ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Optimization of a Viper-Based Verifier

Bachelor Thesis

Till Arnold

Monday 22nd March, 2021

Advisors: Prof. Dr. Peter Müller, Vytautas Astrauskas
Department of Computer Science, ETH Zürich

Abstract

Prusti is a Viper-based static verifier which allows for the verification of functional correctness for Rust programs. The adoption of Prusti as a means to enable a broader user base to write correct system software is currently being held back in part by its slow verification times. A key contributing factor to this phenomenon is the time spent verifying the Viper code generated by Prusti. In this thesis we propose and implement potential performance optimizations aimed at improving the generated Viper code, thus decreasing the verification time. Among those proposed the most promising optimization is the purification optimization which reduces the number of heap-dependent variables and leads to runtime reductions of up to two thirds in selected cases.

Contents

Contents **iii**

1 Introduction **1**

2 Background **3**
- 2.1 Rust 3
- 2.2 Prusti 5
- 2.3 Viper 5

3 Optimizations **9**
- 3.1 Remove predicates which are never used 9
- 3.2 Remove the bodies of predicates which are never folded or unfolded 9
- 3.3 Purifying local variables 10
  - 3.3.1 Motivation 10
  - 3.3.2 Purifying local variables in a simple function 10
  - 3.3.3 Snapshots 11
  - 3.3.4 Purifying local variables with snapshots 12

4 Purification of functions **15**
- 4.1 Motivation 15
- 4.2 Background 15
  - 4.2.1 Prusti translation process 15
  - 4.2.2 Viper 16
- 4.3 Encoding the types 18
  - 4.3.1 Encoding primitive types 18
  - 4.3.2 Encoding structs 19
  - 4.3.3 Encoding enums 21
- 4.4 Encoding mirror functions 24
  - 4.4.1 Translation 24

4.4.2 Correctness 26
4.4.3 Recursive types 28

5 Evaluation **31**
5.1 Implementation 31
5.2 Benchmarks 32
5.2.1 Methodology 32
5.2.2 Results 32

6 Conclusion **37**

A Appendix **39**
A.1 Benchmark data 39
A.1.1 Overview 39
A.1.2 Purification without predicate optimizations 40
A.1.3 Purification with predicate optimizations 40
A.1.4 No Purification with predicate optimizations 41
A.1.5 No Purification without predicate optimizations 41

Bibliography **43**

Chapter 1

Introduction

Writing correct system software is an inherently difficult task. Traditionally it is left up entirely to the programmer to ensure the correctness of their programs. The Rust programming language is a modern system programming language which safeguards the programmer from many common classes of vulnerabilities and bugs by statically ensuring memory safety using its type system. While Rust's type system prevents dangling pointers and buffer overruns, it does not ensure the functional correctness of programs. However checking the functional correctness of programs is possible using static verifiers. The use of a static verifier is particularly difficult in the case of system programming languages, as such languages give the programmer low level control. As a result, formal verification tools for system programming languages often require in-depth expert knowledge of specification languages and formal systems and thus such tools are not widely used.

Prusti [1], a static verifier for Rust, uses simpler specifications, which are at the abstraction level of the programming language and thus suitable to be used by programmers. The programmer specifies assertions and Prusti uses the type information from the Rust compiler together with those assertions to construct a formal proof of the correctness of the program. This fully automatic construction and verification of proofs is directly enabled by Rust's memory safety guarantees: In a programming language without such guarantees inferring all the necessary information might be infeasible.

A defining characteristic of Prusti is, that it is targeted at programmers and aims to make formal verification accessible to a wide audience and as such the quality and ease of use of Prusti is highly important. Effort has gone towards improving the IDE experience and reducing the runtime with the addition of a server mode [3]. Despite these improvements, the long verification times, even for simple programs, might discourage the use of and hinder the adoption of Prusti.

Prusti uses the Viper [5] intermediate verification language and suite of tools in its verification process: Prusti translates Rust code with the user-provided assertions into Viper and then uses the Viper infrastructure to verify that generated code. The time spent verifying generated Viper code is a main contributing factor to the total verification time. This thesis aims to improve the performance of Prusti and thus reduce the verification time by improving the generated Viper code.

This thesis describes and implements optimizations, with a special focus on the *purification* optimization, and implements the necessary infrastructure, namely snapshots and mirror functions, to enable this.

We first introduce some necessary technical background on Rust, Prusti and Viper in Chapter 2. Chapter 3 "Optimizations" describes the proposed optimizations, while Chapter 4 "Purification of functions" details the changes and additions to Prusti which are necessary to implement the purification optimization. We choose to first describe the optimizations, in particular the purification optimization, as a motivation for the changes proposed in Chapter 4. Lastly, Chapter 5 describes the implementation in Prusti and how the optimizations impact performance.

Chapter 2

Background

This chapter introduces the technical background and some basic concepts of Rust, Prusti and Viper which will be used throughout the thesis.

2.1 Rust

Rust is a system programming language which statically guarantees memory safety by using its type system, specifically the concepts of ownership and borrowing. This will be explained with an example below.

Listing 1 shows a simple Rust program. It defines a struct `List` and a function `len`. The `List` contains two fields, the field `value`, which is a 32-bit unsigned integer, and the field `next`, which is is an optional (as indicated by `Option`) heap-allocated (as indicated by `Box`) `List`.

```rust
struct List {
    value: u32,
    next: Option<Box<List>>,
}
fn len(head: &List) -> usize {
    match head.next {
        None => 1,
        Some(box ref tail) => 1 + len(tail)
    }
}
enum Option<T> {
    None,
    Some(T),
}
```

Listing 1: Simple Rust program from the Prusti test suite

```rust
fn prepend_list(x: u32, tail: List) -> List {
    List {
        value: x,
        next: Some(Box::new(tail)),
    }
}
```

Listing 2: Example demonstrating ownership

The enumerated type[1] `Option<T>` is defined in the Rust standard library and is included here solely for explanatory purposes.

The `len` function is a simple recursive function which computes the length of a `List`. The argument `head: &List` declares `head` to be an immutable reference to a `List`. In Rust terminology this is known as an *immutable borrow* or *shared borrow* which allows the body of `len` to access, but not modify, the data behind the reference for the duration of the function call. A mutable reference would be written as `head: &mut List`. Such a *mutable borrow* allows modification. For both kinds of borrows, once the function returns, the borrow ends and the reference passed to the function may no longer be used to access the data behind it. As a consequence, a borrow is inherently temporary and constrained by its lifetime. Lifetimes are used by the Rust compiler to guarantee that references are used only when they are valid, however lifetimes are not relevant for understanding the rest of the thesis; therefore, we will not further explain them here.[2]

At any point in time, Rust enforces that there can only be either multiple immutable *or* one mutable borrow to a specific object. This is enforced at compile time, which is of critical importance to the Rust compiler's ability to ensure memory safety.

Listing 2 shows the function `prepend_list` which take an argument `tail` of type `List` and prepends an element to it. Because the type of `tail` is `List` (and not `&mut List` or `&List`), any list passed to `prepend_list` will be moved instead of borrowed. This means that the ownership of `tail` is transferred to the callee `prepend_list` and can no longer be accessed in the caller. For example the code `prepend_list(42,a); len(&a);` causes an error at compile time because the ownership of the `List` `a` was moved from the caller to `prepend_list` in the call to `prepend_list` and after the move, `a` is borrowed by the call to `len`, which is not allowed.

Note that ownership is not moved for so called `Copy` types such as `u32`, `usize` or `bool`, which are instead copied.

[1] https://doc.rust-lang.org/reference/types/enum.html

[2] For an introduction to lifetimes see https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html

2.2 Prusti

While Rust's type system enforces memory safety using the above explained mechanisms of ownership and borrowing, the general functional correctness of programs cannot be ensured by the Rust compiler.

Listing 3 shows an incorrectly implemented `len` function. It is incorrect because the base case for `None` returns 0 instead of 1. Such an error cannot be checked for by the Rust compiler automatically. It can however be checked by Prusti, a verifier for Rust, using the `ensures(result > 0)` attribute which defines a postcondition for Prusti to verify, where `result` is a special variable which stands for the return value of the function.

```
#[ensures(result > 0)]
fn len(head: &List) -> usize {
    match head.next {
        None => 0,
        Some(box ref tail) => 1 + len(tail),
    }
}
```

Listing 3: Incorrectly implemented `len` function with a Prusti `ensures` attribute

2.3 Viper

To verify the functional correctness of Rust programs, Prusti leverages the Viper infrastructure and the eponymous Viper intermediate verification language. Prusti translates the Rust program with its type information and the user-provided preconditions and postconditions to Viper code and subsequently verifies it with the Viper infrastructure.

