ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Evaluating and Documenting a Rust Verifier

Bachelor Thesis

Patrick Muntwiler

April 2023

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

Abstract

Prusti [1] is a static verifier for the Rust programming language [2], based on the Viper verification infrastructure [3]. Prusti allows programmers to verify program properties such as correct functionality and absence of crashes.

Program verifiers are extremely effective at proving the absence of bugs. Nonetheless, program verification is still regarded as an obscure and challenging field that suffers from a steep learning curve and lack of learning materials aimed at complete beginners. In this project, we aim to evaluate Prusti by translating a subset of examples and exercises from a recent book on program verification called "Program Proofs" by K. Rustan M. Leino [4].

This translation serves as an evaluation of Prusti, as well as the basis for improving the user-level Prusti documentation, with a focus on beginner level documentation. One part of this was to complete an unfinished guided tour in the Prusti user guide, which should serve as an entry point for programmers wanting to learn to verify Rust programs with Prusti.

During the translation we collected data on features that Prusti is missing or that could be improved, to enable or simplify the verification of specific code patterns. This report also contains a summary of features of both Rust and Prusti, that make verification of certain programs easier compared to other verifiers like Dafny.

Acknowledgments

At this point I would like to thank my thesis supervisor, Aurel Bílý, for all his support throughout the work for this thesis.

I would also like to thank Dominik Odzak for helping me with proofreading this report.

Contents

Chapter 1

Introduction

Static program verifiers are extremely effective at proving the absence of bugs. Nonetheless, program verification is still regarded as an obscure and challenging field that suffers from a steep learning curve and lack of learning materials aimed at complete beginners.

Prusti [1] is a prototype static verifier for Rust, built upon the Viper verification infrastructure [3]. Prusti verifies programs with the help of user-provided specifications in the form of macros imported from the `prusti_contracts` crate, while still keeping the Rust file compilable using a normal Rust compiler. Successful verification means that a program cannot reach any statement that causes a crash, like a `panic!()` macro. Absence of integer over- and underflows is also checked by default.

Rust's strong type system and powerful compile time guarantees such as strict aliasing rules, reduce the amount of annotations needed for verification. Before being able to successfully verify programs with Prusti, a beginner has to learn some verification concepts and new syntax.

The goal of this thesis is to provide an evaluation of the current state of Prusti, and to improve the Prusti documentation, including documentation of all features that Prusti currently supports or that are missing. In addition, we reworked and expanded the guided tour in the user guide [5] for Prusti, to explain important features of program verification like function specifications, quantifiers, pledges, assertions, and more. This tutorial should be able to guide new users of Prusti through the verification process from writing to debugging specifications.

As a starting point, examples and exercises written in Dafny [6] were translated from a recent book on program verification called "Program Proofs" by K. Rustan M. Leino [4]. Dafny is a verification-aware programming language originally developed at Microsoft Research.

While translating the examples from the book we noted down any features that were missing or incomplete in Prusti compared to Dafny, and which existing features could benefit from more or better documentation to make them easier to understand and use. Any bugs encountered were reported on the prusti-dev GitHub repository [7], and we also provided fixes for some of them. We also created some feature requests for missing features that would have been useful during our translations. Most issues will be discussed in this report, and a full list is available in the Appendix. We categorized all the examples into four categories according to whether they could be verified or not, and whether they should pass verification or not (i.e., Correct Verification, Correct Failure, Incompleteness, Unsoundness). The results of the categorization are explained in more detail in the evaluation in Chapter 5.

There were also some differences in verification stemming from language design features in Rust compared to Dafny, independent of Prusti. We will go into more depth on that topic in the evaluation chapter.

As the final part of this thesis, we expanded the Prusti user guide [5] by adding or improving documentation for existing Prusti features, and by providing a tutorial for beginners to program verification. This tutorial is written in a similar style and as an extension to the excellent Rust tutorial "Learn Rust with Entirely Too Many Linked Lists" [8]. The main datastructure we used was the same linked list that was explained in the original tutorial, but with some additional functions and annotations to be verifiable by Prusti.

We also added a list of current Prusti limitations with workarounds if available, as an overview and help to users and developers of Prusti.

Chapter 2

Background

2.1 Prusti

Prusti [1] is a prototype verifier for Rust [2], built upon the Viper verification infrastructure [3].

Prusti provides commands for running static verification on either a standalone Rust file or on an entire Rust project (a "crate"). These commands are designed to work in a similar way to how Rust's official package manager Cargo handles compilation. In a crate directory, this command will run Prusti on the entire crate:

```
$ [path]/cargo-prusti
```

Prusti can be run as a Cargo command, if the path to the `cargo-prusti` executable is on the `PATH` environment variable:

```
$ cargo prusti
```

Prusti also provides an extension for Visual Studio Code, called Prusti Assistant. This extension can call `cargo-prusti` in the background and show the error messages in the source code itself.

Prusti can verify the absence or unreachability of any statements that could cause a program to crash at runtime (called a "panic" in Rust). Sources of panics include manual assertions, out of bounds accesses, explicitly panicking statements (such as `panic` or `unreachable`) or integer overflows.

Programmers have to add specifications to the code to enable the verification of functional safety. These specification take the form of annotations, implemented using Rust's powerful macro system, and imported from the `prusti_contracts` crate. When verifying a program with these annotations, Prusti will compile the code and annotations into the Viper language, which will then be verified by a Viper backend. Potential verification errors will then be translated back to be shown in the original Rust source code.

Rust code with these annotations is still perfectly valid Rust code that can be compiled normally. The macros use conditional compilation to completely remove themselves in normal compilation, so there will be no added runtime cost.

Here is an example of a Rust function that calculates the sum of all values from `1` to `x`. The additional Prusti annotations are used to verify that the function's return value matches the summation formula $\sum_{i=1}^{x} = \frac{x \cdot (x+1)}{2}$ for all allowed inputs of `x`:

```
use prusti_contracts::*;

#[requires(0 <= x)]
#[requires(x * (x + 1) / 2 <= i32::MAX)]
#[ensures(x * (x + 1) / 2 == result)]
fn summation(x: i32) -> i32 {
  let mut i = 1;
  let mut sum = 0;
  while i <= x {
    body_invariant!(sum == (i - 1) * i / 2);
    sum += i;
    i += 1;
  }
  sum
}
```

The summation formula only applies to positive inputs, which is checked by the first precondition `requires(0 <= x)`. Without this precondition, Prusti will detect that the postcondition `ensures(x * (x + 1) / 2 == result)` might not hold. The `result` variable is used to refer to the return value of the function.

The second precondition is required to ensure that the `sum += i` in the loop cannot overflow.

Lastly, the `body_invariant` macro is used by Prusti to verify that `sum` contains the correct value at the end of the `while` loop. It must hold on entry to the loop and has to be upheld by the body of the loop.

Prusti has a user guide [5] for programmers to learn how to use Prusti, which explains these annotations in more detail.

2.2 Viper

Viper [3] is a language and a suite of tools developed as an intermediate target for building verifiers for different programming languages. These frontends

don't need to implement the entire verification pipeline by themselves, they only need to translate their code and annotations to Viper. Viper will then take over verification from that point onward, for example by using symbolic execution.

Prusti inherits some of the design of Viper, such as the general goal of automated verification with the help of contracts from the programmer.

2.3 Dafny

Dafny [6] is a verification-aware programming language originally developed at Microsoft Research. Dafny code can also contain specifications in the form of annotations and can then be verified. Dafny code can also be compiled into a more widespread programming language such as C# or Java.

A large part of this thesis is based on a recent book called "Program Proofs" by K. Rustan M. Leino [4], which uses Dafny for explaining verification concepts. The book also contains exercises for the reader to apply the discussed topics.

Dafny has special syntax useful for verification, such as a distinction between pure and impure functions. Both an (impure) `method` and (pure) `function` in Dafny correspond to `fn` in Rust, distinguished with the `#[pure]` Prusti annotation.

All three verifiers suffer from possible incompleteness due to quantifiers (`forall`, `exists`), which require triggers to be instantiated correctly. An incorrect choice of triggers can cause verification to fail or lead to a worse verification time.

Chapter 3

Approach

This thesis consists of two parts. In the first part, examples and exercises were translated from a recent book on program verification called "Program Proofs" from Dafny to Rust. In the second part, the Prusti documentation and the user guide were improved and expanded, based on the findings from using it during the translation process. The main work was to update and expand an existing beginner tutorial on program verification with Prusti.

3.1 Translating Dafny Examples

The first step was to transcribe the Dafny examples from the book, as these where not yet made available in a digital format at the start of the project. Chapters 1 to 5 were already transcribed for another thesis that also involved translating examples from the book. As part of the other thesis, examples from the book were translated into Gobra [9], an automated verifier for Go programs [10] that is also built on the Viper verification infrastructure.

Both projects translated the first five chapters into their respective target language, since these chapters contained general verification concepts. The remaining chapters 6 to 17 were split up between the Prusti and the Gobra project. For this thesis, the chapters 8, 9, 14, 16 and 17 were selected. The chapter selection was made with the intention of testing a diverse array of Rust language features and Prusti verification features.

3.2 Improving Prusti Documentation

Prusti has documentation in the form of a user guide [5] and a developer guide [11]. The user guide contained an incomplete and partially outdated tutorial for beginners to program verification. This tutorial was meant to be built in the style of and as an extension to the excellent Rust tutorial

"Learn Rust With Entirely Too Many Linked Lists" [8]. This tutorial explains programming concepts for Rust by implementing a linked list.

Several verification features offered by Prusti were not yet utilized in the existing parts of the Prusti tutorial, and most of the later chapters had not been written yet. We updated and expanded this tutorial to include a large subset of the verification features currently supported by Prusti. Some of the remaining Prusti documentation was also incomplete or unclear, so we expanded whatever documentation we discovered to be unhelpful during the translation.

Chapter 4

Implementation

4.1 Translation

We decided to use one Rust crate per chapter of the book. Chapters 8 and 9 are both using the same code for a linked list, so we created an additional crate for just the linked list implementation. Using crates was convenient, as it allowed us to verify an entire chapter at once using the Prusti Assistant extension [12].

