Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Supporting Borrows in Specification Functions of a Rust Verifier

Master Thesis

Nicolas Trüssel

September 4, 2019

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

Abstract

Prusti is a Rust verifier based on the Viper verification infrastructure. One of its main limitations is the limited support for shared borrows in specification functions. In this thesis, we present access witnesses—an extension of Viper's permission model that enables us to enrich Prusti's Rust-to-Viper encoding, such that it is now possible to encode specification functions that return shared borrows. While access witnesses were motivated by Rust, they fundamentally improve the expressiveness of Viper and, therefore, offer new encoding possibilities that can be leveraged in Viper frontends for other programming languages as well.

Acknowledgments

I am deeply grateful to my supervisor Vytautas Astrauskas. Without his excellent guidance and support this thesis would not have been possible. Thank you for our interesting and fruitful discussions, your advice and cheerful words when I was stuck once again, and your countless insights and tips over the course of this project. I would also like to thank Professor Müller and his group for the opportunity to work on this topic and for their useful feedback during the design of access witnesses. A special thanks goes to Dr. Malte Schwerhoff for his support during the implementation of access witnesses. His feedback on the implementation proposal and especially his help during debugging were invaluable for the success of this thesis. Another big thank you goes to my friends Maximilian Falkenstein and Jan Veen for their help with proofreading. Finally, I am eternally grateful to my parents and my brothers for their endless support and encouragement.

Contents

Abstract **i**

Acknowledgments **iii**

Contents **v**

1 Introduction **1**

2 Viper **5**

2.1 Intuition 5

2.2 Syntax 6

2.3 Verification 7

2.3.1 Self-Framing Assertions 8

2.4 Predicates 8

2.5 Functions 9

2.6 Quantified Permissions 10

3 Access Witnesses **13**

3.1 Motivation 13

3.2 Intuition 15

3.3 Design 15

3.3.1 Syntax 15

3.3.2 Well-Formedness Condition 16

3.3.3 Program State 16

3.3.4 Semantics 17

3.4 Future Work 18

4 Implementation in a Symbolic Execution Based Verifier **21**

4.1 Background 21

4.1.1 Path Conditions 21

4.1.2 Heap Representation 22

4.2 Access Witness Representation 23
4.3 Producing and Consuming Witnesses 24
4.4 Fold and Unfold 27
4.4.1 Fold Statements 27
4.4.2 Unfold Statements 29
4.5 Joining 30
4.6 Unfolding Expressions 31
4.7 Field Access 33
4.8 Function Verification 33
4.9 Function Axiomatization 33
4.10 Function Evaluation 34

5 Prusti 37
5.1 Background 37
5.2 Type Encoding for Pure Functions 38
5.3 Encoding Function Calls 39

6 Implementation and Evaluation 41
6.1 Challenges With Implementing Snapshot Summarization . . . 41
6.2 Expressiveness of Viper Programs 43
6.2.1 Scenario 43
6.2.2 Witness-Based Encoding 44
6.2.3 Purification 44
6.2.4 Hybrid Approach 46
6.3 Correctness and Performance Tests 48
6.3.1 Silicon 49
6.3.2 Prusti 55

7 Future Work 61
7.1 Supporting Quantified Permissions 61
7.2 Incorporating Magic Wands 61
7.3 Foldable Predicates 62
7.4 Snapshot and Memory Equality 62
7.5 Supporting Access Witnesses in Carbon 62
7.6 Addressing Summarization Incompletenesses 62
7.7 Field Lookups and Unfold Statements 63
7.8 Selective Witness Generation 63
7.9 Calling Pure Functions That Return Shared Borrows from Non-Pure Code 63

8 Conclusion 65

A Why Access Witnesses Must Not Provide Write Access 67

B Symbolic Execution Rules of Utility Methods 69

B.1 `produce'` 69
B.2 `produce-dependent` 70

C Viper Programs from Expressiveness Evaluation 71
C.1 Purified Solution 71
C.2 Hybrid Solution 78

D Additional Plots from Performance Evaluation 83

Bibliography 87

Chapter 1

Introduction

Rust is an emerging systems programming language, aiming to provide a memory safe and type-safe alternative to C and C++. Its language design makes it an interesting target for automated verification: Rust's ownership type system statically enforces restrictions on aliasing which can be leveraged by verification tools to automatically infer essential information about the program that one otherwise would have to provide manually. This significantly reduces the complexity of the verification process [1].

The information that is inferrable from Rust's ownership type system allows verification tools to determine which memory locations may be accessed by a piece of code (for example a function call) and which memory locations cannot be accessed. The problem of determining these memory locations is called the *framing* problem and is one of the main challenges of modular verification.

However, just relying on ownership information and static types is not enough to determine whether a memory location can be accessed because it could be temporarily borrowed. The Rust compiler forbids any access that conflicts with an active borrow as illustrated in Figure 1.1. The technique described in [1] therefore uses *capabilities* to keep track of the access rights to memory locations and to model the Rust compiler's internal representation of this information. Capabilities are computed for each program point and subsequently used for framing.

Prusti—an implementation of the technique described in [1]—translates Rust programs to Viper [2], an intermediate verification language. Viper's logic is based on implicit dynamic frames [3] and uses so-called *permissions* to solve the framing problem. Similar to Rust, where a memory location can be accessed if a reference has the necessary capability, memory locations in Viper can be accessed if one holds the required *permission* to do so. Unlike Rust, Viper's permission accounting is much more precise; operations that are performed implicitly in Rust are explicit operations in Viper code. Moreover,

```
1  struct Pair {
2    fst: i32,
3    snd: i32,
4  }
5
6  fn foo() {
7    let mut p = Pair { fst: 0, snd: 1 };
8    let x = &mut p;
9    p.fst = 2;
10   x.snd = 1;
11 }
```

Figure 1.1: An example Rust program that illustrates why ownership information does not suffice to determine whether a memory access is legal. Even tough `p` owns the created `Pair`, the assignment on line 9 causes a compile error: Since `x` mutably borrows `p`, the read and write permissions are temporarily transferred to `x`. As long as `x` is active, `p` cannot be used to access the underlying `Pair`.

while a shared reference capability in Rust can be duplicated, Viper's permissions are non-duplicable; instead, they can be split and recombined. Therefore, figuring out which Viper operations to emit is an extremely challenging task in some cases, for instance when a borrow expires.

This mismatch between the precise permission accounting in Viper and the more relaxed model used in Rust complicates the verification process. It is also the cause of some of the limitations the approach from [1] currently suffers from.

We developed an extension of Viper's permission model to make it more suitable for modeling Rust code. More specifically, we address the issue that so called *pure* functions cannot return non-primitive types. A pure function is a function that is both deterministic and side-effect free, and can therefore be used in specifications. To allow their usage in specifications, Prusti translates pure functions to Viper functions (see Section 2.5 for an explanation of Viper functions). However, Viper's permission model does not allow functions to return permissions, while this is allowed in Rust. This fundamental difference between the two models is what previously prevented Prusti from supporting non-primitive return types in pure functions.

Our extension—access witnesses—overcomes this limitation and allows Prusti to support shared borrows as return values of pure functions. For Viper itself, it improves the expressiveness of the overall language by allowing functions to make assertions about the permissions of their result. As we will see, this also allows chaining heap-dependent functions—something that was previously impossible in Viper.