Translating a Rust program into Viper involves translating Rust types. Viper's built-in types include booleans (`Bool`), unbounded integers (`Int`) and references to objects (`Ref`). In Viper `Bool` and `Int` are pure values and as such not fit to model mutable borrows. Thus we cannot model a Rust `usize` as a Viper `Int` because we cannot use the later to model a mutable borrow. We instead model a Rust `usize` as a Viper `Ref` with a field `val_int` which allows us to model ownership using an *accessibility predicate* such as for example `acc(self.val_int, write)`, which indicates the permission to a read from and write to a field `val_int` on an object called `self`. For abstracting over permissions we use Viper predicates: A predicate is defined in terms of a name, a list of parameters and a body which is an assertion. An example for this using `usize` is:

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

Such a Viper predicate is generated for each Rust type by Prusti. In a similar vein `acc(u32(self.value), write)` indicates the permission to access the `u32` value `self.value`.

As a further example `List` and `Box<List>` are encoded as seen in Listing 4.

```
predicate List(self: Ref) {
    acc(self.f$value, write) &&
    acc(u32(self.f$value), write) &&
    acc(self.f$next, write) &&
    acc(Option$Box$List(self.f$next), write)
}

predicate Box$List(self: Ref) {
    acc(self.val_ref, write) &&
    acc(List(self.val_ref), write)
}
```

Listing 4: Encoding of `List` and `Box<List>` as Viper predicates

Enum types are encoded with a `discriminant` field indicating which variant the object represents. Listing 5 demonstrates this on the example of `Option<Box<List>>`. Lines 2–4 encode the possible `discriminant` values and lines 5–7 encode the fact, that if `self` is the `Some` variant then the enclosed value can be accessed.

Viper supports a variety of statements including assignments, conditional statements and assertions. In addition to these regular statements, Viper has special statements for manipulating permissions: `fold` and `unfold`, `inhale` and `exhale`.

At any point in a Viper program there are certain predicates that are currently being held by the program state. A Viper program is only allowed to access memory for which it holds the relevant permissions. However Viper predicates are not automatically unfolded, thus a Viper program is not allowed access by permissions which are stored inside a predicate. That is to say, a predicate `P(x)` and the body of `P(x)` are not equivalent. They can however explicitly be transformed into each other using `fold` and `unfold` statements. Unfolding a predicate replaces it with the assertion that is the body of the predicate. Similarly folding a predicate does the opposite, replacing a number of assertions, which make up the body of said predicate, with the predicate.

```
predicate Option$Box$List(self: Ref) {
    acc(self.discriminant, write) &&
    0 <= self.discriminant &&
    self.discriminant <= 1 &&
    acc(self.enum_Some, write) &&
    (self.discriminant == 1 ==>
        acc(Option$Box$List$Some(self.enum_Some), write))
}

predicate Option$Box$List$Some(self: Ref) {
    acc(self.f$0, write) &&
    acc(Box$List(self.f$0), write)
}
```

Listing 5: Encoding of `enum` types as Viper predicates

Informally, the statement `inhale acc(self.val_int, write)` adds the permission to a read from and write to a field `val_int` on an object `self`. Correspondingly the statement `exhale acc(self.val_int, write)` asserts that the given permission is currently being held and removes the permission. The `inhale` and `exhale` statement can not only be used for permissions but also for values: The statement `inhale x > 9` assumes that the variable `x` has a value greater than 9 and the statement `exhale x > 9` asserts that the variable `x` has a value greater than 9.

Once all Rust types have been encoded as Viper predicates, the bodies of Rust functions can be encoded as Viper methods. A Viper method consists of a name, an output parameter and a body of statements. Listing 6 shows exemplary parts of the generated Viper code for the `len` function. Lines 2 and 3 encode the `head` argument of the function. The `inhale` statement on line 3 can be read as an assumption that the `head` variable is a List. Line 5 is the `None` match arm check. Lines 6–8 represent the body of the `None` match arm: Lines 6 and 7 show that the return value is a reference to an int. The `builtin$havoc_ref` function on line 6 returns a `Ref` with no known properties, which can be seen as initializing the variable `ret` with an unknown value.[3] On line 8 we see the erroneous value 0 which is returned. If line 8 were changed to be `ret.val_int := 1` the verification would succeed. Line 12 is the direct translation of the user-provided post condition (`#[ensures(result > 0)]`). This assertion will fail.

[3]In fact the entire implementation of `builtin$havoc_ref` is `method builtin$havoc_ref() returns (ret: Ref)`. That is a method with no body.

```
method len() returns (ret: Ref) {
    var head: Ref
    inhale acc(head.val_ref, write) && acc(List(head.val_ref),
    ↪  read$())
    unfold acc(List(head.val_ref), read$())
    if (Option$Box$List$discriminant(head.val_ref.f$next) == 0)
    ↪  {
        ret := builtin$havoc_ref()
        inhale acc(ret.val_int, write)
        ret.val_int := 0

        fold acc(List(head.val_ref), read$())
        fold acc(usize(ret), write)
        assert (unfolding acc(usize(ret), write) in ret.val_int)
        ↪  > 0
        exhale acc(List(head.val_ref), read$())
        exhale acc(usize(ret), write)
    }
}
```

Listing 6: Part of the encoding of the `len` function

Chapter 3

Optimizations

The overall goal of this project is to improve the performance of Prusti. Three of the identified potential optimizations are discussed in this chapter. The optimization work focuses on producing better Viper code and does not take into account Viper configuration, performance issues within Viper itself or optimizations of the Prusti code itself.

3.1 Remove predicates which are never used

Predicates which are never used in any Viper methods, predicates or functions, can be trivially removed. For this we walk all expressions in the generated Viper methods and functions and identify all predicates which are used directly. We also compute a map of which predicates are used in which predicate. We then identify all predicates that are either directly or indirectly utilized and remove all the unnecessary predicates.

3.2 Remove the bodies of predicates which are never folded or unfolded

As stated above Viper does not automatically unfold predicates. If there is a predicate that is never folded or unfolded, the body of that predicate is not necessary for the correct verification of the program and can be removed as Viper allows predicates without bodies. When the body of a predicate is removed this could potentially make more predicates entirely unused, which is taken into account by the previous optimization.

3.3 Purifying local variables

3.3.1 Motivation

Rust types are modeled by Prusti in Viper using `Ref` and predicates. This is done because Rust allows passing potentially mutable references to functions and thus we model all variables, even the ones that in Rust only exist on the stack and are never mutably borrowed, as heap-dependent. A heap-dependent variable is modeled as a `Ref` with a predicate such as for example `acc(usize(_1), write)`. Opposed to this the equivalent mathematical variable would be of type `Int`. It is to be assumed that it would be beneficial for performance if some Rust values could be modeled as mathematical values. This process of translating Prusti's generated Viper code to such a mathematical representation is referred to as *purifying*. This section describes the process of purifying methods by translating the values of local variables.

3.3.2 Purifying local variables in a simple function

Recall the `len` function which is shown again for convenience in Listing 7. Listing 8 shows the Viper code which is generated for the `Some` match arm. The variable _4 represents the result of the `len(tail)` call. Prusti does not generate a recursive call but instead generates an arbitrary value (`builtin$havoc_int()`) and then assumes the postconditions (line 2). Lines 3–5 create a `Ref` to an integer with the value `1 + len(tail)`. On line 6 the value is assigned to the return value. If `ret` and _6 were of type `Int` instead of `Ref`, the code could look like Listing 9.

```
#[ensures(result > 0)]
fn len(head: &List) -> usize {
    match head.next {
        None => 1,
        Some(box ref tail) => 1 + len(tail)
    }
}
```

Listing 7: `len` function with postcondition

```
1 _4 := builtin$havoc_int()
2 inhale _4 > 0
3 _6 := builtin$havoc_ref()
4 inhale acc(_6.val_int, write)
5 _6.val_int := 1 + _4
6 ret := _6
```

Listing 8: Viper encoding of the `Some` match arm body

```
_4 := builtin$havoc_int()
inhale _4 > 0
_6 := 1 + _4
ret := _6
```

Listing 9: Purified version of Listing 8

This purification works because there is a built-in mathematical Viper type for Rust's `usize`. However, a similar optimization is possible for more complicated types like Rust's structs and enums. To facilitate this, we need a mathematical representation of the types and cannot use the previous `Ref`-based representation. This is allowed by *snapshots* in Prusti.