Generally, each subchapter was put into its own file, except if it made sense to merge multiple subchapters, for example if they built on each other. Whenever possible, we used imports from other subchapters to prevent code duplication. Exercises were generally put into their own files, except when it made more sense to put them in the same file as the rest of the subchapter. To conform to Rust's general style guidelines, we had to rename functions, files and modules slightly. These changes include removing dots from filenames and changing function and module names to be snake case. Since some of the translations could not be verified with the current capabilities of Prusti, we created separate directories for files that pass or fail verification. By using feature flags in the crate, we were able to quickly change between verifying either the passing, the failing or all the examples in the crate. We checked all the translated code using Rust's official linter called "Clippy" with a command like this: `cargo clippy --all-features -- -D warnings`.

We marked any spot in the code with a comment like this "FUTURE: [feature_description]" whenever we encountered a missing, incomplete or broken feature of Prusti. We collected data on all of these features and will present them in the evaluation in Chapter 5.

Each example and exercise from the book was categorized based on whether Prusti could verify it or not, and whether it should be verifiable or not. This lead to the four main categories "Correct Verification", "Correct Failure",

"Incompleteness" and "Unsoundness".

"Correct Verification" was further split into subcategories, based on how much annotation effort was needed for correct verification relative to Dafny. The "Incompleteness" subcategories were split based on the reason for the incompleteness, which was either a Prusti bug, missing features or in one case a limitation of Rust itself.

Any bugs we encountered were reported on the prusti-dev GitHub repository [7] as an issue, and we created feature request for any missing features.

We implemented one bug fix and one feature suggestion ourselves, and they have since been merged into the Prusti codebase. We also provided some pull requests for general code improvements in terms of readability, performance or memory consumption. A list of pull requests created for or related to this thesis can be found in the appendix.

One important decision early on was to disable overflow checking by Prusti for the translated code. The reason for this was that the book used Dafny's `Int` and `Nat` types, which are unbounded, mathematical integers and natural numbers respectively. These are translated to use "BigInteger" when compiled to C# to not have any overflows at runtime. While there are some Rust crates that provide BigInteger functionality, we decided to use standard integers for our translations. To verify programs that utilize unbounded integers, a crate providing them like `num-bigint` [13] would need to be annotated first. We decided to use Rust's `u64` and `i64` types for integer values, and `usize` for indices, but without overflow checking.

To disable overflow checks by Prusti, all chapter crates contain a "Prusti.toml" file with the configuration flag "check_overflows = false".

Dafny functions taking array references were translated to use either mutable or immutable slice references in Rust.

4.2 Documentation

During the translation, we took note of features that were useful for verification, but that were not or incompletely documented. This documentation was then added or improved by us.

We then used the knowledge gained during the translation to create a tutorial to teach new users of Prusti how to verify their Rust code with Prusti. Since it is an extension of the "Learn Rust With Entirely Too Many Linked Lists" tutorial, we did not explain Rust in detail, but instead link to the relevant parts of the original tutorial for readers that are not yet familiar with Rust itself.

A link to the corresponding documentation was added for every Prusti features used in the tutorial. If the linked documentation was missing or not beginner friendly, we improved it as well.

The tutorial now also has a section on running Prusti verification on a Rust project, such as adding the required dependencies and running the Prusti verifier itself and setting configuration flags.

The developer and user guide for Prusti both use mdBook [14]. MdBook enables the creation of documentation containing (Rust) code, which can then be automatically compiled into a website. MdBook has functionality to automatically test any code samples in the documentation for correct compilation and if any assertions fail when executed. We set up and fixed the Prusti documentation to be able to run "mdBook test", and set it up to run automatically on each pull request to the main branch on GitHub. This will check that all Rust code in the documentation compiles and runs without crashing, unless manually marked as `ignore` or `no_run`.

We also extracted any big code examples from the documentation and placed them in Prusti's testing directory. The testing directory contains different subdirectories for code that is expected to pass or fail verification with certain settings. This ensures that any examples in the documentation are not just checked for normal compilation, but also for correct verification. By running these tests on each pull request, any changes to Prusti are checked against the documentation, so that it does not get outdated in the future.

Lastly, we created a "Capabilities" chapter in the user guide, which should serve as a list of capabilities and limitations of Prusti, both for users and for the Prusti developers. Where possible, we tried to provide workarounds for known limitations.

Chapter 5

Evaluation

In this chapter we will first present our findings from translating the book examples, then afterwards the findings made when reworking the documentation and tutorial. Note that all Prusti code examples in this chapter require having the crate `prusti-contracts` as a dependency and imported to each file by adding the line `use prusti_contracts::*;` at the beginning of each file. We omit this line here for brevity and readability.

The appendix contains a list of the GitHub issues and pull requests that were opened over the course of this thesis. The evaluation contains links to the relevant issues where appropriate.

5.1 Results from the translations

Note that the titles used in the following subchapters are not necessarily the titles of the chapters from the book, but what was relevant for the translations.

5.1.1 Chapters 1 and 2: General verification concepts

The first two chapters were mainly about the basics of verification, i.e., pre- and postconditions, assertions, pure functions etc. Except for one bug, where the overflow check for the expression `-x` was not disabled even with the configuration flag set, every example and exercise in these two chapters was able to be verified in Prusti.

There were three main missing features found in these two chapters: ghost code, showing multiple errors per function, and mutable function arguments.

Ghost code

Ghost code is code that is only used in verification, and will not appear in the final executable when compiled normally. At the same time as this thesis, an-

other thesis [15] was underway, for which additional features for Prusti were implemented, including support for ghost code. The ghost code functionality is now available using the configuration flag `unsafe_core_proof = true`, which enables a different, experimental encoding scheme used by Prusti in the background. As the name implies, this new encoding scheme is meant to also be able to verify `unsafe` Rust code, but it is still very incomplete. We only used the standard encoding for our translations, so we translated any ghost code in these chapters into normal code. This has the limitation that we could not use Prusti specific syntax such as quantifiers in our ghost code.

A workaround is the `prusti_assert` macro available in both encoding modes, which can use the full Prusti specification syntax. The `prusti_assert` macro was introduced in the same thesis as the ghost code, but it has since been backported to the standard encoding. Syntactically, `prusti_assert` is the Prusti equivalent of the standard Rust `assert` macro. We noticed the lack of an equivalent for Rust's `assert_eq` and `assert_ne` macros, so we made a feature request for adding the macros `prusti_assert_eq` and `prusti_assert_ne` (see issue #1283). These two macros have since been added and are available at the time this report was written.

Showing multiple errors per function

A second, less important missing feature is showing multiple errors per function, which is available and enabled by default in Dafny. With this enabled, verification does not stop when a failing assertion is encountered in a function. Instead, verification continues with the assumption that the failing condition does hold. This enables slightly easier or faster debugging of specifications in some cases. For example, when attempting to prove a lemma, getting multiple errors can help. Every step in the lemma is checked on each verification run, not just the part up to first failing step.

Showing multiple errors per function is a supported feature of the Viper backend used by Prusti, but a bug currently prevents enabling this setting. The corresponding configuration flag is enabled with `extra_verifier_args = ["--numberOfErrorsToReport=0"]`, but due to the bug, only one error per function will be shown (see issue #1213).

Mutable function arguments

Lastly, mutable function arguments are not supported by Prusti. This does not include mutable references, which are supported. As an example, this swap function cannot currently be verified by Prusti:

```
#[ensures(result === old((y, x)))]
fn swap(mut x: i64, mut y: i64) -> (i64, i64) {
  x = y - x;
  y = y - x;
  x = y + x;
  (x, y)
}
```

The workaround for the moment is to not mutate the function parameters directly, but to instead assign them to a local mutable variable as the first step in the function:

```
#[ensures(result === old((y, x)))]
fn swap_workaround(x: i64, y: i64) -> (i64, i64) {
  let mut x = x;
  let mut y = y;
  x = y - x; y = y - x; x = y + x;
  (x, y)
}
```

Rust allows redefinition of variables with the same name, so nothing has to be changed in the remaining parts of the function. Due to Rust's move semantics, this change does not affect the behavior of the function.

In this specific case, the easiest solution is to return a tuple with the two values swapped:

```
#[ensures(result === old((y, x)))]
fn swap_simple(x: i64, y: i64) -> (i64, i64) {
  (y, x)
}
```

Misleading warning on generated functions

There also was a rather small bug, for which we submitted a fix ourselves. Any function starting or ending in an underscore `'_'` with any of the Prusti annotations attached, caused a `non_snake_case` warning that could not easily be silenced by a user (see issue #1202). This was caused by Prusti generating a function internally for each of the pre- and postconditions, where during name mangling, an underscore was added to the function name, causing it to not be valid snake_case anymore. We fixed this by automatically adding the `#[allow(non_snake_case)]` annotation to each of these internal functions (PR #1269). Silencing this warning will not cause any problems, since users are not meant to see these internal functions anyway.

5.1.2 Chapter 3: Termination checks

Chapter 3 introduced the concept of termination checks for recursive functions. Dafny has termination checks enabled by default and automatically selects a `decreases` clause for each recursive function. The expression in the `decreases` clause must decrease in any recursive function call compared to the calling function. A manual `decreases` clause is sometimes needed, if the automatically chosen one is not enough to prove termination.

Prusti does not currently support termination checks, meaning it only proves partial correctness. Termination checking for pure functions was added in the same thesis as ghost code, but this functionality is also only available at the moment with `unsafe_code_proof = true`.

Due to missing termination checks, Prusti verified the examples that should fail due to infinite recursion, causing unsoundness. Luckily this was the only source of unsoundness that we detected during our evaluation.

Note that for some of the examples with infinite recursion, if not explicitly suppressed using the `#[allow(...)]` annotation, the Rust compiler will warn about unconditional recursion. Running Rust's official linter called "Clippy" on this code will also give the hint that the variable `x` is only ever used in the recursion, so in these cases it should be obvious that the code won't terminate:

```
#[allow(unconditional_recursion)] // stop rustc warning
#[allow(clippy::only_used_in_recursion)] // stop clippy warning

#[ensures(result == 2 * x)]
fn bad_double(x: i64) -> i64 {
  let y = bad_double(x - 1);
  y + 2
}
```

More subtle infinite recursion would still go undetected even by using Clippy, so Prusti should support termination checks for both pure and impure functions, and also for loops at some point.

