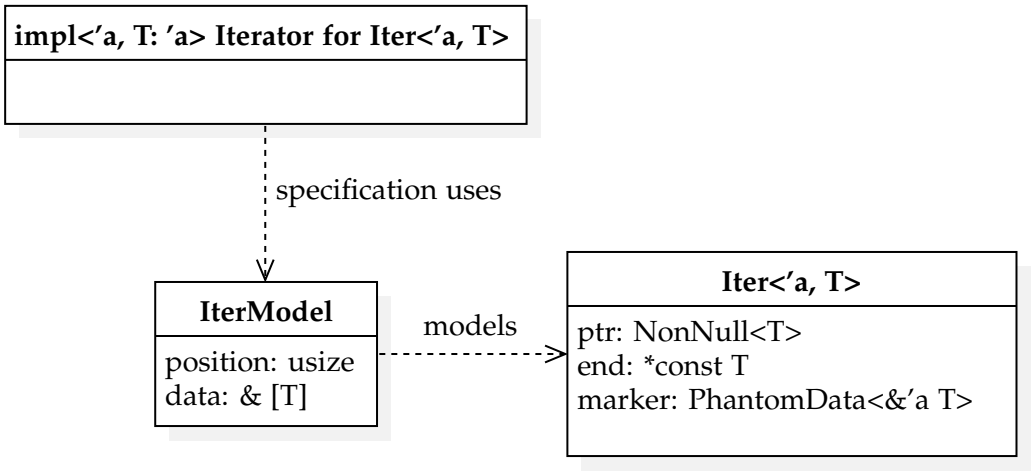


Figure 4.1: Sketch for a type model of `Iter` which acts as an abstraction between its implementation specification and the type.

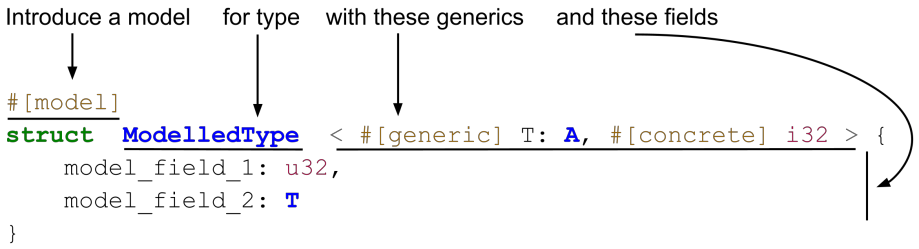
Motivational example

We highlight type models in Listing 31, exemplified on the `Iter` iterator. The type model is introduced on Line 3. For the `Iter` iterator, we need the position to the element which will be returned by the immediately following call to its `fn next` method, as well as the data of the slice.

The model can then be used in an external specification. On Line 9, we initialize the model with suitable defaults and in Line 16 we provide an overly-simplified specification how `Iter`'s model state evolves for a call to its `next` method.

4.3.1 Syntax

The syntax for a type model is as follows:



The type model is introduced with the procedural macro `#[model]`, followed by the model itself in a `struct` code block. This block acts a container for the model fields but also serves as a path to the type we want to model. As of now, it is not possible to use an arbitrary Rust path, e.g., one can not write

```rust
use std::slice::Iter;

#[model]
struct Iter<'a, T> where T: 'a {
    pos: usize,
    data: &'a [T],
}

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

#[extern_spec]
impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T

    #[ensures(!old(self.completed()) ==>
        self.model().pos == old(self.model().pos) + 1)]
    #[ensures(old(self.completed()) ==>
        self.model().pos == old(self.model().pos) + 1)]
    fn next(&mut self) -> Option<Self::Item>
}
```

Listing 31: A type model for `Iter`, including external specifications which initialize and update the model.

`std::vec::Vec` for a model definition. Instead, the type must be imported first with `use std::vec::Vec`.

Type models support modelling generic types. That is, we can either create a type model for all generic type instantiations of a type or a concrete monomorphized version (or a combination of the two). We must indicate this with the `#[generic]` and `#[concrete]` annotations per type parameter. In the example above, we create a type model for `ModelledType` whose first type parameter implements the trait `A` and whose second type parameter is an `i32`.

4.3.2 Implementation

The `#[model]` macro first parses a type model:

```
#[model]
struct Foo<#[generic] T: Copy, #[concrete] i32> {
    field_1: u32,
    field_2: T,
}
```

And then generates the following code:

```
trait FooToModel_UUID<T: Copy> {
    #[pure]
    #[trusted]
    fn model(&self) -> FooModel_UUID<T>;
}

struct FooModel_UUID<T: Copy> {
    field_1: u32,
    field_2: T,
}

// ... Copy implementation ...
// ... Clone implementation ...

impl<T: Copy> FooToModel_UUID<T> for Foo<T, i32> {
    #[pure]
    #[trusted]
    fn model(&self) -> FooModel_UUID<T> {
        unimplemented!()
    }
}
```

We see that the `#[model]` macro is just syntactic sugar for creating a struct (the model) and implementing a method which returns that struct on the to-be-modelled type. This model function is part of a trait, as the to-be-modelled type may be an external type (which is usually the case for iterators). Notice, that this trait needs to be generated as part of the macro desugaring: It is not possible that there exists a general `ToModel` trait (e.g., as part of Prusti), because in Rust it is not possible to implement non-local traits on non-local types[10].

From this desugaring mechanism, we can observe the following limitations for type models:

[10] This is called the orphan rule in Rust. We could restrict that to-be-modelled types need to be local and one could apply Rust's *newtype pattern* to still model external types. However, we concluded that this is too cumbersome, as we generally want to model external types when specifying iterators.

- A type model needs to be `Copy`, as it is used as part of a pure method. Likewise, if a type model uses generics, those are also restricted to be `Copy` types (thus the bound in the example above).
- Only one type model can be created for a type (for any generic instantiation). While we technically can create multiple type models, we are not able to use them, as we would have multiple `fn model` functions and the Rust compiler does not know which one the user meant to use.
- A type model must not and can not be used outside of specification code, since it panics during runtime. This is enforced by Prusti, similar to the check that predicates are only used in specifications. The source of any *knowledge* about a type model is thus always a trusted function.

4.3.3 Encoding

The encoding of a type model is no different from ordinary Prusti encoding. There is one important subtlety though: On the Viper level, the model struct is encoded using Prusti's snapshot-based encoding (see Section 2.3.6) and as a consequence, the model is encoded as a Viper domain with suitable functions and axioms to ensure there exists exactly one snapshot for every possible state of the model.

Likewise, the `fn model` method is encoded as a bodyless Viper function, which maps from a snapshot of the modelled type to the snapshot of the model.