3.3.3 Snapshots

A snapshot is an alternative representation of a Rust type in Viper. The inner workings of snapshots are explained in chapter 4. In this chapter it suffices to understand the inherent properties of snapshots.

The snapshot type of a primitive type such as `usize`, `u32` or `bool` is the corresponding primitive Viper type `Int`, `Int` or `Bool` respectively. For each compound Rust type `T`, such as structs, enums or tuples, we define a type in Viper called `Snap$T` which represents that type as an immutable mathematical value.

Snapshots allow most of the operations that the `Ref`-based representation would allow, specifically for each type `Snap$T` there exists:

- A *constructor function* `Snap$T$cons` which constructs an instance of the snapshot. For example in the case of a struct the function `Snap$T$cons` takes an argument for each field of the struct and returns a `Snap$T`. The types of the arguments are the corresponding snapshot types for each field.
- A *snapshot function* `snap$T` which takes an argument of type `T` and returns a corresponding snapshot of type `Snap$T`.
- For each field `f` of a type `T` there is a function `Snap$T$field$f` which takes a `Snap$T` and returns the value of the corresponding field. For enums the function `Snap$T$discriminant` takes a `Snap$T` and returns an `Int` corresponding to the discriminant.

Because snapshots are immutable they are not suitable as a representation if mutable references are required. Thus only calling functions that do not mutate the arguments is supported for snapshots. In Prusti such pure functions can be marked as `#[pure]`. For each pure function `f` an equivalent *mirror function* `mirror$f` is defined. The mirror function is equivalent to the pure function except that the types of the arguments and return value have been substituted for their corresponding snapshot type.

3.3.4 Purifying local variables with snapshots

The `len` function in Listing 7 can be marked as `#[pure]` and thus a corresponding mirror function `mirror$len` can be constructed.

Listing 10 shows a function which constructs a list with one element, modifies it and calls `len` on it. Listing 11 shows the non-purified Viper encoding of that function.

```rust
fn main() {
    let mut x = List { value: 10, next: None };
    x.value = 30;
    assert!(len(&x) == 1);
}
```

Listing 10: Client code for `len`

At first glance it appears that, by utilizing snapshots, purifying this function should be a trivial matter as all operations can simply be substituted with their corresponding snapshot version. Such a naive approach is shown in Listing 12. Note that `inhale` is used to model assignments to fields, because an assignment statement such as `Snap$List$field$value(x) := 10` is not valid in Viper, since we cannot assign to the result of a function call. This naive approach however is unsound because snapshots are immutable: Thus the statements on lines 9 and 11 contradict each other, which causes the program to always verify as we have essentially assumed false.

To solve this, the Viper code has to be translated to static single assignment form (SSA). Every time a snapshot variable is mutated we instead use a new variable and create a copy of the snapshot as shown in Listing 13.

This translation to SSA solves a further issue with purifying local variables: Old expressions. Old expressions are a Viper language feature used by the Viper code generated by Prusti. The old expression `old[l](e)` evaluates to the value which the expression `e` had at the label `l`. This language feature however cannot be used once the method body has been purified because snapshots are immutable and thus do not have an old value. However because we now translate method bodies to SSA, we can refer to the corresponding SSA variable. Thus `assert old[l](foo(x))` will be translated to `assert mirror$foo(x_n)` where `x_n` is the SSA variable for `x` at label `l`.

```
var _none: Ref
var x: Ref

_none := builtin$havoc_ref()
inhale acc(Option$Box$List(_none),
↪  write)
inhale Option$Box$List$discriminant
↪  (_none) == 0

x := builtin$havoc_ref()
inhale acc(List(x), write)
unfold acc(List(x), write)
_aux_u32 := builtin$havoc_ref()
x.f$value := _aux_u32
inhale acc(x.f$value.val_int,
↪  write)
x.f$value.val_int := 10
x.f$next := _none
x.f$value.val_int := 30

//...
assert len(x) == 1
//...
```

Listing 11: Non-Purified

```
var _none:
↪  Snap$Option$Box$List
var x: Snap$List



inhale
↪  Snap$Option$Box$List$discriminant
↪  (_none) == 0


inhale
↪  Snap$List$field$value(x)
↪  == 10
inhale
↪  Snap$List$field$next(x)
↪  == _none
inhale
↪  Snap$List$field$value(x)
↪  == 30
//...
assert mirror$len(x) == 1
//...
```

Listing 12: Naively Purified

```
var _none: Snap$Option$Box$List
var x_1: Snap$List
var x_2: Snap$List
var x_3: Snap$List

inhale Snap$Option$Box$List$discriminant(_none) == 0
inhale Snap$List$field$value(x_1) == 10
inhale Snap$List$field$value(x_2) == Snap$List$field$value(x_1)
inhale Snap$List$field$next(x_2) == _none
inhale Snap$List$field$next(x_3) == Snap$List$field$next(x_2)
inhale Snap$List$field$value(x_3) == 30

//...
assert mirror$len(x_3) == 1
//...
```

Listing 13: Purified with SSA

Chapter 4

Purification of functions

4.1 Motivation

As mentioned in the previous chapter, purification of method bodies necessitates a snapshot encoding of the relevant types. Furthermore the process of purification requires mirror functions: Functions which are equivalent to their corresponding pure functions, in all aspects except for their formal argument types and return types, which are snapshots instead. Not only are snapshots and mirror functions essential for the optimization described in the previous chapter, but they also allow Prusti to model pure functions that return non-primitive types as opposed to the current implementation.

It is to be noted here that Prusti already contained an implementation of snapshots which this thesis extends further. The existing implementation supports creating snapshots from the `Ref`-based representation (`snap$T`), the constructor function (`Snap$T$cons`) and discriminant access (`Snap$T$discriminant`).

4.2 Background

4.2.1 Prusti translation process

When the Rust compiler compiles a program it uses multiple intermediate representations (IRs): After parsing, Rust code is translated to HIR (High-Level Intermediate Representation) and later to MIR (Mid-Level Intermediate Representation) which is eventually translated into an executable binary.[1]

Prusti leverages the Rust compiler and operates on MIR instead of directly operating on Rust code which simplifies its operations greatly. Prusti trans-

[1]For an overview of the Rust compiler see `https://rustc-dev-guide.rust-lang.org/part-2-intro.html` in the Guide to Rustc Development

lates MIR into a new IR called VIR before translating VIR into Viper code. The process described in this chapter is technically implemented on the VIR level, however in the following examples we chose to use Viper code instead of VIR for the sake of visual clarity and simplicity while being, for the intent of this chapter, equivalent.

4.2.2 Viper

To represent snapshots in Viper we utilize Viper *functions*, *domains*, *domain functions* and *axioms* which are described in this section.

Viper functions

In the `Ref`-based representation Rust functions marked with Prusti's *#[pure]* attribute are translated to Viper functions, while impure Rust functions are translated to Viper methods. Viper methods have been discussed in previous chapters. A Viper function consists of a name, a list of zero or more parameters, any number of preconditions and postconditions, and a function body that is a single expression. Viper function can contain recursion and evaluating a Viper function does not cause any side effects. Viper functions are declared on the top-level of a Viper program. Listing 14 shows an example Viper function for the Euclidean algorithm as well as a method that demonstrates how the function can be used.

```
function gcd(a: Int, b: Int) : Int
  requires a >= 0
  requires b >= 0
  ensures result <= a || result <= b
  {
     b == 0 ? a : gcd(b, a % b)
  }

method test(x: Int, y: Int) {
    assert gcd(22, 12) == gcd(12, 10)
    assert gcd(12, 10) == gcd(10, 2)
    assert gcd(10, 2) == gcd(2, 0)
    assert gcd(2,0) == 2
    assert gcd(12, 22) == 2

    inhale x > 0 && y > x
    assert gcd(x, y) == gcd(y, x)
}
```

Listing 14: Viper function implementing the Euclidean algorithm for the greatest common divisor

Viper Domains

Domains are a Viper language feature allowing the flexible modeling of new types and mathematical functions operating on them. A domain has a name and can contain any number of *domain functions* and *axioms*.

A functions declared inside of a domain is called a domain function and as opposed to a standard Viper function cannot have preconditions, postconditions or a body. Since domain functions cannot have preconditions they are total functions. The value of the evaluation of domain functions is defined using axioms.