By using the fact that termination is not checked, arbitrary code can be verified. Termination and explicitly trusted functions were the only sources of potential unsoundness we found in Prusti.

5.1.3 Chapter 4: Recursive types and allocations in pure functions

Type definitions

Chapter 4 introduces Dafny's `enum` type. Like in Rust, Dafny's `enums` are algebraic data types, meaning that they can have data associated with each

`enum` variant. The translation of the types themselves was mostly one-to-one, with the exception of Rust requiring an indirection through a `Box` in the case of recursive types like trees or linked lists.

In Dafny, the memory structure of the `Node` variant is handled transparently, as seen in this Blue-Yellow-Tree:

```
datatype BYTree = BlueLeaf | YellowLeaf | Node(BYTree, BYTree)
```

In Rust, we cannot just put two `BYTrees` into a Node directly, because this would generate an infinitely sized type. Instead, we add an indirection using a `Box` containing a `BYTree` (i.e., a unique pointer to a `BYTree`):

```
enum BYTree {
  BlueLeaf, YellowLeaf,
  Node(Box<BYTree>, Box<BYTree>),
}
```

Pure functions

Functions that take the tree by immutable reference and return some integer were easily translated and could also be marked as being pure functions:

```
impl BYTree {
  #[pure]
  fn blue_count(&self) -> u64 {
    use BYTree::*;
    match self {
      BlueLeaf => 1, YellowLeaf => 0,
      Node {l, r} => l.blue_count() + r.blue_count(),
    }
  }
}
```

Note that this function does terminate, but Prusti does not currently check this.

These pure functions can then also be used in specifications.

Note that in the second match arm `Node {l, r}`, both `l` and `r` have the type `&Box<BYTree>`. `Box` implements the `std::ops::Deref` trait, which enables `&Box<BYTree>` to be automatically dereferenced into `&BYTree`, and thus `l.blue_count()` can be called on the subtrees directly. Prusti also handles these automatic dereferentiations, so no additional annotations or workarounds are needed here.

Allocation in pure functions

The rest of the chapter showed one of the bigger limitations of Prusti. In Dafny, new `BYTree` enums can be created in pure functions, as is done in the `Node` case in this example:

```
function ReverseColors(t: BYTree): BYTree {
  match t
  case BlueLeaf => YellowLeaf
  case YellowLeaf => BlueLeaf
  case Node(left, right) =>
    Node(ReverseColors(left), ReverseColors(right))
}
```

Prusti currently only allows allocation in pure functions for types that implement the `Copy` trait. The `Copy` trait denotes types that can be cloned by making a bitwise copy, without running any additional code. This includes primitive types such as `char`, `i32`, `u128`, and types that are built from other `Copy` types. The type `BYTree` cannot implement the `Copy` trait, because `Box` is a unique pointer, which would not be unique anymore if it was copied bitwise. Creating or cloning a boxed value requires allocating memory on the heap, which is currently considered a side-effect and thus not allowed in pure functions.

The same `Copy` trait limitation also applies to parameters and return values of pure functions, so the `Box::new()` and the `reverse_colors` function are not supported by Prusti:

```
#[pure]
fn reverse_colors(t: &BYTree) -> BYTree {
  use BYTree::*;
  match t {
    BlueLeaf => BlueLeaf, YellowLeaf => YellowLeaf,
    Node(left, right) => Node(
      Box::new(reverse_colors(left)),
      Box::new(reverse_colors(right)),
    ),
  }
}
```

The input parameter `t: &BYTree` does not cause a problem here, because an immutable reference to any type is `Copy`.

Structures containing references

Code using any structures containing references are another limitation of Prusti that we encountered in this chapter. This limitation affects any enums,

tuples or structs that contain references, such as `Option<&T>`, `(&T, &T)` or `struct{ a: &T }`.

One example of where this limitation can be encountered is in the following function, which checks two inputs of type `&BYTree` for equality:

```
#[pure]
fn eq(a: &BYTree, b: &BYTree) -> bool {
  use BYTree::*;
  match (a, b) {
    (BlueLeaf, BlueLeaf) | (YellowLeaf, YellowLeaf) => true,
    (Node(a_left, a_right), Node(b_left, b_right)) => {
      eq(a_left, b_left) && eq(a_right, b_right)
    }
    _ => false,
  }
}
```

The tuple `(a, b)` used to match on both trees simultaneously, has the type `(&BYTree, &BYTree)`, which is not supported by Prusti. This limitation can sometimes be worked around, in this case by matching first only on the tree `a`, then matching on just the tree `b` in each of the outer match arms. This workaround enables verifying this example, but will make the code more verbose and harder to read. In some cases, this limitation cannot be worked around, such as when trying to destructure a value of type `Option<&T>`, like in this toy example, where the function optionally gets a reference to some value:

```
fn option_reference(opt_value: Option<&mut u64>) {
  if let Some(value) = opt_value {
    *value = 5;
    // ...
  }
}
```

The verification fails with the following error message:

```
[Prusti: unsupported feature]
access to reference-typed fields is not supported.
```

Types like this often appear as the return type for lookup functions into some data structure, where a value might not always get returned. An example for this is a `peek` function on a linked list, that will return `Some(&head)` when the list is non-empty, and `None` otherwise. Note that this limitation affects both `&T` and `&mut T`.

Shallow borrows

Shallow borrows are not supported by Prusti. They appear when matching on a reference, in a match arm with a guard condition:

```
struct Wrapper {x: i32}

fn test(t: &Wrapper) {
  match t { // <== Error
    Wrapper{ x } if x % 2 == 0 => {},
    Wrapper{ x } => {}
  }
}
```

```
[Prusti: unsupported feature] unsupported creation of shallow borrows
  (implicitly created when lowering matches)
```

This limitation was never an issue in any of the translation, since we were able to work around it by moving the condition into the match arm. This workaround might not be possible in every case, but we did not encounter such cases.

5.1.4 Chapter 5: Lemmas

Just like `function` and `method`, Dafny also has a specific identifier `lemma` for denoting lemmas, often used to prove some non-trivial property of a function or method to then be used in specifications later.

Prusti does not have a dedicated syntax for lemmas at the moment, so we used normal `#[pure]` functions with a "lemma_" prefix in their name. A big difference here however, is that the Dafny lemmas are ghost code, and are checked to not be used in normal code, but since we just used normal pure functions in Prusti, lemmas can be used anywhere. Prusti has the `predicate` macro that can be used to create a ghost predicate, but these currently have the limitation that they cannot have pre- or postconditions (more on predicates in a later section).

Automatic induction proofs

Here we have a function `more` where the output is always bigger than the input, and a corresponding lemma to prove that property:

```
#[pure]
fn more(x: i64) -> i64 {
  if x <= 0 { 1 } else { more(x - 2) + 3 }
}
```

```
#[pure]
#[ensures(x < more(x))]
fn lemma_increasing_automatic(x: i64) {}

#[pure]
#[ensures(x < more(x))]
fn lemma_increasing_manual(x: i64) {
  if x <= 0 {
    // Base Case
  } else {
    lemma_increasing(x - 2)
  }
}
```

The first lemma is identical to the one from the book, but Dafny can prove the lemma postcondition automatically by using an induction proof. Prusti currently requires some manual help as seen in the second lemma. Automatic induction proofs were not used much in the book, as the goal was often to be able to do the proofs by hand. The automatic induction was often disabled in the book by using `lemma {:induction false}`. We suspect that induction proofs would be used more often when verifying a real codebase, when the goal is not to teach verification. Automatic induction proofs by Prusti would have made verifying some translations easier.

Calc blocks

Another feature in Dafny is the `calc`-block, which allows writing multiple proof steps together in a concise way. Each step in the `calc`-block consists of an expression, then an operator optionally with some assertions or lemma applications, and then the next expression.

Here we have the same lemma from before, but with every proof step written explicitly:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{ if x > 0 {
    calc {
      More(x);
      ==
      More(x - 2) + 3;
      > { Increasing(x - 2); }
      x - 2 + 3;
      >
      x;
} } }
```

These `calc`-blocks can also have a default operator, which can then be omitted between the expressions in each step.

In Prusti, this feature can be emulated by using multiple `prusti_assert` statements with the corresponding expressions, and adding the assertions or lemma applications when needed:

```
#[pure]
#[ensures(x < more(x))]
fn lemma_increasing(x: i64) {
  if x > 0 {
    prusti_assert!(more(x) == more(x - 2) + 3);
    lemma_increasing(x - 2);
    prusti_assert!(more(x - 2) + 3 > x + 1);
    prusti_assert!(x + 1 > x);
} }
```

5.1.5 Chapter 8: Sorting Linked Lists

In this chapter, the goal was to write and verify different sorting algorithms that work on linked lists.

The types for Dafny and Rust were again very similar:

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>)
```

Like for the `BYTree`, the main difference in Rust is the explicit indirection through a box:

```
enum List<T> {
  Nil,
  Cons(T, Box<List<T>>)
}
```

Both are now generic over type `T`, so they can be instantiated with values of any type.

Predicates

The book defined a predicate `Ordered`, which checks if each consecutive pair of values in the list is correctly ordered:

```
predicate Ordered(xs: LL.List<int>) {
  match xs
  case Nil => true
  case Cons(x, Nil) => true
  case Cons(x, Cons(y, _)) => x <= y && Ordered(xs.tail)
}
```

Prusti has dedicated syntax for writing predicates, by wrapping a function in a `predicate` macro. The function then gets turned into a ghost function that can be used in specifications, but not in normal code. At the time of the translations, the function in the predicate macros always had to return a `bool`, but this restriction has since been loosened to support any return type, as long as it implements `Copy`.

One limitations of these predicates is that they currently cannot have any pre- or postconditions. If this restriction gets lifted at some point, it should be possible to use the `predicate` macro to turn the lemmas we wrote into ghost functions as well. If a dedicated lemma syntax is introduced in the future, it might be possible to desugar it to a `predicate` macro with annotations.