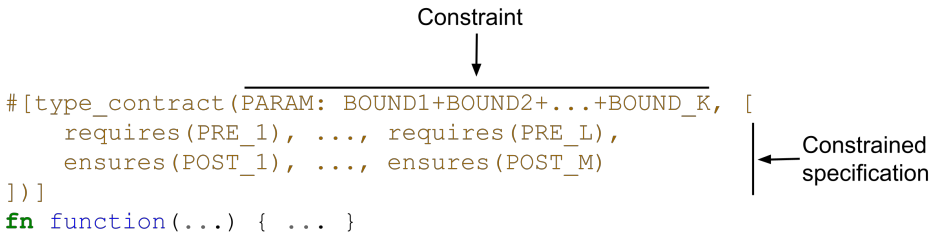
A consequence of the snapshot encoding is that both the model and the modelled type *need* to have at least one field. If this were not the case, Prusti could not set up a bijective relation between a snapshot of the model and a snapshot of the modelled type, which can lead to unsound verifications as outlined in Appendix A.5.

4.4 Type-dependent contracts

In this section, we discuss implementation-related details of type-dependent contracts which were introduced in Section 3.2.1.

4.4.1 Syntax

As already illustrated in Section 3.2.1, a type-dependent contract is introduced with the `type_contract` procedural macro:



This macro has two parameters: A *constraint* and a *constrained specification*. The constraint denotes under which condition the constrained specification should hold. It is syntactically a `where` clause which appear in Rust when we add bounds to generics (see Section 2.1.1). The constrained specification consists of an unbounded number of pre- and postcondition assertions, appearing inside `requires(.)` and `ensures(.)` expressions.

4.4.2 Limitations

Before we look at implementation details, we emphasize that the current implementation is a prototype, and the usage of type-dependent contracts is limited:

- It is disallowed to have multiple type-dependent contracts on the same function.
- Type-dependent contracts must not appear as a part of trait method refinements.
- Type-dependent contracts can only appear on trusted functions.

These restrictions are permissive enough to allow us to use type-dependent contracts in our iterator setting.

The reason for disallowing multiple type-dependent contracts on a function is that their constraints could overlap: For a callsite, multiple type-dependent contracts could be active[11], and we would need to implement a strategy to pick one of the contracts.

The reason for disallowing type-dependent contracts on non-trusted functions is that this would raise many questions for modular verification. Assume we had a non-constrained specification *and* a constrained specification on a function. It is not immediately obvious which of the two specifications should be used for verifying the function. Furthermore, we would need to define how the non-constrained specification and the constrained specification are related. Generally, we require that the constrained specification is a strengthening of the non-constrained specification, but we do not yet enforce this.

[11] A type-dependent contract is active for a callsite when its constraint is satisfied on the callsite.

As a consequence, type-dependent contracts are highly experimental and users need to explicitly enable them with a feature flag.

4.4.3 Desugaring and type-checking

The `type_contract` macro desugars the type-dependent contract into Rust code which contains enough information such that it can be type checked and that Prusti can enable it on callsites.

Consider the following function with a type-dependent contract:

```
1 #[type_contract(T: A, [requires(x >= 5), ensures(result >= 7)])]
2 fn foo<T: B>(x: i32) -> i32 { ... }
```

The desugarring of the macro leads to the following code:

```
1  #[prusti::spec_only]
2  #[prusti::spec_id = "UUID_PRE"]
3  #[prusti::type_dep_contract_trait_bounds_in_where_clause]
4  fn prusti_pre_item_foo_UUID_PRE<T: B>(x: i32) -> bool
5      where T: A {
6      !!((x >= 5): bool)
7  }
8  #[prusti::spec_only]
9  #[prusti::spec_id = "UUID_POST"]
10 #[prusti::type_dep_contract_trait_bounds_in_where_clause]
11 fn prusti_post_item_foo_UUID_POST<T: B>(x: i32, result: i32) -> bool
12     where T: A {
13     !!((result >= 7): bool)
14 }
15 #[prusti::pre_spec_id_ref = "UUID_PRE"]
16 #[prusti::post_spec_id_ref = "UUID_POST"]
17 fn foo<T: B>(x: i32) -> i32 { ... }
```

This is mostly the same as ordinary specification desugaring. There are two important distinctions: First, the constraint is attached as a `where` clause to the specification function on Line 5 and Line 12. Second, there is a marker attribute on Line 3 and Line 10 which indicates that the `where` clause is to be interpreted as a constraint of a type-dependent contract.

This desugaring enables automatic type-checking of type-dependent contracts. Since the constraint is part of the desugared Rust program, using a method of `trait A` as part of a specification successfully type-checks.

4.4.4 Collection

After having desugared all macros, Prusti collects user-annotated specifications per function. When collecting the specification for such a function, Prusti partitions it into a base specification and a constrained specification. This partitioning is based on the marker attribute attached to the specification function.

Since we currently can not annotate a method as pure or trusted as part of a type-dependent contract, we simply inherit these properties from the base contract. Additionally, all postconditions are automatically copied to the constrained specification. Copying the postconditions is merely a convenience feature to require less annotation overhead[12]. These details might need to be reworked as this current prototype implementation matures.

After collection, we have a mapping from Rust functions to a base specification and a constrained specification which contains at least the same information as the base specification (modulo preconditions). The constraint is not explicitly part of the constrained specification. Instead, it is inherent to the specification functions themselves, as part of their function signature.

Example

Consider the following specifications for a function `foo`:

```
#[requires(x >= 10)]
#[ensures(result >= 0)]
#[type_contract(T: A, [
    requires(x >= 5),
    ensures(result >= 7)
])]
fn foo<T>(x: i32) -> i32 { ... }
```

As we have seen in Section 4.4.3, we get two specification functions for each pre- and postcondition of the type-dependent contract, say `fn foo_pre_c` and `fn foo_post_c`[13]. We furthermore get two specification functions for the base contract, say `fn foo_pre_b` and `fn foo_post_b`.

The function-to-specification mapping will then contain the following information, as shown in Table 4.2.

[12] While still maintaining behavioral subtyping rules, because then the postcondition only gets stronger. We do not do this for preconditions, as this would certainly lead to invalid weakenings.

[13] In the previous listing, this was `prusti_pre_item_foo_UUID_PRE_C` and `prusti_post_item_foo_UUID_POST_C`.

Function	Base specification		Constrained specification	
`foo`	Kind	Impure	Kind	Impure (*)
	Trusted	`false`	Trusted	`false` (*)
	Preconditions	`foo_pre_b`	Preconditions	`foo_pre_c`
	Postconditions	`foo_post_b`	Postconditions	`foo_post_b` (*)
				`foo_post_c`
...	...		...	
			(*) Inherited from base specification	

Table 4.2: Mapping of functions to specifications with type-dependent contracts.