Listing 15 contains an example domain `Nat`, modeling natural numbers as peano numbers, using the domain functions `zero` and `succ`, as well as a domain function `even` that represents whether or not a `Nat` is even. The properties of the `even` function are defined by the two axioms `even_axiom_0` and `even_axiom_1`. The names of axioms have no significance and only aide in visual clarity. The axiom `even_axiom_1` contains an `forall` quantifier which is used to declare that the contained assertion holds for all values `n` of type `Nat`. The expression after the `::` in the quantifier is known as the trigger expression, the quantifier is only instantiated when a trigger expression is encountered. The choice of triggers is critical as a bad choice of triggers can lead to infinite loops or can cause Viper to fail the verification of a program. As this example shows, a domain is not just a syntactic container for axioms and domain functions but also declares a new type with the name of the domain.

```
domain Nat {
  function zero(): Nat
  function succ(val: Nat): Nat
  function even(val: Nat): Bool
  axiom even_axiom_0 { even(zero()) }
  axiom even_axiom_1 {
    forall n: Nat :: { even(n) }
      even(succ(n)) == !even(n)
  }
}
method test() {
  assert zero() == zero()
  assert even(zero())
  assert !even(succ(zero()))
  assert even(succ(succ(zero())))
}
```

Listing 15: Viper domain encoding natural numbers

4.3 Encoding the types

Following is a description of how snapshots are encoded for different kinds of Rust types.

For all applicable types `T` we need to define the corresponding snapshot type `Snap$T` and functions to operate on snapshots. In particular we need to define the *constructor function* `Snap$T$cons` which allows the creation of snapshots, as well as the *snapshot function*, `snap$T` which converts from a `Ref`-based `T` to a corresponding snapshot of type `Snap$T`.

As we aim to model `#[pure]` functions with snapshots we need to ensure that all properties that hold for the `Ref`-based encoding also hold for the snapshot encoding. This also necessitates functions to access the values of fields of stucts and enums as well as the discriminant values of enums.

Additionally we need to defined a *validity function* `Snap$T$valid` with the purpose of capturing the validity invariant [6] of a type. A validity invariant is an invariant that always holds, for example that an enum is always one of its variants. This validity function is needed because Viper domain functions are total functions and as such one can call a domain function even with an argument for which the validity invariant is violated. Using the validity function we can ensure that mirror functions are only defined for valid arguments.

4.3.1 Encoding primitive types

The simplest case for defining snapshots are primitive Rust types. As stated before, the snapshot type of a primitive type such as `u32` or `bool` is simply the corresponding Viper primitive type `Int` or `Bool`. We also do not need a constructor function for primitive types as it, in the example of `u32`, would be the identity function with the signature `function Snap$u32$cons(arg: Int):Int`. Thus instead of defining a constructor function and calling it as `Snap$u32$cons(arg)` we directly use `arg`.

The valid function `Snap$bool$valid` for booleans is defined in such a way that `Snap$bool$valid(x) == true` for all `x`. This is correct as all validity constraints of booleans are already captured by the Viper `Bool` type. For integers however, further constraints are necessary as Viper `Int`s are unbounded as opposed to Rust integers. Listing 16 contains an axiom that encodes this validity constraint for `u32` such that `Snap$u32$valid(self)` only holds for `self` in the valid range of a `u32` namely from 0 to $2^{32} - 1$.

The snapshot function for a primitive type takes a `Ref`-based representation as an argument and reads the contained value and returns it. Listing 17 shows the snapshot function for a `u32`. The `requires acc(u32(arg), read$())` ensures that the argument is the `Ref`-based representation of a `u32`

```
function Snap$u32$valid(self: Int): Bool

axiom Snap$u32$valid$axiom {
    forall self: Int :: { Snap$u32$valid(self) }
     Snap$u32$valid(self) == (self >= 0 && self <= 4294967295)
}
```

Listing 16: Valid function and axiom for `u32`

```
function snap$u32(arg: Ref): Int
  requires acc(u32(arg), read$())
  ensures Snap$u32$valid(result)
{
    unfolding acc(u32(arg), read$()) in arg.val_int
}
```

Listing 17: Snapshot function for `u32`

which then allows us to use the `unfolding` expression to access the `val_int` field of the argument and use that as a snapshot. Notice that the postcondition ensures that the returned snapshot is valid. All snapshot functions for all types have the valid function as a postcondition as the validity constraints hold for the `Ref`-based representation as enforced by the precondition.

4.3.2 Encoding structs

For a struct `T`, the snapshot type `Snap$T` is declared by defining a domain with the coresponding name. Inside that domain the domain functions and axioms are declared to allow for the following operations.

Constructor function

Listing 18 shows how the constructor for `Snap$List` is encoded as a domain function. The `Snap$List$injectivity` axiom ensures that the constructor is injective, that is to say that to construct two snapshots that are equivalent their arguments have to be equivalent too.[2]

Field access

Listing 19 shows the domain function that defines access to the `next` field of the `List` struct. The domain function `Snap$List$field$next` takes a `Snap$List` as an argument and returns the value of the `next` field. This is ensured by the axiom `Snap$List$field$next$axiom` which defines the prop-

[2]As remarked above, this domain function and axiom were not developed as part of this thesis and existed before.

```
 function Snap$List$cons( value: Int
                         , next: Snap$Option$Box$List): Snap$List

 axiom Snap$List$injectivity {
    forall value_1: Int,
            next_1: Snap$Option$Box$List,
            value_2: Int,
            next_2: Snap$Option$Box$List
     :: { Snap$List$cons(value_1, next_1)
        , Snap$List$cons(value_2, next_2)
        }
    (Snap$List$cons(value_1, next_1)
      == Snap$List$cons(value_2, next_2))
        ==> value_1 == value_2 && next_1 == next_2
}
```

Listing 18: Constructor function and related axiom for `Snap$List`

```
function Snap$List$field$next(self: Snap$List):
↪   Snap$Option$Box$List

axiom Snap$List$field$next$axiom {
    forall value: Int, next: Snap$Option$Box$List ::
    { Snap$List$field$next(Snap$List$cons(value, next)) }
    Snap$List$field$next(Snap$List$cons(value, next)) == next
```

Listing 19: Snapshot Viper encoding of the `next` field of `List`

```
function Snap$List$field$value(self: Snap$List): Int

axiom Snap$List$field$value$axiom {
    forall value: Int, next: Snap$Option$Box$List ::
    { Snap$List$field$value(Snap$List$cons(value, next)) }
    Snap$List$field$value(Snap$List$cons(value, next)) == value
}
```

Listing 20: Snapshot Viper encoding of the `value` field of `List`

erties of `Snap$List$field$value`, namely the relation between the constructor and field access: If a snapshot is constructed with `Snap$List$cons(v, n)` then the `next` field of that snapshot has the value `n`. Listing 20 shows the domain function and axiom for the `value` field of the `List` struct, which work in the same way.

Valid function

A struct is valid if all of its fields are valid. Listing 21 shows how this is encoded as an axiom for the `Snap$List$valid` domain function.

```
function Snap$List$valid(self: Snap$List): Bool
axiom Snap$Snap$List$valid$axiom {
    forall self: Snap$List ::
    Snap$List$valid(self) ==
     ↪  (Snap$u32$valid(Snap$List$field$value(self)) &&
     ↪  Snap$Option$Box$List$valid(Snap$List$field$next(self)))
}
```

Listing 21: Valid function for `List`

Snapshot function

Listing 22 shows, on the example of the `List` struct, as first shown in Listing 1, how the snapshot function for `Snap$List` is contructed. The function creates snapshots of each field of the `Ref`-based argument and uses the constructor function of `Snap$List` to create a snapshot.

```
function snap$List(arg: Ref): Snap$List
    requires acc(List(arg), read$())
    ensures Snap$List$valid(result)
{
    unfolding acc(List(arg), read$()) in
        Snap$List$cons(snap$u32(arg.f$value),
         ↪  snap$Option$Box$List(arg.f$next))
}
```

Listing 22: Snapshot function for `List`

4.3.3 Encoding enums

The snapshot type for enums is also encoded using Viper domains. In general snapshots for enums are encoded similarly to snapshots for structs with the main difference that an enum has multiple constructors.

