ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Specifying and Verifying Sequences and Array Algorithms in a Rust Verifier

Master Thesis

Johannes Schilling

July 15th, 2021

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

Prof. Dr. Wolfgang Schröder-Preikschat, Simon Schuster, Phillip Raffeck
Department of Computer Science, FAU Erlangen-Nürnberg

Abstract

Arrays and slices are the fundamental sequence types in the Rust programming language. The ability to specify and verify properties about them has the potential to benefit a wide range of programs.

Prusti is a Rust verifier. It supports automatic derivation of a core safety and non-interference proof from information available to the compiler, thanks to the strong guarantees that Rust's memory and thread safety mechanisms provide. User-provided functional specifications can be added on top of the core proof and verified with low effort.

Prusti lacks support for arrays and slices so far. In this thesis, we design and implement support for specifying and verifying array programs in Prusti. This includes not only support for encoding types and program constructs related to arrays and slices, but also predicates as specification constructs supporting Prusti's full specification syntax including quantification and implication to simplify reasoning about sequences.

Our approach integrates well with Prusti's existing encoding technique. We demonstrate its usefulness by showing the verification of a sort algorithm for arrays.

Kurzfassung

In der Programmiersprache Rust sind Arrays und Slices die elementaren Listendatentypen. Die Möglichkeit ihr Verhalten zu spezifizieren und zu verifizieren hat das Potential, für eine große Bandbreite von Programmen nützlich zu sein.

Prusti ist ein Verifizierer für Rust-Programme. Dank der weitreichenden Garantien, die die Speicher- und Aktivitätsträger-Schutzmechanismen in Rust bieten, unterstützt Prusti die automatische Ableitung eines grundlegenden Beweises der Sicherheit und der Nicht-Beeinträchtigung aus Informationen des Rust-Übersetzers. Darüber hinausgehende funktionale Spezifikationen können durch die Benutzer hinzugefügt und mit geringem Aufwand verifiziert werden.

Bisher bietet Prusti keine Unterstützung für Arrays und Slices. In der vorliegenden Arbeit wird die Unterstützung für Spezifikation und Verifikation von Array-Programmen in Prusti konzeptioniert und implementiert. Das beinhaltet nicht nur Mechanismen zur Kodierung von Datentypen und Programm-Konzepten mit Bezug zu Arrays und Slices, sondern auch Prädikate als Spezifikations-Werkzeuge, welche die komplette Spezifikationssyntax von Prusti inklusive Allquantifizierungen und Implikationen unterstützen, um Beweisführungen über Listen zu vereinfachen.

Die Lösung ist gut in die bestehenden Kodierungstechniken von Prusti integriert. Ihre Zweckmäßigkeit wird anhand einer Verifikation eines Sortieralgorithmus aufgezeigt.

Contents

Chapter 1

Introduction

Rust [1] is a relatively new systems programming language with a strong static type system, paired with a novel ownership and reference tracking system to enforce memory and thread safety at compile time.

These strong guarantees greatly simplify verification efforts, making it possible to automatically deduce and encode the basic foundation of safety and non-interference properties. Astrauskas et al. have implemented this automatic deduction in their verification tool Prusti [2]. For advanced, functional properties, further annotations by the user can be added.

Prusti is based on the Viper verification infrastructure [3]. It encodes the semantics of Rust programs to the Viper language and uses Viper to verify the result.

Given a simple Rust function like `calculate` in Listing 1.1, Prusti can check for the absence of panics (unexpected exceptions in Rust). Conditions in which a Rust program will panic include arithmetic overflows or division by zero, accesses outside of array bounds or explicit invocations of the `panic!` macro. In this case, as we have no further information about the inputs, `a + b` might overflow and `c` might be zero, leading to a division by zero. With further annotations these conditions could be excluded, and callers would need to make sure they only pass valid inputs.

```
fn calculate(a: i32, b: i32, c: i32) -> i32 {
  (a + b) / c
}
```

Listing 1.1: Rust function operating on integers.

While this automatic derivation is well supported for scalar variables and some aggregate types, support for sequences has been missing so far.

1.1 Support for Sequence Types

Consider the `sort` function outlined in the following example:

```
1 /// Sorts the sequence a.
2 fn sort(a: &mut [i32]) {
3   // ...
4 }
```

The comment declares that this function sorts the input sequence. But to make sure it actually does, we would need to be able to specify and verify that formally. In order to accomplish that using Prusti, we add support for arrays and slices and specification constructs to talk about sequences of data meaningfully.

1.2 Contributions

The main contributions of this thesis are:

- Syntax and implementation for freestanding specification items called *predicates* which support extended syntax including quantification and implication. This allows specifying properties like the sortedness of a sequence.
- Design of an encoding for arrays and slices to the Viper verification language [3]. Implementation of significant parts of the design in the Prusti verifier.
- Verification of two sort algorithms on arrays, in addition to a large number of simpler test cases, showing suitability of our designed encoding and implementation.

1.3 Outline

In the next chapter we present the necessary background on Rust, the Viper verification infrastructure and the Prusti Rust verifier, concluded by a short section on related work.

We then show the design of our encoding of array and slice types and operations as well as the new predicate syntax into Viper's verification language in Chapter 3.

In Chapter 4 we describe details of the implementation, which is evaluated in Chapter 5. The thesis concludes with Chapter 6.

Chapter 2

Background

This chapter introduces the Rust programming language and in particular its sequence types—namely arrays and slices. We then discuss the Viper verification infrastructure and the Prusti Rust verifier based on it, and finally some related work.

2.1 Rust

Rust is a relatively new systems programming language providing compile-time checked safety via its type system and by tracking ownership and borrowing of memory locations.

The focus in this section is to introduce some of Rust's safety and type system features, and go into more detail on its array and slice data types.

2.1.1 Ownership, Borrowing and Mutability

Rust's safety guarantees are in large part based on the way ownership and mutability of memory is handled. An object in memory always has exactly one variable as owner. If the variable goes out of scope without transferring ownership of the underlying memory, the memory will be deallocated. Access to the object can be passed by shared (immutable) or exclusive (mutable) reference, or by passing full ownership of the memory to or from function calls or other variables. The compiler will statically check that exactly one owner, at most one mutable reference or in the absence of a mutable reference, any number of immutable references to the same memory location exist. Coupled with Rust's strong type system, this allows relatively fine-grained abstractions with locally (function-scope) enforced rules and safe interfaces exposed.

Listing 2.1 shows an example of code the borrow checker rejects. In this small example not much needs to change to make the compiler accept the program;

```
let mut numbers = vec![1, 2, 3, 4];
let first = &numbers[0];
numbers.push(7);
println!("The first number is: {}", first);
```

Listing 2.1: Example of code the borrow checker rejects: appending an element might need to reallocate memory, invalidating the original reference `first`.

reordering statements is enough. In general, safety mechanisms like the borrow checker encourage programming along data accesses and modeling closely which operations need which level of access to data (immutable, mutable) in type definitions and function signatures.

This fits well with Viper's modular verification approach. Viper tracks memory permissions and verifies that all accesses happen with sufficient permissions. It verifies for each method separately that given the preconditions and permissions the postconditions will be fulfilled and returned resource permissions are held at the end of a method body. Then the specification is used in place of the body at call sites. Modular verification is also discussed in Section 2.2.8.

2.1.2 Arrays and Slices

Rust has two fundamental sequence types: arrays and slices. Arrays are fixed-size sequences. Both the element type and length are part of the data type; an array of 5 signed 32-bit integers is written as `[i32; 5]`.

Slices are dynamically sized sequences, in the sense that their length is not known at compile time. They can be thought of as windows into a part of another array or slice. The type of slices of booleans is written `[bool]`. To find the length of a slice, we need to call its `len` method. Neither arrays nor slices support dynamic resizing.

The dynamically resizing sequence type `Vec`, which is part of Rust's standard library, makes significant use of a number of Rust features still unsupported by Prusti.

Slices cannot be directly constructed (there is no slice literal syntax), but are created by slicing a backing array or slice. That means they usually[1] only exist as references to their backing sequence, that is as type `&[T]`, where `T` is the type of the elements. Examples of array and slice creation in Rust are shown in Listing 2.2.

[1] There are cases in the Rust standard library where slices are created from other owned data like a `Vec`'s contents through `Vec::into_owned_slice`. Here the result is a `Box<[T]>`, a sequence of elements on the heap without length information in the data type.

```
// initialize using explicit elements
let mut a = [1, 2, 4];    // type: [i32; 3]

// slice: take a sub-range
let b = &a[1..3];         // [2, 4], type: &[i32]

// take a mutable reference to an element
let c = &mut a[2];        // 4, type: &mut i32
```

Listing 2.2: Array and slice creation in Rust.

References to array elements—just like references to other memory locations—can be mutable or immutable. At any given time, either one mutable or any number of immutable references are permitted. As it is not in general clear at compile time which values an index variable will have (so it is impossible to determine at compile time whether references will access the same memory), a reference into an array blocks the whole array[2], as shown in the following example:

```
let mut a = [1, 2, 3];

let b = &a[0];     // OK
let c = &a[1];     // OK: multiple shared refs
let d = &mut a[2]; // Error: already borrowed immutably

println!("{}", b + c);
```

2.2 Viper

The Viper verification infrastructure [3, 4] is the foundation for a family of verification tools. This section will summarize the parts of Viper and its functionality relevant for this thesis: first a general overview of its architecture, followed by a description of some of the language constructs that Viper offers which we will use later on.

Figure 2.3 shows a schematic overview of Viper's architecture in the context of Prusti. A number of language frontends exist for Viper. Among these are Prusti for Rust programs and Gobra for the Go language. These frontends translate the semantics of programming constructs from their respective source languages into the Viper intermediate language *Silver*. When we talk about Viper programs we mean programs in the Silver language.

[2]Note that even if we could improve Prusti's handling of arrays here, such that we could prove indices distinct for some cases, we would still need the code to pass Rust's borrow checker. This means, in effect, that this is a limitation to be lifted by the Rust compiler, not Prusti alone, should this ever be attempted.

Viper provides different verification backends to verify such programs. Prusti exclusively uses the *Silicon* backend, which is based on symbolic execution. Both currently existing back-ends ultimately use the SMT solver Z3 to discharge proof obligations.

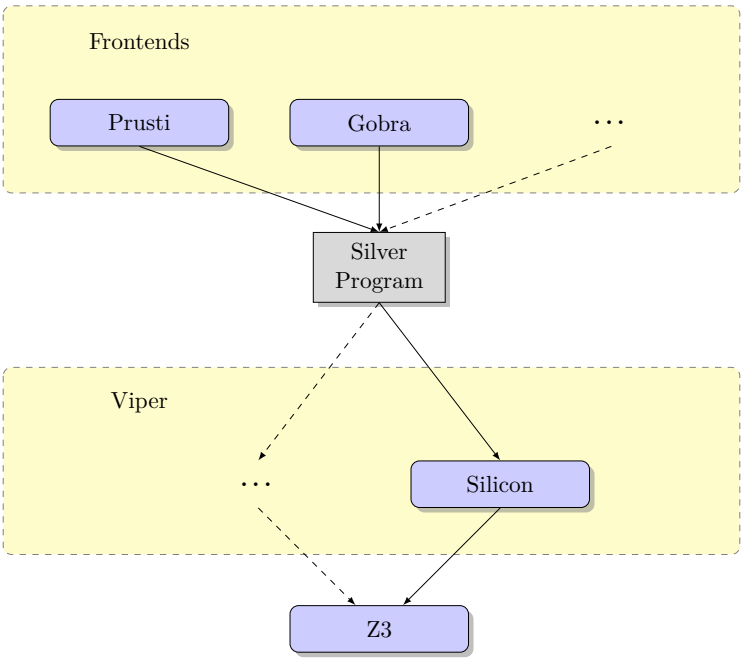

Figure 2.3: Overview of the Viper verification framework.

2.2.1 Structure of a Viper Program

Viper allows relatively high-level encoding of program behavior, resource permissions and verification conditions. A Viper program consists of functions, methods, predicates and data type declarations.

Methods can be viewed as sequences of operations. They form a modular boundary for the verification backend, in that behavior of methods is checked to fulfill its postconditions given the preconditions, but users of the method only interact with what is specified in pre- and postconditions, ignoring the body. Details on these are explained in Section 2.2.8.

Functions can be viewed as abstractions over expressions. They may not contain loops, but recursion is allowed. Functions cannot change any program state, just access it. They are presented in Section 2.2.4.

Predicates can be viewed as abstractions over assertions. They are described in more detail in Section 2.2.3.

Data type declarations include field declarations and domain definitions. They are explained further in the next subsection.

We discuss these in more detail in the following subsections, as well as a number of other features available in the Viper language.

2.2.2 Builtin Data Types and Permissions

Viper supports primitive datatypes like `Int` (unbounded integers) and `Bool`, some composite datatypes like `Seq[T]` for sequences of items of type `T` and `Multiset[T]`, and the special `Ref` and `Perm` types.