This is our translated `ordered` predicate:

```
predicate! { fn ordered(xs: &List<i64>) -> bool {
  match xs {
    List::Nil => true,
    List::Cons(_, box List::Nil) => true,
    List::Cons(x, tail @ box List::Cons(y, _)) =>
      { *x <= *y && ordered(tail) }
  }
} }
```

Here we make use of Rust's (still unstable) box syntax. To use the box syntax, `#![feature(box_syntax)]` must be enabled and the nightly compiler has to be used instead of the stable version. Without the box syntax, this example would be a lot more verbose. Even though the box syntax is a nightly only feature for now, Prusti did not seem to have any problems using it.

In the third match arm we use the `tail @` syntax to bind the tail of the list to a variable, so that we can recursively call the `ordered` predicate. We had to dereference both the `x` and the `y` variables before the comparison, due to the missing external specification for the `std::cmp::PartialOrd::le` function, which corresponds to `<=`. We attempted to write the external specification ourselves, but we could not get it to work. External specification will be discussed in more detail in this chapter.

The macro used for predicates cannot be an annotation macro like for example the `#[pure]` annotation. The reason is that predicates allow the full Prusti specification syntax, which contains parts that are not valid Rust syntax (such as `===`). When using an annotation, `rustc` still attempts to parse the function before passing it to the macro code, which will fail due to the additional syntax. The `predicate` macro used now makes using the additional syntax possible, by parsing the function by itself before `rustc` does.

Viper encoding of pure functions

We ran into a known limitation when trying to use the `ordered` predicate.

```
fn test_ordered() {
  let initial_list = List::Cons(15, Box::new(List::Nil));
  prusti_assert!(ordered(&initial));

  let list = List::Cons(10, Box::new(initial_list));
  prusti_assert!(ordered(&list));
}
```

In this test, the second assertion only passes if the first assertion is also present. The Viper backend seems to only unfold the `ordered` predicate once, in this case checking `10 <= 15`, but the recursive call is not checked. Viper therefore has no information on `ordered(tail)`, which in turn causes the condition `10 <= 15 && ordered(tail)` to fail.

The Prusti developers are planning to rework how Prusti encodes pure functions into Viper, which should allow for more flexibility, for example in choosing the unfolding depth.

This rework might also fix another issue related to pure function, which is that Prusti currently does not support mutually recursive pure functions, or chaining pure functions that have references in the return value. The lemma `all_ordered` aims to prove that if each consecutive pair of elements in a list is ordered, then the entire list is sorted:

```
#[pure]
#[requires(ordered(xs) && i <= j && j < xs.len())]
#[ensures(*xs.at(i) <= *xs.at(j))]
fn lemma_all_ordered(xs: &List<i64>, i: usize, j: usize) {
  if i != 0 {
    lemma_all_ordered(xs.tail_ref(), i - 1, j - 1);
  } else if i == j {
  } else {
    lemma_all_ordered(xs.tail_ref(), 0, j - 1);
  }
}
```

This lemma is almost identical to the Dafny version, but Prusti cannot currently verify it. The main problem here appears to be the function `tail_ref`, as it returns a reference.

`tail_ref` is a function for getting a reference to the second node of the list, given that the list is non-empty. In Dafny, specific parts of any enum variant can be named; in this case the second part of the `Cons` variant is called `tail`. By using `xs.tail`, Dafny will check if the enum variant is guaranteed to be

`Cons` and if so will directly return a reference to that named variable. We wrote the `tail_ref` function ourselves to have similar functionality in Prusti:

```
impl<T> List<T> {
  #[pure]
  #[requires(!matches!(self, List::Nil))]
  #[ensures(result.len() + 1 == self.len())]
  fn tail_ref(&self) -> &Self {
    match self {
      List::Cons(_, tail_ref) => tail_ref,
      List::Nil => unreachable!(),
    }
  }
}
```

The `tail_ref` and all the other functions in this chapter were written with a generic type parameter `T` and the functionality for lists was implemented as associated functions (i.e., in an `impl` block). Prusti was able to handle these generic and associated functions, without any differences to verifying standalone or non-generic functions.

Prusti also correctly identifies that the `unreachable` macro in the `tail_ref` function is actually unreachable, given that the precondition holds. Prusti can also prove that the length of the tail is one element less than the length of the entire list. If overflow checks are not disabled, Prusti correctly identifies `1 + tail.len()` as potentially causing an overflow.

5.1.6 Chapter 9: Module system and contract visibility

Contract visibility

In a Dafny module, any function can either be private to the module, be completely revealed including its specifications (keyword `reveal`), or have only its specifications exported (keyword `provides`). Other modules importing that module can then do different things with a function depending on this export set.

Prusti does not implement this functionality itself, but can inherit most of it from Rust's module system. The two available options for any type or function is being marked as either `pub`, or not. A `pub` function will have its specifications provided when imported, and `#[pure] pub` functions also reveal their implementation. The only missing combination is providing the specifications, but not the body of a `#[pure]` function.

Differing inlining decisions

An interesting difference between the two verifiers was found in this function, which doubles the input value using only additions:

```
function Double(x: int): nat
  requires x >= 0
{ if x == 0 then 0 else 2 + Double(x - 1) }

method Test() {
  assert Double(32) == 2 * 32;
  assert Double(33) == 2 * 33;
}
```

Dafny appears to inline the `Double` function 32 times. Without the first assertion in `Test`, the second assertion fails, as it would require inlining `Double` 33 times. However, if the first assertion is present, Dafny will continue inlining from `Double(32)`, which enables verifying `Double(33)` and larger, up to a higher limit.
In contrast, with the current encoding of pure functions, Prusti seems to only unfold the `double` function once, just like the `ordered` predicate from the previous chapter.

```
#[pure]
#[requires(x >= 0)]
// #[ensures(result == 2 * x)]
fn double(x: i64) -> i64 {
  if x == 0 { 0 } else { 2 + double(x - 1) }
}

fn test() {
  assert!(double(0) == 0); // Required for the next assertions
  assert!(double(1) == 2); // Required for the next assertion
  assert!(double(2) == 4);
}
```

In this case, a simple fix is to add the postcondition `result == 2 * x` to `double`, which enables verifying assertions for any value of `x`.

5.1.7 Chapter 14: Array operations

Aliasing rules

The next example is not from the book, but serves to illustrate the differing aliasing rules between the two languages and how they influence the required annotations. The example function takes three slice references `a`, `b` and `c`, the first two of them mutable, and has two postconditions:

- The first element of `a` and `b` will be different.
- The slice/array pointed to by the immutable reference `c` will be unchanged after the method returns.

In both verifiers, we need to ensure that the slices/arrays have appropriate lengths using a precondition. Prusti does not require any additional annotations to verify the postconditions:

```
#[requires(a.len() != 0 && b.len() != 0)]
#[ensures(a[0] != b[0])]
#[ensures(forall(|i: usize| i < c.len()
  ==> old(c[i]) == c[i]))]
fn aliasing(a: &mut [i64], b: &mut [i64], c: &[i64]) {
  if a[0] == b[0] {
    a[0] = 5;
    b[0] = 10;
} }
```

The reduced annotation overhead is possible due to Rust's strong compile-time guarantees on aliasing. The property Prusti uses here, is that mutable references are not allowed to be aliased in Rust, i.e., if a mutable reference to some data exists, there cannot be any other references to this data. This has the following effects:

- Changes to either `a` or `b` will not have any effects on any other mutable or immutable variable.
- No other part of the program can have a mutable reference to `c` as long as this function holds a reference to it, so `c` will not change during the entire duration of the function call (even in a multithreaded program).

Dafny requires additional annotations to get the same result. First off, mutability is denoted in a separate `modifies` clause, rather than in the code itself. Secondly, `a`, `b` and `c` might alias, so we have to manually prohibit this:

```
method aliasing(a: array<int>, b: array<int>, c: array<int>)
  requires a.Length != 0 && b.Length != 0
  requires a != b && a != c && b != c // Non-aliasing condition
  ensures a[0] != b[0]
  ensures forall i :: 0 <= i < c.Length ==> old(c[i]) == c[i]
  modifies a, b // modifies clause
{
  if a[0] == b[0] {
    a[0] := 5;
    b[0] := 10;
} }
```

Without the non-aliasing precondition, writing to `b[0]` might also write to `a[0]`, which would cause the first postcondition to fail. Additionally, any writes to `a` or `b` could influence `c`, breaking the second postcondition.

Checking the references for equality is enough to show non-aliasing, since Dafny does not allow getting a reference into another array, which is possible with Rust's slices.

Note that both the array and slice references are guaranteed to be non-`null`, so this does not have to be checked manually. Dafny allows taking references that might be null by using `array?<int>`, but Rust's references can never be `null`.

The Rust compiler enforces the annotation of mutability in function signatures and variable declarations, and warns about unnecessary `mut` annotations. This combined with the strong aliasing guarantees means that any Rust codebase to be verified will most likely have full mutability and aliasing information present already, which should reduce the additional effort required for verification.

Effects of ownership models on verification

In addition to the `modifies` clause for mutability, Dafny requires adding a `read` clause to be able to read from a reference.

Dafny methods returning a reference to a newly allocated object also require adding them to a `fresh()` postcondition:

```
method NewArray() returns (a: array<int>)
  ensures fresh(a) && a.Length == 20
{
  a := new int[20];
  var b := new int[30];
  a[6] := 216;
  b[7] := 343;
}

method Caller() {
  var a := NewArray();
  a[8] := 512;
}
```

Without the `fresh(a)` postcondition on `NewArray`, the method `Caller` is not allowed to modify the returned array, because then the returned array reference might point to an array owned by someone else.

Rust has these conditions built into its ownership model and type system. A function returning some type by value makes the caller the new owner of the

returned value. Returning a reference from a function means that the caller does not own the value, and is only allowed to mutate through the reference if it is mutable:

```
fn make_owner() -> Vec<i64> { /* */ }
fn allow_reading() -> &[i64] { /* */ }
fn allow_modifications() -> &mut [i64] { /* */ }

fn caller() {
  let mut vec: Vec<i64> = make_owner();
  // `vec` is now fully owned by the current function

  let slice: &[i64] = allow_reading();
  // can read from `slice`', but not write to it

  let slice_mut: &mut [i64] = allow_modifications();
  // can read from and write to `slice_mut`
}
// (Type annotations in function `caller` are not required)
```

Like aliasing in the previous section, Rust's ownership model and strong type system are very useful for verification in reducing annotation overhead.