Constructor functions

The snapshot of an enum has as many constructors as it has variants. Listing 23 shows this on the example of `Option<Box<List>>`. For each of the two variants of `Option` a constructor function is generated. The constructor

```
function Snap$Option$Box$List$cons$None(): Snap$Option$Box$List
function Snap$Option$Box$List$cons$Some(arg: Snap$List):
↪ Snap$Option$Box$List

axiom injectivity$Some {
    forall arg1: Snap$List, arg2: Snap$List ::
    { Snap$Option$Box$List$cons$Some(arg1),
    ↪ Snap$Option$Box$List$cons$Some(arg2) }
    Snap$Option$Box$List$cons$Some(arg1) ==
    ↪ Snap$Option$Box$List$cons$Some(arg2) ==> arg1 == arg2
}
```

Listing 23: Constructor functions for `Option<Box<List>>`

function `Snap$Option$Box$List$cons$None` takes no arguments because the `None` variant of `Option` has no fields while the constructor function `Snap$Option$Box$List$cons$Some` takes a `Snap$List` because the `Some` variant of `Option<Box<List>>` takes one argument of type `List`. Notice that the `Box` here is ignored, as for the purposes of snapshots the indirection of a `Box` is of no consequence. The `injectivity$Some` axiom serves the same purpose as in the case of a struct. In theory there also is an `injectivity$None` axiom, but since `None` has no fields the axiom body is simply `true` and thus the axiom can be omitted.

Discriminant function

The *discriminant function* is the snapshot equivalent of the `discriminant` field in the `Ref`-based representation. Listing 24 shows on the example of `Option<Box<List>>` how the discriminant function `Snap$Option$Box$List$discriminant` is defined: The `None$cons$axiom` and `Some$cons$axiom` axioms encode that a snapshot constructed with a constructor function has the corresponding discriminant.

Field access

Field access is implemented by essentially treating each variant as a struct. The corresponding axioms are then only defined for the correct constructor as seen in Listing 25 on the example of `Option<Box<List>>`.

Valid function

An enum value is valid when all the fields of its variant are valid. Listing 26 shows the valid function for `Option<Box<List>>` and the axiom `Snap$Option$Box$List$valid$axiom` which ensures that `Snap$Option$Box$List$valid(x)` is true if the discrimiant of `x` is in the valid range and:

```
function Snap$Option$Box$List$discriminant(s: Snap$Option$Box$List): Int
axiom None$cons$axiom {
  Snap$Option$Box$List$discriminant(Snap$Option$Box$List$cons$None()) == 0
}
axiom Some$cons$axiom {
  forall arg: Snap$List :: {Snap$Option$Box$List$cons$Some(arg)}
  Snap$Option$Box$List$discriminant(Snap$Option$Box$List$cons$Some(arg)) == 1
}
```

Listing 24: Discriminant function for `Option<Box<List>>`

```
function Snap$Option$Box$List$Some$field$0(self: Snap$Option$Box$List): Snap$List

axiom Snap$Option$Box$List$_0$Some$field$axiom {
    forall arg: Snap$List ::
    { Snap$Option$Box$List$Some$field$0(Snap$Option$Box$List$cons$Some(arg)) }
    Snap$Option$Box$List$Some$field$0(Snap$Option$Box$List$cons$Some(arg)) == arg
}
```

Listing 25: Field access function for `Option<Box<List>>`

```
function Snap$Option$Box$List$valid(self: Snap$Option$Box$List): Bool
axiom Snap$Option$Box$List$valid$axiom {
    forall self: Snap$Option$Box$List ::
    Snap$Option$Box$List$valid(self) ==
    (
    0 <= Snap$Option$Box$List$discriminant(arg) && Snap$Option$Box$List$discriminant(arg) < 2 &&
    (Snap$Option$Box$List$discriminant(self) == 0 ||
    (Snap$Option$Box$List$discriminant(self) == 1 && Snap$List$valid(Snap$Option$Box$List$Some$field$0(self))))
    )
}
```

Listing 26: Valid function for `Option<Box<List>>`

- x has the discriminant 0: x represents a None value and since None has no fields, no conditions except Snap$Option$Box$List$discriminant (self)==0 has to hold for x to be valid.
- x has the discriminant 1 and Snap$Option$Box$List$Some$field$0(x) is a valid List: This means that x represents a Some value and the contained value is valid.

Snapshot function

Listing 27 shows how the snapshot function for an enum is constructed: We match on the discriminant field and then construct the corresponding variant with the correct constructor using snapshot functions recursively for the fields. This is the same approach as used for structs, except for the matching of the discriminant.

```
function snap$Option$Box$List(arg: Ref): Snap$Option$Box$List
  requires acc(Option$Box$List(arg), read$())
  ensures Snap$Option$Box$List$valid(result)
{
    unfolding acc(Option$Box$List(arg), read$()) in
    (arg.discriminant == 0)
    ? Snap$Option$Box$List$cons$None()
    : unfolding acc(Option$Box$List$Some(arg.enum_Some),
    ↪   read$()) in
    unfolding acc(Box$List(arg.enum_Some.f$0), read$()) in
    Snap$Option$Box$List$cons$Some(
    ↪   snap$List(arg.enum_Some.f$0.val_ref))
}
```

Listing 27: Snapshot function for Option<Box<List>>

4.4 Encoding mirror functions

Using the snapshot encoding as defined above we can now construct the mirror functions.

4.4.1 Translation

Mirror functions are constructed by translating the existing standard Viper functions which encode *#[pure]* functions. For this we need to define the process of translating a Viper expression into an equivalent *mirror expression* which can be used in the mirror function.

```
1  function mirror$f(arg: Snap$T): Int
2
3  axiom f$axiom {
4      forall arg: Snap$T :: { mirror$f(arg) }
5      (Snap$T$valid(arg) && 𝓜(pre))
6        ==>
7      𝓜(post) && (mirror$f(arg) == 𝓜(b))
8  }
```

Listing 28: Mirror function for an arbitrary function `f` with an argument of type `T` and a body b. *pre* is the conjunction of all preconditions of `f` and *post* is the conjunction of all postconditions of `f`.

Listing 28 shows how a function `f` which takes an argument of type `T` and returns an `Int` is translated into a mirror function, under the assumption that $\mathcal{M}$ is a function which, given a Viper expression, returns the corresponding mirror expression. Assume that the body of `f` is the expression b while *pre* and *post* are the conjunctions of all preconditions respectively postconditions of `f`. The function `mirror$f` is declared as a domain function which takes an argument of type `Snap$T` since `f` has an argument of type `T`. The axiom `f$axiom` describes the properties of `mirror$f` in the form of an implication for all arguments `arg`. For the implication to hold `arg` has to be a valid `Snap$T` and the precoditions of `f` have to hold (line 5). If these two conditions hold we can assume the postconditions as well as that the value of `mirror$f(arg)` is equal to the function body.

Let $\mathcal{M}$ be a function that takes a Viper expression and returns the corresponding mirror expression, then $\mathcal{M}$ for an expression `e` of type `T` is defined as follows:

- $\mathcal{M}$(`unfolding(acc(...)) in e`) = $\mathcal{M}$(`e`), since the unfolding is no longer necessary as we have no more predicates because we are using snapshots.
- $\mathcal{M}$(`acc(...)`) = `true`, for the same reason as the point above.
- $\mathcal{M}$(`e.val_bool`) = $\mathcal{M}$(`e`), since the `val_bool` field can only be accessed on an expression `e` that represents a Rust `bool`, the mirror expression of e must be of Vipers type `bool`.
- $\mathcal{M}$(`e.val_int`) = $\mathcal{M}$(`e`), for the same reason as $\mathcal{M}$(`e.val_bool`).
- $\mathcal{M}$(`e.val_ref`) = $\mathcal{M}$(`e`), since snapshots are immutable the indirection of references is no longer required.
- $\mathcal{M}$(`e.discriminant`) = `Snap$T$discriminant(`$\mathcal{M}$`(e))`, since this is the snapshot equivalent of the discriminant.