`Ref` is the type of references to objects. Objects have fields, which are declared globally. Which fields a concrete instance can access is encoded using permissions.

`Perm` is the type of permissions to memory locations. Permissions to a specific location are given as a fractional value between 0 and 1, where 0 equals no access, 1 means full (write) access, and any fractional value strictly between 0 and 1 amounts to read access. The symbolic names `none` and `write` are predefined in Viper.

Listing 2.4 shows an example declaring appropriate permissions for what the method `increase_if_active` needs. Note that permissions that are not passed back to the caller via postconditions are consumed. The `acc()` function denotes that we are talking about the permission to the named expression, not the expression itself.

```
field amount: Int
field active: Bool

method increase_if_active(self: Ref)
  requires acc(self.amount, write) && acc(self.active, 1/2)
  ensures acc(self.amount, write) && acc(self.active, 1/2)
{
  if (self.active) {
    self.amount := self.amount + 1
  }
}
```

Listing 2.4: Example method demonstrating permissions.

2.2.3 Named Permissions: Predicates

We can abstract over permissions using predicates. A predicate is a named permission to an optional set of fields or other predicate permissions. A typical

use case is to encode all permissions necessary to access a composite data type from the source language into one logically grouped item. An example of a predicate declaration for a tuple type looks as follows:

```
predicate tuple(self: Ref) {
  acc(self.left, write) && acc(self.right, write)
}
```

When working with predicate permissions, `acc(some_pred(self), write)` may be written as just `some_pred(self)`.

Folding and unfolding. In Viper, a predicate instance is not directly equivalent to the corresponding instantiation of the predicate's body, but these two assertions can be explicitly exchanged for one another. The statement `unfold P(...)` exchanges a predicate instance for its body; `fold P(...)` performs the inverse operation.

To minimize the necessary state space search, Viper requires this explicit folding and unfolding. While this is useful to simplify the verifier's task, we will omit the folding and unfolding parts from now on to improve readability.

Abstract predicates. Predicates do not necessarily need to have a body, that is, they do not need to name any fields or other permissions. We will use this in Section 3.5.1 to encode an abstract permission to an array. Abstract predicates are written as:

```
predicate Array(self: Ref)
```

They can be used just like regular predicates in pre- and postconditions, be added, asserted or removed from the verifier state, but cannot be unfolded.

2.2.4 Named Expressions: Functions

Just as predicates are an abstraction over assertions, functions are an abstraction over Viper expressions. They allow referring to an expression by name, including recursion in the function body. An example for a length function of a linked-list type `MyList` is shown here:

```
function len(l: Ref): Int
  requires MyList(l)
  ensures result >= 0
{
  l.next == null ? 0 : 1 + len(l.next)
}
```

Side effects. Function invocations are side-effect free; they are sometimes explicitly called *pure* functions to differentiate them from related concepts. They cannot modify permissions or values of variables.

Abstract functions. Functions do not necessarily need to have a body; those without a body are called *abstract* or *uninterpreted* functions. Invocations of uninterpreted functions are treated as symbolic values.

2.2.5 The Magic Wand Operator

Besides the usual logical operators `&&`, `||` and `==>` for conjunction, disjunction and implication, Viper supports the *magic wand*[3] operator `--*` known from separation logic [5, 6]. A magic wand `A --* B` represents a resource which, if combined with resource `A`, consumes `A` and yields resource `B`.

A typical use case for magic wands in the context of encoding Rust programs is when we have a reference to a `struct` instance and take a reference to a field of that `struct`. The encoding of taking a reference to a field will add both the new reference (resource `A`) and the ability to regain the original reference to the full structure (resource `A --* B`) to the state. Prusti encodes applying the magic wand at the location that the reference to the field expires, restoring full access to the original struct. This mirrors what the borrow checker enforces in Rust into the encoding in Viper.

2.2.6 Quantification and Triggers

Viper supports universal quantification of expressions and permissions as shown in this example:

```
forall i: Int :: { f(i) } i < f(i)
```

The expression inside the curly braces is called *trigger* and controls when Viper instantiates quantified expressions. In particular, when a universally quantified assertion is a hypothesis for a proof goal, the triggers cause the SMT solver to instantiate the quantifier only when it encounters expressions matching the trigger. When determining whether a trigger matches an expression, quantified variables act as wildcards.

Assuming the current proof goal is `2 * f(4) > 8` then `i` is matched to `4` and the quantified fact is instantiated for this concrete case, allowing verification of the goal. The function `f` here might be any function on integers that we do not know much about, other than its result being strictly greater than its input.

Matching loops. Matching loops are infinite loops that occur when a trigger expression immediately matches on an instantiation again. As an example, consider the following Viper program snippet:

[3] Also sometimes called *separating implication*, because it acts like an implication $A \Rightarrow B$ when disregarding resource assertions.

```
domain MyInt {
  function create(x: Int): MyInt
  function get(a: MyInt): Int
  function sum(a: MyInt, b: MyInt): MyInt

  axiom MyInt_sum {
    forall a: MyInt, b: MyInt :: { sum(a, b) }
      sum(a, b) == create(get(a) + get(b))
  }
}
```

Listing 2.5: Example of a custom integer type as a domain.

```
assume forall i: Int :: { magic(i) }
  magic(magic(i)) == magic(2 * i) + i

assert magic(magic(10)) == magic(12345) + 10
```

Verification of this snippet should fail, because our definition says nothing about the equality we are asserting. But due to the trigger, the solver will instantiate the quantified expression with `i == magic(10)`, matching the outer `magic` call on the left-hand side to the wildcard `i`.

As the instantiation immediately matches the trigger again, this results in an infinite loop.

Quantifiers, triggers and matching loops are explained in more detail in [4, Quantifiers Section].

2.2.7 Domain definitions

In order to define new datatypes in a Viper program, we use domains. A domain consists of a name, a number of uninterpreted functions and a number of axioms describing the behavior of the functions.

As an example, the Viper tutorial [4] shows the axiomatization of a custom integer type, reproduced in Listing 2.5.

The axiom `MyInt_sum` puts the uninterpreted functions `create`, `get` and `sum` in relation to each other.

2.2.8 Methods, Modular Verification and Program State

When verifying whether a method conforms to its specification, Viper (to be more exact: *Silicon*, the backend used by Prusti) will start with an empty program state. The program state includes the values of all variables in scope and the permissions currently held. At the beginning of the verification of a

method the resource and value assertions from the preconditions are added to the program state.

Then the method body is interpreted step by step, applying modifications to the program state or making assertions about it. Statements include assignments, labels, method calls, control structures like conditionals and loops, assertion checking and state modification statements and the `fold` and `unfold` statements which we have seen earlier.

Of particular interest are the `inhale` and `exhale` statements.

The informal semantics of `inhale A` is as follows:

- Add the permissions denoted by `A` to the program state.
- Assume that all value constraints in `A` hold.

The informal semantics of `exhale A` is as follows:

- Assert that all value constraints in `A` hold; if not, verification fails.
- Assert that all permissions denoted by `A` are currently held; if not, verification fails.
- Remove the permissions denoted by `A`.

These semantics match those of pre- and postconditions and make the same functionality available within method bodies. They are intended to enable the encoding of features of the verified source language that are not directly supported by Viper.

Note that any information about values of expressions is only valid as long as we have nonzero permission to access the respective expression. In other words, giving up access to an expression simultaneously gives up any knowledge about the expression.

Old expressions. It is possible to refer to an expression at an earlier point in the program using `old(x)`. Without any further information, this refers to the value `x` had at the beginning of the current method.

It is often useful to refer to the state of the left-hand side of a magic wand just before its expiry. This is what the implicitly defined label `lhs` in magic wand applications does: `old[lhs](x)` refers to the value of `x` just before applying the magic wand.

2.3 Prusti

Prusti is a Rust verifier building on top of the Viper verification infrastructure. It aims to abstract from the underlying verification language and allows users to give specifications in (augmented) Rust syntax.

This section gives an overview of how Prusti works and gives context to how our contributions fit into its overall architecture. Figure 2.6 shows a schematic overview of Prusti.

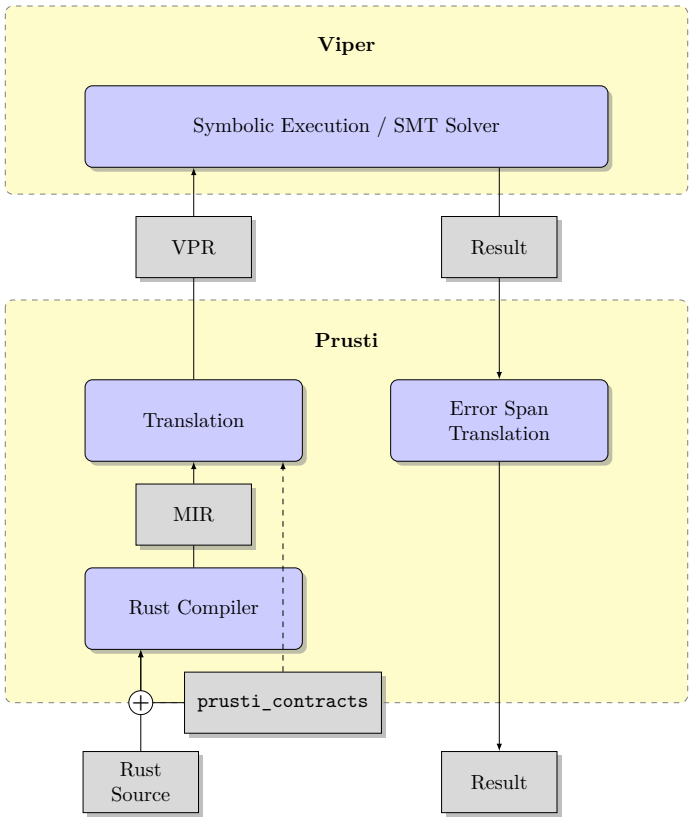

Figure 2.6: Prusti architectural overview.

When running Prusti on a Rust source file the following steps take place:

- Procedural macros and attributes (including for example `#[requires(..)]`) provided by the `prusti_contracts` crate that is part of Prusti) are evaluated by the Rust compiler. They may generate additional code, for example from pre- or postconditions, in order to have the Rust compiler typecheck them. This is represented by the dashed line from `prusti_contracts` through the Rust Compiler block to the Translation block. An example of additionally generated code can be found in Section 4.1.
- The Rust compiler runs compilation stages up until the generation of MIR[4], an intermediate code format.

[4]For an overview of MIR, see the MIR chapter in the Rust Compiler Developer's Guide

- Prusti interprets the MIR, including the additional code generated by the `prusti_contracts` macros, and outputs a Viper program consisting of the *core proof* and translations of user annotations of the input Rust program.
- The Viper verifier will try to verify the given program, returning a result to Prusti containing an error or a success message.
- Prusti will translate errors, if any, back to Rust source locations, and display them to the user.

Even without any user annotations, Prusti will generate a Viper output program containing the so-called *core proof*. The core proof is the translation of the semantics of the Rust program (and further information like borrow analysis calculated from it) to Viper. Successful verification of the core proof implies correctness of the Rust original with regard to memory safety. It provides the foundation for additional checks such as panic (unexpected exception) freedom, absence of overflows and user-provided assertions.

Abstract read permission. We have introduced in Section 2.2 that any permission amount strictly between 0 and 1 represents a read permission, and that Viper has symbolic names `none` for no permissions and `write` for permission amount 1.

For read permissions, Prusti does not follow the full model of fractional permissions, where partial permissions are added up again to a full write permission. Instead, an abstract read permission is used. This read permission is not actually added and subtracted, instead Prusti only models the permissions the Rust compiler assigns to variables and references. In other words, a write permission to a variable is regained when the last reference to it expires, not by adding up fractional permissions. The abstract read permission is defined like this:

```
function read(): Perm
  ensures none < result && result < write
```

More details on why this is a sound way of encoding permissions can be found in [2, Section 5.2].

2.3.1 Encoding of Rust Types

Prusti includes two ways of encoding types to Viper: *Heap-based encoding* and *Snapshot encoding*. Heap-based encoding models ownership of memory locations through permissions.

As not all operations in Viper are possible with heap-dependent data, there is the alternative snapshot encoding. It encodes values, without permissions

at https://rustc-dev-guide.rust-lang.org/mir/index.html.

or heap dependence. This allows types without direct Viper equivalents to be used anywhere in Viper pure expressions (including quantifiers, postconditions and pure function bodies).

Heap-based encoding using predicates. We will take a look at the encoding of some Rust types to Viper by Prusti. While for some of Rust's primitive types equivalents exist in Viper, the heap-based encoding needs to account for permissions and borrowing as well, so in this context all types are encoded using predicates on `Ref`-typed variables and adding or removing permissions to the respective fields. The `usize` type is the type of machine-word-sized unsigned integers, used for example for array indexing and length.

```
predicate usize(self: Ref) {
  acc(self.val_int, write)
}
```

All integer types use the `val_int` field for their contained value. When encoding references, we use the `val_ref` field.

```
predicate ref$usize(self: Ref) {
  acc(usize(self.val_ref), write)
}
```

Notice how the permission to access a `&mut usize` includes the `usize` sub-predicate's permissions.