More precisely, our contributions are:

1. The design of a new feature for the Viper language.
2. The implementation of this feature in Silicon, a Viper verifier based on symbolic execution.
3. The extension of Prusti to allow pure functions to return shared borrows.
4. An evaluation discussing the impact of access witnesses on the expressiveness of Viper.
5. A performance analysis of the updated versions of both Silicon and Prusti.

After providing the necessary background information about Viper in Chapter 2, Chapter 3 will introduce the design of access witnesses. In Chapter 4 we will describe how to implement them in a verification framework based on symbolic execution, followed by an explanation of the required changes in Prusti's encoding of pure Rust functions to Viper such that shared references can be used as return values in Chapter 5. An overview about implementation issues in a real-world Viper verifier and the evaluation of both the design and its implementation is presented in Chapter 6, before we provide more details about potential future work in Chapter 7 and conclude this thesis in Chapter 8.

Chapter 2

Viper

Viper is an intermediate verification language, designed to support fully automated verification of concurrent, heap manipulating programs. It features a global heap, as well as well as an assertion language based on implicit dynamic frames [3]. This chapter is based on the online tutorial on Viper [4] and provides the necessary details about the Viper language required to understand the remainder of this thesis. Many details are omitted and can be found in either the original paper on Viper [2] or the online tutorial.

2.1 Intuition

As previously mentioned, Viper programs have access to a heap. This heap is *object based*, which means that each heap location is uniquely identified by a tuple consisting of a reference and a field identifier. All references have access to every field that is declared in a Viper program; there is no distinction between different kinds of references that have access to different fields.

Access to heap locations is guarded by *access permissions*: to read from or write to a heap location, one needs to hold the required permission to do so. Permissions allow Viper verifiers to exactly determine which memory locations may be read or modified by each piece of code, thereby solving the framing problem for Viper programs. An access permission is a tuple of a heap location and a *permission amount*. A permission amount is a fractional, non-negative number. Permission handling is explicit; that is, permissions have to be added and removed explicitly using the corresponding statements.

To read from a heap location, one needs to hold a permission for said location with a permission amount that is strictly positive. Write access requires the permission amount to be equal to 1, which is denoted by the constant `write` in Viper programs. Having a total permission amount that is greater than 1 means that the Viper program is in an inconsistent state. As a consequence, holding a non-zero permission amount to any heap location means it cannot be modified concurrently (since otherwise the total permission amount would be greater than 1). Dropping all permissions to a memory location means that it may be arbitrarily modified and thus all knowledge about said memory location is havocked if the permission amount drops to zero.

The program state of a Viper program consists of three components:

1. The value of each variable in the current scope. The current scope includes all local variables, as well as arguments and return parameters.
2. The heap, which stores the value of each field location that is currently accessible.
3. The permission mask, which stores how much permission to which resource (heap location) is currently held.

2.2 Syntax

As we have seen, all fields are available to all references. References are variables of the built-in `Ref` type. The field declaration `field f: T` declares a new field with identifier `f` and type `T`. Besides field declarations, Viper programs may also contain predicate declarations (predicates are explained in Section 2.4), methods, and functions (which will be explained in Section 2.5).

To denote permissions in a Viper program, *accessibility predicates* are used. They have the shape `acc(r.f, p)`, where `r.f` denotes a heap location (here the field `f` of reference `r`) and `p` denotes a permission amount. We have already seen that `write` can be used to denote a permission amount of 1. It is also possible to specify fractional permission amounts, such as `write / 2` or an unspecified, strictly positive permission amount denoted by `wildcard`. To add a permission to the program state, `inhale` statements can be used. Removing permissions can be done by the means of `exhale` statements.

However, inhale and exhale statements do not just add and remove permissions from the program state. It is possible to inhale and exhale arbitrary *assertions*. An assertion is a boolean expression which may contain accessibility predicates. Inhaling an assertion a adds all permissions that occur in a to the program state. Additionally, it is assumed that a currently holds (evaluates to true). Exhaling a checks whether a holds and removes all permissions that occur in a from the program state. In contrast to `exhale`, `assert` merely checks whether the given assertion currently holds, but does not remove any permissions from the program state. The basic use of permissions is demonstrated in Figure 2.1.

```
1 inhale acc(r.f, write)
2 r.f := 1                    // Assignment needs write permission
3 exhale acc(r.f, write / 2)
4 assert r.f == 1             // Reading is still allowed
5 exhale acc(r.f, write / 2)
6 inhale acc(r.f, write)
7 assert r.f == 1             // Fails
```

Figure 2.1: Basic permission usage in Viper programs. The `assert` statement on line 7 fails, because we drop all of the remaining permission amount to `r.f` on line 5. The previous `assert` statement succeeds, since we only dropped a fraction of the permission.

2.3 Verification

Each method in a Viper program is equipped with a *specification*, which is composed of a precondition and a postcondition. Verifying a Viper program means proving that each method implementation adheres to the method's specification; that is, if the method is executed starting in an arbitrary program state that satisfies the method's precondition, the method's postcondition holds after the execution of the body. Essentially, verifying a method

```
method m()
  requires pre
  ensures post
{
  <body>
}
```

is equivalent to the verification of the following code block starting in an empty program state.

```
inhale pre
<body>
exhale post
```

As one may have noticed, Viper's program state does not include a call stack. This is because the verification of Viper programs is *method-modular*, meaning that each method is verified independently. Method calls are not executed during verification; instead each method call is simulated by exhaling the method's precondition and then inhaling its postcondition on the caller side. Exhaling the precondition means that we check whether the precondition of the called method holds. Inhaling the postcondition means that the simulated execution of the method call now guarantees that the method's postcondition holds.

This allows Viper programs to contain *abstract* methods. Such methods only have a specification, but no implementation. As a consequence, they are not

```
field value: Int
field next: Ref

predicate list(r: Ref) {
  acc(r.value, write) && acc(r.next, write)
    && (r.next != null ==> acc(list(r.next), write))
}
```

Figure 2.2: A Viper predicate for a linked list with integer values.

verified, but still can be used in other methods, since method calls are only simulated.

Method pre- and postconditions may contain accessibility predicates. Such accessibility predicates model a permission transfer. For accessibility predicates in the precondition, said permissions are passed to the called method. Accessibility predicates in the postcondition specify that the method returns some permissions to its caller.

2.3.1 Self-Framing Assertions

Precondition and postcondition assertions are required to be *self-framing*. An assertion is self-framing if it contains all permissions that are required to evaluate it; that is, if an assertion wants to read a field `r.f`, it must also contain an accessibility predicate for `r.f`. Thus the assertion `r.f == 1` is not self-framing, while `acc(r.f, write) && r.f == 1` is. Since assertions are evaluated from left to right, the accessibility predicate must occur on the left hand side the heap access.

2.4 Predicates

So far, we have only seen field permissions. Given the primitives presented up to here, it would not be possible to represent (statically) unbounded heap structures, such as lists. To overcome this issue, Viper supports *predicates*. A predicate binds a name to an assertion (the predicate body), and allows it to have parameters. Furthermore, predicates may be recursive. It is this property that enables the modeling of unbounded heap structures. Like assertions in specifications, the body of a predicate must be self-framing. An example of a predicate declaration is shown in Figure 2.2.