Since Prusti is built on top of the standard Rust compiler, these compile-time guarantees are directly checked by `rustc`, before Prusti starts the actual verification. This means that Prusti does not need to check these guarantees and can instead use them in the verification process without requiring manual annotations for them. On the other hand, this means that bugs in the Rust compiler could influence verification with Prusti, since Prusti does not check all properties of a program again.

Loops and quantifiers

Chapter 14 of the book also made heavy use of loops. The Dafny code used `while` loops and we did the same in our translations. Using `for` loops with iterators would be the more idiomatic way for writing Rust code, but iterators are currently not supported by Prusti. We will talk more about iterators in this section, when we look at constructs from the Rust standard library in general. Using `while` loops did not matter too much in this case, since that way the code is closer to the original Dafny implementations.

Here is one example showing a few differences between `while` loops in Dafny and Prusti:

```
method InitArray<T>(a: array<T>, d: T)
  modifies a
  ensures forall i :: 0 <= i < a.Length ==> a[i] == d
{
  var n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
    invariant forall i :: 0 <= i < n ==> a[i] == d
  {
    a[n] := d;
    n := n + 1;
  }
}
```

Note that the type `T` is assumed to be copyable by Dafny. In Rust, we limited our implementation to only take `Copy` types:

```
#[ensures(forall(|i: usize| i < a.len() ==> a[i] === d))]
fn init_slice<T: Copy>(a: &mut [T], d: T) {
  let mut i = 0;
  while i < a.len() {
    // body_invariant!(i < a.len());
    body_invariant!(forall(|j: usize| j < i ==> a[j] === d));
    a[i] = d;
    i += 1;
  }
}
```

One difference here is that Prusti uses a body invariant, which is checked at the place it is written. This means that the loop condition is known to hold at the start of the body, so the first `body_invariant` checking the range of `i` can be omitted.

Dafny uses classical loop invariants, and in this example a range annotation is required at least for the upper bound of `i`. The loop invariant in Dafny must hold at the beginning and end of the loop, so the final value of `i == a.Length` must be included, unlike in Prusti, where the final value does not need to be part of the range.

For comparison of values with type `T` we use Prusti's snapshot equality operator `===`, which can be applied even for types that do not implement Rust's `PartialEq` trait that is required for comparisons using the standard equality operator `==`.

In this example we used an invariant with a quantifier over the lower part of the array, to denote that it has already been correctly initialized. Except for differing syntax, the two verifiers handle quantifiers quite similarly. In the

Dafny example, we do have to add the lower and upper bound for `i` in the quantifier, but since the quantifier in Prusti contains type information for the index `i`, we can omit the lower bound.

At the start of this thesis, this omission was not possible with overflow checks turned off, since in that case Prusti encoded all signed and unsigned integers as unbounded mathematical integers. We created a request to change the that behavior (see issue #1215). Instead of being unbounded, unsigned integers should still be encoded and checked to be non-negative. We submitted a fix ourselves (see PR #1281), which changes the default value of the config flag `encode_unsigned_num_constraint` to `true`. This new default behavior is more in line with what a programmer expects for unsigned integers, even when not checking overflows.

Triggering quantifiers

Quantifiers (`forall`, `exists`) require trigger expressions to be correctly instantiated during verification. An incorrect choice of triggers may lead to a failing verification or increased verification times.

These triggers are currently not chosen automatically by Prusti itself, but by the Viper backend. These automatically chosen triggers are not communicated back to the programmer, so a slow verification or verification failures caused by incorrectly chosen triggers can be hard to debug.

The ability to show the quantifiers chosen by Viper is getting implemented for Prusti Assistant as part of pull request #216, among other improvements to Prusti Assistant.

Triggers can be supplied manually in the quantifier annotation, like in this example with a quantifier over `slice` with type `&mut[T]`:

```
forall(|i: usize| i < slice.len()
  ==> slice[i] == old(slice[i - 1]), triggers=[(slice[i - 1])])
```

The current trigger selection is limited to only function calls, quantified variables (here: `i`) or slice accesses. This should get expanded at some point, and the syntax reworked.

5.1.8 Chapter 16: Objects, type invariants and the Rust standard library

Chapter 16 of the book discusses objects and classes with validity requirements. These were translated into `structs` with `impl` blocks in Rust.

In this and the previous chapters, we encountered many types and functions from the Rust standard library that would have been useful during the translations. Due to functions being handled opaquely by Prusti, these

standard library functions can often not be used in verified code without extra effort. This chapter will contain a summary of these types.

Standard library annotations

In the absence of annotations, Prusti treats a function as impure, with pre- and postcondition `true`. Since currently no parts of the Rust standard library contain Prusti annotations, all of them will be treated opaquely.

To still use functionality of the standard library or any other external code, Prusti provides a way of writing specifications for foreign code. This is done by using the `#[extern_spec]` annotation provided by the crate `prusti_contracts`:

```
#[extern_spec(std::mem)]
#[ensures(snap(dest) === src)]
#[ensures(result === old(snap(dest)))]
fn replace<T>(dest: &mut T, src: T) -> T;
```

This creates a trusted specification for the external code, enabling it to be used in verification. Note that `#[extern_spec]` are implicitly trusted, i.e., not actually checked for correctness.

The syntax used here is the newer way of writing external specification, which was made available at some point during this thesis. The older syntax still works, but is more verbose.

At the same time as this thesis, a Master's thesis was ongoing, which worked on writing specifications for parts of the Rust standard library (See pull request #1249). The goal was to provide a crate called `prusti-std`, which can be imported into a project and will provide external specifications for the standard library. That Master's thesis was complete by the time of writing this report, but the changes had not yet been fully merged into Prusti. Many of the specifications that were missing in the translations should be available once the `prusti-std` crate is available.

We categorized the standard library functionality in two ways in this project.

- *Enhancement functionality* are functions, which would have made some of the translations shorter, more readable or closer to being idiomatic Rust. Not having them available often made the code more verbose, but it did not really hinder verification. Examples:
  - `std::cmp::min` and `max` functions for integers.
  - `is_empty`, which is more idiomatic than `len() == 0`
- *Required functionality* contains functions which were required for some verifications. For some of these functions, the specifications were

already completed in PR #1249, so we could use them. Examples of required functionality:

- `std::mem::replace` and `std::mem::swap`, which are required for some operations in Rust. One example is the sentinel pattern, which is used to temporarily take ownership of a value behind a mutable reference, by using `mem::replace` to take the value and store a sentinel value in its place. This pattern is not possible without these functions, because taking a value normally would leave uninitialized memory behind, which is not allowed in Rust.
- `std::option::Option::take`, which is also used for the sentinel pattern, but specifically for `Options`, by returning the value in `Option::Some(value)` and storing `None` in its place.
- `slice::swap`, which enables swapping elements in a slice containing non-`Copy` values.
- `std::collections::HashMap`, which was required in some of the examples in the book.

A list of all the functions and types that would have been useful in any of the translations or the documentation can be found in the appendix.

Some missing features of Prusti depend on other external specifications and on having a way to provide specifications for use in a program. We will list the main features dependent on this here:

Iterators are one Rust feature we missed a lot, since they are required to use `for` loops. Iterators desugar to multiple functions during compilation such as `std::iter::Iterator::next`, all of which require annotations to be used in verification. `next` and many other functions in Rust have the return type `std::option::Option<T>`, which is one of the types that is part of PR #1249.

Iterators on references also require support for structures containing references, such as `Option<&T>`.

Closures are not supported at the moment. Supporting them also depends on having a way of providing external specifications to the user.

Strings and string slices are also not supported at the moment. If the strings are not part of the code to be analyzed, this can sometimes be worked around, for example by using `println` only in `#[trusted]` functions. Verifying code that actively works with strings, like the tokenizer in chapter 16.1, is not possible in Prusti at the moment.

Deriving trait implementations is often done for common types like `Clone` for cloning a type, or `PartialEq` comparing values. This is done by using "derive macros", which can be attached to a type declaration to automatically generate a trait implementation for that type:

```
#[derive(Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Debug)]
struct Test {
  x: i64,
  y: u32,
}
```

Two of these that would have often been useful in our translations are `Debug` and `PartialEq`. The code generated by these macros does not have any Prusti annotations, and was therefore not very useful in verification. In some cases, the generated code also used Rust features not yet supported by Prusti, like references in structures. In most of these cases, writing the trait implementation manually including annotations fixed the issue, but this requires more effort. A way of either manually annotating derived implementations or automatic annotations for some of the more common traits would reduce verification effort, especially in codebases that already make heavy use of derive macros.

Specifications in trait impl blocks

We had some trouble using `impl` blocks with specifications. Writing a trait implementation block and then attempting to add a pre- or postcondition resulted in an unclear error. The fix is to add the `#[refine_trait_spec]` annotation to the `impl` block. This was not documented at the time.

```
trait TestTrait {
  fn trait_fn(self) -> i64;
}

// #[refine_trait_spec] // <== Fix
impl TestTrait for i64 {
  #[ensures(result >= 0)] // <== Error here
  fn trait_fn(self) -> i64 { 5 }
}
```

Attempting verification leads to this error:

```
[E0407] method
`prusti_post_item_trait_fn_75f3a99c74f9401583ad05ee8d8dd027`
is not a member of trait `TestTrait`.
```

This error is caused by the `#[ensures(...)]` annotation, which generates a function for the verification in the same place as the parent function, in this

case the `impl` block. Since this additional function is not part of the trait to be implemented, `rustc` throws this error. With the `#[refine_trait_spec]` annotation, this is resolved and Prusti is able to verify this code.

To prevent new users of Prusti from having the same problem, we added documentation for this annotation to the user guide. A more helpful error message hinting at using `#[refine_trait_spec]` would help new users with this problem.

Type invariants and representation sets

Chapter 16 demonstrated verifying code using objects, which can consist of other objects and can have validity predicates.