For composite types like tuples and structures, the encoding becomes more elaborate, but still follows the same basic approach using `Ref` and predicates. Given this Rust structure definition:

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

Prusti will encode the following predicate describing the type and its permissions:

```
predicate Range(self: Ref) {
  usize(self.f$start) && usize(self.f$end)
}
```

One thing to note here is that `$` is a valid character in identifiers in Viper, providing an easy way to avoid name clashes when encoding.

Snapshot-based encoding. For primitive types (all integer primitive Rust types, including `char` and `bool`) their snapshot is just their value as an `Int` or `Bool`, respectively.

For composite types like tuples and structures, a corresponding snapshot domain[5] is generated.

The components of a snapshot domain for a Rust type `T` are:

- a domain type `Snap$T` representing deep copies of values of type `T`,
- a constructor function `cons$Snap$T` that takes all components that make up type `T` and returns a deep copy,
- domain axioms to make sure each instance of `Snap$T` corresponds to exactly one instance of `T`, i.e. there is a bijection between heap-dependent and heap-independent values[6] and
- for each field, a field access function and corresponding axioms specifying the field access function's behavior.

The snapshot encoding for the `Range` type is shown in Listing 2.7.

In addition to the domain definition itself, Prusti defines a function to take the snapshot of a heap-dependent value at some point in the program.

The snapshot function for primitive types like `usize` is just unfolding its predicate and extracting the contained value:

```
function snap$usize(self: Ref): Int
  requires acc(usize(self), read())
{
  unfolding acc(usize(self), read()) in self.val_int
}
```

2.3.2 Havocking: Erasing Knowledge about a Value

For some operations it is necessary or useful to explicitly erase all knowledge about a variable's contents. The operation to assign such an arbitrary value to a variable is commonly known as `havoc`.

At the time of writing, Viper does not have an explicit `havoc` statement, but we can build a method with the same effect:

```
method havoc_ref(): Ref
```

This method makes no guarantees or assertions about its return value. Therefore that value is arbitrary; the solver can make no assumptions about it.

Similar methods can be defined for other types. In Prusti they are named according to the return type, for example `havoc_int` or `havoc_bool`.

[5] The Prusti Developer Guide at https://viperproject.github.io/prusti-dev/dev-guide/encoding/types-snap.html describes snapshot encoding in more detail.

[6] In practice the injectivity axiom seems sufficient, so an axiom for surjectivity, encoding that all values of type `Snap$T` must correspond to an application of the `cons` function, is currently not generated.

```
domain Snap$Range {
  function cons$Range(start: Int, end: Int): Snap$Range

  axiom injectivity {
    forall s1: Int, s2: Int, e1: Int, e2: Int ::
      { cons(s1, e1), cons(s2, e2) }
      cons(s1, e1) == cons(s2, e2)
      ==> s1 == s2 && e1 == e2
  }

  function field_start(self: Snap$Range): Int

  function field_end(self: Snap$Range): Int

  axiom field_start_read {
    forall s: Int, e: Int :: { field_start(cons(s, e)) }
      field_start(cons(s, e)) == s
  }

  axiom field_end_read {
    forall s: Int, e: Int :: { field_end(cons(s, e)) }
      field_end(cons(s, e)) == e
  }
}

function snap$Range(self: Ref): Snap$Range
  requires acc(Range(self), read())
{
  unfolding acc(Range(self), read()) in
    cons$Range(snap$usize(self.f$start), snap$usize(self.f$end))
}
```

Listing 2.7: Snapshot domain for the `Range` type.

Another way of erasing knowledge from the program state is to (temporarily) drop all permissions to a location. As discussed in Section 2.2.8, information about values of expressions is only valid as long as nonzero permissions to that expression are held.

2.3.3 Encoding of Rust Functions and Methods

Prusti contains two styles of encoding functions from Rust. By default, functions from Rust are encoded to **methods** in Viper that capture the behavior of the original Rust code. Rust expressions in specifications and in functions with the *#[pure]* attribute are encoded to Viper **functions**.

Encoding inside a Viper method. When using the `Range` struct predicate in encoding Rust statements (that is, in a heap-dependent encoding context), Prusti creates a new variable of type `Ref` for each Rust variable. Adding the correct predicate permission to the variable makes it usable as the intended type.

```
let range = Range { lower: 3, upper: 5 };
```

Given this assignment, and the struct declaration from above, Prusti will generate the following (simplified) encoding:

```
var lower: Ref
lower := havoc_ref()
inhale usize(lower)
lower.val_int := 3

var upper: Ref
upper := havoc_ref()
inhale usize(upper)
upper.val_int := 5

var range: Ref
range := havoc_ref()
inhale Range(range)
range.f$lower := lower
range.f$upper := upper
```

Fields are encoded with the `f$` prefix to avoid name conflicts with `struct` field names in Rust. The basic mechanism for initializing variables is similar every time: a new variable is declared and initialized with an explicitly arbitrary value. Then we add predicate permissions for the target type and (with the unfolding of the predicate omitted) assign the desired values to the now accessible target fields.

As the encoding of this single statement is already much longer in Viper, we do not show an example of a full function being encoded to a Viper method using this encoding style.

Encoding a Viper function. A typical example for of a *#[pure]* function is the maximum of two numbers:

```
#[pure]
fn max(a: i32, b: i32) -> i32 {
  if a > b { a } else { b }
}
```

This will be encoded to the following (simplified) pure function in Viper:

```
function max(a: Int, b: Int): Int {
  a > b ? a : b
}
```

This concludes the background sections. We have introduced Rust and its sequence types, presented an overview of Viper and Prusti and shown core concepts. Finally, we have seen encodings of basic datatypes and functions from Rust to the Viper language.

2.4 Related Work

This section will present some related work: the Gobra verifier and especially its handling of arrays and slices and a short introduction of the Creusot Rust verifier.

Arrays and Slices in the Gobra Verifier

Gobra [7, 8] is an automated, modular verifier for Go programs, based on the Viper verification infrastructure.

Arrays and Slices in Go programs are nullable[7]. Gobra's array axioms account for that by only axiomatizing injectivity over the valid indices of the array, and `null` arrays having length 0 trivially fulfill that.

Go's arrays do not carry their length as part of the type (like arrays in Rust do), so Gobra needs to keep track of the length for both arrays and slices separately from the actual type.

Slices in Go allow re-slicing beyond the end of an existing slice, up to the length of the underlying array. The length from the beginning of a slice to the last valid index of the underlying array is called a slice's *capacity* in Go.

In contrast to Prusti's approach, permissions need to be handled explicitly by the user in Gobra. This simplifies the definition of the actual array datatypes, but in turn moves complexity to the user. On the other hand, due to the Go language not tracking mutability, multiple references into the same backing array are possible, and more than one of them may mutate elements.

Gobra has some different design constraints than Prusti, both due to the different semantics of arrays and slices in the Go language compared to Rust, and because of design choices such as making the user of the tool explicitly annotate permissions.

[7] *Nullable* here means arrays in Go inherently behave somewhat like references, being either `null` or an actual array.

Creusot Rust Verifier

Creusot [9, 10] is a tool for deductive verification of Rust code. It translates code—optionally annotated with specifications, invariants and assertions—to Why3 and then semi-automatically solves these verification conditions.

It does not currently support handling of arrays or slices, and instead exclusively uses linked lists for sequences of data.

Chapter 3

Design

In the last chapter we have presented the background required for our contributions: Rust and the semantics of its array and slice types, an overview and some of the verification features of the Viper verification infrastructure and finally how Prusti uses them in encoding Rust programs and specifications to the Viper language.

This chapter describes the design and the encoding of the added constructs: predicates as first-class items and arrays and slices as supported built-in Rust types, both in code and in specifications.

3.1 Predicate Syntax

Prusti specification syntax is an extension to the existing Rust pure expression syntax that additionally allows specification constructs like quantification and implications. While this syntax was available in pre- and postconditions already, we now added predicates as freestanding, named, function-like constructs that support this extended syntax.

Predicates make sense in the context of adding array and slice support, as sequences of elements are the prime example for quantified assertions. Specifying properties about a sequence without using quantification is difficult (it may be possible for fixed-size sequences by writing out all required properties on elements by hand).

As there is already support for quantified expressions in Prusti as well as for freestanding functions that can be used in specifications, the main design goal for freestanding predicates is to both allow reusing existing code as well as being intuitive to use.

The existing `#[pure]` functions in Prusti use function syntax, so predicates do that as well but allow the extended specification syntax in the function body:

```
fn sorted(a: &[i32]) -> bool {
  forall(|i: usize, j: usize| (0 <= i && i < j && j < a.len())
      ==> a[i] <= a[j])
}
```

Another important driver for the decision to make predicates use Rust's function call syntax is that it makes them usable in more contexts. The way the parser for Prusti specification syntax currently works does not support the extended operators in arbitrary positions, just the top level of the syntax tree. It does not currently support constructs like `result == (forall(..))`. This is a limitation in the current specification parser and will be lifted in the future. With predicates as function-like freestanding items, we can already use the extended syntax in arbitrary positions through the indirection of defining a predicate function: `#[ensures(result == sorted(input))]` is a valid postcondition.

3.1.1 Desugaring to Precondition

Note that the semantics of a predicate are different from a pure function that has the contained expression as a precondition. We initially considered this simple desugaring as a possibility, because it would have meant fewer necessary implementation changes. The reason this approach does not work as expected is because an unfulfilled precondition immediately fails the verification, whereas we want a boolean value as result. This is the desugared code when encoding to a precondition:

```
#[pure]
#[requires(forall(|i: usize, j: usize|
  (0 <= i && i < j && j < a.len()) ==> a[i] <= a[j]))]
fn sorted_precond(a: &[i32]) -> bool {
  true
}
```

Consider using both in a more complex expression like `#[requires(sorted(a)) || some_other_prop(a)]`. While `sorted` only returns a boolean value, `sorted_precond` will immediately error out if its precondition is not satisfied, making this approach unusable.

As we want to be able to use predicates as parts of expressions, we need to encode the actual assertion as the function body, not a precondition. We want to encode the sortedness example to Viper as follows (the array encoding shown here is just a placeholder, our encoding will be presented in the next sections):

```
function sorted_body(a: Array) -> Bool
{
  forall i: Int, j: Int ::
    (0 <= i && i < j && j < a.len()) ==> a[i] <= a[j]
}
```

The rest of this chapter will be about the encoding of arrays and slices.

3.2 Goals of the Array and Slice Encoding

Arrays and slices are ubiquitous in Rust code, so supporting them in Prusti significantly increases its applicability to real-world code.

Integration with Prusti's existing approach. We have seen the encoding of an example structure in Section 2.3. The encoding we choose for array operations integrates well with the existing approach for encoding data types and functions to Viper. That includes a way to encode array operations both in specifications and in regular code, that supports modeling permissions like Prusti does for other composite data types and pure, value-only snapshots.

As another important type of integration, the code written as part of this thesis should integrate well with the existing codebase.

General applicability. The encoding should work well with all constructs that come up in encoding Rust's array and slice operations. The typical operations include creation, element lookup, taking references to single elements, shared and mutable slicing and taking the length of a sequence.

3.2.1 Encoding of Arrays and Slices in Gobra

In Section 2.4 we have seen that Gobra has some different design constraints from Prusti. Arrays and slices in Go are different from arrays and slices in Rust in some aspects, and permissions in Gobra are not handled automatically as in Prusti.

As it is possible in Go to have multiple references mutating arrays (directly or through slices), Gobra models individual elements of arrays. Array allocation, slicing and lookups are fundamentally defined by uninterpreted functions and axioms regarding length, capacity and offsets, backed by permissions to elements quantified over these lengths.

In Rust the semantics are different. For mutable accesses, write access to the whole array is needed. And even for read access, the whole array is relevant, not just a single element: if a conflicting mutable reference exists, reading

from the array is not allowed. Therefore we do not gain much from encoding elements or element access separately in Prusti.

Nullability of arrays is not possible in Rust, so we do not need to account for it in our encoding in Prusti.

3.2.2 Design Choices for our Encoding

We have seen in the previous subsection that we do not gain much for our encoding of Rust arrays from modeling individual elements. Arrays and slices in Rust act as one unit when we look at how references to elements block the whole array or slice for modification, or in the case of mutable references for any other reference.

Therefore, we base our encoding around arrays and slices as entities and not single elements. We encode permissions and looking up or modifying elements on these entities.

In practice Rust avoids some of the issues with blocking the whole array by providing functions like `split_at_mut` on slices. The `split_at_mut` function splits a mutable slice up into two mutable slices at a given index, as a safe abstraction around unsafe code. Our approach supports this kind of methods by adding support for arrays and slices in specifications.