`P(`e_1`, ..., `e_n`)` is called an *instance* of predicate `P`. In addition to field permissions, predicate instances are another resource type that is supported by Viper. Like field permissions, they can be inhaled and exhaled. Viper treats predicates *iso-recursively*, which means that a predicate instance is not treated as if it was equal to its body. Instead, one has to explicitly convert between

```
1  field first: Int
2  field second: Int
3
4  predicate pair(r: Ref) {
5    acc(r.first, write) && acc(r.second, write)
6  }
7
8  method example() {
9    var r: Ref
10   inhale pair(r)
11   r.first := 1 // Fails
12   unfold pair(r)
13   r.first := 2
14   assert acc(pair(r), write) // Fails
15   fold pair(r)
16   assert acc(pair(r), write)
17 }
```

Figure 2.3: A Viper program illustrating the effects of folding and unfolding predicate instances. The assignment on line 11 fails, since the permission to `r.first` is not directly available. After unfolding the predicate, the assignment succeeds (line 13). However, now we don't have access to the the predicate instance anymore, as it was unfolded (line 14). After folding it again, the same assertion succeeds.

the two using `fold` and `unfold` statements. The former converts the predicate body to a predicate instance, the latter does the opposite. Figure 2.3 illustrates this process.

If one only wants to temporarily unfold a predicate instance, for example to read from a field whose accessibility predicate is in the body of a predicate, one can also use `unfolding` expressions, which temporarily unfold a predicate instance and then allow the evaluation of a single expression in this state before undoing the unfold operation.

2.5 Functions

Aside from methods, Viper also supports *functions*. A function models a deterministic, side-effect free method. Like a predicate binds a name to an assertion, a function binds a name to an expression. Functions may have parameters, contain recursive function calls, and are equipped with a precondition and a postcondition. While function preconditions may contain accessibility predicates (such functions are called *heap-dependent* functions), function postconditions must not contain any accessibility predicates. The reason is, that functions cannot change the program state, and therefore also cannot change any permissions.

```
function length(r: Ref): Int
  requires acc(list(r), write)
  ensures result > 0
{
  unfolding acc(list(r), write) in
    r.next == null ? 1 : 1 + length(r.next)
}

function n_th(r: Ref, n: Int): Int
  requires acc(list(r), write) &&
             0 <= n && n < length(r)
{
  unfolding acc(list(r), write) in
    n == 0 ? r.item : n_th(r.next, n - 1)
}
```

Figure 2.4: Two heap-dependent Viper functions expressing properties of the list structure defined in Figure 2.2. The first function computes the length of a given list, while the second function returns the list element at a given position.

The semantics of a function call are also different to a method call: Instead of treating function calls solely by their specification, function calls are unrolled; that is, a function call is replaced by its implementation. Before unrolling, the function precondition is *asserted*, hence we only check whether the precondition holds but do not remove any permissions from the program state. After unrolling, the function postcondition is inhaled, but since it must not contain any accessibility predicates, no permissions are added to the program state.

Functions are essential to reason about unbounded heap structures. Figure 2.4 shows some examples of heap-dependent functions that express properties of the list structure defined in Figure 2.2.

2.6 Quantified Permissions

Viper also supports universal (and existential) quantification. Quantified expressions are essential to express and check certain properties of Viper programs. For example, one may want to check whether all list elements are positive, which can be expressed using the following assertion (the definitions of the used functions can be found in Figure 2.4):

```
assert forall i: Int :: { n_th(r, i) } 0 <= i && i < length(r)
                  ==> n_th(r, i) > 0
```

`{ n_th(r, i) }` is called a *trigger*. Triggers are used to control when the body of a quantifier is instantiated. In the given example, this happens whenever `n_th` is called with `r` as its first argument. More information about triggers can be found in [5].

Viper's universal quantifier also offer an additional way to model statically unbounded heap structures called *quantified permissions*. Quantified permissions are essentially just accessibility predicates that occur in the body of a universal quantifier. They are typically used when the modeled heap structure does not have a simple recursive structure. A quantified permission that describes write access to field `f` of each element in a set `s` looks like this:

```
forall x: Ref :: { x.f } x in s ==> acc(x.f, write)
```

Chapter 3

Access Witnesses

Now that we have gained the necessary background knowledge about Viper, we are ready to introduce our extension to Viper's permission model, called *access witnesses*. This chapter will explain the motivation, design considerations, intuition, and finally the detailed syntax and semantics of this new concept.

3.1 Motivation

As we have seen, to access a heap location in Viper, one needs to hold the corresponding permission. We also learned that Viper treats predicates iso-recursively and thus distinguishes between a predicate instance and its body. This means, that a permission might be available in the current program state, but not directly, because it is hidden inside the body of a predicate instance. To obtain this permission, one has to explicitly unfold that predicate instance.

It is Viper's iso-recursive predicate treatment that makes it impossible to chain heap-dependent functions beyond trivial cases. Figure 3.1 illustrates this issue with an example. Besides limiting Viper itself, not being able to chain heap-dependent functions is the main obstacle preventing the implementation of `pure` methods that return shared references in Prusti, the primary goal of this thesis.

The reason why chaining heap-dependent functions is impossible, is that the outer function application usually requires the permissions to be in a different shape than the nested function application. Since function applications cannot modify the program state, they also cannot modify the shape of permissions. In few cases, the shape of the permissions can be modified using `unfolding` expressions; however, this is only possible if one can statically determine how the permissions have to be unfolded. For recursive and abstract

```
 1  field x: Int
 2  field y: Int
 3  predicate point(r: Ref) { acc(r.x, write) && acc(r.y, write) }
 4
 5  field container_value: Ref
 6  predicate container(r: Ref) {
 7    acc(r.container_value, write) &&
 8      acc(point(r.container_value), write)
 9  }
10
11
12  function get_internal_value(r: Ref): Ref
13    requires acc(container(r), write)
14  {
15    unfolding acc(container(r), write) in r.container_value
16  }
17
18  function get_x(r: Ref): Int
19    requires acc(point(r), write)
20  {
21    unfolding acc(point(r), write) in r.x
22  }
23
24  method example() {
25    var p: Ref
26    inhale acc(p.x, write) && acc(p.y, write)
27    p.x := 0
28    p.y := 1
29    fold acc(point(p), write)
30
31    var c: Ref
32    inhale acc(c.container_value, write)
33    c.container_value := p
34    fold acc(container(c), write)
35
36    assert get_x(get_internal_value(c)) == 0
37  }
```

Figure 3.1: A Viper program demonstrating why chaining heap-dependent functions is not possible. The predicates `point` and `container` are used to model heap allocated two-dimensional points and a container which wraps a single point. The functions `get_internal_value` and `get_x` model getter functions for the respective fields of containers and points. Due to the way Viper handles permissions, the `assert` statement on line 36 fails: The current program state only contains permission to the predicate instance `container(c)`. To apply function `get_x`, we would need to have permission to `point(c.container_value)`.

functions this is not possible. Our solution to address this limitation is to introduce a new means to talk about permissions: *access witnesses*.

3.2 Intuition

There are many situations where we actually know that we hold some permission, but it is currently not accessible because it is in an unsuitable shape. Access witnesses allow Viper programs to reason about such permissions: They specify that some permission `acc(l, wildcard)` is currently held, but not directly accessible. The current design limits access witnesses to express that `acc(l, wildcard)` is obtainable through a sequence of unfold operations.

Holding an access witness for `p` allows one to read from the memory location guarded by `p`. Write accesses are not allowed, since `wildcard` permissions do not allow write access either. Note that it is unsound to allow write accesses to memory locations for which one does not directly own permission. The reason is that there could be unknown constraints associated with the memory location which are not directly visible in the current program state. Furthermore, write access to such memory locations could also be abused to create cycles in recursively defined datastructures, breaking some of the fundamental assumptions behind Viper's permission model. Compare Appendix A for more details.