4.4.5 Resolving

Resolving to the correct specification when encoding a method call is implemented as part of the `specification` module (see Section 4.1).

The query to get a specification for encoding a method call contains information that is available on the callsite, for example the bounds that appear on a generic parameter. With this information, the `specification` module can check whether the constraint is fulfilled for this specific callsite. If this is the case, then the constrained specification is returned, otherwise the base specification.

Chapter 5

Evaluation

In this chapter, we qualitatively outline what we have achieved throughout this work by showing the specification and verification of a custom-written iterator and the standard library iterators `Iter` and `Map`.

For all following iterator evaluations, we externally specified `Option<T>` as shown in Appendix B.1.

5.1 Custom-written iterator

The custom-written iterator `Counter` that we verify counts from 0 up to a user-provided bound. The implementation is listed in Appendix B.3, the specification in Appendix B.4.

Verification of implementation

Due to a bug in the predicate encoding of Prusti we were unable to verify the implementation of `Counter` against the specification of `Iterator::next`. We thus verified the implementation against an equivalent specification which does not contain predicates.

The implementation itself was adjusted slightly such that the verification succeeded: As part of `Counter::next`, we update the ghost sequence `visited` as we observe a new value.

Verification of clients

We verified three clients using `Counter` which are all shown in Appendix B.5 appearing in the same order as they are described in this section.

The looping clients for `Counter` were all written in `loop { ... }` syntax[1].

For the first client we create a new `Counter` with a bound of 3 and then invoke its `fn next` method four times. We `unwrap` the first three results and check the contained value. For the last returned value we check that it is a `None`. Prusti is able to prove that `fn unwrap` does not panic. Prusti is also able to prove the converse, namely that calling the `fn unwrap` method on a `Counter` with a bound of 1 panics on the second element returned from `fn next`.

The next client uses a loop to iteratively compute the sum of all values of a `Counter` with a concrete bound. We use a helper function `sum_rec` which calculates the sum of a `GhostSeq` recursively. This helper function is then used as part of the invariants of the loop. After the loop, we can recover the value of the iteratively computed sum with the help of this function.

In the last client, we iteratively compute the maximum of a `Counter` with a concrete bound. We store the index of the maximum value and the maximum value itself as we observe new values from the `Counter` and can thus recover the value with our provided loop invariants after the loop.

It is important to note that the looping clients are not using internals of `Counter`. All used invariants either refer to `Counter::enumerated` or to the `visited` sequence of the iterator.

5.2 Specification of `Iter`

The specification for the `Iter` iterator, including the verified client is shown in Appendix B.6.

We additionally included a specification for the creation of `Iter` from a slice type `&[T]`. Our non-looping client takes a slice with a known size and known values, creates an `Iter` from that slice and then uses the iterator to iterate over the slice.

A major challenge for this iterator was the fact that `Iter` returns references. References as part of data structures are known to be not fully supported in Prusti[2]. A concrete problem we encountered during our evaluation was that matching an `Option<&i32>` currently causes a Prusti crash. We circumvented this issue with a combination of `OptionPeeker` (described in Appendix B.2) and a custom snapshot type `Snap`.

[1] The compiler converts `for ... in ... { ... }` or `while ... { ... }` expressions automatically to `loop { ... }` expressions. Even though `for` loops are most commonly seen as part of iterator traversal, we can use all three equivalently to accomplish this task.

[2] For example, Prusti currently does not support the reborrowing of a field of a struct which is passed as a borrow to a function.

5.3 Specification of `Map`

As a last iterator, we specified `Map`. The specification is shown in Appendix B.7. The specification does not use the generalized version of the iterator adapter specification (see Section 3.3.6). Instead, we use the encoding described in Section 3.3.3, and we thus require that nested iterators implement `IteratorSpec` as well.

Call descriptions for closures are not yet fully integrated into Prusti. As a consequence, we model closures as custom structs. The imitation of closures with these *fake* closure structs is a reasonable workaround: Internally, the Rust compiler represents a closure as a type "approximately equivalent to a struct which contains the captured variables"[6]. We then model state changes of the captured closure state via specification functions, which are attached to the struct. In the specification of `Map`, we then use these specification functions in a way equivalent to call description encoding. Details about this workaround are explained in Appendix B.7.2.

As for the clients, we used `Map` in non-looping clients with non-capturing and capturing closures. We used an `Iter` iterator as the nested iterator. In the following, we list the verified clients[3].

Map with non-capturing state closure This client uses a `Map` which simply doubles every element. The closure thus has no captured state:

```
let vec = vec![3, 6, 8];
let mut iter = vec.iter().map(|x| 2*x);
assert!(iter.next().unwrap() == 6);
assert!(iter.next().unwrap() == 12);
assert!(iter.next().unwrap() == 16);
assert!(iter.next().is_none());
```

Map with captured state closure For the second and third client, we tested whether our specification can handle captured state as part of the closure of a `Map` iterator.

The following client adds the value of a captured value to every element prior to increasing it:

```
let vec = vec![3, 6, 8];
let mut count = 5;

let mut iter = vec.iter().map(|x| {
```

[3]The actual clients we verified are syntactically different due to the mentioned workarounds.

```
    let res = x + count;
    count += 1;
    res
});

assert!(iter.next().unwrap() == 3 + 5);
assert!(iter.next().unwrap() == 6 + 6);
assert!(iter.next().unwrap() == 8 + 7);
assert!(iter.next().is_none());
```

The following client shows that we can imitate an accumulator as part of the `Map` iterator:

```
let vec = vec![3, 6, 8];
let mut acc = 0;

let mut iter = vec.iter().map(|x| {
    acc += x;
    acc
});

assert!(iter.next().unwrap() == 3);
assert!(iter.next().unwrap() == 9);
assert!(iter.next().unwrap() == 17);
assert!(iter.next().is_none());
```

Captured and non-captured state In our fourth client, we tested whether we can nest a `Map` inside a `Map` iterator:

```
let vec = vec![3, 6, 8];
let mut count = 5;

let mut iter = vec.iter()
    .map(|x| {
        let res = x + count;
        count += 1;
        res
    })
    .map(|x| 2*x);

assert!(iter.next().unwrap() == 2*(3 + 5));
assert!(iter.next().unwrap() == 2*(6 + 6));
assert!(iter.next().unwrap() == 2*(8 + 7));
assert!(iter.next().is_none());
```

Offsetting The fifth client tests whether our specification supports offsetting as described in Section 3.3.4:

```
let vec = vec![3, 6, 8];

let mut iter = vec.iter();
assert!(*iter.next().unwrap() == 3);

let mut iter = iter.map(|x| 2 * x);
assert!(iter.next().unwrap() == 2*6);
assert!(iter.next().unwrap() == 2*8);
assert!(iter.next().is_none());
```

Different output type It is not uncommon that Map yields elements which are not of the same type as the input elements. We thus verified a client which converts integers to booleans:

```
let vec = vec![1, 2, 3];

let mut iter = vec.iter().map(|x| *x >= 2);

assert!(iter.next().unwrap() == false);
assert!(iter.next().unwrap() == true);
assert!(iter.next().unwrap() == true);
assert!(iter.next().is_none());
```

Chapter 6

Conclusion

In this thesis we have made a first step towards supporting iterator verification in Prusti by adapting a general framework from Filliâtre and Pereira [5].