3.3 Encoding of Arrays and Slices in Prusti

This section introduces the `lookup_pure` function which is at the heart of our encoding of arrays and slices in Prusti. Starting from its definition we will give an overview of the design of the encoding of arrays and slices, both in regular code and in specifications.

3.3.1 Abstract Predicate Encoding of an Array and Lookup

The encoding is based on abstract predicates. While the resource predicates for other types like structures and tuples name explicit fields, we cannot name fields for arrays. Instead, we represent arrays or slices as a whole using an abstract predicate. The abstract predicate for the Rust array type `[i32; 3]` is:

```
predicate Array$3$i32(self: Ref)
```

This encodes the permission to use `self` as an array of size `3` with element type `i32`.

The semantics of operations on arrays are encoded via an abstract function `lookup_pure` that is defined for each array type (i.e. element type and length) similar to this one for `[i32; 3]`:

```
function lookup_pure(self: Ref, idx: Int): Int
  requires acc(Array$3$i32(self), read())
  requires 0 <= idx && idx < 3
```

This function requires read access to the array, and needs the index to be within bounds for the specific array type's length. It returns the value at position `idx` in the array. The return type is the snapshot of the indexed element, which happens to be `Int` for `i32`. For an array of `struct Foo { .. }` elements, the corresponding `lookup_pure` function would return a `Snap$Foo`. We have seen an example of such a snapshot encoding in Section 2.3.1.

As it has no function body, similar to the abstract predicate, `lookup_pure(array, idx)` is treated as an abstract, symbolic value of type `Int`. We use this symbolic value to encode the semantics of other Rust constructs.

The intuition behind the name is that it looks up a value in an array at a given index, and returns a pure, heap-independent representation of the element. The `_pure` suffix also distinguishes it from the `lookup_shared` and `lookup_mut` methods that we will define in Section 3.5.1 to encode shared and mutable array access operations from Rust, respectively.

Note that even though there exists one such function per element type and length (and slightly different ones for arrays and slices), we will use the same identifier `lookup_pure` for readability.

3.3.2 Heap Independence: Snapshots

While the `self` parameter to `lookup_pure` is of type `Ref`, that is, it depends on data on the heap, the index parameter and importantly the return value do not. This way we can use them throughout specifications without having to worry about permissions to the underlying array or index variable.

3.3.3 Operations on Arrays and Slices in Rust

The main operations we want to support are

Array creation `let a = [0, 1, 2];`

Element lookup `let b = a[1];`

Element referencing `let c = &mut a[i];`

Shared slicing `let d = &a[1..3];`

Mutable slicing `let e = &mut a[2..4];`

Sequence length `let n = a.len();`

Of particular interest when encoding mutable referencing and mutable slicing operations is that they change the backing sequence's values. In a compiled

Rust program, this happens somewhat implicitly by writing to the corresponding memory location. In Viper we need to encode this explicitly.

In the following sections we will discuss the encoding of array and slice types and corresponding operations both snapshot-based and heap-based, corresponding to specification and procedural code in more detail. Finally, we will discuss how this approach generalizes to multidimensional sequences and take a look at an alternative encoding approach centered around treating all array operations like method calls and providing specifications for those.

3.4 Snapshot Encoding

We start with the snapshot encoding of types. The return value of the `lookup_pure` function that we started with is a snapshot of the element at the given index. In general, snapshots are used when encoding specifications. This includes both internal specifications like the `lookup_pure` function we use to describe the array and slice operations as well as user-added specification annotations and specification functions (that is, those declared with the `#[pure]` attribute or predicates).

We have seen the heap-independent encoding of composite types in Section 2.3.1. The example there translated the `Range` type into a domain definition `Snap$Range` and a snapshot function `snap$Range`.

3.4.1 Snapshots of Arrays

When comparing to a struct like the `Range` type, we have some different requirements for arrays. When encoding the values for an array type `[T; N]`, we need to store the value of all elements in order to have a complete copy of its contents. While each array has a fixed number of elements, we ideally want to use an encoding scheme that works for any size, to be able to snapshot arbitrary array types. Therefore, we use Viper's builtin sequence type `Seq` to store snapshots of the elements.

Listing 3.1 shows the snapshot domain definition for the array type `[i32; 5]`. The length of the snapshot is encoded in the snapshot's type, just like the length of the array in Rust is encoded in the array's type.

Using the uninterpreted functions `cons` and `read`, we define in the axioms the semantics of these functions and how they interact. The `read` function is to a snapshot of type `Snap$Array$5$i32` what `lookup_pure` is to a heap-based array instance of type `Array$5$i32`: it looks up an element, given an index.

The `read` function works similar to how `read_start` and `read_end` are used in the snapshot encoding for `Range`, with the difference of the additional index parameter. The corresponding axiom looks similar as well: while the read axioms for `Range` specify which of the constructor parameters each read function

```
domain Snap$Array$5$i32 {
  function cons(data: Seq[Int]): Snap$Array$5$i32

  function read(arr: Snap$Array$5$i32, idx: Int): Int

  axiom injectivity {
    (forall _l_data: Seq[Int], _r_data: Seq[Int] ::
        cons(_l_data) == cons(_r_data) ==> _l_data == _r_data)
  }

  axiom read_indices {
    forall data: Seq[Int], idx: Int ::
      { read(cons(data), idx) } { data[idx] }
        read(cons(data), idx) == data[idx]
  }
}
```

Listing 3.1: Snapshot domain definition for `[i32; 5]`.

returns, the read axiom for the array snapshot domain specifies which element is returned from the `data` sequence.

As length of the array is part of the type name, we do not need to encode it separately. We will have to do that for slices later on, even though their encoding is overall very similar to that of arrays.

Snapshot function. To take a snapshot of a heap-dependent array instance at some point in the program, we define a snapshot function.

```
function snap$Array$5$i32(self: Ref): Snap$Array$5$i32
  requires acc(Array$5$i32(self), read())
  ensures forall i: Int ::
    { read(result, i) } { lookup_pure(self, i) }
    read(result, i) == lookup_pure(self, i)
{
  cons(Seq(
    lookup_pure(self, 0),
    lookup_pure(self, 1),
    // ...
    lookup_pure(self, 4),
  ))
}
```

To bridge the gap between the heap-based and snapshot-based world we have

to collect all values before calling the `cons` function. For arrays we can just encode calls to `lookup_pure` for indices from 0 to the length of the array (which is encoded in from the array's type). Remember that the return value of `lookup_pure` is already of some snapshot type, so there is no need to call snapshot functions for the elements.

The postcondition was added to solve certain triggering issues we found during evaluation.

Tuple-like encoding of array snapshots. An alternative design we initially considered would be to snapshot arrays similar to how we snapshot tuples or structures—using one parameter to the constructor per element. This would avoid constructing an intermediate `Seq` of the elements.

The `cons` function for `Snap$Array$5$i32` would be:

```
function cons(x0: Int, x1: Int, x2: Int, x3: Int, x4: Int)
```

The read axiom would become significantly more complicated however, as we can no longer pass on the indexing operation to the `Seq`.

We would either have to explicitly enumerate equalities for all allowed indices like this:

```
axiom read_indices_enumerate {
  forall x0: Int, x1: Int, x2: Int, x3: Int, x4: Int ::
    { /* triggers omitted */ }
    let snap == (cons(x0, .., x4)) in
      read(snap, 0) == x0 &&
      read(snap, 1) == x1 &&
      read(snap, 2) == x2 &&
      read(snap, 3) == x3 &&
      read(snap, 4) == x4
}
```

Alternatively, we could try to use just one `read` call with a dynamic index, but that index would still need to be matched to its respective constructor parameter. We would then end up with a large cascade of case distinctions:

```
axiom read_indices_ifelse {
  forall i: Int, x0: Int, x1: Int, x2: Int, x3: Int, x4: Int ::
    { /* triggers omitted */ }
    read(cons(x0, .., x4), i) == (i == 0 ? x0
      : (i == 1 ? x1
      : (i == 2 ? x2
      : (i == 3 ? x3 : x4))))
}
```

As both of these are not ideal, the solution utilizing the `Seq` type makes most sense in our eyes. It also seems the best of the alternatives in handling large arrays, like buffers of 1024 or 4096 elements, common in Rust code that interacts with streams of data.

3.4.2 Snapshots of Slices

Snapshots of slices work similarly to snapshots for arrays. The most significant difference is that slices have dynamic length. That means that we need to add a `len` function to our slice snapshot domain, along with axioms to specify what it does:

```
domain Snap$Slice$i32 {
  // parts similar to arrays omitted

  function len(slice: Snap$Slice$i32): Int

  axiom slice_data_len {
    forall d: Seq[Int] :: { len(cons(d)) } len(cons(d)) == |d|
  }

  // added to solve certain triggering issues discovered during
  // evaluation
  axiom len_positive {
    forall s: Snap$Slice$i32 :: len(s) >= 0
  }
}
```

While for the heap-dependent encoding we have a separate abstract function `Slice$len`, for snapshots the length needs to be part of the `Snap$Slice$i32` value. As the contained `Seq[Int]` already contains the length of the slice implicitly, we encode that as an axiom. The syntax `|d|` designates the sequence length of `d`.

Collecting Heap-Dependent Values for Snapshot Construction. When we snapshot a slice, we cannot just take the length from the type like we did for arrays and collect that many `lookup_pure` calls as the slice's `data`. Instead, we need to look up `Slice$len(self)` elements, with `Slice$len(self)` being an uninterpreted function. To collect that dynamic number of elements, we define a recursive function `slice_collect` shown in Listing 3.2. Note that while `idx` is a parameter of the `slice_collect` function, we always collect a full slice; in case we only need a subsequence of its elements, the slicing will be applied on the resulting snapshot using snapshot slicing machinery.

```
function slice_collect(slice: Ref, idx: Int): Seq[Int]
  requires acc(slice$i32(slice), read())
  requires 0 <= idx
  ensures idx >= Slice$len(slice)
    ==> result == Seq[Int]()
  ensures idx < Slice$len(slice)
    ==> |result| == Slice$len(slice)-idx
  ensures idx < Slice$len(slice)
    ==> result[0] == lookup_pure(slice, idx)
  ensures idx < Slice$len(slice)
    ==> forall i: Int, j: Int ::
      idx <= i && i < Slice$len(slice) &&
        0 <= j && j < Slice$len(slice)-idx && i == j + idx
          ==> lookup_pure(slice, i) == result[j]
{
  (idx >= Slice$len(slice))
  ? Seq[Int]()
  : Seq(lookup_pure(slice, idx))
      ++ slice_collect(slice, idx + 1)
}
```

Listing 3.2: Recursive element collection function for slice snapshots.

Once we have collected the elements, the slice snapshot is created just like an array snapshot from the `Seq` of elements:

```
function snap$Slice$i32(self: Ref)
  requires acc(Slice$i32(self), read())
{
  cons(slice_collect(self, 0))
}
```

Now that we have seen the encoding of values as snapshots, we will take a more in-depth look at the heap-based encoding. To recap, heap-based encoding is used when encoding permissions is required, that is, encoding of all Rust functions and methods outside of specifications.

3.5 Heap-Based Encoding

This section describes the encoding of array operations when we are encoding in a heap-dependent method context; that means we encode arrays as `Ref`-typed variables, to which we have abstract array predicate permissions.

3.5.1 Array Operations

This subsection will go over the basic operations we introduced in Section 3.3.3 and show how to encode them in heap-based encoding context.

Array creation. When encoding an array creation, we have a left-hand side variable that we need make usable as an array with the contents given as the right-hand side operands.

```
let a = [1, 3, 5];
```

We have seen in Section 2.3.1 when encoding the `Range` type's initialization that a variable is declared, then initially cleared using `havoc` and then the correct access predicates are added. Assuming the local variable for `a` is `_1`, this is what lines 1-2 do. Next we need to make the array's contents known in lines 3-5:

```
1  _1 := havoc_ref()
2  inhale Array$3$i32(_1)
3  inhale lookup_pure(_1, 0) == 1
4  inhale lookup_pure(_1, 1) == 3
5  inhale lookup_pure(_1, 2) == 5
```

Note that the right-hand side of these equalities is of the type of snapshots of the array elements. For integer constants that does not change much, as they are the same in Rust and Viper. If a right-hand side operand is not a constant, we would see the snapshot application explicitly; the inhaled equality for the local variable `x: i32` encoded to Viper variable `_3` would then be:

```
inhale lookup_pure(_1, 2) == snap$i32(_3)
```

This works the same way for primitive as for composite data types, making this approach generally applicable and integrate well with the existing (snapshot) encoding in Prusti.

Element lookup. When looking up a single element from an array, we have an array and an index as inputs, the array with array predicate permission, and the index with `usize` permissions. Rust only allows `usize`-typed indices for arrays, so we can hardcode the `usize` predicate into the precondition.