Since the permission `p` an access witnesses represents is not directly available, each access witness has a set of *dependencies*. The dependencies specify from which resources `p` can be obtained (which predicate instance has to be unfolded to obtain `p`). Dependencies can be provided by accessibility predicates with non-zero permission or access witnesses. However, all dependencies have to be eventually backed by permissions that are available directly in the program state. Access witnesses expire as soon as one of their dependencies cannot be provided by the program state anymore, since we can no longer guarantee that `p` is being held.

We allow access witnesses to occur in all assertions, and additionally also in function postconditions. This makes it possible for functions to specify which permissions are associated with their return values. By using access witnesses in function specifications, we can achieve our initial goal of chaining heap-dependent function applications. We will demonstrate this in Figure 3.2 after explaining both the syntax and semantics of access witnesses.

3.3 Design

3.3.1 Syntax

There are several ways to denote access witnesses in Viper programs.

- `dep(r, p(a))` denotes that `r` is obtainable from `p(a)`.
- `dep(r,` p_1`(`a_1`),` p_2`(`a_2`), ...)` is a generalization of the former case and denotes that `r` can be obtained from one of the given dependencies $p_i(a_i)$, but it is not statically known which one.
- `dep(r)` states that it is possible to obtain `r`, but we do not know on which accessibility predicates it is dependent on. As a consequence it is treated as if it was dependent on all permissions that are available in the current program state. This kind of access witnesses is required for function preconditions.

Note that $p_i(a_i)$ not necessarily denotes a predicate instance, but can also refer to a field permission, where we denote `a.f` as `f(a)`. All three forms of access witnesses can be used in pre- and postconditions, unfolding expressions, as well as inhale and exhale statements.

3.3.2 Well-Formedness Condition

Similar to heap accesses, which are only valid if the program state contains the necessary permissions, an access witness is only valid if the program state contains its dependencies. As previously mentioned, dependencies can be provided by either accessibility predicates or other access witnesses. For assertions that are required to be self-framing, all dependencies of access witnesses have to be provided by either accessibility predicates or access witnesses occurring within the same assertion. If an access witness is inhaled, exhaled, or asserted, it suffices if its dependencies are provided by the program state.

3.3.3 Program State

To support access witnesses, we extend Viper's program state. In addition to the heap, the permission mask, and the values of all variables, the program state also consists of all access witnesses. Each access witness is stored as a tuple (resource, dependencies). If an accessibility predicate is completely exhaled or the corresponding memory memory is modified (for example by a field assignment), all access witnesses that are directly or indirectly dependent on said accessibility predicate are removed from the program state.

Note that the introduction of access witnesses means, that it is no longer true that evaluating an expression does not affect the program state. By evaluating function applications and unfolding expressions, new access witnesses may be added to the current program state. However, no other parts of the program state may change during expression evaluation.

3.3.4 Semantics

Now that we described the syntax and the new program state, we explain the semantics of access witnesses.

Expiration

After every operation that removes permissions from the program state, we check whether there are access witnesses with dependencies that can no longer be provided by the updated program state. Such access witnesses are removed from the new program state.

Inhale

Inhaling an access witness first checks whether the dependencies are currently available and then adds it to the program state. If no dependencies are provided (`dep(r)`), all currently available permissions are recorded as dependencies of the created witness.

Exhale

Exhaling an access witness `dep(r,` $p_1(a_1)$`,` $p_2(a_2)$`, ...)` checks whether there is an access witness for `r` for which the given set of dependencies is sufficient to determine when it expires. For example, this can be checked by removing all vertices representing $p_i(a_i)$ from the dependency graph and then checking whether it is possible to reach a permission from the access witness for `r`. In case this is impossible, the given set of dependencies is sufficient and the exhale statement succeeds.

If no dependencies are provided we simply check whether such an access witness exists. Since access witnesses are not resources, they are not removed from the program state when they are exhaled, which means that it is possible to exhale them repeatedly.

Heap Lookup

Reading from a heap location no longer necessarily requires that there is some positive permission amount to said heap location. In case one does not have enough permission to read a heap location, we additionally check whether there is an access witness for said memory location. If this is the case, the heap location can be accessed.

Fold

Access witnesses cannot be folded. However, their introduction requires a slight modification of the semantics for normal fold statements: folding a predicate instance creates access witnesses for every permission occurring in the its body. The created witnesses are dependent on the folded predicate instance.

Unfold

While access witnesses cannot be folded, it is possible to unfold an access witness for a predicate instance. Doing so creates access witnesses for all permissions that occur in the body of the unfolded predicate instance. The created access witnesses are dependent on the unfolded predicate instance. The witness for the unfolded predicate instance remains in the program state.

Furthermore, the semantics of unfolding a normal permission have to be modified. Permanently unfolding a predicate instance `p(a)` causes a modification of all access witnesses that are dependent on it. Since the predicate instance is consumed, the dependent access witnesses might get removed from the program state (as the dependency is missing). However, unfolding does not really consume the dependency, it only exchanges it for something equivalent. Thus the access witnesses can be preserved. We update the dependencies of all access witnesses that are dependent on `p(a)` as follows: Let `bdy` denote the set of accessibility predicates that occur in the body of `p(a)`. We remove `p(a)` as a dependency for all access witnesses, and replace it by `bdy`. If the resource represented by an access witness occurs in `bdy`, the access witness is deleted. This modification allows us to preserve access witnesses across repeated fold and unfold operations.

Unfolding

Using access witnesses for predicates in unfolding expressions actually does exactly the same as if the access witness was unfolded by an unfold statement: It creates access witnesses for all resources that occur in the predicate body and then the evaluates the given expression. Being able to use access witnesses in unfolding expressions is an important cornerstone of supporting function chaining.

3.4 Future Work

The current design does not support quantified permissions. There are two different ways quantified permissions and quantifiers interact with access witnesses: Access witnesses may be dependent on a quantified permission, or they may occur within the body of a quantifier. The former case happens if we unfold a predicate instance whose body contains a quantified permission. We will see how the current implementation handles this case in Section 4.6. Access witnesses that occur within the body of a quantifier (*quantified access*

```
field x: Int
field y: Int
predicate point(r: Ref) { acc(r.x, write) && acc(r.y, write) }

field container_value: Ref
predicate container(r: Ref) {
  acc(r.container_value, write) &&
    acc(point(r.container_value), write)
}


function get_internal_value(r: Ref): Ref
  requires dep(container(r))
  ensures dep(point(result), container(r))
{
  unfolding dep(container(r)) in r.container_value
}

function get_x(r: Ref): Int
  requires dep(point(r))
{
  unfolding point(r) in r.x
}

method example() {
  var p: Ref
  inhale acc(p.x, write) && acc(p.y, write)
  p.x := 0
  p.y := 1
  fold acc(point(p), write)

  var c: Ref
  inhale acc(c.container_value, write)
  c.container_value := p
  fold acc(container(c), write)

  assert get_x(get_internal_value(c)) == 0
}
```

Figure 3.2: A modification of the Viper program shown in Figure 3.1, leveraging access witnesses to achieve the original goal of chaining heap-dependent functions. The access witness returned by function `get_internal_value` satisfies the precondition of function `get_x`, leading to a successful verification of the given program.

witnesses) naturally occur in lifted functions, for example functions that apply a single function to an entire list or array.