- $\mathcal{M}$(`e.f$foo`) = `Snap$T$field$foo(`$\mathcal{M}$(`e`)`)`, because Prusti uses the convention of prefixing Rust field names with `f$` to form the Viper field name.
- $\mathcal{M}$(`e.enum_Var.f$foo`) = `Snap$T$Var$field$foo(`$\mathcal{M}$(`e`)`)`, since `e.enum_Var.f$foo` represents accessing the `foo` field of the `Var` variant of an enum and the snapshot equivalent is this field access function.
- $\mathcal{M}$(`f(e)`) = `mirror$f(`$\mathcal{M}$(`e`)`)`, since a pure function can only call other pure functions and thus a mirror function for f exists and can be called.
- $\mathcal{M}$(`result`) = `mirror$f(arg1, ..., argn)` where f is the function that is being translated. The special variable `result` is used in post conditions of Viper functions to refer to the result of the function. Since we are translating the body of the pure Viper function into an expression to be used in a domain axiom, a postcondition like `ensures result > 0` for a function `f(x)` would be represented by `mirror$f(x) > 0`.
- $\mathcal{M}$(`c ? builtin$unreach(): e`) = $\mathcal{M}$(`c ? e : builtin$unreach()`) = $\mathcal{M}$(`e`) for a boolean expression `c`. One of the properties that Prusti verifies is that the Rust macro `unreachable!` will never be called. To accomplish this, Prusti translates `unreachable!` to a call to the `builtin$unreach` function. The `builtin$unreach` function is a Viper function with a precondition of `false`, thus calling `builtin$unreach` causes the verification to fail. Since mirror functions are Viper domain functions and thus total functions we cannot model the function call failing and we thus ignore calls to `builtin$unreach` by translating a conditional with an `builtin$unreach()` as one arm into the other arm. This approach causes unsoundness, as this allows functions that can potentially call `unreachable!` to verify. This issue is addressed later in the chapter.

For a concrete example of a pure function and its corresponding mirror function see Listing 29 and Listing 30. In this particular example the precondition `acc(List(x), read$())` was translated to `true` as it existed purely to ensure that the argument was of the right type, which is already ensured by the valid function. Informally speaking the valid function is the translation of `acc(List(x), read$())`.

4.4.2 Correctness

To ensure that the translation process is sound two more steps are required. Despite the fact that the precondition is on the left hand side of the implication in the definitional axiom, verification will not fail if `mirror$f` is called with a value `e` for which the precodition does not hold. We will simply have no information about the value of `mirror$f(e)`. However such a

```
function len(x: Ref): Int
requires acc(List(x), read$())
ensures result > 0
{
  (unfolding acc(List(x), read$()) in
  (unfolding acc(Option$Box$List(x.f$next), read$()) in
  (x.f$next.discriminant == 0 ?
  1 :
  (unfolding acc(Option$Box$ListSome(x.f$next.enum_Some),
  ↪   read$()) in
  (unfolding acc(Box$List(x.f$next.enum_Some.f$0), read$()) in
  1 + len(x.f$next.enum_Some.f$0.val_ref))))))
}
```

Listing 29: Ref-based encoding of the pure len function

```
function mirror$len(arg: Snap$List): Int

axiom len$axiom {
  forall arg: Snap$List :: { mirror$len(arg) }
  Snap$List$valid(arg) && true ==>
  mirror$len(arg) > 0 && mirror$len(arg) ==
  (Snap$Option$Box$List$discriminant(Snap$List$field$next(arg))
  == 0 ? 1
  : 1 + mirror$len(Snap$Option$Box$List$Some$field$0(arg))
  )
}
```

Listing 30: Mirror function for Listing 29

function call should fail verification. As a mirror function is a domain function and thus a total function we cannot restrict for which values a mirror function can be called. To address this we define a standard Viper function `caller$mirror$f`, with the translated preconditions and postconditions of `f`, which directly calls `mirror$f`. We then use `caller$mirror$f` instead of `mirror$f` when calling `f` from a Viper method.

The second problem is, that if a user specifies a function where the precondition and the function body together do not imply the postcondition Prusti should fail verification. With the previous implementation using Viper functions this was the case, as Viper automatically verifies the correctness of functions. With the new domain function based implementation, Viper can no longer verify this for us, as the function body becomes part of an axiom. At first one might suspect that it would be enough to simply emit both representations. The domain function `mirror$f` would then be used in all method bodies (indirectly via `caller$mirror$f`) while the standard Viper function `f` would cause the verification to fail if the postcondition is not valid. This does not work because if the postcondition is invalid then the axiom derived from it might cause a global contradiction, which allows Viper to prove `false` thus the verification of the standard Viper function `f` might succeed. Note that such a global contradiction does not only influence the verification of the function `f` but also causes the entire program to verify automatically. Instead one has to use a two phase approach where in the first phase the standard Viper function `f` is verified and in a second phase `mirror$f` is used. This two phase approach also solves the previously mentioned issue of $\mathcal{M}$ ignoring calls to `builtin$unreach` in the translation process as functions that might call `unreachable!` would be rejected in the first phase.

4.4.3 Recursive types

A critical point that has not been mentioned so far is, that there is no automation for mirror functions: For standard Viper functions, Viper automatically handles the instantiation of the function, for domain functions however this has to be handled explicitly with the triggers for the quantifiers.

The approach as described in Listing 28 has the problem that the trigger is contained in the body of the quantifier which might cause an infinite loop.

An approach for solving this is the use of limited functions as described in [4]: For each mirror function we define a limited and an unlimited form. They are logically equivalent, however only the unlimited form of the function causes the axiom to be instantiated because only the unlimited form of the function is used in triggers, while the limited form of the function is used in the body of the axiom. We however choose to use a more generalized approach of limited functions inspired by the implementation in Dafny [2]. Instead of defining a limited and an unlimited form of a mirror function we add a formal parameter `count` to the mirror function. This additional parameter indicates how many times the axiom should be instantiated. Thus the signature of the mirror function for a function `f` with one parameter would look like `mirror$f(arg: Snap$T, count: Nat)`. Since `count` represents a natural number we define an encoding of natural numbers as peano numbers as seen in Listing 31.

```
domain Nat {
    function zero(): Nat
    function succ(val: Nat): Nat
}
```

Listing 31: Viper domain for `Nat`

Following the change of signature of mirror functions we need to adjust the definition of the axiom for the mirror function: In addition to all arguments of a function we now also quantify over the additional variable `count` of type `Nat`. And instead of the trigger *{mirror$f(arg)}* we use the trigger *{mirror$f(arg,succ(count))}*. Recursive calls to `mirror$f` inside `mirror$f` are translated to `mirror$f(x, count)`. Intuitively a call to a mirror function with a `count` argument of 3[3] would only contain recursive calls with a `count` argument of 2. Since the axiom will not get instantiated once `count` reaches 0 this approach successfully prevents the infinite loop.

An extra axiom is required to ensure that the `count` parameter does not affect the result of a mirror function. We ensure this by declaring that `mirror$f(arg, count) == mirror$f(arg, succ(count))` for all `arg` and `count`.

As mentioned above, standard Viper functions are instantiated automatically: In particular Viper automatically instantiates a standard function each time a predicate in its precondition is unfolded. Thus simply instantiating a mirror function n times at each call site is not sufficient: To emulate this behavior of standard functions for mirror functions, we declare an additional domain function `Snap$T$UnfoldWitness` for each type `T`, which will be used for the sole purpose of causing additional instantiations of mirror functions. This is accomplished by adding an additional trigger {`Snap$T$UnfoldWitness(arg,count),mirror$f(arg,zero())`} to mirror functions and emitting calls to `Snap$T$UnfoldWitness` in method bodies where appropriate.

With these changes in place the real mirror function for `len` is depicted in Listing 32.

[3]Which would be encoded as `succ(succ(succ(zero())))`.

```
function mirror$len(arg: Snap$List, count: Nat): Int
function Snap$List$UnfoldWitness(self: Snap$List, count: Nat):
↪  Bool

axiom len$axiom {
  forall arg: Snap$List, count: Nat ::
  { Snap$List$UnfoldWitness(arg,count),mirror$len(arg,zero())}
  { mirror$len(arg, succ(count)) }
  Snap$List$valid(arg) ==>
  mirror$len(arg, succ(count)) > 0
  && mirror$len(arg, succ(count)) ==
  (Snap$Option$Box$List$discriminant(Snap$List$field$next(arg))
   ↪  == 0 ? 1 : 1 +
   ↪  mirror$len(Snap$Option$Box$List$Some$field$0(
   ↪  Snap$List$field$next(arg)), count))
}

axiom len$nat_axiom {
  forall arg: Snap$List, count: Nat ::
  { Snap$List$UnfoldWitness(arg,count), mirror$len(arg,zero())}
  { mirror$len(arg, succ(count)) }
  mirror$len(arg, count) == mirror$len(arg, succ(count))
}
```

Listing 32: Real mirror function for `len`

Chapter 5

Evaluation

5.1 Implementation

In order to evaluate the impact of the proposed optimizations, they were implemented within Prusti. The Implementation work consisted of two main tasks: Implementing the optimizations in Prusti and creating a continuous integration task which regularly benchmarks verification performance.