The output is a `Ref`-typed variable holding the element at the specified index in the array. The name `lookup_shared` expresses that we lookup from a shared (that is, immutable) reference to an array here.

```
method lookup_shared(self: Ref, idx: Ref): Ref
  requires acc(Array$2$i32(self), read())
  requires usize(idx)
  ensures acc(Array$2$i32(self), read())
  requires (0 <= idx.val_int && idx.val_int < 2)
  ensures acc(i32(result), write)
  ensures snap$i32(result) == lookup_pure(self, idx.val_int)
```

The method requires array permissions and returns those same permissions again. This reflects the fact we can lookup from shared references, without blocking the array for further shared lookups.

Note that we do not rely on the Rust compiler here more than anywhere else, we just encode statements as they appear in the MIR. If it turns out we do not have read access to the array at the point in the program where we want to look up an element (and the Rust compiler made an error in encoding the array access anyway), the precondition will not be fulfilled and the verification will consequently fail.

Note that we have full permissions for the resulting variable. This is because we could have code like:

```
let a = [0; 3];
let mut x = a[i]:
x += 3;
```

where a variable gets its initial value from a shared array lookup, but will be modified later on in the program. This does not modify any of the values inside the array.

Note that there is a distinction between taking a shared reference to an array element and looking up an element by-value. For types that can be trivially copied, both are permitted by the Rust compiler. For others, the lookup would mean a transfer of ownership out of the array, so it is not permitted in that case.

As far as Prusti is concerned, we encode the array access the same way for both cases, resulting in a fresh temporary variable that holds the contents of the array at the given index. Encoding a reference to that is encoded like any other reference, using the `val_ref` field. For shared array element access we do not have to encode any value update. For mutable element references, the update mechanism is shown in the next paragraph.

Mutable element referencing. When we want to modify entries in an array, we can do so in two ways: taking a mutable reference into the array and assigning to that, or assigning to the array directly. These two work very similarly from a design standpoint, but we can use a simpler encoding in the second case, that is, if no temporary reference is created.

```
method lookup_mut(self: Ref, idx: Ref): Ref
  requires acc(Array$2$i32(self), write)
  requires 0 <= idx.val_int && idx.val_int < 2
  ensures ref$i32(result)
  ensures lookup_pure(self, idx.val_int) == snap$ref$i32(result)
  ensures ref$i32(result) --*
    // regain array permission
    acc(Array$2$i32(self), write) &&
    // all other elements unchanged
    forall i: Int :: { lookup_pure(self, i) }
      0 <= i && i < 2 && i != old(idx.val_int)
      ==> lookup_pure(self, i)
        == old[lhs](lookup_pure(self, i)) &&
    // indexed element updated
    lookup_pure(self, old(idx.val_int))
        == old[lhs](snap$ref$i32(result))
```

The first three items are very similar to the `lookup_shared` case, except that we require full permission to the array now. This encodes the fact that by Rust's semantics a mutable reference into an array blocks the whole array.

Just like in the `lookup_shared` case we need to encode the value of the returned reference.

The last `ensures` encodes what happens when the reference into this array is expired. We have introduced the magic wand operator `--*` in Section 2.2.5. In this case it encodes the capability to exchange the result of `lookup_mut` (the reference to the element) for the conjunction.

The magic wand instance is automatically applied by Prusti at the location where the reference expires. It has three components: regaining permission to the array, value equalities for all array indices other than the one looked up, and finally one about the value of the indexed element.

The regained array access is the same as the one we initially consumed when calling `lookup_mut` and it is regained at the point where the result reference expires.

As we had temporarily lost any permission to the array, the verifier state has also cleared any information about the array such as value equalities we had previously established. To recover the values of the elements that remain unchanged, we `inhale` equalities with their respective values before the method call.

The indexed element may have been changed through the returned reference, so we need to equate it to the last value the reference had before its expiry, using `old[lhs]` in the magic wand.

We need the `old(..)` around `idx.val_int` because we do not have permissions to it anymore at the point where the magic wand is applied.

Direct array assignment. When directly assigning a new value into an array, we could just encode it as a `lookup_mut` call followed immediately by applying the resulting magic wand.

As there is no reference generated by the Rust compiler that could expire, we can use an updated version of the method that does not generate an intermediate magic wand.

```
method update_array(self: Ref, idx: Ref, val: Ref)
  requires acc(Array$2$i32(self), write)
  requires 0 <= idx.val_int && idx.val_int < 2
  requires acc(i32(val), read())

  ensures acc(Array$2$i32(self), write)
  // all other elements unchanged
  ensures forall i: Int :: { lookup_pure(self, i) }
      0 <= i && i < 2 && i != old(idx.val_int)
      ==> lookup_pure(self, i)
        == old(lookup_pure(self, i))
  // indexed element updated
  ensures lookup_pure(self, old(idx.val_int)) == snap$i32(val)
```

We snapshot the value to be assigned, as always when `inhale`ing an equality with a `lookup_pure` invocation.

This method is somewhat simpler in that in directly encodes the new state in postconditions instead of encoding them in a magic wand and does not need to encode the old value at the indexed array position in a returned reference. It also makes sense to use this simpler variant because the Rust compiler does not encode a reference for this case, so we would have to take care of applying a magic wand ourselves.

Regarding its effects on the given array, this variant still has the same effect as the previous version of `lookup_mut` and immediate expiry.

3.5.2 Slice Operations

To recall, slices are views into an underlying array or slice. They can be mutable or shared, just like single-element array references. In fact, the encoding of the slicing operation is a generalization of the encoding of array indexing to a range of indices.

Abstract slice length function. As the length of slices is not known at compile time, we need some other way to encode the length, for example in bounds checks.

The slice length is encoded using an abstract function `Slice$len`. When we create a slice, we have information about the value of the (symbolic) slice length, so we can encode it in an `inhale` statement to make it known to Viper.

Encoding of shared slicing operation. The slicing operation is typically[1] invoked by indexing an array with a range of indices. The encoding of this in MIR passes the range as an instance of the `Range` type (or a similar type) we have seen earlier. Given this slicing operation:

```
let s = &a[i..j];
```

We can encode the difference of the given variables as the newly created slice's length. This works even if we do not know at compile time what the values of the indices are going to be.

For the elements, we can refer to what we know about the elements of the backing array (or slice).

Using `N` as placeholder for the backing array's length and `T` for the type, we have for the encoding of the slicing method:

```
method slice_shared(self: Ref, lo: Ref, hi: Ref): Ref
  requires acc(Array$N$T(self), read()) && usize(lo) && usize(hi)
  requires 0 <= lo.val_int && lo.val_int <= hi.val_int
      && hi.val_int <= N
  ensures acc(Slice$T(result), read())
  ensures Slice$len(result) == hi.val_int - lo.val_int
  ensures forall i: Int, j: Int ::
    { lookup_pure^A(self, i), lookup_pure^S(result, j) }
      lo.val_int <= i && i < hi.val_int &&
      0 <= j && j < (hi.val_int - lo.val_int) &&
      i == j + lo.val_int
    ==> lookup_pure^S(result, j) == lookup_pure^A(self, i)
```

Listing 3.3: Shared slicing operation.

The `slice_shared` method has some similarity with the `lookup_shared` method shown earlier: both require read access to the backing array, and need the indices to be within bounds. For encoding the length of the slice, we have the

[1] There are implicit conversions, for example when calling slice methods like `.len()` on an array. These are encoded as if the full length of the array was passed into the slice operation.

abstract function `Slice$len`, and define its value for this particular slice in the postcondition.

The last `ensures` is to define the values of the slice elements. We map the `lookup_pure` calls on the resulting slice to the corresponding shifted indices on the original sequence.

Note that the two `lookup_pure` calls here are to different versions: one for the array being sliced, the other for the new slice. We have annotated them in Listing 3.3 using `S` and `A` for the slice and array variant, respectively.

The two quantified variables in the postcondition instead of just one serve to avoid matching loops. Assume we encoded in the postcondition of `slice_shared` an equality with just one index variable:

```
forall i: Int :: { lookup_pure^A(self, i + lo.val_int) }
                 { lookup_pure^S(result, i) }
  // indices
  lookup_pure^A(self, i + lo.val_int) == lookup_pure^S(result, i)
```

Then having `lookup_pure(result, i)` as an expression instantiates the quantified fact, yielding `lookup_pure(self, i + lo.val_int)`. This will lead to an instantiation of `lookup_pure(self, i + lo.val_int + lo.val_int)` and so on. Having separate variables avoids this issue.

Encoding of mutable slicing operation. We have seen both the mutable array indexing and the shared slicing. The mutable slicing is the combination of those two: encoding lookup of multiple elements, including a magic wand to encode the update of element values after the expiry of the slice. The special case for assigning an array element without an intermediate reference does not exist for slices, as there is no range assignment operation in Rust.

The code for the mutable slicing operation—shown in Listing 3.4—starts the same way as that for `slice_shared`: require proper permissions for the inputs and encode the values and length of the resulting slice.

The magic wand in lines 15-30 encodes regaining access to the original array and the updated values when the slice expires. Similar to the magic wand for a mutable reference into an array we need to encode that all elements that were not referenced (inside the slice) have the same values as before the slicing (lines 16-19 and 21-23). The elements inside the slice receive their new values from the values at corresponding indices at the expiring slice just before the expiry (lines 25-30). Note that while the `old` expression for the unchanged elements does not mention any specific label, and so refers to the state at the begin of the `slice_mut` call, the updated elements refer to `old[lhs]`, that is the state of the left-hand side of the magic wand just before it is applied.

```
method slice_mut(self: Ref, lo: Ref, hi: Ref): Ref
  requires Array$N$T(self) && usize(lo) && usize(hi)
  requires 0 <= lo.val_int && lo.val_int <= hi.val_int
    && hi.val_int <= N

  ensures Slice$T(result)
  ensures Slice$len(result) == hi.val_int - lo.val_int
  ensures forall i: Int, j: Int ::
    { lookup_pure^A(self, i), lookup_pure^S(result, j) }
      lo.val_int <= i && i < hi.val_int &&
      0 <= j && j < (hi.val_int - lo.val_int) &&
      i == j + lo.val_int
    ==> lookup_pure^S(result, j) == lookup_pure^A(self, i)

  ensures Slice$T(result) --* Array$N$T(self) &&
    // 0..lo unchanged
    forall i: Int :: { lookup_pure^A(self, i) }
      0 <= i && i < old(lo.val_int) ==>
        lookup_pure^A(self, i) == old(lookup_pure^A(self, i)) &&
    // hi..end unchanged
    forall i: Int :: { lookup_pure^A(self, i) }
      old(hi.val_int) <= i && i < N ==>
        lookup_pure^A(self, i) == old(lookup_pure^A(self, i)) &&
    // indexed updated
    forall i: Int, j: Int ::
      { lookup_pure^A(self, i), lookup_pure^S(result, j) }
      old(lo.val_int) <= i && i < old(hi.val_int) &&
      0 <= j && j < old(hi.val_int - lo.val_int) &&
      i == j + old(lo.val_int)
    ==> lookup_pure^A(self, i) == old[lhs](lookup_pure^S(result, j))
```

Listing 3.4: Encoding of mutable slicing operation to Viper.

Slicing a slice. When the backing storage for a slicing operation is not an array but itself a slice, we need to account for that in the encoding. The basic idea remains the same.

As we have seen for the other slicing operations, there is one `lookup_pure` instance for each array or slice type. If the sequence being sliced is a slice itself, we encode its respective `lookup_pure` function. We also need to replace the permissions with corresponding array permissions, and the simple length expression from the array type with a call to `Slice$len(self)`. With those changes, the rest of the `slice_shared` and `slice_mut` encodings are the same.

3.6 Multidimensional Arrays

The design presented here works for multidimensional arrays as well, contributing to the generality of our approach. We will outline in the following subsections what the previously presented array operations look like for multidimensional arrays.

Note that in Rust there is no multidimensional slicing operation, that is, it is not currently possible to slice each element in a slice of arrays in one go. In separate statements, the encoding described for the slicing operation before works as expected for each step.

3.6.1 Creation

Given the following Rust code snippet:

```
let a: [[i32; 2]; 2] = [ [0, 1], [2, 3] ];
```

We have three basic operations to encode: the creations of the two nested arrays, and then the creation of `a`. This is also reflected in the MIR encoding generated for this snippet by the Rust compiler:

```
_2 = [const 0_i32, const 1_i32];
_3 = [const 2_i32, const 3_i32];
_1 = [move _2, move _3];
```

We have already seen how to encode the first two statements in Section 3.5.1, so we only show one of the two here:

```
_2 := havoc_ref()
inhale Array$2$i32(_2)
inhale lookup_pure(_2, 0) == 0
inhale lookup_pure(_2, 1) == 1

// similar for _3
```

When encoding the creation of the outer array, we first remind ourselves that the type of the `lookup_pure` invocation (i.e. the return type, and consequently the type of the symbolic value of the invocation) is the type of snapshots of the elements. That means that the `lookup_pure` instance for an array with arrays as elements looks like this:

```
function lookup_pure(self: Ref, idx: Int): Snap$Array$2$i32
  requires acc(Array$2$Array$2$i32(self), read())
  requires 0 <= idx && idx < 2
```

And with that we can encode the values at indices of the outer array as:

```
_1 := havoc_ref()
inhale Array$2$Array$2$i32(_1)
inhale lookup_pure(_1, 0) == snap(_2)
inhale lookup_pure(_1, 1) == snap(_3)
```

The application of the snapshot function on the right-hand side returns a snapshot of a whole array here, as that is the element type of the outer array. That is, the approach translates transparently to multi-dimensional arrays.