Besides designing the syntax for both quantified access witnesses and witnesses that are dependent on other quantified permissions, extending access witnesses with support for quantified permissions also requires addressing other open questions. For example, one has to decide how quantified access witnesses should expire. There are at least two approaches here: One could try to model fine grained expiration, which precisely tracks which parts of a quantified witness are dependent on which resources and only expires quantified witnesses partially. This requires designing a sophisticated mechanism that allows them to track which parts are dependent on which resources. Alternatively, one could chose a more coarse grained approach in which the entire quantified witness expires once a dependency of an item is missing. Furthermore, one has to design a syntax for both quantified witnesses and witnesses that are dependent on a quantified permission.

Chapter 4

Implementation in a Symbolic Execution Based Verifier

In this chapter, we describe how to implement access witnesses in a Viper verifier. We focus on Viper's symbolic execution based verifier, which was originally described in [6], and provide new and updated symbolic execution rules that extend it with support for access witnesses.

4.1 Background

We assume some familiarity with the original description in [6]. Nevertheless, the necessary context and background will be explained as we present the updated symbolic execution rules. This background section (Section 4.1) is paraphrased from the original description.

Viper's program state is modeled as a symbolic program state which will be denoted as σ. σ consists of

- a symbolic store γ. γ maps variables to symbolic values.
- a *path condition stack* π. Path conditions record constraints about the symbolic state (for example assumptions in a Viper program).
- a symbolic heap h. It is organized in *chunks* and is used to store both values and permission amounts for fields and predicate instances.

4.1.1 Path Conditions

Path conditions are stored as a stack of *path condition scopes*. Each scope is a triple consisting of a unique identifier called scope identifier, a branch condition, and a set of path conditions. The scope identifiers can be used to determine what branch conditions and path conditions were discovered between two points during symbolic execution. Branch conditions record why a certain branch was taken and typically stem from `if` conditions. Path conditions, on the other hand, record all other constraints that were learned.

There are multiple utility functions that allow querying and manipulating the path condition stack:

- `pc-add`(π, v) adds assumption v to the current top scope of path condition stack π and returns the updated path condition stack.
- `pc-push`(π, id, c) pushes a new scope with identifier *id* and branch condition c to the path condition stack π and returns the updated path condition stack.
- `pc-after`(π, id) returns all scopes that have been pushed to π after the scope with identifier *id* has been pushed (including the scope with identifier *id*).
- `pc-segs`(π) returns a summary of all branch conditions and path conditions in π.

4.1.2 Heap Representation

As already mentioned, the symbolic heap h is organized in chunks. A heap chunk has the shape $id(\overline{v};\overline{w})$, where *id* uniquely identifies a resource (a field or a predicate), $\overline{v}$ are the input arguments, and $\overline{w}$ are the output arguments. Overlined variables are used to denote lists. If *id* denotes a field, the corresponding chunk is called a *field chunk*, if *id* denotes a predicate, it is called a *predicate chunk*. For field chunks, the input argument list $\overline{v}$ is a singleton list consisting of the receiver (the symbolic value of the reference `r` in the heap access `r.id`). For predicate chunks, $\overline{v}$ is the symbolic argument list of the predicate instance.

The output argument list $\overline{w}$ is always composed of two elements, a *snapshot* and a *permission value*. The permission value stores the permission amount to the represented memory location, while the snapshot stores its value. Occasionally, we will denote field chunks as $id(r; s, p)$ and predicate chunks as $id(\overline{args}; s, p)$, where r denotes the receiver, $\overline{args}$ denotes the predicate arguments, s denotes the snapshot, and p denotes the permission amount.

Besides the chunks we already presented, there is a second kind of heap chunks called *quantified chunks*. They are a generalization of the chunks we previously presented and are used to represent quantified permissions. Quantified chunks therefore typically represent multiple heap locations. To be able to do so, their output arguments consist of a *snapshot map* and a *permission function*, the input arguments $\overline{v}$ are obsolete. Instead, the permission function p maps the field receiver (in case of a quantified field chunk) or the predicate arguments (for quantified predicate chunks) to a permission value $p(\overline{args})$.

The snapshot map sm maps the same arguments to a snapshot $sm(\overline{args})$ if $p(\overline{args})$ is non-zero. Since every non-quantified permission

```
acc(r.f, p)
```

can be replaced by a quantified permission

```
forall x: Ref :: x == r ==> acc(x.f, p)
```

we assume that all heap chunks are quantified chunks. This simplifies the symbolic execution rules presented in this chapter.

4.2 Access Witness Representation

To represent access witnesses in the state of the symbolic verifier, we add a new type of heap chunks: *witness chunks*. They have the same basic structure as other chunks. To be able to distinguish between witness chunks and normal chunks, the identifier id of witness chunks is uniquely marked ($\hat{id}$). Thus they have the basic shape $\hat{id}(\overline{v};\overline{w})$. The output arguments $\overline{w}$ of witness chunks are identical to the ones of non-quantified chunks, consisting of a snapshot and a permission value. The input arguments $\overline{v}$ start with the same entries that a corresponding non-quantified chunk would contain: the predicate arguments or the field receiver. These entries are followed by a list of dependencies (parents). Each dependency is a tuple $(id, \overline{args})$ which uniquely identifies another resource ($\overline{args}$ corresponds to the input arguments for the resource, while id is its identifier). Thus field witnesses have the shape $\hat{id}(r, \overline{parents}; s, p)$ and predicate witnesses have the shape $\hat{id}(\overline{args}, \overline{parents}; s, p)$.

Even though access witnesses do not have a permission amount on the Viper level, our representation still stores a permission amount in the chunks for witnesses to provide a uniform interface for all chunks. This significantly reduces the number of symbolic execution rules that have to be updated. The actual permission amount is irrelevant, as we only care whether a witness exists in the current program state or not. This can be checked by testing whether the symbolic permission amount p is positive.

To access the specific parts of a generic chunk c we will use the following utility functions:

- id(c) returns the chunk identifier. Note that it ignores the witness chunk marker for its result.
- is_witness(c) returns whether the chunk is a witness chunk.
- arguments(c) is only defined for witness chunks and returns a singleton list containing the receiver for field witness chunks, and the list of predicate arguments for predicate chunks.

- parents(c) is only defined for witness chunks and returns the list of dependencies.
- perm$(c)(\overline{args})$ returns the permission value stored by c for arguments $\overline{args}$. For quantified chunks, this corresponds to a lookup in the permission value function of c. For witness chunks, the result corresponds to the expression $ite(\overline{args} = \text{arguments}(c), p, 0)$, where p is the permission value of chunk c. ite denotes the symbolic conditional expression.
- snap$(c)(\overline{args})$ returns the snapshot stored by c for arguments $\overline{args}$. It is only defined if $perm(c)(\overline{args}) > 0$ and evaluates to the stored snapshot value for witness chunks and to the snapshot map lookup $sm(\overline{args})$ for quantified chunks.

The symbolic execution rules make sure the following invariant holds for all witness chunks *after* the execution of each statement: Whenever the permission value of a witness chunk is non-zero, the symbolic heap also holds non-zero permission to every dependency of this witness chunk. This is necessary to make sure no witness allows access to a memory location to which we do not have permission. Figure 4.1 shows the symbolic execution rule that removes all witnesses for which this invariant no longer holds. The utility function `check(`π`,`$cond$`)` tests whether a given symbolic boolean expression $cond$ holds, given the path conditions π.