Significant efforts went towards the implementation of the snapshot features and mirror functions as described in Chapter 4. As a consequence parts of the "Purifying local variables" optimization were not fully implemented; specifically the translation to SSA, the handling of `old` expressions, the two phase approach and emitting `Snap$T$UnfoldWitness` in method bodies. Due to the incomplete implementation, a sizable percentage of the Prusti test suite causes an error when run with purification enabled, in particular test cases relying on `old` expressions and test cases where the axioms are not instantiated a sufficient number of times because of the missing `Snap$T$UnfoldWitness` calls.

Prusti's source code is hosted on GitHub and uses GitHub Actions for continuous integration (CI). As part of the implementation work, a GitHub Action was written to automatically benchmark Prusti when pull requests are merged. This CI task verifies a set of benchmark cases multiple times, and stores the time measurements in a file. This data can be used to estimate the performance of Prusti, which can then be used to prevent unwanted performance regressions.

It is to be noted that the benchmarks presented here were ran locally, because using the results generated in the CI decreases reproducibility, as we cannot control any external influences, such as the load on the CI system.

5.2 Benchmarks

5.2.1 Methodology

To evaluate the performance of the implemented optimizations, 16 tests from the Prusti test suite were selected.

The time Prusti takes to verify a benchmark case was measured in two ways:

- *total verification time*: The time it takes Prusti to verify the benchmark, including the time it takes to analyze the input and generate the Viper code.
- *Viper-only verification time*: The time spent by Viper verifying the generated code.

First the total verification time was measured directly by warming up a Prusti server instance and verifying the benchmark Rust files against it. The Viper-only verification time was measured by using the generated Viper files from the Prusti server and querying a warmed up ViperServer using `viper_client` to measure the time Viper takes to verify the generated code. The performance is measured in these two ways to see the impact Prusti itself has on the verification time. For running Prusti and ViperServer the same version of Viper was used. The servers were warmed up by running a specific benchmark case for 6 iterations. Afterwards the same, now warmed up, server instance was used to gather the benchmark data presented below, by running each benchmark case for 10 iterations.

The benchmarks where ran on an Intel i7-4790K CPU @ 4.00GHz with 24GiB DDR3 @ 1600 MHz running Fedora 33 (Kernel `5.10.22-200.fc33.x86_64`) on OpenJDK 64-Bit 11.0.10 and using the Prusti commit `508fc6e22f4b41e10b9aed233bdb974bacdda79d` and `viper_client` commit `bb19f104daee74bdc2fd57d75d687e7ae613a662`. The Viper nightly build from the 16. March 2021 with the silicon backend was used.

5.2.2 Results

Figure 5.1 shows the average runtime of each benchmark case. The entire bar represents total verification time while the colored part indicates the Viper-only verification time. The chart suggests that the predicate optimizations have a rather small impact in all cases. The impact of purification is more significant, in nearly all cases it appears to decrease performance by a factor of up to two. It is however notable that purification significantly improves performance for the `Knights_tour` benchmark, where the average total verification time is reduced to a third from 65.024 to 21.499 seconds. This benchmark is of significance, as it has the longest runtime by a significant margin. Similarly, medium-sized benchmarks such as `Binary_search_shared`

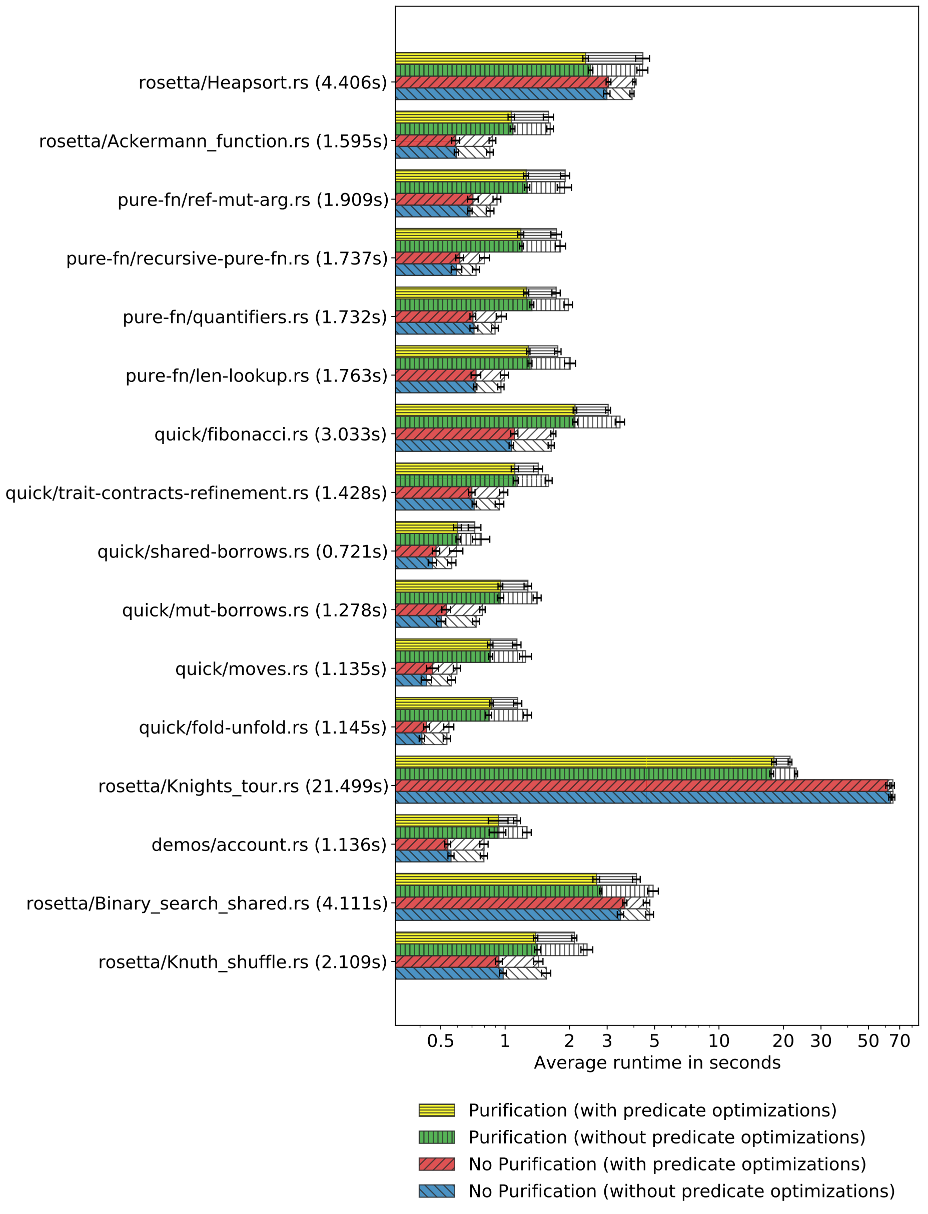

Figure 5.1: Benchmark results for the average runtime over 10 iterations on a log scale. The entire bar represents the total verification time while the colored part indicates the Viper-only verification time. The time listed next to the file name is the total verification time for “Purification (with predicate optimizations)”.

or `Heapsort`, although performing worse when using purification, suffer a smaller performance decrease compared to small benchmarks.

Possible explanations for this behavior are:

- There is an overhead to creating snapshots, mirror functions and performing the purification, which for the smaller benchmarks with runtimes < 10 seconds outweighs the performance gains of purification, while larger and slower benchmarks such as `Knights_tour` significantly benefit from the purification optimization. Additionally the purification optimization is not fully implemented, and as such there are still potentially significant performance improvements to be achieved by fully implementing it.
- In the current implementation both mirror functions and pure functions are emitted for each `#[pure]` function, as mirror functions cannot yet be used in some cases, which causes some overhead. Eventually Prusti can be switched to solely using mirror functions which will allow the pure functions to be removed.

To show the performance benefits of purification a synthetic example was created: Listing 33 shows a Rust function which takes an argument and reassigns it to a variable 70 times. Using the purification optimization the Viper-only verification time improves from an average of 48.209 seconds down to 1.996 seconds. This program exemplifies which problem the current implementation of the purification optimization successfully solves. The non-purified version is slow since a lot of heap-dependent variables have to be kept track of, while the purified version can avoid that.