3.6.2 Element Lookup

Looking up values in multidimensional arrays has similar adjustments from the single-dimensional case. When encoding this Rust fragment (in the context of the array `a` created just before):

```
let b = a[1];
```

We again use a snapshot equality to encode the fact that a new local variable now has certain contents:

```
1 // _1 is the MIR local variable for a
2 // _5 is the MIR local variable for b
3 _5 := havoc_ref()
4 inhale Array$2$i32(_5)
5 inhale snap(_5) == lookup_pure(_1, 1)
```

That means the snapshot equality in line 5 encodes the information that the array `_5` (now) has the contents that `lookup_pure` returned as snapshot of the second entry in the outer array. From there a further lookup can be encoded for the elements of `_5` in the same way.

3.6.3 Mutation

When mutating elements in multidimensional arrays, we need to act step by step. Accessing a nested element is encoded as step-wise access of sub-arrays, one index operation per step, until we reach the inner array we want to modify. Then modify that array in the usual way. It does not conceptually matter at this point if we are replacing a complete sub-array or just an element, as both amount to a single `inhale` statement. Then we encode the assignment of each element (array or single value) into the next outer array as we did for single elements, using snapshot equality.

So we encode the assignment `a[i][j] = new_val` in three separate steps:

```
let mut tmp = a[i];
tmp[j] = new_val;
a[i] = tmp;
```

Encoding that to Viper looks like this:

```
// create and initialize tmp variable
tmp := havoc_ref()
inhale Array$2$i32(tmp)
inhale lookup_pure(a, i) == snap(tmp)

// clear info about tmp's contents
label old_tmp
exhale Array$2$i32(tmp)
inhale Array$2$i32(tmp)

// encode updated info about contents
inhale forall k: Int :: 0 <= k && k < 2 && k != j
    ==> lookup_pure(tmp, k) == old[old_tmp](lookup_pure(tmp, k))
inhale lookup_pure(tmp, j) == snap(new_val)

// clear info about a's contents
exhale Array$2$Array$2$i32(a)
inhale Array$2$Array$2$i32(a)

// encode updated info about contents
inhale forall k: Int :: 0 <= k && k < 2 && k != i
    ==> lookup_pure(a, k) == old[old_tmp](lookup_pure(a, k))
inhale lookup_pure(a, i) == snap(tmp)
```

Mutable References. Taking a mutable reference into a multidimensional array follows the same basic structure. We know how to encode taking a mutable reference into a one-dimensional array: encode the value of the reference as in the immutable case, and encode a magic wand that will re-enable access to the original array as well as encode that the indexed item in the array now has a new value and all other values keep their previous values.

Given the array from earlier, encoding a reference like this:

```
let b = &mut a[i][j];
*b = new_val;
```

consists of the following steps:

```
let subarray = &mut a[i];
let elem_ref = &mut subarray[j];
*elem_ref = new_val;
// elem_ref expires, updating subarray[j]
// subarray expires, updating a[i]
```

Encoding of the value of a single element to one lookup per dimension happens the same way as we have seen for the multi-dimensional lookup earlier. Updating after the reference expires is encoded in corresponding magic wands, just as we have done previously for mutable access. The change for the multidimensional case is that we have one magic wand per dimension, just as we had to encode one lookup per dimension.

We encode the innermost array reference—`elem_ref` in the example—which is a reference into a one-dimensional array, so we already know how to encode it from Section 3.5.1. Its magic wand will update the referenced element in `subarray` on expiry.

At this point we are left with `subarray`—a mutable reference to an element of `a`—that is about to expire. But we already know how to encode the update of an array when a mutable reference to an element expires: using a magic wand. In fact, the same magic wand definition we used in Section 3.5.1, with the element type specialized to the type of `subarray`.

This way we encode all nested updates. Each application of a magic wand returns permissions and updated values of a subarray that we use in updating the next array outwards.

As snapshots of elements and arrays work the same way with the `lookup_pure` function on which we based our encoding, this design works on both equally well and allows us to nest arrays in the way we have just shown.

Chapter 4

Implementation

This chapter discusses the implementation of our design. It starts with an overview of Prusti's architecture and then show how each new part fits into that architecture and which changes were made.

4.1 Prusti Architecture Overview

At a high level, Prusti is a plugin to the Rust compiler that converts Rust code enriched with Prusti specifications into Viper code, verifies the code with an external verifier, and then reports the results back to the user.

As shown in Figure 4.1, the first stage is the handling of the specification attributes, which are implemented as Rust procedural macros in the `proc-macros` stage. Depending on the type of specification, additional code is generated to

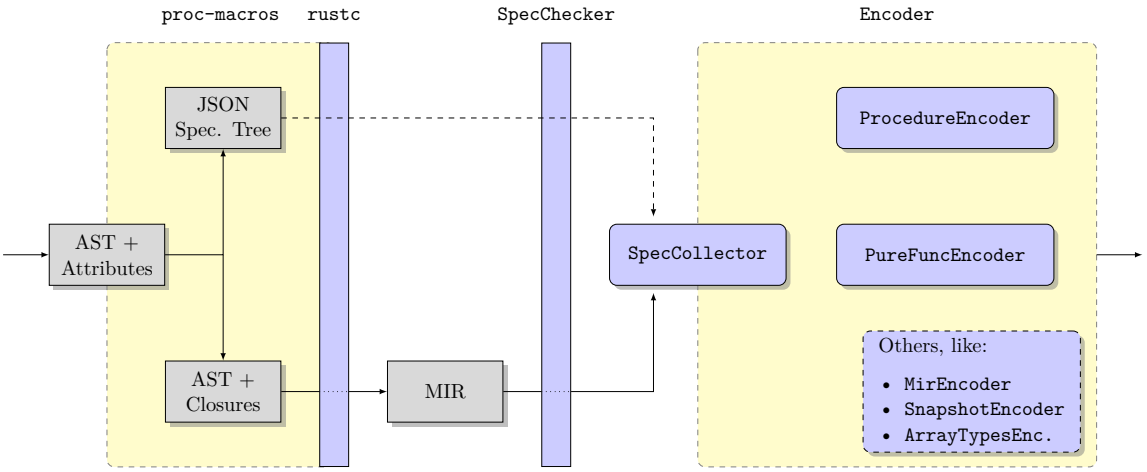

Figure 4.1: Prusti implementation overview.

typecheck the specifications. After the Rust compiler has performed all checking and compilation steps up to the generation of MIR, the MIR is inspected by the specification checker module and all specifications that were generated for a function are collected for encoding.

The specification checker is a new module added during this thesis, it is described in Section 4.2.

Specification attributes. The input file may contain annotations from the `prusti_contracts` crate such as `#[requires(..)]`, `#[ensures(..)]` or `#[pure]`. These are procedural macros that are evaluated by the compiler. Procedural macros are one way of preprocessing available in Rust, described in more detail in the Rust Reference[1]. In short, procedural macros are Rust functions that receive a `TokenStream`, an early representation of Rust syntax during parsing, as input, and generate a new `TokenStream` as output. The output is fed back to the compiler and interpreted in place of the original code.

Two main things happen in the procedural macro implementations: specifications are parsed from attributes and additional code is generated to typecheck specifications and make them available for later stages.

To illustrate, let us look at an example of a simple function with a postcondition:

```
#[ensures(result == a + b)]
fn add(a: i32, b: i32) -> i32 {
  a + b
}
```

The `#[ensures(..)]` macro will generate a new specification id, parse the ensured expression and generate a new function with the purpose of having the Rust compiler typecheck it, and later to be able to translate the resulting MIR code snippet into Viper specifications. The result is shown in Listing 4.2.

Once the typechecking is done and Prusti starts processing the MIR, the `#[prusti::post_spec_id_ref = ".."]` identifier references are used to match generated specification snippets with the functions they are specifying. The structure of the specification (in this case just one expression) is represented in the `#[prusti::assertion]` attribute, referencing the generated closures by the generated `spec_id` and a sequential `expr_id`. This module is called `SpecCollector` (see Figure 4.1).

[1] https://doc.rust-lang.org/reference/procedural-macros.html

```
// generated to typecheck the asserted expressions
#[allow(unused_must_use, unused_variables, dead_code)]
#[prusti::spec_only]
#[prusti::spec_id = "c5ac4263"]
#[prusti::assertion = "{\"kind\":{
  \"Expr\":{\"spec_id\":\"c5ac4263\",\"expr_id\":101}}}"]
fn prusti_post_item_add_c5ac4263(a: i32, b: i32, result: i32) {
    #[prusti::spec_only]
    #[prusti::expr_id = "c5ac4263_101"]
    || -> bool { result == a + b };
}

// the actual function, with reference to specifications by id
#[prusti::post_spec_id_ref = "c5ac4263"]
fn add(a: i32, b: i32) -> i32 {
  a + b
}
```

Listing 4.2: Intermediate Rust code for the encoding of the `add` function.

Encoding of specifications and functions. Once all specifications have been collected, they need to be encoded from MIR into Viper expressions, pre- and postconditions, functions and methods. In the encoder, there are two main modules relevant to our contributions: the `pure_function_encoder` which is responsible for encoding MIR to the different kinds of Viper specifications, and `procedure_encoder`, responsible for encoding MIR to Viper methods. The procedure encoder will call out to the pure function encoder to encode specification snippets. Both of these use the `mir_encoder` module to encode common building blocks like single expressions or types from MIR to Viper.

The encoded program will be checked by Viper and the results reported back to the user, mapping source locations and possible errors back to the respective Rust code. While we did add an extra error variant for array and slice index bounds checks, this part of Prusti was not our main focus.

The rest of this chapter will present the additions and changes made to incorporate predicate syntax as well as array and slice support.

4.2 Predicate Syntax

The ingredients for predicates as freestanding function-like items are already present in Prusti: functions with the `#[pure]` attribute are encoded to Viper `function`s and usable in specifications, just like we designed predicates to be. The full Prusti specification syntax, including quantification and implications is available in pre- and postconditions already. So to implement predicates,

we need to bring these two existing concepts together. An extra step that we need to add is a check that predicates are only used inside specifications.

4.2.1 Syntax

Ideally, we would have wanted predicates to use the `#[predicate]` attribute for consistency with Prusti's other specification attributes. This does not work due to the way the Rust compiler handles procedural macro attributes. The compiler performs a certain level of parsing (for example for name analysis) on all toplevel items, even before procedural macro attributes are evaluated. The extended Prusti syntax contains constructs that are not valid Rust expressions, like the implication operator `==>`, and thus would fail to parse. This parse error would occur before our `#[predicate]` macro even applied, leading to a compile error.

Instead, predicates use the `predicate!{}` syntax. Wrapping predicate functions in this macro call prevents them from the very early parsing before the macro is evaluated, so Prusti syntax can be parsed and rewritten.

4.2.2 Stages of Encoding

We have seen the stages of encoding a specification attribute in Section 4.1. When translating a predicate instead of a postcondition, a similar sequence of steps is performed.

Procedural Macro Translation. Using the function-like syntax to denote a predicate function, we can translate this sortedness predicate:

```
predicate! {
  fn sorted(a: &[u32]) -> bool {
    forall(|i: usize, j: usize|
      (0 <= i && i < j && j < a.len()) ==> a[i] <= a[j])
  }
}
```

into the output of the `predicate!` procedural macro as shown in Listing 4.3.

The result of the procedural macro contains two functions: a generated function containing parts of the predicate body and a version of the original predicate function without the body.

The generated function's purpose is to have the compiler typecheck the contained expressions and generate MIR for them. The `#[prusti::assertion = ".."]` attribute in lines 4-13 preserves the structure of the predicate's body in JSON format, referencing the snippets by their `expr_id` attribute.

The body of the predicate function is replaced with `unimplemented!()`, to make the function pass the Rust compiler's parser and further checks. The

#[prusti::pred_spec_id_ref] attribute ties the additional generated specification function back to the original predicate function, just like *#[prusti::post_spec_id_ref]* did for the postcondition and its generated function item.

Specification Checker. The difference between predicates and pure functions is that predicates support Prusti's extended syntax, which goes beyond Rust's side-effect free expression syntax. So while there is no reason not to use *#[pure]* functions outside of specifications as well—as they have clearly defined semantics in Rust—this is not true for predicates.

We have added a new specification checker module (see Figure 4.1) to Prusti that makes sure that predicate functions are only used within specification code. We expect there to be further semantic checks on specifications that the specification checker could perform in the future.