4.3 Producing and Consuming Witnesses

In Silicon, chunks are created and removed by two symbolic execution rules called *produce* and *consume*, respectively. Now that we know how access witnesses are modeled, we are ready to describe how to extend and update these rules to accommodate access witnesses. First note that many of the symbolic execution rules use a continuation passing style with Q denoting the continuation.

The rule for producing access witnesses is stated in Figure 4.2 and very similar to the rule for producing non-quantified accessibility predicates that is presented in [6]. To establish the aforementioned invariant for witness chunks, we check whether the permission amount for each dependency is positive prior to creating the witness chunk.

The used `eval` rule is an appropriately lifted version of the original one and symbolically evaluates a list of lists. `assert(`π`, p)` checks whether the given predicate p holds under the path conditions in π and aborts the symbolic execution in case the check fails. Note that the new heap h_3 is derived from $\sigma_1.h$ and not from $\sigma_2.h$, since the evaluation of the predicate arguments may produce additional witnesses as an unwanted side effect.

```
remove-expired-witnesses($\sigma$) =
  $witnesses := \{c \mid \in \sigma.h \wedge \text{is\_witness}(c)\}$
  $alive := \sigma.h - witnesses$
  $updated :=$ true
  while $updated$
    $updated :=$ false
    foreach $c \in witnesses$
      $p_c := \text{perm}(c)(\text{arguments}(c))$
      if check($\sigma.\pi, p_c = 0$) then
        $witnesses := witnesses \setminus \{c\}$
      else
        $condition :=$ true
        foreach $(id, \overline{args}) \in \text{parents}(c)$
          $r := \{x \mid x \in alive \wedge \text{id}(x) = id\}$
          $P := \sum_{x \in r} \text{perm}(x)(\overline{args})$
          $condition := condition \wedge (0 < p_c \Rightarrow 0 < P)$
        if check($\sigma.\pi, condition$) then
          $updated :=$ true
          $witnesses := witnesses \setminus \{c\}$
          $alive := alive \cup \{c\}$
  $\sigma\{h := alive\}$
```

Figure 4.1: Symbolic execution rule to remove all witnesses that expired. It reestablishes the invariant for witness chunks.

```
produce($\sigma_1$,dep($p(\overline{a}), \overline{q(\overline{b})}), s, Q) =$
  eval($\sigma_1$,$\overline{a} :: \overline{\overline{b}}, \lambda\ \sigma_2, \overline{a'} :: \overline{\overline{b'}}\ \cdot$
    Let $v$ be $e' \neq null$ if $p$ denotes a field, and $true$ otherwise
    $parents := \varnothing$
    foreach $id(\overline{args'}) \in \overline{q(\overline{b'})}$
      $chunks := \{c \mid c \in \sigma_1.h \wedge \text{id}(c) = id\}$
      $perm := \sum_{c \in chunks} \text{perm}(c)(\overline{args'})$
      assert($\sigma_2.\pi, perm > 0$)
      $parents := parents \cup \{(\text{id}(c), \overline{args'}) \mid c \in chunks\}$
    $c := p(\overline{a'}, parents, s, 1)$
    $h_3 := \sigma_1.h \cup \{c\}$
    $Q(\sigma_2\{h := h_3\}))$
```

Figure 4.2: The new symbolic execution rule to produce witnesses.

```
consume'($\sigma_1$,$h$,dep($p(\overline{a})$,$\overline{q(\overline{b})}$),$Q$) =
  eval($\sigma_1$,$\overline{a} :: \overline{\overline{b}}$,$\lambda\ \sigma_2$,$\overline{a'} :: \overline{\overline{b'}}$ ·
    $chunks :=$ dependent-on($\sigma_2$,$\{c \mid c \in h \wedge \mathrm{id}(c) = p\}$,$\overline{q(\overline{b'})}$)
    $snap, snap_{def}, perm :=$ summarise-chunks($chunks$,$\overline{a'}$)
    assert($\sigma_2.\pi$,$0 < perm$)
    $\pi_3 :=$ pc-add($\sigma_2.\pi$,$snap_{def}$)
    $Q(\sigma_1\{\pi := \pi_3\}, h, snap)$)
```

Figure 4.3: The new symbolic execution rule to consume witnesses.

Note that producing an access witness with no dependencies leads to the creation of a witness chunk with an empty dependency list, meaning that it will never expire. As a consequence, we prohibit access witnesses without dependencies from occurring in inhale statements, method preconditions and method postconditions. This prevents us from ever producing access witnesses without dependencies in a state in which doing so would be unsound with our implementation.

Since access witnesses have a different mechanism that removes them from the heap, the consume operation for witnesses does not delete any chunks. However, the consume rule not only removes chunks, but it also returns a snapshot of the deleted chunks (more precisely, it passes it to the continuation). So instead of removing the chunks and returning a snapshot, the consume rule for witnesses merely returns the snapshot. Its implementation is listed in Figure 4.3.

`dependent-on` filters a given set of heap chunks. It keeps all non-witness chunks and all witness chunks that are directly or indirectly dependent on the given list of dependencies. To determine whether a single chunk is dependent on a list of dependencies, we check whether they form a vertex cut in the graph of the dependency relation. The actual implementation can be found in Figure 4.4.

The remaining chunks are then summarized. This summarization step is necessary to prevent severe incompletenesses: For performance reasons, branches that are created during expression evaluation are merged into a single branch. A detailed explanation can be found in Section 3.4.3 of the original description [6]. During the merging process, it may be possible that different branches contain different access witness chunks. By using summarization we are able to preserve those chunks in the merged branch. More details about how access witnesses affect the merging of different branches will be provided in Section 4.5.

The summarization is an adaptation of the approach presented in [7] (Section 6.2). The main difference is that it not only considers witness chunks,

```
dependent-on($\sigma$, $chunks$, $\overline{p(\overline{a})}$) =
  if $\overline{p(\overline{a})}$ = [] then
    return $chunks$

  $result := \{c \mid c \in chunks \land \neg\,\text{is\_witness}(c)\}$
  $visited := \varnothing$
  foreach $chunk \in \{c \mid c \in chunks \land \text{is\_witness}(c)\}$
    $todo := \{(chunk, \text{arguments}(chunk))\}$
    $skip := false$
      while $todo \neq \varnothing \land \neg skip$
        Select $(c, \overline{args}) \in todo$
        $todo := todo \setminus \{(c, \overline{args})\}$
        if $(c, \overline{args}) \notin visited$ then
          if check($\sigma.\pi, \exists i \cdot p_i = \text{id}(c) \land \overline{a_i} = \overline{args}$) then
            $visited := visited \cup \{(c, \overline{args})\}$
          else if is_witness($c$) then
            foreach $(id, \overline{v}) \in \text{parents}(c)$
              $todo := todo \cup \{(c', \overline{v}) \mid c' \in \sigma.h \land \text{id}(c') = id\}$
          else
            $skip := true$
      if $\neg skip$ then
        $result := result \cup \{chunk\}$
  $result$
```

Figure 4.4: The symbolic execution rule to filter a given set of heap chunks for the ones that are either dependent on the provided list of dependencies, or chunks that are not witnesses.

but it also summarizes them with non-quantified and quantified chunks as well. The symbolic execution rule is presented in Figure 4.5.

Since consuming permissions may lead to the expiration of witnesses, the symbolic execution rule for permission consumption must be updated as well. Thus, we extend the original rule with a call to `removed-expired-witnesses` before calling the continuation. Since this is the only change, we omit the implementation of the updated rule.