Our specification extension trait `IteratorSpec` can be seen as a framework for iterator specification in Rust. Type-dependent contracts allow for a modular plug-in architecture of these specifications.

Using call descriptions in combination with type-dependent contracts allows for assembling the combined specification of an iterator adapter and its nested iterator at callsites while still maintaining modularity of the specifications.

With external specifications we can specify iterators from the standard library without needing to copy their implementation to the to-be-verified crate. Additionally, type models introduce an abstraction between specifications and implementations. This is needed as part of external specifications when the external code is impossible or hard to specify.

We demonstrated and evaluated our approach on three iterators. The first iterator is a custom iterator implementation. We verified the implementation and the usage of this iterator in looping and non-looping clients. Next, we specified the two iterators `Iter` and `Map` from the standard library with external specifications and verified several non-looping clients with these specifications.

6.1 Future Work

In this section we talk about points which were not implemented or discussed as part of this work. They are nevertheless important for iterator verification. These points can be built on top of this thesis as part of future work.

Specifications for more standard library iterators

We have demonstrated how to apply our methodology on the standard library iterators `Iter` and `Map`. This gave us confidence that our approach works in general. Nevertheless, these are arguably *simpler* iterators in terms of the expected specification complexity.

We thus propose to specify more involved iterators such as `Filter`, `Take`, `Repeat`, or `Zip`.

Verify implementations of standard library iterators

As an addition to the last point we suggest to check whether these specifications of the standard library iterators can be used to verify their implementation. In our work we only have verified the implementation of a custom-written iterator.

The verification of the implementation would contribute in the following two points: First, we would know that the specifications indeed describe the behavior of the iterators. Second, the specifications become more reliable. Currently, a wrong specification causes soundness problems on the client-side which can only be circumvented with excessive testing. Knowing that the implementation verifies against the specification decreases this risk.

Library of iterator specifications

Ideally Prusti users do not need to specify standard library iterators themselves. These specifications can become quite large and, by the nature of external specifications, are easily to get wrong because they are not verified against their implementation.

Therefore, we suggest shipping iterator specifications as part of a *specification library* to Prusti users. As a consequence, users do not need to write specifications for the iterators themselves but can verify their code which uses iterators.

Better support for loops

We experimented mainly with plain `loop` loops and to some extent with `while` loops in this thesis. Using `for el in iter` syntax is currently not supported in Prusti. Even though the former loop types are 'equivalent' to `for` loops, it would be cumbersome if Prusti was not able to support the latter because these are the ones commonly appearing in the context of iterators.

Mutable iteration

We have only considered immutable traversal of iterators in our work. The support for mutable traversal would be a good addition, because it is a common use case in Rust programs.

Stabilize type-dependent contracts

Our implementation of type-dependent contracts is minimal and experimental (see Section 4.4.2) and thus gated behind a feature flag. As a first step, we propose to evaluate in which other scenarios type-dependent contracts are applicable, and then generalize and stabilize the current implementation.

One specific challenge that we think is a suitable use case for type-dependent contracts is shown in Appendix A.4.

Experiment with non-fused iterators

Our approach relies on the fact that iterators are fused. The adaption to non-fused iterators poses several challenges:

- How should the `visited` ghost sequence of a non-fused iterator look like?
- How can we formulate the termination criterion in the generalized specification which uses call descriptions in an iterator adapter?

All of these challenges, and probably many more, induce an adaption of our general approach for iterator verification.

Appendix A

Supplementary material

A.1 An implementation of the `Iterator` trait

In the following, we illustrate the implementation of `Iterator` on a custom type `BoundedCounter`. This iterator starts by counting from 0 to the given bound. The implementation of the trait on Line 12 is straightforward; whenever the current `count` is strictly smaller than the provided `bound`, return the count and increase it by one. Otherwise, return `None` for all successive calls. Also note that since `BoundedCounter` implements `Iterator` and thus `IntoIterator`, we can use it in a loop as shown on Line 27.

```
1  struct BoundedCounter {
2      count: u32,
3      bound: u32,
4  }
5
6  impl BoundedCounter {
7      fn with_bound(bound: u32) -> Self {
8          // initialize with provided bound and count = 0
9      }
10 }
11
12 impl Iterator for BoundedCounter {
13     type Item = u32;
14
15     fn next(&mut self) -> Option<Self::Item> {
16         if self.count < self.bound {
17             let count = self.count;
18             self.count += 1;
19             return Some(count);
20         }
21         None
22     }
```

```
}

fn use_counter() {
    let mut c = BoundedCounter::with_bound(3);
    for el in c {
        print!("{el}, ");
    }
}
```

A.2 Implementation of a `Map` iterator

In the following, we illustrate how a `Map` iterator adapter can be implemented. The implementation is similar to the implementation in the standard library.

```
struct Map<I, F> {
    iter: I,
    cl: F,
}

impl<B, I, F> Iterator for Map<I, F>
where
    I: Iterator,
    F: FnMut(&I::Item) -> B,
{
    type Item = B;

    fn next(&mut self) -> Option<B> {
        let n = self.iter.next();
        match self.iter.next() {
            Some(next) => self.cl(next),
            None => None,
        }
    }
}
```