Encoding to Viper. The next stage after the specification checker is the specification collector. It will walk the whole typechecked MIR program and collect assertion fragments into procedure contracts by their identifier references. When encoding, a predicate will be encoded as the body of a Viper function, instead of as a pre- or postcondition. The sorted predicate from above is encoded by Prusti to this Viper function:

```
function sorted(s: Snap$Slice$u32): Bool
{
  forall i: Int, j: Int ::
    0 <= i && i < j && j < len(s)
      ==> (i < len(s)
        ==> (j < len(s) ==> read(s, i) <= read(s, j))
        && j < len(s))
      && i < len(s)
}
```

Which is equivalent to what we specified initially, with the length constraints somewhat complicated. The extra roundtrip of requiring `i < len(s)` and having it given later in the same expression is due to the bounds check that the Rust compiler will insert.

```
#[allow(unused_must_use, unused_variables, dead_code)]
#[prusti::spec_only]
#[prusti::spec_id = "e5d7aa"]
#[prusti::assertion = "{\"kind\":{\"ForAll\":[
  {\"spec_id\":\"e5d7aa\",\"expr_id\":101,\"count\":2},
  {\"kind\":{\"Implies\":[
    {\"kind\":{\"And\":[
      {\"kind\":{\"Expr\":{\"spec_id\":\"e5d7aa\",\"expr_id\":102}}},
      {\"kind\":{\"Expr\":{\"spec_id\":\"e5d7aa\",\"expr_id\":103}}},
      {\"kind\":{\"Expr\":{\"spec_id\":\"e5d7aa\",\"expr_id\":104}}}
    ]}},
    {\"kind\":{\"Expr\":{\"spec_id\":\"e5d7aa\",\"expr_id\":105}}}
  ]}},[]]}}"]
fn prusti_pred_item_sorted_e5d7aa(a: &[u32]) {
  #[prusti::spec_only]
  #[prusti::expr_id = "e5d7aa_101"]
  |i: usize, j: usize| {
    #[prusti::spec_only]
    #[prusti::expr_id = "e5d7aa_102"]
    || -> bool { 0 <= i };

    #[prusti::spec_only]
    #[prusti::expr_id = "e5d7aa_103"]
    || -> bool { i < j };

    #[prusti::spec_only]
    #[prusti::expr_id = "e5d7aa_104"]
    || -> bool { j < a.len() };

    #[prusti::spec_only]
    #[prusti::expr_id = "e5d7aa_105"]
    || -> bool { a[i] <= a[j] };
  };
}

#[allow(unused_must_use, unused_variables, dead_code)]
#[prusti::pure]
#[prusti::trusted]
#[prusti::pred_spec_id_ref = "e5d7aa"]
fn sorted(a: &[u32]) -> bool {
  unimplemented!()
}
```

Listing 4.3: Intermediate Rust code for the encoding of the `sorted` predicate, as generated by the `predicate!` procedural macro.

4.3 Arrays and Slices

Adding support for arrays and slices requires changes mostly in the procedure and pure function encoder modules. As we have seen in Sections 3.5.1 and 3.5.2, arrays and slices in the design we propose integrate well with Prusti's existing encoding technique.

We will put focus on a few key changes during the development, starting with changes due to the fact that arrays are encoded using the `lookup_pure` function instead of fields like on `struct` types. Section 4.3.4 will talk about value preservation in loops, where special care is needed for arrays and slices as compared to other variables, followed by error reporting changes for index bounds checks in Section 4.3.5. Finally, Section 4.3.3 will discuss an alternative implementation approach that we initially considered.

4.3.1 Places in the Rust Compiler

In the Rust compiler, an important type for holding memory locations are *Places*. A place is a local variable with a (possibly empty) sequence of projections, such as field accesses, array element accesses or dereferences. In Prusti's encoding to Viper, most of the time a place is translated to a variable or a field of a variable, as for example referencing and enums are encoded to fields instead of as separate concepts.

With the way we designed the encoding of arrays and slices, we do not have explicit variables or fields for each element, only knowledge through symbolic `lookup_pure` or `read` calls mentioning the array. As these are encoded differently depending on whether we are in procedure encoding or specification encoding context, we added an abstraction between the common code in the `MirEncoder` module and the consumer modules. As field accesses, dereferencing and array or slice indexing are all valid operations and can be grouped, when encoding a place we get a possibly recursive structure as a result. The common structure is called `PlaceEncoding` and shown in Listing 4.4.

4.3.2 Encoding Expressions into Statements

This change—while enabling encoding of array accesses—has two effects that we need to adjust to. Code that previously expected a simple `Expr` now needs to handle `PlaceEncoding` instances, and needs to either ignore cases other than simple expressions and fields on them (that is, anything related to arrays or slices) or encode the array and slice accesses, resulting in sequences of statements in the encoding of procedures where we previously only had expressions.

As an example, a struct field access like `range.start` on the right-hand side of an assignment will be encoded to `_3.f$start`, in a Viper expression.

```
enum PlaceEncoding {
  Expr(Expr),
  Field {
    base: Box<PlaceEncoding>,
    field: Field,
  },
  Variant {
    base: Box<PlaceEncoding>,
    variant_field: Field,
  },
  ArrayAccess {
    base: Box<PlaceEncoding>,
    index: Expr,
  },
  SliceAccess {
    base: Box<PlaceEncoding>,
    index: Expr,
  }
}
```

Listing 4.4: PlaceEncoding structure definition.

An array element lookup like `a[i]` on the right-hand side will be encoded to the following, assuming the local variable holding the array is encoded to `_1` and the index variable `i` is encoded to `_5`:

```
var _tmp: Ref
inhale usize(_tmp)
inhale _tmp.val_int == lookup_pure(_1, _5.val_int)
```

And the fresh variable `_tmp` being used as the resulting expression.

Due to every invocation creating new temporary variables, we need to make sure to not rely on the encoding of a place to Viper being the same when encoding it multiple times. It is possible to compare the `PlaceEncoding` instead of the final sequence of statements if necessary.

4.3.3 Encoding as Methods with Specifications

A number of testcases for Prusti already contain a set of specifications for a `Vec`-like type with operations including `push`, `len`, `lookup` and `lookup_mut`. An initial idea for implementation of array and slice support was to encode the required operations as method calls that specifications could be attached to. We did end up not following through with the idea, for a number of reasons.

Patching AST In order to encode array accesses as function calls, we would have had to patch the program's syntax tree at some point during the compilation to insert function calls in places where array accesses happen.

- For the AST, the first syntax tree representation in the Rust compiler pipeline, we would have had the main issue that it is not clear whether an array access is mutable or immutable, or even whether an expression is related to arrays at all; the `Index` and `IndexMut` traits allow overriding the sequence index operator for custom types as well. Without that knowledge, it is impossible to patch in the correct dummy function.
- The following intermediate representations—HIR and typed HIR—are not intended for modification from outside the compiler (that includes from plugins like Prusti).
- The MIR, the representation Prusti uses, already has array accesses grouped up with other projections like field accesses and dereferencing, so preprocessing of the MIR would have been necessary before the actual encoding, increasing the risk for errors during encoding.

Specification Mechanisms The current `#[extern_spec]` mechanism that in principle allows attaching specifications to methods outside the current crate is not powerful enough to specify what we needed for arrays. At the very least, there are currently issues with generics. In addition, support for the `impl<T> [T]` syntax for implementation blocks for slices is missing in `#[extern_spec]`. This syntax seems to need the `#[lang = "slice"]` attribute in the standard library. Introducing the required specifications programmatically at the start of Prusti would give up an important advantage of this approach by making it impossible to use the concise existing specification syntax directly.

Adaption to array types One part that cannot be specified in `#[extern_spec]` annotations is choosing the specific matching `lookup_pure` instance matching the array or slice type.

We agree that it would be great to reuse existing mechanics like those for function specifications directly, but in practice we deemed it not worth the effort. We are nonetheless reusing significant parts of the code.

4.3.4 Value Preservation in Loops

The encoding of loops in Prusti uses a rewriting where loops are replaced with nested `if` statements and clearing of local variables to simulate an arbitrary loop iteration. The transformation is described in more detail in [11]. In short, a loop of the form:

```
while (b) {
  body_invariant!(I);
  B;
}
```

is rewritten into the following:

```
1  if (b) {
2    exhale I;
3    // havoc local vars
4    assume I;
5    if (b) {
6      assert I;
7      assume false;
8    }
9    assume !b;
10 }
```

The idea behind this is to make sure the invariant holds when initially entering the loop (line 2). Then, havocking all local variables and assuming the loop invariant emulates an arbitrary loop iteration by erasing all prior knowledge about variables except for the invariant. The `if` in line 5 encodes the case that the loop is entered another time, in which case we only need to make sure that the loop invariant holds again. If that is the case, `assume false` will make this branch of the verification succeed in any case, as we have already checked the case for an arbitrary loop iteration. Otherwise, `assume !b` will make sure that after the loop, the invariant will hold, but the loop condition will not anymore, the expected outcome when the loop has finished.

For regular variables, encoded as simple expressions, Prusti will just generate calls to `havoc_ref` or `havoc_int`.

As we have seen earlier, the encoding of an array access is a fresh temporary variable. Havocking those would not fulfill the intention here: we need to clear information about array contents, not the temporary variables, in order to match the behavior on other variables.

So the current solution is to treat array accesses separately when encoding havocking of the local variables. Read accesses to arrays do not lead to any havocking, as those cannot change across loop iterations. This is how variables that are only ever read during the loop are treated as well. Write accesses lead to the whole array being havocked, as we cannot tell which indices have been written during the loop. This fulfills the requirements, but is overly conservative, in that it might lose more information about the array contents than necessary.

4.3.5 Error Reporting

Prusti performs error translation from Viper errors back to the source location in the original Rust program. This is done by creating a mapping of pairs of locations in the generated Viper program and types of Viper errors to Prusti errors that are returned to the user when a matching error has occurred at a matching location in the viper program.

Adding support for arrays and slices also included adding a Prusti error kind for failed bounds checks on a sequence. In the context of adding this, we successfully contributed a small change[2] to the Rust compiler.

While the implementation does not cover all parts of the design yet, we have added solid foundational support. We have highlighted in this chapter a number of additional topics that became relevant during implementation and how we addressed them; what benefits or future work we see in them.

[2] https://github.com/rust-lang/rust/pull/84392

Chapter 5

Evaluation

The main goal of this thesis was to add support for arrays and slices to Prusti, and adding predicate syntax to improve and simplify specification of sequence-using code.

We have implemented predicate syntax and significant parts of the array and slice support we designed. This chapter will evaluate this implementation by showing the verification of a sort algorithm and measuring the effect of added support on the amount of real-world code that is supported by Prusti.

5.1 Implementation Status

Predicate syntax has been implemented as proposed. We can express sortedness of an array or slice like this:

```
predicate! {
  fn sorted(s: &[i32]) -> bool {
    forall(|i: usize, j: usize|
      (0 <= i && i < j && j < s.len()) ==> s[i] <= s[j]
  }
}
```

Arrays. Arrays are supported well in Prusti currently. In particular

- creation,
- both shared and mutable element lookup,
- mutation through references or directly and
- bounds checks and taking the length

are implemented for arrays both in specifications and in regular code. Support for multidimensional arrays is not currently implemented.

Note that taking the length of an array using `a.len()` involves a slicing operation that the Rust compiler automatically inserts, as `len` is a method on slices, not arrays.

We encode expressions containing array accesses using Viper statements (see Section 4.3.2). There are some parts in the encoding of expressions that do not handle the generated statements yet, so currently array or slice accesses on the right-hand side of assignments are not supported in compound expressions. That means that `let b = a[i] + 1` and `a[i] += 1` (which is short for `a[i] = a[i] + 1` are not supported, but `let tmp = a[i]; a[i] = tmp + 1` is.

Slices. Slices are supported in specifications—as we have seen above in the sortedness example—and basic operations are implemented in regular code. In particular,

- creation,
- element lookup,
- bounds checks and taking the length and
- explicit and implicit slicing of arrays and slices

are implemented for slices in specifications. In regular code,

- shared slicing of arrays,
- taking the length of slices and
- element lookup

are supported, while mutable slicing is not yet supported.

Integration into the main Prusti codebase. All of the features implemented are merged or part of an open pull request to the Prusti git repository[1]. A selection of changes is listed here, with 48 total pull requests merged and 2 still open.

- **#390: Add predicate syntax** Initial pull request to add predicate syntax.
- **#391: Check for extra parameters to *#[pure]*, *#[trusted]* attributes** Error message for invalid use like *#[trusted(foo)]*.
- **#435: Add section about predicates to the user guide**
- **#455: Builtins: add array lookup pure function** This PR added the `lookup_pure` function and array type predicates.