The example is a `class CoffeeMaker`, with a predicate `Valid` and a representation set `Repr`. This representation set contains any objects that are part of this `CoffeeMaker`. The representation set is then used to denote which parts of a `CoffeeMaker` are accessed or modified by each of its methods (code for other classes in this example omitted):

```
class CoffeeMaker {
  var g: Grinder
  var w: WaterTank
  ghost var Repr: set<object>

  predicate Valid()
    reads this, Repr
  {
    this in Repr &&
      g in Repr && g.Repr <= Repr &&
        this !in g.Repr && g.Valid() &&
      w in Repr && w.Repr <= Repr &&
        this !in w.Repr && w.Valid() &&
      g.Repr !! w.Repr
  }

  constructor ()
    ensures Valid() && fresh(Repr)
  {
    g := new Grinder();
    w := new WaterTank();
    Repr := {this, g, w};
    new;
    Repr := Repr + g.Repr + w.Repr;
  }
```

```
  predicate method Ready()
    requires Valid()
    reads Repr
  {
    g.HasBeans && 2 <= w.Level
  }
}
```

The translation of this example was able to make good use of Rust's ownership model, by not requiring a representation set at all. For the invariant, we decided on creating the trait `Invariant` with a single member function `is_valid`. This trait could then be implemented for any type and this clearly showed which `impl` block contained the type invariants of any structure. Using a trait here also tests Prusti's ability to verify trait implementations.

```
pub trait Invariant {
  #[pure]
  fn is_valid(&self) -> bool;
}
struct CoffeeMaker {
  g: Grinder,
  w: WaterTank,
}

impl Invariant for CoffeeMaker {
  #[pure]
  fn is_valid(&self) -> bool {
    self.g.is_valid() && self.w.is_valid()
  }
}

impl CoffeeMaker {
  #[ensures(result.is_valid())]
  fn new() -> Self {
    let g = Grinder::new();
    let w = WaterTank::new();
    Self { g, w }
  }

  #[pure]
  #[requires(self.is_valid())]
  fn is_ready(&self) -> bool {
    self.g.has_beans && self.w.level >= 2
  }
}
```

The Rust code requires a lot less effort to verify than the Dafny counterpart, since we do not need to denote the ownership relations contained in the representation set. Like in chapter 14, here we also do not require adding additional annotations for the read, write and fresh sets of each function. The ownership information checked manually in the `Valid` predicate is already contained in Rust's type system, like for example that the representation of the sub-objects is a subset of the representation of the parent `CoffeeMaker` object, or that the representation sets of the two sub-objects are disjoint.

Keeping track of and correctly updating this representation set can be tedious, so Prusti not requiring it for verification is very useful.

Representation sets might still be needed for verifying code where ownership information cannot be checked at compile time, like for example a graph datastructure using `Rc<RefCell<T>>` or in unsafe code using raw pointers. Future research would be required for checking the requirement of a representation set in such examples. The impact of having to check the representation set in each method on verification performance could also be an interesting topic of future research.

5.1.9 Chapter 17: Mutable data structures

The datastructures discussed in chapter 17 were not a great match for Rust or Prusti.

Lazily initialized arrays

This subchapter discussed creating an array datastructure that will return some default value on each uninitialized index. This clashes with safe Rust's rule about not having any uninitialized memory accessible. This means that any array used in the datastructure would either need to be fully initialized, store an `Option<T>` at each index, or use unsafe code. In addition, in order to create an array with dynamic size required by such a datastructure, either a `Vec` or a boxed slice would need to be used. Both of these require annotations to be used in verifications.

Lastly, for keeping the type invariants, several ghost fields are required in the datastructure, which have not yet been implemented in Prusti. Two of these ghost fields were `ghost var Elements: seq<T>` for storing which index corresponds to which value, and `ghost var s: set<int>` to store which indices are already initialized. Prusti has support for these types, but only with the incomplete setting `unsafe_core_proof = true`. Some work towards backporting the types `Map`, `Seq` and `Int` for use without having to enable `unsafe_core_proof` has been done in pull request #1178. These types would enable a few more of the examples in the book to be verified.

A positive aspect of our translation was again the possibility of omitting the representation set from the datastructure.

Extensible Array

The next subchapter was about implementing an array that dynamically allocates more memory as more elements are pushed onto it. This is basically the same functionality that Rust's `Vec` provides, but in a more complicated way. The `ExtensibleArray<T>` consists of a front storage of type `array<T>` with a fixed length of `256` elements, and a depot of type `ExtensibleArray?<array<T>>`. This kind of recursive type did not mesh well with Rust's strict type system, and we were unable to get even the unverified code working. Creating the type itself was not a problem, we gave the front storage the type `Option<Box<[T; 256]>>`. To create the depot, we had to extract all the functionality of the `ExtensibleArray` into a trait and then use the `dyn` keyword to store a trait object as the depot. The final type we decided on was this:

```
const ARR_LEN: usize = 256;
struct ExtensibleArray<T> {
  // ghost_elements: Seq<T>,
  front: Option<Box<[T; ARR_LEN]>>,
  // depot: Option<Box<ExtensibleArray<Box<[T; arr_len]>>>>,
  depot: Option<Box<dyn ExtensibleArrayTrait<Box<[T; ARR_LEN]>>>>,
  len: usize,
  m: usize,
}

trait ExtensibleArrayTrait<T: Clone> {
  fn is_valid(&self) -> bool;
  fn get(&self, i: usize) -> &T;
  fn get_mut(&mut self, i: usize) -> &mut T;
  fn update(&mut self, i: usize, t: T);
  fn append(&mut self, t: T);
}
```

Using the commented out version of the type for depot that is not using trait objects, we get an error from `rustc` as soon as we try to use `ExtensibleArray` in any functions:

```
[E0320] overflow while adding drop-check rules for
example_17_1::ExtensibleArray<T>. example_17_1.rs(34, 17):
overflowed on std::boxed::Box<example_17_1::ExtensibleArray<
std::boxed::Box<[std::boxed::Box<[ ... ]>; 256]>; 256]>>>
```

Note that we removed some of the repeated `Box` type in the error message for readability.

The problem here is that implementing any function for `ExtensibleArray<T>` also requires an implementation of that function for the recursively generated type `ExtensibleArray<Box<T; 256>>`. Since all functions to be compiled or verified need to be known at compile time, this type triggers an infinite chain of functions requiring an implementation, so this code cannot be compiled or verified.

Using a trait object made implementing some of the required functions possible, but we got stuck when trying to implement the `append` function. If the `front` is full, `append` needs to create a new `depot`, which would either have to use unsafe memory again to create a `depot` with a `front` containing uninitialized memory, or somehow fill each slot, for example by using a default value or by cloning the current front. All three methods have problems, unsafe code can not be verified by Prusti using the standard encoding (and `unsafe_core_proof` is still incomplete). Requiring that the ExtensibleArray is cloneable requires having infinite implementations for the `Clone` type. Analogously, requiring that the `ExtensibleArray` implements `Default` would lead to the same problem.

The initialization problem can be solved by replacing the front storage with a `Vec<T>`. In that case the datastructure would make even less sense, since `Vec` is a dynamic array, and thus already provides the functionality implemented in `ExtensibleArray`. Using `Vec` to build the `ExtensibleArray` would also require annotating the required methods on `Vec`, such as `push`.

Binary search tree with iterator

The results from this chapter are similar to the previous parts. The tree consists of nodes, which require a representation set again in Dafny:

```
class Node<Data> {
  ghost var M: map<int, Data>
  ghost var Repr: set<object>
  var key: int
  var value: Data
  var left: Node?<Data>
  var right: Node?<Data>
}
```

```
struct Node<Data> {
  // ghost_m: Map<i64, Data>
  key: i64,
  value: Data,
  left: Option<Box<Node<Data>>>,
  right: Option<Box<Node<Data>>>,
}
```

The Dafny version of this example required checking the representation set a lot. For the validity predicate, approximately half was for dedicated to checking the properties of the representation set. The translation of the first part of this example was not successful, because Prusti's `Map` datastructure was not yet available.

The second part of this example was implementing an iterator over the binary search tree, which was not possible as it depended on the successful verification of the binary search tree itself.

Chapter 17 will be very interesting to revisit once Prusti has support for maps, ghost code and iterators.

5.1.10 Summary of the example and exercise categorization

We classified every example and exercise into the four main categories "Correct Verification", "Correct Failure", "Incompleteness" and "Unsoundness". The classification was done per Rust file, independent of how much code was contained in a file.

A bit more than 53 percent of the final files were categorized as either "Correct Verification", "Correct Failure". Of these, most took a similar amount of effort for verification as their Dafny counterpart. A few of the files were categorized as requiring more effort to verify, mostly due to it requiring more code to work around some Prusti limitation. We classified bit less than 5 percent of the files as easier to verify in Prusti than Dafny, mostly due to less annotation overhead stemming from more aliasing or ownership guarantees by Rust.

Approximately 43 percent of files fall into the "Incompleteness" category, which is further divided into the reason for why Prusti could not verify it. In approximately 37 percent of the files, the reason was a missing Prusti feature and in slightly less than 5 percent the reason was a bug in Prusti. Only one example and one exercise failed due to a Rust limitation, which was the recursive type for the `ExtensibleArray` used in chapter 17.

The last category is "Unsoundness", which appeared in a bit more than 3 percent of files. The only examples this was encountered in were demonstrating

the termination checks performed by Dafny. Prusti does not currently check for termination, so all of these examples incorrectly passed verification.

5.2 Findings from rewriting the documentation

5.2.1 General documentation improvements

During the first part of this thesis project, we took notes on every feature that was insufficiently documented or hard to understand. When reworking the user guide and especially the beginner tutorial, we tried to make sure that every feature is explained and documented.

We will list some of the missing features and potential improvements for Prusti in this chapter.

Counterexample

Prusti has a feature for printing a counterexample to any assertion which may not hold. This feature is quite helpful for debugging failing verification, but we noticed that it was insufficiently documented. We wrote a chapter in the tutorial on how to use the counterexample printing, and expanded the documentation to include how to enable the setting, an example and some other useful information.

Pledges

Pledges are used on functions returning mutable references into an input structure. They denote the effect of changes to the returned reference on the original datastructure. This is an example of using pledges from the verified linked list in the user guide tutorial:

```
#[requires(!list.is_empty())]
#[ensures(snap(result) === old(snap(list.peek())))]
#[after_expiry(
      old(self.len()) === self.len()
  && forall(|i: usize| 1 <= i && i < self.len()
    ==> old(snap(self.lookup(i))) === snap(self.lookup(i)))
  && snap(self.peek()) === before_expiry(snap(result))
  )]
fn peek_mut<T>(list: &mut List<T>) -> &mut T {
  if let Some(node) = &mut list.head {
    &mut node.elem
  } else {
    unreachable!()
  }
}
```

This pledge denotes that after the returned value expires (gets dropped), the length of the list must be unchanged, all elements other than the head of the list are unchanged, and the head of the list is now the value that the reference had right before it expired.