4.4 Fold and Unfold

4.4.1 Fold Statements

By design, folding a predicate instance creates new witnesses for all permissions that occur in the body of the folded predicate. This requires redesigning the symbolic execution rule for fold statements. Furthermore, the old rule would also lead to the expiration of witnesses since it consumes the predicate body. This may result in the deletion of witnesses that are dependent

```
summarise-chunks($chunks, \overline{args}$) =
  $snap$ := fresh
  $snap_{def}$ := $\varnothing$
  $p := 0$
  foreach $chunk \in chunks$
    $snap_{def} := snap_{def} \cup$
      $\{0 < \mathrm{perm}(chunk)(\overline{args}) \Rightarrow snap = \mathrm{snap}(chunk)(\overline{args})\}$
    $p := p + \mathrm{perm}(chunk)(\overline{args})$
  $(snap, snap_{def}, perm)$
```

Figure 4.5: The symbolic execution rule to summarize both witness chunks and normal chunks to obtain a combined snapshot, its definition, and the overall permission amount.

```
exec($\sigma_1$, fold acc($pred(\overline{e}), p$), $Q$) =
  eval($\sigma_1, p :: \overline{e}, \lambda\ \sigma_2, p' :: \overline{e'}$ ·
    assert($\sigma_2.\pi, 0 \leq p'$)
    $bdy$ := scale($pred_{body}[\overline{x \mapsto e'}], p'$)
    consume-preserving($\sigma_2, bdy, \lambda\ \sigma_3, s$ ·
      produce-dependent($\sigma_3, bdy, pred(\overline{e'}), s, \lambda\ \sigma_4$ ·
        produce($\sigma_4$, acc($pred(\overline{e'}), p'$), $s, \lambda\ \sigma_5$ ·
          $Q$(remove-expired-witnesses($\sigma_5$)))))
```

Figure 4.6: Updated rule for executing fold statements. $pred_{body}$ denotes the body of predicate $pred$, and $\overline{x}$ its formal arguments.

on the consumed permissions. To overcome this issue, we introduce a new symbolic execution rule named `consume-preserving`. Its implementation is equal to the rule stated in Figure 3.9 of the original description [6] renamed to `consume-preserving`. The difference between this new rule and the updated `consume` rule is that `consume-preserving` does not remove witnesses after consuming the predicate body. This is required to enable the preservation of witnesses during the fold operations. The `scale` operation multiplies all permission amounts in an assertion by a given symbolic permission value.

We use `produce-dependent` to create witnesses for all permissions occurring in the predicate body. `produce-dependent` is a utility method that allows the production of accessibility predicates as access witnesses and is listed in Appendix B.2. This is necessary because predicate bodies only contain accessibility predicates, but we have to produce access witnesses. Note that it is not possible to perform syntactic rewriting on the predicate body and reuse the existing production rule to achieve the same goal, since the syntax for access witnesses does not allow listing permission amounts. If the predicate body contains an accessibility predicate with a conditional permission amount (for example `condition ? write : none`), respecting this permission amount is crucial for the soundness of the symbolic execution engine.

```
1  exec($\sigma_1$,unfold acc($id(\bar{e}), p$), $Q$) =
2    eval($\sigma_1, p :: \bar{e}, \lambda\ \sigma_2, p' :: \overline{e'}\ \cdot$
3      assert($\sigma_2.\pi, 0 \leq p'$)
4      $bdy$ := scale($pred_{body}[\overline{x \mapsto e'}], p'$)
5      if $bdy$ contains quantified permissions then
6        consume($\sigma_2$,acc($pred(\overline{e'}), p'$), $\lambda\ \sigma_3, s\ \cdot$
7          produce($\sigma_3$, $bdy, s, Q$))
8      else
9        consume-preserving($\sigma_2$,acc($pred(\overline{e'}), p'$), $\lambda\ \sigma_3, s\ \cdot$
10         produce'($\sigma_3$, $bdy, s, \lambda\ \sigma_4, replacements\ \cdot$
11           $\sigma_5$ := replace-dependencies($\sigma_4$, $(pred, \overline{e'}), replacements$)
12           $Q$(remove-expired-witnesses($\sigma_5$))))
13     )
```

Figure 4.7: The updated symbolic execution rule for unfold statements.

4.4.2 Unfold Statements

While the updated folding rule is rather simple, unfold statements require more bookkeeping in the presence of witnesses. While it is not necessary to introduce new witnesses when a predicate instance is unfolded to be able to preserve the existing witnesses, the existing rule for unfold statements may still lead to the expiration of witnesses nonetheless, as we have seen in Section 3.3.4.

Figure 4.7 shows the updated rule for predicate unfolding. The `if` statement on line 5 is necessary, as there is currently no way to preserve witnesses if the body of a dependency contains quantified permissions, as this would require quantified access witnesses. We therefore use the old rule when such a predicate is unfolded, which removes all witnesses that are dependent on the unfolded predicate instance. In our implementation, we also emit a warning.

The updated rule is also quite similar to the original one, but only because the bookkeeping code is outsourced to `replace-dependencies` and `produce'`. `replace-dependencies` replaces the missing dependencies of all witnesses if necessary. Its implementation is shown in Figure 4.8. `produce'` is identical to `produce`, but records which permissions were produced and passes them to the continuation as a set of predicate identifier and symbolic argument tuples. Note that it is not possible to determine this set of replacements statically, as the `produce` rule may create different branches. Its implementation is shown in Appendix B.1.

Both the updated fold and unfold rule use `remove-expired-witnesses` to delete witnesses with missing dependencies. For the fold rule it is only used to be sure that no expired witnesses remain, but it should never remove any witnesses in practice. In the unfold case however, it may be the case that the

```
replace-dependencies($\sigma$, ($pred$, $\overline{args}$), $replacements$) =
  $chunks := \{c \mid c \in \sigma.h \wedge \mathrm{id}(c) = pred\}$
  $perm := \sum_{c \in chunks} \mathrm{perm}(c)(\overline{args})$
  if check($\sigma.\pi$, $perm > 0$) then
    return $\sigma$

  $h_1 := \{c \mid c \in \sigma.h \wedge \neg\,\mathrm{is\_witness}(c)\}$
  $witnesses := \sigma.h \setminus h_1$
  foreach $c \in witnesses$
    $d' := \varnothing$
    foreach $(p, \bar{e}) \in \mathrm{parents}(c)$
      if $p = pred \wedge$ check($\sigma.\pi$, $\bar{e} = \overline{args}$) then
        replacements
      else
        $d' := d' \cup \{(p, \bar{e})\}$
    $h_1 := h_1 \cup \{w\{parents := d'\}\}$
  $sigma\{h := h_1\}$
```

Figure 4.8: A utility method to replace dependencies of witnesses if the given dependency is no longer satisfiable.

predicate body contains some permissions which are not strictly positive. In some cases, this may lead to the removal of access witnesses, as our previously stated invariant is violated. While this introduces incompletenesses, we didn't find a more precise invariant that allows the preservation of witnesses across unfold operations.

4.5 Joining

As we mentioned earlier, branches that are created during symbolic expression evaluation are merged into a single branch. To do so, the `join` operation is used. It takes a function that may create different branches (Q_{branch}), executes it and then merges the different branches into a single one. Before access witnesses were introduced, it was guaranteed that the heaps in different branches were equal, as expression evaluation cannot change permissions or write to heap locations. Access witnesses invalidate this assumption, as different branches may create different witnesses, making it necessary to update the `join` rule.