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

- **#466: Add arrays to TypeVisitor**
- **#467: Work around AssertKind::description panicking for BoundsCheck**
- **#470: Introduce PlaceEncoding machinery**
- **#475: Implement obtaining array length**
- **#476: Array access projection** This PR added array element access; the term *projection* is used by the Rust compiler for field access, array and slice accesses and others like dereferences.
- **#478: Implement encoding of array creation**
- **#484: Implement taking references into arrays**
- **#489: Encoding of mutable array indexing**
- **#495: Add the slice builtins**
- **#505: Array snapshot encoding and specs**
- **#521: Implement array initialization from repeat expressions** Repeat expressions are a shortcut for array initialization giving one copyable element and a count as in `[0; 1024]`.
- **#532: Refactor array types encoding** This PR added a separate `array_types_encoder` module.
- **#537: Change predicate syntax to allow nested implications** This PR changed the syntax for predicates from the `#[predicate]` attribute to `predicate!{}` to allow syntax rejected by the Rust compiler (like `==>`) inside predicates.
- **#538: Implement enough of array slicing to support .len() calls** The `len` method is a method on slices, so calling it on arrays needs a minimum level of slice support for the conversion the compiler automatically inserts.
- **#564: Add snapshot encoding for slices**
- **#566: Verify selection sort**
- **#568: Verify mergesort's merge function on static arrays**

5.2 Verification of a Sort Algorithm

We provide a verification of a selection sort implementation, showing that our design and implementation work. The code is shown in Listing 5.1.

Selection sort is a sort algorithm with a run-time quadratic in the number of elements. The basic idea of selection sort is to go through the positions in the array in ascending order. For each position `i`, select the next-smallest of the remaining elements in a linear scan, then swap it to the position `i` and go to position `i + 1`.

When verifying this sort algorithm, we need to encode the following conditions:

- Array indices are within bounds. This is encoded by the invariants in lines 6 and 20.
- For the outer loop, we know that elements at previous indices (smaller than `i`) are already sorted, while all remaining elements are bigger (or equal) than those already sorted. This is encoded by the invariants in lines 9-10 and 12-14 respectively.
- To make sure these outer invariants are kept intact by the inner loop, we need to repeat them in the inner loop as well (lines 20-25).
- The inner loop has to find the next-smallest item among the remaining elements. The control variable `j` must be between `i` and the length, and the index `min` of the smallest element so found so far must be within these bounds as well (lines 27-28).
- While searching for the next-smallest element among the remaining ones, we know that
  - all previously sorted elements (indices below `i`) are smaller than what we have as our currently smallest found element. This is similar to the general invariant in line 12, but gives us specific information about the value at index `min`, the currently smallest element found (lines 30-31).
  - among the not yet sorted elements that we already checked in our search for the minimum, the element at index `min` is the smallest (lines 35-36).
- Therefore, at the end of the inner loop `min` is the index of the smallest remaining element. Swapping it to the smallest remaining index `i` will make the array sorted at the end of the outer loop.

```
fn selection_sort(mut a: [i32; 10]) {
    let mut min;
    let mut i = 0;

    while i < a.len() {
        body_invariant!(0 <= i && i < a.len());

        // sorted below i
        body_invariant!(forall(|k1: usize, k2: usize|
            (0 <= k1 && k1 < k2 && k2 < i) ==> a[k1] <= a[k2]));
        // all below i are smaller than all above i
        body_invariant!(forall(|k1: usize, k2: usize|
            (0 <= k1 && k1 < i && i <= k2 && k2 < a.len())
                ==> a[k1] <= a[k2]));

        min = i;
        let mut j = i + 1;
        while j < a.len() {
            // these three are the same as for the outer loop
            body_invariant!(0 <= i && i < a.len());
            body_invariant!(forall(|k1: usize, k2: usize|
                (0 <= k1 && k1 < k2 && k2 < i) ==> a[k1] <= a[k2]));
            body_invariant!(forall(|k1: usize, k2: usize|
                (0 <= k1 && k1 < i && i <= k2 && k2 < a.len())
                    ==> a[k1] <= a[k2]));

            body_invariant!(i < j && j < a.len());
            body_invariant!(i <= min && min < a.len());
            // all previously sorted are smaller than the current min
            body_invariant!(forall(|k: usize|
                (0 <= k && k < i) ==> a[k] <= a[min]));

            // all not-yet-sorted checked so far are bigger
            // than the current min
            body_invariant!(forall(|k: usize|
                (i <= k && k < j && k < a.len()) ==> a[min] <= a[k]));

            if a[j] < a[min] {
                min = j;
            }
            j += 1;
        }

```

```
44            let a_i = a[i];
45            let a_min = a[min];
46            a[i] = a_min;
47            a[min] = a_i;
48            i += 1;
49        }
50    }
```

Listing 5.1: Verification of a selection sort implementation.

Necessary changes for Prusti support. In order to avoid features not yet supported by Prusti, some changes from a canonical Rust implementation of selection sort had to be made. These are

Array as input. Due to the implementation of slices not being as complete as the one for arrays, namely around mutable slicing, the input to our selection sort here is an array instead. The verification does not depend on the fixed length in any way; we use `a.len()` in all conditions and invariants.

Iterators. As iterators are not supported in Prusti, the loops are not written more succinctly, for example as `for i in (0..a.len())`, but instead use a separate variable that is manually incremented.

Element swap. The swapping of two elements in lines 44-47 only works in this form (with temporary copies) for Rust types that can be freely copied (that is, implement the `Copy` trait). For others the Rust standard library provides a `swap` method on slices that exchanges the values at two given indices without requiring them to implement `Copy`.

Generics The algorithm is monomorphized to `i32` elements. As the only operations on the elements are comparing and swapping, much the same verification approach should work for any element type that has a total order (that is, implementing the `Ord` trait).

Rust standard library sort. The sort algorithm in Rust's standard library[2] is a modified version of merge sort. It keeps track of sequences of already sorted increasing and decreasing sequences and merges pairs of adjacent sequences. Very short sequences are sorted in-place using insertion sort.

In order to verify its functional correctness, the following features would be required in Prusti or the corresponding code rewritten without them in `sort`:

Unsafe code is used in multiple places throughout the algorithm for slice access without bounds checks (as they have already been checked). These parts can be rewritten relatively easy to avoid unsafe code.

[2] https://doc.rust-lang.org/std/primitive.slice.html#method.sort

crate name	before	after	total methods
siphasher	36	38	105
crc	13	13	37
libc	29	29	164

Table 5.2: Number of methods fully encodable in crates that previously had methods unsupported because of missing array or slice support.

Loop conditions Matching on an `Option` in a loop condition is not supported yet, as are loop conditions involving iterators like `for i in 2..v.len()`. These can be rewritten as well.

Generics and Closures While there is support for both generics and closures in Prusti, there have been issues, so it might still be better to monomorphize to one element type and replace the `is_less` closure with a simple comparison.

5.3 Quantification of Improvements on Real-World Code

In this section we evaluate the impact that supporting arrays and slices has on the amount of code supported at least to the degree that Prusti successfully generates a corresponding core proof. As programming language constructs are not usually used in isolation, this turns out to be difficult.

Prusti regularly runs checks on a number of Rust packages from the set of the most downloaded packages on the official registry `crates.io`. These tests are run using a mode where an unsupported construct does not abort a Prusti run, but instead replaces the current function with an empty one and continues with translation.

Of those crates where missing array or slice support was the reason for failure to encode methods, we can report a slight increase in the number of successfully encoded methods.

On the remaining 35 crates, no change in the number of fully supported methods was measured. This is not surprising, as the reason they were not fully supported yet was not array or slice support.

The reason for this small amount of change, we believe, is that methods are still very coarse-grained units of measure when it comes to measuring the amount of supported constructs like array accesses.

Consider for example the function `make_table_crc16`, taken from the `crc` crate. While it seems like a prime example of code that should be fully supported, now that Prusti has array support, it still contains two types of constructs that are not supported yet: iterators and bitwise integer operations. This makes the method as a whole not supported by Prusti yet.

```rust
// from crc crate, utils.rs
pub fn make_table_crc16(poly: u16) -> [u16; 256] {
    let mut table = [0u16; 256];
    for i in 0..256 {
        let mut value = i as u16;
        for _ in 0..8 {
            value = if (value & 1) == 1 {
                (value >> 1) ^ poly
            } else {
                value >> 1
            }
        }
        table[i] = value;
    }
    table
}
```

Listing 5.3: An example method from the `crc` crate calculating a lookup table.

Chapter 6

Conclusion

Sequences of data are ubiquitous in a large amount of real-world software. Adding support for foundational data structures like arrays and slices to the Prusti verifier makes it applicable to a wider range of Rust programs.

We have shown that our design and implementation work on a suite of tests and allow verifying nontrivial array programs such as sort algorithms.

Some aspects of the design are not yet fully implemented. Nonetheless, we have shown an increase in the amount of real-world code supported in Prusti.

The nature of arrays and slices as foundational Rust types means supporting arrays is an important step not just because of its immediate positive results but also as a starting point for further progress.

6.1 Future Work

Apart from implementing missing parts of the design presented in this thesis, there are further extensions that could be added in future projects.

6.1.1 Smarter Value Preservation of Arrays in Loops

Consider the code snippet in Listing 6.1, repeatedly modifying the same array element in a loop.

Note that we only ever write to index `0` of the array. Due to the way loops are encoded (see Section 4.3.4), we lose all information about the array between loop iterations if any of the accesses is writing to any element.

A more fine-grained analysis of which loop indices are actually accessed could help here. As a first step, if only constant indices are accessed, the remaining unchanged indices could be preserved across loop iterations.

```
fn foo() {
  let mut a = [1, 2, 3];
  let mut i = 0;

  while i < 3 {
    a[0] = 0;
  }

  // true, but not verifiable currently.
  assert!(a[1] == 2);
}
```

Listing 6.1: Array element value preservation currently not verifiable.

A good design of such a preservation mechanism would have to be easily explainable and follow user expectations.

6.1.2 Iterators and Automatic Loop Invariants

A common operation for arrays and slices is to iterate over their contents, and especially now that Prusti has array and slice support, iterators are one of the next hurdles to take.

This would add support for Rust syntax like `for i in 0..N` which is desugared into an iterator over that range of indices.

Another step that would be possible then is to use the information the iterator has to automatically add invariants to loops. Currently, a simple loop accessing an array fails to verify without annotations, because the information about loop indices is lost (see the explanation of the loop rewriting in Section 4.3.4). It could be worth investigating whether it is possible to automatically add a loop invariant if the loop bounds are controlled only by an iterator like in this example:

```
for i in 0..array.len() {
  array[i] = i-2;
}
```

6.1.3 Extern Specifications for Array and Slice Methods

We have discussed in Section 4.3.3 that we do not think encoding all array and slice operations as function calls is the best approach. Therefore we encoded the basic array operations directly, as other MIR constructs like field accesses are encoded as well.

There are however a large number of methods defined on the builtin array and slice types, many of them to avoid issues with mutability or lifetimes that are

better solved once in the standard library. These include `split_at_mut`, a method to return two mutable non-overlapping subslices, the `sort` and `swap` methods that both need to reorder elements in-place and various methods to construct different iterators over the slice's elements.

As these are all method calls, it might be the best option for specifying them to extend the existing *#[extern_spec]* mechanism, now that the fundamentals of arrays and slices are available. As an example, we have specified a monomorphized version of `split_at`:

```
#[trusted]
#[requires(0 <= idx && idx < slice.len())]
#[ensures(result.0.len() == idx)]
#[ensures(result.1.len() == slice.len() - idx)]
#[ensures(forall(|i: usize| (0 <= i && i < idx)
    ==> slice[i] == result.0[i]))]
#[ensures(forall(|i: usize, j: usize|
    (idx <= i && i < slice.len() && 0 <= j && j < result.1.len())
    ==> slice[i] == result.1[j]))]
fn split_at(slice: &[i32], idx: usize) -> (&[i32], &[i32]) {
    slice.split_at(idx)
}
```

Bibliography

[1] The Rust contributors. The Rust programming language. https://www.rust-lang.org. Accessed 2021-07-10.

[2] 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.

[3] 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.

[4] The Viper Tutorial. https://viper.ethz.ch/tutorial/. Accessed 2021-06-09.

[5] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[6] Peter W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR 2004 - Concurrency Theory*, pages 49–67, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[7] Felix A Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C Pereira, and Peter Müller. Gobra: Modular specification and verification of go programs (extended version). *arXiv preprint arXiv:2105.13840*, 2021.

[8] Gobra. https://www.pm.inf.ethz.ch/research/gobra.html. Accessed 2021-06-14.

[9] Creusot: deductive verification of Rust code. https://github.com/xldenis/creusot.git. Accessed 2021-06-14.

[10] Xavier Denis. *Deductive program verification for a language with a Rust-like typing discipline.* PhD thesis, Université de Paris, 2020.

[11] Peter Müller. Building deductive program verifiers. 2018.

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):

Specifying and Verifying Sequences and Array Algorithms 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):**
Schilling	Johannes

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)**
Erlangen, 2021-07-15	

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

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.
I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Johannes Schilling)
Erlangen, July 15th, 2021