It is possible that further implementation work on the purification optimization can improve performance to the degree where runtimes are shorter or equal to the non-purified version, even for the small benchmark cases. Even if that is not possible, it might be an acceptable trade-off to increase verification time for programs with verification times of under 2 seconds, if in exchange verification time of programs with long verification times of over 60 seconds can be significantly reduced.

```
pub struct VecWrapperI32{
    v: Vec<i32>
}

impl VecWrapperI32 {
    //...
}

fn test(arr: &VecWrapperI32) -> usize {
    let arr = arr;
    let arr = arr;
    let arr = arr;
    // ... total of 70 repetitions
    let arr = arr;
    arr.len()
}
```

Listing 33: Rust program that reassigns a variable 70 times

Chapter 6

Conclusion

This thesis proposes and implements techniques to improve the performance of the Prusti verifier.

The purification optimization presented is of particular interest, as it shows promising performance results for larger programs, with reductions in execution time by up to two thirds. For small programs the purification optimization decreases the performance, which may be effected by the incomplete implementation. Even under the assumption that the fully implemented purification will slightly decrease performance for small programs, this is an acceptable trade-off in exchange for the decrease in verification time for large programs. The implementation of the purification optimization required extending the functionality of snapshots and introducing mirror functions. This not only enabled the optimization but also allows more functions to be labeled as `#[pure]` in Prusti.

Since not all parts of the purification optimization as described in this thesis were implemented, finishing and extending it is left up to future work. Once the implementation is finished and sufficiently mature the existing pure function can be fully replaced by mirror functions.

Appendix A

Appendix

A.1 Benchmark data

The times in the tables are the total verification time in seconds.

A.1.1 Overview

This table shows a comparison of average total verification time. *pur.* stands for purification optimization while *pred.* stands for predicate optimization.

Filename	neither	pred.	pur.	pur. + pred.
`rosetta/Knuth_shuffle.rs`	1.558	1.432	2.416	2.109
`rosetta/Binary_search_shared.rs`	4.745	4.587	4.920	4.111
`demos/account.rs`	0.796	0.797	1.266	1.136
`rosetta/Knights_tour.rs`	65.135	65.024	22.961	21.499
`quick/fold-unfold.rs`	0.534	0.546	1.275	1.145
`quick/moves.rs`	0.562	0.595	1.247	1.135
`quick/mut-borrows.rs`	0.732	0.784	1.413	1.278
`quick/shared-borrows.rs`	0.563	0.592	0.776	0.721
`quick/trait-contracts-refinement.rs`	0.942	0.985	1.599	1.428
`quick/fibonacci.rs`	1.643	1.682	3.445	3.033
`pure-fn/len-lookup.rs`	0.956	0.993	2.015	1.763
`pure-fn/quantifiers.rs`	0.897	0.960	1.974	1.732
`pure-fn/recursive-pure-fn.rs`	0.731	0.801	1.817	1.737
`pure-fn/ref-mut-arg.rs`	0.851	0.916	1.898	1.909
`rosetta/Ackermann_function.rs`	0.849	0.873	1.624	1.595
`rosetta/Heapsort.rs`	3.922	4.023	4.393	4.406

A.1.2 Purification without predicate optimizations

Filename	Min.	Max.	Avg.
`rosetta/Knuth_shuffle.rs`	2.175	2.695	2.416
`rosetta/Binary_search_shared.rs`	4.546	5.459	4.920
`demos/account.rs`	1.122	1.342	1.266
`rosetta/Knights_tour.rs`	22.172	23.396	22.961
`quick/fold-unfold.rs`	1.198	1.366	1.275
`quick/moves.rs`	1.171	1.411	1.247
`quick/mut-borrows.rs`	1.291	1.496	1.413
`quick/shared-borrows.rs`	0.655	0.922	0.776
`quick/trait-contracts-refinement.rs`	1.505	1.681	1.599
`quick/fibonacci.rs`	3.222	3.720	3.445
`pure-fn/len-lookup.rs`	1.856	2.268	2.015
`pure-fn/quantifiers.rs`	1.840	2.122	1.974
`pure-fn/recursive-pure-fn.rs`	1.638	1.975	1.817
`pure-fn/ref-mut-arg.rs`	1.690	2.149	1.898
`rosetta/Ackermann_function.rs`	1.531	1.699	1.624
`rosetta/Heapsort.rs`	3.978	4.739	4.393

A.1.3 Purification with predicate optimizations

Filename	Min.	Max.	Avg.
`rosetta/Knuth_shuffle.rs`	2.043	2.242	2.109
`rosetta/Binary_search_shared.rs`	3.920	4.584	4.111
`demos/account.rs`	1.072	1.197	1.136
`rosetta/Knights_tour.rs`	20.615	22.158	21.499
`quick/fold-unfold.rs`	1.056	1.230	1.145
`quick/moves.rs`	1.057	1.215	1.135
`quick/mut-borrows.rs`	1.181	1.336	1.278
`quick/shared-borrows.rs`	0.671	0.823	0.721
`quick/trait-contracts-refinement.rs`	1.303	1.507	1.428
`quick/fibonacci.rs`	2.913	3.157	3.033
`pure-fn/len-lookup.rs`	1.650	1.839	1.763
`pure-fn/quantifiers.rs`	1.552	1.832	1.732
`pure-fn/recursive-pure-fn.rs`	1.529	1.906	1.737
`pure-fn/ref-mut-arg.rs`	1.714	2.084	1.909
`rosetta/Ackermann_function.rs`	1.469	1.735	1.595
`rosetta/Heapsort.rs`	3.970	4.882	4.406

A.1.4 No Purification with predicate optimizations

Filename	Min.	Max.	Avg.
`rosetta/Knuth_shuffle.rs`	1.337	1.594	1.432
`rosetta/Binary_search_shared.rs`	4.342	4.946	4.587
`demos/account.rs`	0.752	0.871	0.797
`rosetta/Knights_tour.rs`	62.800	67.303	65.024
`quick/fold-unfold.rs`	0.512	0.604	0.546
`quick/moves.rs`	0.552	0.635	0.595
`quick/mut-borrows.rs`	0.743	0.821	0.784
`quick/shared-borrows.rs`	0.530	0.669	0.592
`quick/trait-contracts-refinement.rs`	0.916	1.064	0.985
`quick/fibonacci.rs`	1.618	1.773	1.682
`pure-fn/len-lookup.rs`	0.887	1.053	0.993
`pure-fn/quantifiers.rs`	0.872	1.044	0.960
`pure-fn/recursive-pure-fn.rs`	0.716	0.869	0.801
`pure-fn/ref-mut-arg.rs`	0.864	0.993	0.916
`rosetta/Ackermann_function.rs`	0.814	0.933	0.873
`rosetta/Heapsort.rs`	3.899	4.156	4.023

A.1.5 No Purification without predicate optimizations

Filename	Min.	Max.	Avg.
`rosetta/Knuth_shuffle.rs`	1.491	1.703	1.558
`rosetta/Binary_search_shared.rs`	4.547	5.116	4.745
`demos/account.rs`	0.759	0.868	0.796
`rosetta/Knights_tour.rs`	62.728	67.747	65.135
`quick/fold-unfold.rs`	0.509	0.580	0.534
`quick/moves.rs`	0.529	0.615	0.562
`quick/mut-borrows.rs`	0.695	0.767	0.732
`quick/shared-borrows.rs`	0.527	0.624	0.563
`quick/trait-contracts-refinement.rs`	0.875	1.043	0.942
`quick/fibonacci.rs`	1.557	1.754	1.643
`pure-fn/len-lookup.rs`	0.905	1.012	0.956
`pure-fn/quantifiers.rs`	0.850	0.957	0.897
`pure-fn/recursive-pure-fn.rs`	0.691	0.776	0.731
`pure-fn/ref-mut-arg.rs`	0.794	0.938	0.851
`rosetta/Ackermann_function.rs`	0.814	0.913	0.849
`rosetta/Heapsort.rs`	3.788	4.073	3.922

Bibliography

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

[2] Dafny Developers. Dafny. `https://github.com/dafny-lang/dafny`. [Online; accessed 15-March-2021].

[3] Julian Dunskus. Developing IDE support for a rust verifier, 2020.

[4] Stefan Heule, Ioannis T Kassios, Peter Müller, and Alexander J Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. *Technical Report / ETH Zurich, Department of Computer Science, 776*, 2012.

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

[6] R. Jung. Two kinds of invariants: Safety and validity. `https://www.ralfj.de/blog/2018/08/22/two-kinds-of-invariants.html`, 2018. [Online; accessed 15-March-2021].

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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):

Optimization of a Viper-Based Verifier

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Arnold	Till

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, 22 Mar 2021	[illegible]

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