A.3 GhostSeq<T>

The following listing shows our implementation of `GhostSeq<T>`. Notice that we have added useful helper predicates to `GhostSeq<T>` which can decrease the specification overhead for iterators significantly. However, there is currently a bug in Prusti which causes that deeply nested predicates will

not be encoded. However, we still show our specifications with the use of these predicates.

```
pub struct GhostSeq<T> {
    phantom: std::marker::PhantomData<T>,
}

// Implementation of `Clone` and `Copy` omitted.

impl<T: Copy + PartialEq> GhostSeq<T> {
    #[pure]
    #[trusted]
    #[ensures(result >= 0)]
    pub fn len(&self) -> usize {
        unimplemented!()
    }

    #[trusted]
    #[ensures(self.len() == 0)]
    pub fn initialize(&self) { unimplemented!() }

    #[pure]
    #[trusted]
    #[requires( 0 <= i && i < self.len() )]
    #[ensures(self.contains(result))]
    pub fn lookup(&self, i: usize) -> T {
        unimplemented!()
    }

    #[trusted]
    #[ensures(self.len() == old_seq.len() + 1)]
    #[ensures(forall(|i: usize|
        (0 <= i && i < prepush_self.len()) ==> (
            self.lookup(i) == old_seq.lookup(i)
    )))]
    #[ensures(self.lookup(self.len() - 1) == val)]
    pub fn push(&self, old_seq: &Self, val: T) {
        unimplemented!()
    }
}

// Helper predicates for specifications
impl<T: Copy + PartialEq> GhostSeq<T> {
    predicate! {
        pub fn equals(&self, other: &GhostSeq<T>) -> bool {
            self.len() == other.len() &&
            forall(|i: usize|
                (0 <= i && i < self.len()) ==>
                    (self.lookup(i) == other.lookup(i)))
```

```
47          }
48      }
49
50      predicate! {
51          pub fn is_prefix_of(&self, other: &GhostSeq<T>) -> bool {
52              self.len() <= other.len() &&
53              forall(|i: usize| (0 <= i && i < self.len()) ==>
54                  (self.lookup(i) == other.lookup(i)))
55          }
56      }
57
58      // Specifies that the ghost sequence increases by
59      // exactly one element if `increases` is true.
60      // Otherwise, ensures that `self` is equal to `old_self`.
61      predicate! {
62          pub fn seq_increases_if(&self,
63                                  increases: bool,
64                                  old_seq: &GhostSeq<T>) -> bool {
65              (increases ==> (
66                  self.len() == old_seq.len() + 1
67                  && old_seq.is_prefix_of(self)
68              )) &&
69              (!increases ==> (
70                  old_seq.equals(self)
71              ))
72          }
73      }
74  }
```

A.3.1 Pushing new values to `GhostSeq<T>` in iterators

Notice that the `fn push` method has a special parameter `old_seq`. Currently, we only need `fn push` to verify an iterator implementation (see Appendix B.3 for an example). The `visited` sequence of an iterator is accessed via the `fn visited` function, defined on `IteratorSpecVisited`. This function relates one iterator state with one `visited` sequence. When we mutate the state of the iterator as part of `fn next`, the `visited` sequence after the state update loses the information of the `visited` sequence prior to the state update. We thus need to frame the old `visited` sequence (prior to the state update) of the iterator around the state update. We can then use this additional knowledge as part of `fn post` to assemble the new `visited` sequence.

A.4 Other use cases for type-dependent contracts

There are other interesting applications for type-dependent contracts which are not directly tied to iterator verification.

One use case are marker traits which either act as additional information for the compiler (think of the `Copy` trait) or act as an *untyped contract* for the implementor.

One such example is the `ExactSizeIterator`[1] trait. Its documentation states that any type implementing this trait must uphold the guarantee that `fn Iterator::size_hint`[2] does not panic and returns a tuple `(a,a)`, i.e., that the size of the iterator is exactly known. There is no way of checking this statically and implementors need to make sure that this guarantee is uphold.

Let us now look at the method at the `Extend`[3] trait which is implemented for vectors and the following code snippet:

```
let mut v1 = vec![1,2,3,4];
let v2 = vec![3,4,5];
v1.extend(v2.into_iter());
assert!(v1.len() == 7)
```

How should a specification for `fn extend` look like such that the assertion verifies? This method can take any iterator, even ones which have an unknown size (think of a `Filter` iterator). The best guarantee that a specification thus can provide is that the size of the receiving vector is non-decreasing. This guarantee could be strengthened with a type-dependent contract: We could conditionally strengthen the specification *if* the passed iterator implements `ExactSizeIterator`:

```
impl<T, A> Extend<T> for Vec<T, A> {
    #[ensures(self.len() >= old(self.len()))]
    #[type_contract(I::IntoIter: ExactSizeIterator, [
        ensures(self.len() == old(self.len()) + iter.len())
    ])]
    fn extend<I: IntoIterator<Item = T>>(&mut self, iter: I);
}
```

[1] `ExactSizeIterator`: https://doc.rust-lang.org/stable/std/iter/trait.ExactSizeIterator.html

[2] `fn Iterator::size_hint`: https://doc.rust-lang.org/stable/std/iter/trait.Iterator.html#method.size_hint

[3] `Extend`: https://doc.rust-lang.org/stable/std/iter/trait.Extend.html

Notice that this does not work yet: The type parameter `I` is not directly an `Iterator`, but a type which can be turned into an iterator. Hence we would need a type-dependent contract for the associated type `I::IntoIter`.

A.5 Unsoundness when dealing with type models

We impose the restriction that type models and the modelled type has fields. In the following listing, we create a model for a type where the type has no fields, which leads to unsound verification.

```
1  struct Foo; // This type has no fields!
2
3  #[model]
4  struct Foo {
5      val: i32,
6  }
7
8  #[trusted]
9  #[ensures(result.model().val = val)]
10 fn create_foo(val: usize) -> Foo {
11     Foo {}
12 }
13
14 fn main() {
15     let foo1 = create_foo(0);
16     let foo2 = create_foo(1);
17
18     // foo1.model() and foo2.model() refer to the same
19     // snapshot of the Foo model, say S.
20     // As a consequence, the inhaling of the postcondition
21     // assumes S_val = 0 ∧ S_val = 1; a contradiction.
22     assert!(false); // Verifies
23 }
```

Appendix B

Evaluation code

B.1 `Option<T>` specifications

In our evaluation, we used the following external specifications for `Option<T>`. Notice that reasoning about `Option<&T>` currently requires certain workarounds which are described in Appendix B.2.

```
#[extern_spec]
impl<T: PartialEq> std::option::Option<T> {
    #[pure]
    #[ensures(matches ! (* self, Some(_)) == result)]
    pub fn is_some(&self) -> bool;

    #[pure]
    #[ensures(self.is_some() == ! result)]
    #[ensures(matches ! (* self, None) == result)]
    pub fn is_none(&self) -> bool;
}

#[extern_spec]
impl<T: PartialEq + Copy> std::option::Option<T> {
    #[requires(self.is_some())]
    #[ensures( *(old(self.peek().get())) == result )]
    fn unwrap(self) -> T;
}
```

B.2 Interacting with `Option<&T>`

When a function returns an `Option`, we usually directly compare it to one of its variants in a specification:

```
#[ensures(result == Some(...))]
fn foo() -> Option<i32> { /* body */ }
```