This way of writing pledges was well documented, but there is also a way of adding an assertion right at the point of expiry, which was not documented. The syntax for this is `assert_on_expiry(condition, invariant)`. Prusti checks that the condition holds on expiry, and ensures that the invariant holds given the changes to the reference and the condition.

We added this way of asserting a condition on expiry to the user guide tutorial and to the documentation of pledges.

Suggestion for improving pledges

At the moment, functions using pledges cannot be made `#[pure]`, because the parameter and return type must contain a mutable reference to even require a pledge, but then the function cannot be pure since mutable references are not `Copy`.

Conceptually, a function like `peek_mut` above should be pure, since it does not actually make any changes to any state in the program. If the function was pure, Prusti would be able to use the function body in the verification, which could reduce the amount of pledges that a programmer has to write.

There are several prerequisites before functionality like this could be implemented, for example supporting non-`Copy` types in pure functions.

Separate under- and overflow errors

The current error message for integer overflows includes operations going over the upper bound as well as below the lower bound of the integer range. We made a feature request for adding a setting to enable separate error messages for the two cases, which should make debugging overflow errors easier (see issue #1356).

5.2.2 Writing the beginner Prusti tutorial

The last part of this thesis was rewriting and expanding a tutorial for beginners to learn using Prusti for verification. The general structure of the tutorial already existed, as well as some of the earlier chapters. The existing chapters were not using the latest Prusti features, and had to be updated. We wanted to make sure that this tutorial stays up to date with future development on Prusti, so we put several automatic tests in place to ensure proper maintenance. For this we extracted any code snippets bigger than a few lines into a separate file, which was then imported with mdBook's `#rustdoc_include`

syntax. We placed these examples in the `prusti-tests` directory, which is automatically tested by Prusti on each pull request. This way we can ensure that all code that should fail verification actually fails, and all correct code can still be verified.

Documentation testing with mdBook

To ensure that the code examples in the documentation still compile with the standard Rust compiler, we set up an automatic action to run mdBook documentation testing on both the user guide and the developer guide on each pull request. After implementing this, we discovered that some of the code examples in the developer guide did not compile, so we fixed them. This should not happen in the future, since the failing documentation test will prevent merging the incorrect changes into the codebase.

We ran into an annoying limitation of mdBook's documentation testing. Dependencies, like in our case the `prusti-contracts` crate, will not be automatically handled like with Rust's package manager Cargo. Instead, a path to the required build artifacts of the dependency needs to be given as a parameter. A fix for this was to create a dummy crate with `prusti-contracts` as a dependency, build it before running the tests, then link to its target directory. This is not the most elegant workaround, but it works.

A proper fix would be to implement Cargo's dependency management in mdBook, which is getting discussed in mdBook issue #706. A plugin for mdBook to implement this functionality called "mdBook-keeper" is mentioned in the discussion of this issue. We decided against using this plugin, as it is still missing some features, such as automatically adding a `main` function to each of the tested code blocks, as is done by `mdBook test`. This would currently require manually adding a `main` function to each code block that should be tested.

Automatically testing Prusti examples in the book

Another limitation of mdBook is that the test command it runs is fixed. If it was possible to choose to run a different command like `cargo-prusti` on the code blocks instead of the standard Rust test, we could likely use the same mechanism for testing the entire documentation. Our current approach is not optimal, since it spreads some of the files used in the documentation into the testing directory, which may cause problems in future refactors. On the other hand it should not cause any silent errors, since the automatic tests will not succeed if any of the required files is not available.

Capabilities and limitations of Prusti

We created a capabilities chapter in the user guide, which lists verification features that Prusti supports, and features that we discovered to be missing in Prusti. The limitations chapter contains links to relevant GitHub issues and pull requests, and if possible, explains workarounds for some of the missing features. The capabilities chapter should be able to serve as an overview of the state of Prusti for both the users and the developers of Prusti. It should also help to make sure that comparisons of Prusti to related work in other projects and papers are accurate and fair to the current state of Prusti.

Smaller improvements to Prusti

Over the course of the thesis, we sometimes noticed some potential improvements to the source code of Prusti, like better readability or slightly better performance and memory usage. One of these changes was swapping the more expensive standard library `HashMap` and `HashSet` with the faster `rustc_hash::FxHashMap`. Other changes include preallocating vector capacities when the final size was known, and switching from `sort` to `sort_unstable` where possible. We provided these improvements in the pull requests #1259 and #1314.

5.3 Miscellaneous

5.3.1 Prusti Assistant extension for Visual Studio Code

Like Dafny, Prusti also provides a VS Code extension for helping with verification, which is called Prusti Assistant [12]. It handles running `cargo-prusti` in the background, and it provides inline error messages in the code, similar to the Rust Analyzer VS Code extension for writing normal Rust code.

Prusti Assistant can also automatically update the installed version of Prusti, which works well, except for one bug involving multiple active instances of VS Code (see #198). We also reported a bug about incorrect error reporting when using a Cargo workspace (see #218).

Prusti Assistant is intended to be used together with another extension providing warnings, errors and inferred type information (for example Rust Analyzer). This works well overall, with only some minor issues. One such issue is that when both extensions are set up to run on file save or change, both will show warnings and errors for the code, duplicating entries in the `problems` list.

Some of the currently missing features are getting implemented in pull request #216. Some of these features are selective verification of single functions, and showing triggers chosen by Viper for any particular quantifier.

Overall, Prusti Assistant was quite helpful for writing and verifying the translated examples.

5.3.2 Prusti verification times

Over the course of the project, we noticed that Dafny was generally faster at verifying any particular chapter compared to Prusti. The verification time of Prusti is composed of multiple parts, mainly compiling the dependencies, translating the code and annotations to Viper, and finally verifying the generated Viper code.

Verification from a cold state

The worst verification times are encountered when starting verification from a clean state, with all caches cleared (i.e., removed the `./target` directory and with all Prusti caches empty). Just compiling the `prusti-contracts` dependency took approximately 12 seconds on our test machine [1]. This crate is always on the critical path for any clean state verification, since it provides the functionality required for translating the annotations in the code to be verified.

Cargo provides a way to measure the time taken by each step of the compilation of a crate, and since Prusti builds on top of Cargo, we were able profile the verification by using `cargo-prusti --timings`. This provides a summary of the steps taken for verification, but it cannot show details in the parts of the verification that run in Viper. Using this profiling tool, we discovered that a dependency of `prusti-contracts` called `serde` was taking up approximately 4 to 5 seconds for a cold compilation.

We then checked in the Prusti source code if this dependency is actually required or if its functionality can be provided in a more lightweight way. The `serde` dependency seems to have been used in a previous version of Prusti, but was not required anymore. We were able to completely remove `serde` from `prusti-contracts` in pull request #1377. After this change, compiling `prusti-contracts` from a cold state only takes up approximately 8 seconds.

Note that this change does not affect compilation with the standard Rust compiler, since `serde` and other dependencies of `prusti-contracts` are only included during verification.

Verification from a cached state

After an initial verification, Prusti is able to reuse the results of from previous verification run. Any unchanged dependencies (such as `prusti-contracts`)

[1] Test machine: AMD Ryzen 9 5900X @ 4.4GHz (24 logical cores), 32GB DDR4 RAM @ 3600MT/s, running Ubuntu 22.04 through WSL1 on Windows 10

will not need to be recompiled, and functions with unchanged code and specifications will not have to be verified again. This speeds up repeated verification by a lot, leading to faster writing and debugging of specifications.

The ability to selectively verify specific functions in a file, instead of the entire crate, is getting implemented for Prusti Assistant in pull request #216. Similar functionality is already available in the VS Code extension for Dafny. This feature should lead to faster verification times while working on the code or specifications, since Prusti will not have to verify the entire crate before showing results. Pull request #216 will also introduce asynchronous processing of verification results, which will show errors as soon as they are available, not after Prusti finishes verifying the entire crate.

Potential future speedups

We suspect that there may be potential for improving the verification times of Prusti by better utilizing multithreading. On our test machine with 24 logical cores, Prusti was almost never fully utilizing all of them. Full utilization may not be entirely possible from within Prusti, and could require changes to the Viper backend. More testing is needed to come to a concrete conclusion on potential speedups from improved parallelism.

Chapter 6

Conclusion

Here we will summarize our findings about the strengths and weaknesses of Prusti.

6.1 Strengths of Prusti

The way Prusti makes use of Rust's strong compile time guarantees is a major strength of Prusti. The reduced annotation overhead for aliasing and ownership compared to Dafny result in some examples being easier to verify. The main benefit we discovered in our translations was the possibility to omit storing a representation set for every object, which shortened some conditions considerably.

The good integration with Rust's package manager Cargo also made running Prusti easy and intuitive. Being able to use almost the same commands for verification as for compilation, with no changes needed to the code, is very helpful.

The snapshot equality operator `===` and the snapshot function `snap` helped a lot by not requiring types to implement the `PartialEq` trait, and to work around the Rust borrow checker in specifications.

Prusti Assistant was quite useful for verification, and showing the verification errors in the actual source code helped a lot during our translations.

6.2 Contributions to Prusti from this thesis

6.2.1 Improved (beginner level) documentation

The Prusti documentation, especially at a beginner level, has been updated and improved to show new users which features that Prusti provides and how to use them. This documentation is also better protected against future

breaking changes to Prusti, as it will be automatically tested on every pull request. These automated tests cannot protect against new features not getting documented, which is why we suggest that the Prusti developers implement some protocol to check that all new features are getting properly added to the documentation.

6.2.2 More intuitive behavior for unsigned integers

With overflow checking turned off, Prusti will now still model unsigned integers as being non-negative, in contrast to modeling them as unbounded as it did in the past. The old behavior is still available with the setting `encode_unsigned_num_constraint = false`, but we expect this setting to rarely be used in practice.