This however, is currently not possible in Prusti if the method returns an `Option` which contains a reference. In this thesis, this usually happens when we interact with the `Iter` iterator. We worked around this limitation by introducing the `OptionPeeker` trait, defined as follows:

```
pub trait OptionPeeker<T> {
    #[pure]
    fn peek(&self) -> Snap<T>;
}

#[refine_trait_spec]
impl<T> OptionPeeker<T> for Option<T> {
    #[pure]
    #[trusted]
    #[requires(self.is_some())]
    fn peek(&self) -> Snap<T> {
        unimplemented!()
    }
}
```

This `OptionPeeker` allows to specify the contained value of an `Option` with the `fn peek` method. It returns a special type `Snap<T>` which pretends to hold a reference to type `T`. It is implemented as follows:

```
pub struct Snap<T>(std::marker::PhantomData<T>);

impl<T> Snap<T> {
    #[pure]
    #[trusted]
    pub fn get(&self) -> &T {
        unimplemented!()
    }
}
```

With the `OptionPeeker`, we can specify a method which returns an `Option<&T>`:

```
#[ensures(result.is_some())]
#[ensures(*result.peek().get() == ...)]
fn foo() -> Option<i32> { /* body */ }
```

On callsites, when dealing with an `Option<&T>`, we always copy the `Option` to verify the value of the option. A natural choice would be to use the method `Option::copied` which consumes an `Option<&T>` and returns an `Option<T>`. This, however, is also not supported in Prusti. We thus use the following function as a workaround when we want to copy an `Option<&T>`:

```
// This method is a wrapper for `Option::copied`.
// It exists because Prusti can not encode a method which
// takes an `Option<&T>` (in contrast to `&Option<&T>` here).
#[pure]
#[trusted]
#[ensures(o.is_some() == result.is_some())]
#[ensures(o.is_some() ==>
    (**o.peek().get() == *result.peek().get()))]
pub fn deref_copy_option<T>(o: &Option<&T>) -> Option<T>
    where T: Copy + PartialEq {
    o.copied()
}
```

B.3 Verified custom-written iterator `Counter`

We use the following implementation for our custom-written iterator. Note that the altering of the `visited` sequence on Line 32 requires framing the `visited` sequence prior to the state update of `Counter`. The reason for this is described in Appendix A.3.1.

```
pub struct Counter {
    count: usize,
    up_to: usize,
}

impl Counter {
    #[ensures(result.count == 0)]
    #[ensures(result.up_to == up_to)]
    #[ensures(result.visited().len() == 0)]
    #[ensures(result.enumerated(&result))]
    fn new(up_to: usize) -> Self {
        let c = Counter {
            count: 0,
            up_to
        };
        c.visited().initialize();
        c
    }
}

#[refine_trait_spec]
```

```rust
impl Iterator for Counter {
    type Item = usize;

    // Specifications inherited from Iterator::next
    fn next(&mut self) -> Option<Self::Item> {
        if self.count < self.up_to {
            let res = self.count;

            let pre_vis = self.visited();
            self.count = self.count + 1;
            self.visited().push(&pre_vis, res);

            return Some(res);
        }
        None
    }
}
```

B.4 Specification of `Counter`

```rust
impl IteratorSpec for Counter {
    type IterItem = usize;

    predicate! { fn enumerated(&self, prev: &Self) -> bool {
        // monotonicity
        self.visited().len() >= prev.visited().len() &&
        self.count >= prev.count &&
        self.up_to == prev.up_to &&

        // visited
        self.count == self.visited().len() &&
        forall(|i: usize|
            (0 <= i && i < self.visited().len()) ==> (
            self.visited()[i] == i
        ))
    } }

    predicate! { fn completed(&self) -> bool {
        self.count == self.up_to
    } }

    predicate! { fn post(old_self: &Self,
                         new_self: &Self,
                         res: &Option<Self::IterItem>) -> bool {
        (old_self.completed() ==> (
```

```
            new_self.count == old_self.count )) &&
        (!old_self.completed() ==> (
            new_self.count == old_self.count + 1 )) &&

        (res.is_some() ==> (*res.peek().get() == old_self.count))
    } }
}
```

B.5 Verified clients of `Counter`

For our looping clients, we use the following code snippet to create a snapshot of the state of `Counter` before the loop:

```
#[trusted]
#[ensures(*c === result)]
fn snap_counter(c: &Counter) -> Counter { unimplemented!() }
```

B.5.1 Direct traversal

```
fn test_direct_traversal() {
    let mut c = Counter::new(3);

    assert!(c.next().unwrap() == 0);
    assert!(c.next().unwrap() == 1);
    assert!(c.next().unwrap() == 2);
    assert!(c.next().is_none());

    // Check visited values
    let vis = c.visited();
    assert!(vis.len() == 3);
    assert!(vis.lookup(0) == 0 &&
            vis.lookup(1) == 1 &&
            vis.lookup(2) == 2);
}

fn test_direct_traversal_fail() {
    let mut c = Counter::new(1);

    assert!(c.next().unwrap() == 0);
    assert!(c.next().unwrap() == 1); // Verification fails
}
```

B.5.2 Iterative sum computation

```
#[pure]
#[requires(0 <= from && from <= to && to <= g.len())]
fn sum_rec(from: usize, to: usize, g: &GhostSeq<usize>) -> usize {
    if from == to {
        0
    } else {
        g.lookup(from) + sum_rec(from + 1, to, g)
    }
}

fn test_loop_sum() {
    let mut c = Counter::new(3);
    let snapped = snap_counter(&c);

    let mut sum = 0;
    let mut i = 0;

    loop {
        body_invariant!(c.enumerated(&snapped));
        body_invariant!(i == c.visited().len());
        body_invariant!(sum == sum_rec(0, i, &c.visited()));
        let el = c.next();

        match el {
            Some(val) => {
                i += 1;
                sum += val;
            },
            None => break,
        }
    }

    assert!(i == 3);
    assert!(c.completed());
    assert!(c.visited().len() == 3);
    assert!(c.visited().lookup(0) == 0);
    assert!(c.visited().lookup(1) == 1);
    assert!(c.visited().lookup(2) == 2);

    assert!(sum == sum_rec(0, 3, &c.visited()));
    assert!(sum == 0 + sum_rec(1, 3, &c.visited()));
    assert!(sum == 0 + 1 + sum_rec(2, 3, &c.visited()));
    assert!(sum == 0 + 1 + 2 + sum_rec(3, 3, &c.visited()));
    assert!(sum == 0 + 1 + 2 + 0);
    assert!(sum == 3);
}
```

B.5.3 Iterative maximum computation

```
fn test_loop_max() {
    let mut c = Counter::new(3);
    let snapped = snap_counter(&c);

    let mut max = c.next().unwrap();
    let mut max_idx = 0;
    let mut i = 1;

    loop {
        body_invariant!(c.enumerated(&snapped));
        body_invariant!(i == c.visited().len());
        body_invariant!(0 <= max_idx && max_idx < c.visited().len());
        body_invariant!(max == c.visited().lookup(max_idx));
        body_invariant!(forall(|i: usize|
            (0 <= i && i < c.visited().len()) ==>
            (c.visited().lookup(max_idx) >= c.visited().lookup(i))
        ));
        let el = c.next();

        match el {
            Some(val) => {
                i += 1;
                if val > max {
                    max = val;
                    max_idx = i;
                }
            },
            None => break,
        }
    }

    assert!(i == 3);
    assert!(c.completed());
    assert!(c.visited().len() == 3);
    assert!(c.visited().lookup(0) == 0);
    assert!(c.visited().lookup(1) == 1);
    assert!(c.visited().lookup(2) == 2);

    assert!(max_idx == 2);
    assert!(max == 2);
}
```

B.6 Specification and verification of `Iter`

The type model of `Iter` is defined as follows:

```
#[model]
struct Iter<'a, #[generic] T: Copy + PartialEq> {
    position: usize,
    data: GhostSeq<T>,
}
```

B.6.1 Slice to `Iter` specification

```
type SliceTy<T> = [T];
#[extern_spec]
impl<T: Copy + PartialEq> SliceTy<T> {
    #[pure]
    fn len(&self) -> usize;

    // Initialize the model
    #[requires(self.len() >= 0)]
    #[ensures( result.model().position == 0 )]
    #[ensures( result.model().data.len() == self.len() )]
    #[ensures(
        forall(|i: usize| 0 <= i && i < self.len() ==> (
            self[i] == result.model().data.lookup(i)
        ))
    )]
    // Initialize ghost sequence of visited values
    #[ensures(result.visited().len() == 0)]
    fn iter(&self) -> std::slice::Iter<'_, T>;
}
```

B.6.2 Specification extension and model

We use the following type model definition and `IteratorSpec` implementation for `Iter`.

```
impl<'a, T: Copy + PartialEq + 'a> IteratorSpec
    for std::slice::Iter<'a, T> {
    type IterItem = &'a T;

    predicate! { fn completed(&self) -> bool {
        self.model().position == self.model().data.len()
```

```
    } }

    predicate! { fn enumerated(&self) -> bool {
        self.visited().len() <= self.model().data.len() &&
        self.visited().len() == self.model().position &&
        forall(|i: usize| (0 <= i && i < self.visited().len()) ==>
            self.model().data.lookup(i) == *self.visited().lookup(i)
        )
    }}

    predicate! { fn post(old_self: &Self,
                         new_self: &Self,
                         res: &Option<Self::IterItem>) -> bool {
        // Data does not change
        new_self.model().data.equals(&old_self.model().data) &&

        // Visited ghost sequence is up to date
        new_self.visited().seq_increases_if(
            !old_self.completed(),
            old_self.visited()) &&

        // Evolution of position
        (!old_self.completed() ==
            (new_self.model().position ==
                old_self.model().position + 1)) &&
        (old_self.completed() ==
            (new_self.model().position ==
                old_self.model().position)) &&

            // Result
        (res.is_some() && !old_self.completed()) ==> (
            *res.peek().get() ==
                new_self.visited().lookup(
                    new_self.visited().len() - 1)
        )
    } }
}
```

B.6.3 Verified non-looping client

```
#[requires(slice.len() == 2)]
#[requires(slice[0] == 42)]
#[requires(slice[1] == 777)]
fn test_slice_iter(slice: &[i32]) {
    let mut iter = slice.iter();
```

```
6
7       let el = iter.next();
8       assert!(el.is_some());
9       assert!(deref_copy_option(&el).unwrap() == 42);
10
11      let el = iter.next();
12      assert!(el.is_some());
13      assert!(deref_copy_option(&el).unwrap() == 777);
14
15      let el = iter.next();
16      assert!(el.is_none());
17  }
```

B.7 Specification and verification of `Map`

The type model of `Map` is defined as follows:

```
1   #[model]
2   struct Map<#[generic] I: Iterator, #[generic] C: Copy + PartialEq> {
3       // The current nested iterator
4       iter: I,
5       // The current state of the closure
6       cl: C,
7       // All observed closure states during iteration
8       cl_states: GhostSeq<C>,
9       // Offset flag
10      offset: usize,
11  }
```

B.7.1 Creation of `Map` from an iterator

A `Map` iterator is created with the method `Iterator::map`. We thus need a specification for this method to create a `Map` iterator from another iterator:

```
1   #[extern_spec]
2   trait Iterator {
3       // We require `self` to be enumerated
4       #[requires(self.enumerated(&self))]
5
6       // Init closure state
7       #[ensures(result.visited().len() == 0)]
8       #[ensures(result.model().cl_states.len() == 1)]
9       #[ensures(result.model().cl_states.lookup(0) ===
```

```
                                        result.model().cl)]
    #[ensures(old(f) === result.model().cl)]

    // Init the nested iterator:
    // We establish equality, but set the offset if
    // the nested iter already contains visited elements
    #[ensures(result.model().iter === old(&self))]
    #[ensures(result.model().iter
                    .enumerated(result.model().iter))]
    #[ensures(result.model().offset ==
                    old(self).visited().len())]

    // `enumerated` should hold now for the resulting Map iterator.
    fn map<B, F>(self, f: F) -> Map<Self, F>
    where
        F: FnMut(Self::Item) -> B,
        Self::Item: Copy + PartialEq,
        B: Copy + PartialEq,
        Self: IteratorSpec<IterItem = Self::Item>,
        F: ClosureSpecExt<Self::Item, Output = B>;
}
```

B.7.2 A specification extension for describing closures

Recall from Section 5.3 that our specification of `Map` does not use real closure call description syntax, because this feature is not yet merged into Prusti. To circumvent this, we use yet another specification extension trait `ClosureSpecExt` which has two required method `fn pre` and `fn post`:

```
pub trait ClosureSpecExt<Arg>: FnMut<(Arg,)> {
    #[pure]
    fn pre(&self, args: Arg) -> bool;

    #[pure]
    fn post(old_self: &Self, new_self: &Self,
            arg: Arg, res: Self::Output) -> bool;
}
```