6.2.3 Documentation of current state of supported features

With the new capabilities and limitations chapter in the user guide, Prusti developers and users can more easily see which areas of Prusti are in need of improvements.

6.2.4 Smaller code improvements

Over the course of this thesis, we sometimes implemented some small code improvements to Prusti, which improved code readability, performance or memory usage.

6.2.5 Bug reports

Several bug reports have been made, which should help improve Prusti. Some smaller bugs were not mentioned in this report, but they can be found in the appendix.

6.3 Areas of improvement for Prusti

6.3.1 Termination checks

Termination checks were automatically performed by Dafny on any function or method using loops or recursion, which include the majority of the examples in the book. Since most of them were supposed to terminate correctly, the fact that Prusti did not check for termination on the translated code did not cause any unsoundness in most of the examples. However, non-terminating pure functions can be used to prove any property during verification, so termination checks should be implemented for Prusti at some point.

6.3.2 Standard library annotations

The project that aimed to provide Prusti annotations to the Rust standard library appears to be an important step in adding many other features, as well as being very useful by itself.

Some of the functionality in the standard library is used often in Rust programs, so having annotations for these will enable verification of many more projects with a heavily reduced manual annotation effort.

6.3.3 Iterators

Rust code makes heavy use of iterators, since they are required for using `for` loops. Iterators require quite a bit of functionality from the standard library, like for example the `Option` type. This makes iterator support dependent on the annotation of the standard library.

6.3.4 Allocation in pure functions

The current limitation of pure functions of only taking or returning `Copy` types can be quite limiting, especially for recursive datastructures like trees or lists. Due to Rust's type system, these types have to use an indirection through a pointer type like a `Box`. Being able to use these types in pure functions would make verifying many more programs possible. The `snap` function provided by Prusti can be a helpful workaround for some pure functions, but it cannot make all programs verifiable.

This limitation may have appeared at a higher frequency in the translations compared to a regular Rust codebase. One reason for this is the heavy use of linked lists and trees in the book, which are a lot less common in real Rust code. Types like `std::collections::Vec` and `BTreeMap` would likely be used rather than implementing a collection type from scratch.

6.3.5 References in structures

This limitation can be hard to work around, which may make the resulting code much less readable and potentially also less performant. Certain functions are completely unusable, for example any lookup function on a collection that returns `Option<&T>`.

Lifting this restriction could greatly improve the usability of Prusti.

6.3.6 Shallow borrows

The only way we found to trigger this limitation is by using match guard, which did not happen often during the translation. Not using match guards

made some `match` statements more complicated, but did not prevent us from verifying any of the examples.

6.3.7 Closures

We did not use any closures in the translated examples, but they are commonly used in normal codebases.

6.4 Future work

Potential future work for Prusti could be to implement some of the missing features discussed in this report.

Implementing these features in decreasing frequency of occurrence in the translations, the order would be:

- Implementing (automatic) termination checks for pure and impure functions, for both recursion and loops.
- Writing the specifications for standard library functionality not included in pull request #1249, such as iterators, functions for the trait `std::cmp::PartialOrd`, and the remaining functions on slices.
- Lifting the limitation of `Copy` types in pure functions, including allowing allocations like `Box::new()` in pure function.
- Implementing support for verifying code that uses types containing references.
- Adding lemma syntax with automatic induction proofs.
- Completing support for ghost code, ghost functions and ghost variables.
- Finish backporting maps, sets, sequences and unbounded integers (some work has been done already in pull request #1178).
- Fixing the bugs in the created GitHub issues.

Once some of these features are implemented, certain chapters of the book could be revisited, such as chapters 8, 14, 16, 17. The remaining chapters could also be translated to Prusti as part of future work.

Other future work could be improving the current handling of quantifiers and triggers by Prusti.

Appendix A

Appendix

A.1 Relevant issues and pull requests

This appendix contains a list of pull requests, issues and feature requests related to this thesis. Some issues were not mentioned in the report, but are added here for completeness. Issues that were resolved at the time of writing this report are marked `(*)`.

Prusti bug reports

Description	Link
`(*)` `non_snake_case` warning from functions starting or ending in an underscore	#1202
Error caused by mutable function arguments	#1203
Extra verifier argument `numberOfErrorsToReport` gets ignored	#1213
Internal error from mutually recursive pure functions	#1214
Missing type limits for unsigned integers when overflow checks turned off	#1215
`(*)` `i16` incorrect overflow check	#1223
Inaccurate Cast Encoding	#1224
Internal error from using references in `old()`	#1251
Internal error from old(result)	#1252
Char reference in pure function triggers unreachable code	#1262
Internal error from pure function calling itself in its contract	#1267

(*) Unsupported constant type `Ref(...)`	#1268
Unhelpful error reporting for out-of-bounds errors	#1273
Unexpected panic from trusted pure function with non-copy parameter	#1274
Incorrect verification depending on version of Rust toolchain	#1279
(*) Cannot detect if code is getting verified by Prusti or compiled normally	#1295
(*) Predicate function can only take a single expression	#1297
`snap` function can be used in normal code	#1298
Add a hint for when `result` is used in a precondition	#1300
Unreachable code reached when using reference to Option in predicate	#1305
Consistency error from marking `Option::unwrap` as `#[pure]`	#1306
Internal error from multiple pledges per function	#1322
Internal Error with `counterexample = true`	#1326
Different verification behavior with `counterexample = true` vs —false—	#1327
Error with `extern crate prusti_contracts`	#1357
No counterexample for `prusti_assert`	#1358
`extern_spec` panic with `pub` attribute	#1359
`prusti_assert` macros incorrect expansion during normal compilation	#1371
Consistency error when running `cargo-prusti --release`	#1383
Internal Error from matching on generic `enum` in pure function	#1387
Shallow Borrows not supported	#1388
Windows executable crashes during verification	#1390

Prusti feature requests

Description	Link
(*) Add `prusti_assert_eq` and `prusti_assert_eq`	#1283
(*) Support for an if-and-only-if operator `<==>`	#1299
Separate Errors for Integer Under- and Overflows	#1356

Prusti Assistant bug reports

Description	Link
`Prusti: update verifier` fails when running multiple instances of VS Code	#198
Incorrect Error Reporting when Verifying Project containing Cargo Workspace	#218

Other related issues (not created by us)

Description	Link
Current state of floating point support by Prusti	#575
(mdBook) Add ability to reference 3rd-party crates	#706

Pull requests created during this thesis

Description	Link
Improvements to the Prusti source code	#1259
Fix for issue #1202 (`non_snake_case` warning)	#1269
Enable `encode_unsigned_num_constraint` by default	#1281
Completion of issue 711 and general code improvements	#1314
Enable cargo sparse index protocol	#1367
Rework User-Guide and Documentation	#1355
Remove unnecessary serde dependency	#1377
Bump `prusti-contracts` version number	#1378

Related pull requests (not created by us)

Description	Link
[WIP] Backport mathematical types	#1178
Provide Specs for the Standard Library	#1249
(Prusti Assistant) New features and improvements	#216

A.2 Useful standard library functionality

This section contains Rust standard library functions that would have been useful in the translations. They require external specifications for use in verifications. Some of these specifications will be available in the `prusti-std` crate once pull request #1249 is merged. We mark them in the list depending on if they will be available `[++]`, partially available `[+]` or not available `[-]` after the merge.

- `[+]` functions on slices, such as:
  - `[-]` `swap`
  - `[++]` `is_empty`
  - `[-]` `rotate_right` (rotate_right was implemented in the book, but is likely not commonly used)
- `[-]` `std::collections::HashMap`, e.g.:
  - `[-]` `contains_key`
- `[+]` Operations on `std::vec::Vec`, e.g.:
  - `[++]` `Vec::new`, `Vec::with_capacity`
  - `[++]` `Vec::push`, `Vec::len`
  - `[-]` `Deref::deref(&self) -> &[T]` (automatic dereferentiation of a `Vec` to a `slice`)
- `[++]` `std::clone::Clone` (note that this is not required for use in specifications, due to the `snap()` function in Prusti)
- `[++]` `std::mem::swap`
- `[++]` `std::option::Option`:
  - `[++]` `is_none`, `is_some`
  - `[++]` `take`, `unwrap`
- `[++]` `std::result::Result`

- [+] `std::String` and `std::str`
- [-] functions in `std::cmp::PartialEq`
- [-] functions in `std::cmp::PartialOrd`
- [-] `std::iter::Iterator`
- [-] `std::ops::Range` (for use in `for` loops)

Bibliography

[1] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[2] Nicholas D. Matsakis and Felix S. Klock. The rust language. *Ada Lett.*, 34(3):103–104, oct 2014.

[3] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *International conference on verification, model checking, and abstract interpretation*, pages 41–62. Springer, 2016.

[4] K. Rustan M. Leino. *Program Proofs*. The MIT Press, 2023.

[5] Prusti Devs. Prusti user guide. https://viperproject.GitHub.io/prusti-dev/user-guide/.

[6] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer, 2010.

[7] Prusti Devs. Prusti. https://github.com/viperproject/prusti-dev.

[8] Alexis Beingessner. Learn rust with entirely too many linked lists. https://rust-unofficial.GitHub.io/too-many-lists/index.html.

[9] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. Gobra: Modular specification and verification of go programs. In *International Conference on Computer Aided Verification*, pages 367–379. Springer, 2021.

[10] Alan A. A. Donovan and Brian W. Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[11] Prusti Devs. Prusti developer guide. https://viperproject.GitHub.io/prusti-dev/dev-guide/.

[12] Prusti Devs. Prusti assistant extension for vs code. https://marketplace.visualstudio.com/items?itemName=viper-admin.prusti-assistant.

[13] hauleth cuviper. Crate num-bigint. https://crates.io/crates/num-bigint.

[14] mdBook contributors. mdbook. https://github.com/rust-lang/mdBook.

[15] Jonas Maier. Towards verifying real-world rust programs. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Jonas_Maier_BS_Thesis.pdf.

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):

Evaluating and Documenting a Rust Verifier

Authored by (in block letters):

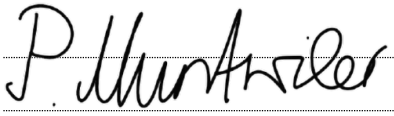
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Muntwiler	Patrick

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)**
Zürich, 2023-04-10	

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