This specification extensions models the behavior of a unary closure. For example, consider the closure:

```
let mut count = 0;
let cl = |x: i32| -> i32 {
    let res = x + count;
    count += 1;
```

```
    res
};
```

We can create a fake closure type `AddIncreasingCounter` which models the mutably captured state of `cl` with a field:

```
struct AddincreasingCounter {
    count: i32
}
```

We then implement `ClosureSpecExt` on this type:

```
impl ClosureSpecExt<&i32> for AddIncreasingCounter {
    #[pure]
    fn pre(&self, arg: &i32) -> bool {
        true
    }

    #[pure]
    fn post(old_self: &Self,
            new_self: &Self,
            arg: &i32,
            res: Self::Output) -> bool {
        new_self.count == old_self.count + 1 &&
        *res == *arg + old_self.count
    }
}

impl<'a> FnMut<(&'a i32,)> for AddIncreasingCounter {
    // Implement the closure
}
```

As part of our implementation of `IteratorSpec` of `Map`, we can then add a bound on the type parameter which represents the type of the closure of `Map`. Concretely, we implement `IteratorSpec` conditionally, only if the closure type parameter `C` implements `ClosureSpecExt` (see Appendix B.7.3). With that additional knowledge, we can encode the behavior of a *real* closure as part of our closure call description encoding.

B.7.3 Specification extension

We use the following `IteratorSpec` implementation for `Map`:

```
#[refine_trait_spec]
impl<
        B: Copy + PartialEq,
        I: Iterator + IteratorSpec<IterItem = I::Item>,
        C: ClosureSpecExt<I::Item, Output = B> + Copy,
    > IteratorSpec for Map<I, C>
where
    I::Item: Copy + PartialEq,
{
    type IterItem = B;

    predicate! {
        fn completed(&self) -> bool {
            self.model().iter.get().completed()
        }
    }

    predicate! { fn enumerated(&self, prev: &Self) -> bool {
        // The nested iterator is enumerated
        self.model().iter.enumerated(prev.model().iter) &&

        // Link the visited elements of the adapter
        // to the nested iter
        self.visited().len() + self.model().offset ==
            self.model().iter.visited().len() &&

        // cl_states length is always one larger than
        // the amount of visited elements
        self.model().cl_states.len() == self.visited().len() + 1 &&

        // Link visited elements of Map with visited
        // elements of the nested iterator
        forall(|i: usize| (0 <= i && i < self.visited().len()) ==> (
            // Encoding of the call description of the closure
            exists(|r: Snap<Self::IterItem>,
                    a1: Snap<I::Item>,
                    cl_state_prev: C,
                    cl_state_next: C| {

                let cl_state_prev = self.model().cl_states
                                                .lookup(i);
                let cl_state_next = self.model().cl_states
                                                .lookup(i + 1);

                (*a1.get() ==
                    self.model().iter.visited().lookup(
                        i + self.model().offset)) &&
                (*r.get() == self.visited().lookup(i)) &&

```

```
50                (ClosureSpecExt::pre(&cl_state_prev, *a1.get())) &&
51                (ClosureSpecExt::post(&cl_state_prev,
52                                      &cl_state_next,
53                                      *a1.get(),
54                                      *r.get()))
55            })
56        ))
57    } }
58
59    predicate! { fn post(old_self: &Self,
60                         new_self: &Self,
61                         res: &Option<Self::IterItem>) -> bool {
62        // The offset never changes
63        new_self.model().offset == old_self.model().offset &&
64
65        // Visited sequence is up to date
66        new_self.visited().seq_increases_if(
67            !old_self.completed(),
68            &old_self.visited() ) &&
69
70        // cl_states is up to date
71        new_self.model().cl_states.seq_increases_if(
72            !old_self.completed(),
73            &old_self.model().cl_states ) &&
74
75        // Link `cl` to `cl_states`
76        (!old_self.completed() ==>
77            (new_self.model().cl ===
78                new_self.model().cl_states.lookup(
79                    new_self.model().cl_states.len() - 1) )) &&
80
81        // Describe the evolution of the nested iterator and the
82        // closure with `IteratorSpec` and `ClosureSpecExt`.
83        exists(|nested_res: Option<I::Item>|
84            res.is_some() == nested_res.is_some() &&
85
86            IteratorSpec::post(
87                old_self.model().iter,
88                new_self.model().iter,
89                &nested_res ) &&
90
91            ((!old_self.completed() && res.is_some()) ==>
92                ClosureSpecExt::post(
93                    &old_self.model().cl,
94                    &new_self.model().cl,
95                    *nested_res.peek().get(),
96                    *res.peek().get()
97            ))
98        )
```

```
99      } }
100 }
```

Bibliography

[1] 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.

[2] Cees-Bart Breunesse and Erik Poll. Verifying jml specifications with model fields. In *In Formal Techniques for Java-like Programs. Proceedings of the ECOOP 2003 Workshop*, 2003.

[3] Y Cheon, GT Leavens, M Sitaraman, S Edwards, and Model Variables. Cleanly supporting abstraction in design by contract", department of computer science, iowa state university. Technical report, TR# 03-10a, March 2003, revised September, 2003.

[4] Matthias Erdin, Vytautas Astrauskas, and Federico Poli. Verification of rust generics, typestates, and traits. Master's thesis, Eidgenössische Technische Hochsschule - ETH, 2019.

[5] Jean-Christophe Filliâtre and Mário Pereira. A modular way to reason about iteration. In *NASA Formal Methods Symposium*, pages 322–336. Springer, 2016.

[6] The Rust Foundation. Closure types. https://doc.rust-lang.org/stable/reference/types/closure.html#closure-types. Accessed: 2022-06-14.

[7] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[8] Barbara H Liskov and Jeannette M Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[9] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[10] Prusti. Heap-based type encoding. https://viperproject.github.io/prusti-dev/dev-guide/encoding/types-heap.html. Accessed: 2022-06-18.

[11] Prusti. Loop body invariants. https://viperproject.github.io/prusti-dev/user-guide/verify/loop.html. Accessed: 2022-06-18.

[12] Prusti. Snapshot-based type encoding. https://viperproject.github.io/prusti-dev/dev-guide/encoding/types-snap.html. Accessed: 2022-06-18.

[13] Christopher Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13(1):11–49, 2000.

[14] 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)*, volume 5, New York, NY, USA, oct 2021. ACM.

[15] Fabian Wolff. Verification of closures in rust programs. Master's thesis, Eidgenössische Technische Hochsschule - ETH, 2020.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Specification and Verification of Iterators in a Rust Verifier

Authored by (in block letters):

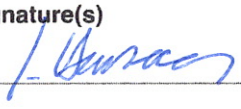
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Hansen	Jonas

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Olten, 18.06.2022	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.