While it would be sound to simply drop all witness chunks from different branches, the arising incompleteness would render our implementation useless. Instead, we also join the different heaps. To do so, we update the permission expression of all witness chunks that were produced in different branches before adding them to the merged state. It is not necessary to update witness chunks that are already present in the original heap. For a witness chunk c from a branch with the branch condition bcs its permission amount is updated to $ite(bcs, c.perm, 0)$, where $c.perm$ denotes the symbolic permission value stored in c. Hence c can only be used if we can show that bcs holds.

Remember that the invariant for witness chunks states that whenever we have non-zero permission to a witness chunk, the heap also contains non-zero permission to every dependency of said chunk. Since the proposed heap update changes the permission values of witness chunks, we have to show that this update preserves the invariant. To do so, we proceed by an informal case analysis for a generic witness chunk and dependency pair.

- If we update neither the permission amount of the witness chunk nor the dependency, the invariant is trivially preserved.
- If we update the permission amount of the witness chunk, but without modifying the dependency, the invariant is also preserved: Since the updated permission amount is less or equal than the original amount, we know that if the updated permission amount is non-zero, then so was the original permission amount. Since the invariant did hold before merging, the permission amount of the dependency must therefore also be non-zero.
- The case that we update the permission amount of the parents without updating the permission amount of the witness chunk is irrelevant: If the permission of the witness chunk remains untouched, this implies that the witness chunk was already present in the original heap. Since the invariant holds in the initial heap, it must contain enough permission to the dependency. Due to being in the original heap, this permission amount to the dependency is also not updated, therefore trivially preserving the invariant. The updated parent chunks may only increase the new overall permission of the dependency.
- If both the permission amount of the witness chunk and the permission amount of the parents is modified, the invariant is preserved as well, since the applied permission update is identical for both permission amounts.

The updated version listed in Figure 4.9 replaces the original rule. In addition to performing the same tasks as the original rule, it also updates the permission value of witness chunks as described above.

4.6 Unfolding Expressions

Unfolding expressions with access witnesses behave like normal unfolding expressions, but instead of producing the predicate body as normal chunks

```
join($\sigma_1$, $Q_{branch}$, $Q$) =
  $id$ := fresh
  $data$ := $\varnothing$
  $Q_{branch}$(
    $\sigma_1\{\pi$ := pc-push($\sigma_1.\pi$, $id$, $true$)$\}$,
    $\lambda\ \sigma_2, \omega \cdot$
      $data := data \cup \{(\sigma_2.h$, pc-after($\sigma_2.\pi$, $id$), $\omega)\}$
      success()
    ) $\wedge$ (
      $(h_2, \pi_2, \omega s)$ := foldl($data$, $(\sigma_1.h, \sigma_1.\pi, \varnothing)$,
                       $\lambda\ (h_i, \pi_i, \omega_i), (h_{joined}, \pi_{joined}, \omega s_i) \cdot$
        $(cnds, bcs_{all})$ := pc-segs($\pi_i$)
        $h_{diff} := h_i \setminus \sigma_1.h$
        foreach $c \in h_{diff}$
          $h_{joined} := h_{joined} \cup \{c\{perm := ite(bcs_{all}, c.perm, 0)\}\}$
        ($h_{joined}$, pc-add($\pi_{joined}$, $cnds$), $\omega s_i \cup \{(bcs_{all}, \omega_i)\}$))
      $Q(\sigma_1\{\pi := \pi_2, h := h_2\}, \omega s)$)
```

Figure 4.9: The updated join rule. The *perm* field of witness chunks denotes the stored permission value.

```
eval($\sigma_1$, unfolding dep($pred(\bar{e})$) in $b$, $Q$) =
  if the unfolding is explicit then
    eval($\sigma_1$, $\bar{e}$, $\lambda\ \sigma_2, \overline{e'} \cdot$
      $bdy := pred_{body}[\overline{x \mapsto e'}]$
      join'($\sigma_2$,
        $\lambda\ \sigma_3, Q_{join} \cdot$
          consume($\sigma_3$, dep($pred(\overline{e'})$), $\lambda \sigma_4, s \cdot$
            produce-dependent($\sigma_4$, $bdy$, $pred(\overline{e'})$, $s$, $\lambda\ \sigma_5 \cdot$
              eval($\sigma_5$, $b$, $\lambda\ \sigma_6, b' \cdot$
                $Q_{join}(\sigma_6, b')$))))),
    $Q$)
  else
    identical to the unfolding rule for accessibility predicates
```

Figure 4.10: The symbolic execution rule for unfolding witnesses.

```
eval($\sigma_1$, $e.f$, $Q$) =
  eval($\sigma_1$, $e$, $\lambda\ \sigma_2, e' \cdot$
    $snap, snap_{def}, perm :=$ summarise-chunks($\{c \mid c \in \sigma_2.h \land \mathrm{id}(c) = f\}, e'$)
    assert($\sigma_2.\pi$, $0 < perm$)
    $Q(\sigma_2\{\pi :=$pc-add$(\sigma_2.\pi, snap_{def})\}, snap))$
```

Figure 4.11: The updated field access rule.

and temporarily removing the predicate instance from the heap, they keep the predicate witness and permanently produce the predicate body as witness chunks.

If the body of the unfolded predicate contains quantified permissions, the production of the respective chunk is skipped. The symbolic execution rule for unfolding witnesses is listed in Figure 4.10.

4.7 Field Access

Since witnesses provide read access to fields, we have to update the field access rule. The updated rule stated in Figure 4.11 is an adaptation of the field access rule presented in Figure 4.4 of [6]. It just combines all snapshots from normal and witness chunks using the `summarise-chunks` method from Figure 4.5.

4.8 Function Verification

We also have to slightly change how function verification works. However, only minor changes are required because we only have to modify the self-framedness checks for function postconditions. Since these may now contain witnesses, it is no longer possible to just evaluate the postcondition, instead we have to produce it. Unlike method verification, the postcondition is not produced into an empty heap, but into the heap resulting from the production of the precondition. This is because the function precondition is only asserted on the caller side. The updated implementation can be found in Figure 4.12.

4.9 Function Axiomatization

The addition of access witnesses makes it necessary to introduce an additional step during function axiomatization. Viper functions are translated to symbolic functions in the underlying SMT solver. Since the SMT solver has no access to the symbolic heap (it is only known to the symbolic execution engine), each translated function has an additional argument, which represents the part of the heap that is accessible by said function. This additional ar-

```
1  verify($\pi_0$, function $fun(\overline{x : T}) : T_r$) =
2    $\overline{x'}$ := fresh
3    $\sigma_1 := \{\gamma := \varnothing[\overline{x \mapsto x'}], h := \varnothing, \pi := \pi_0\}$
4    produce($\sigma_1$, exh-variant($fun_{pre}$), fresh, $\lambda\ \sigma_2 \cdot$
5      produce($\sigma_2$, $\{\gamma := \sigma_2.\gamma[$result $\mapsto$ fresh$]\}$, $fun_{post}$, fresh, $\lambda\ \_, \_ \cdot$
6        success())) $\wedge$
7    produce($\sigma_1$, inh-variant($fun_{pre}$), fresh, $\lambda\ \sigma_2 \cdot$
8      eval($\sigma_2$, $fun_{body}$, $\lambda\ \sigma_3, \_ \cdot$
9        consume($\sigma_3\{\gamma := \sigma_3.\gamma[$result $\mapsto$ fresh$]\}$, $fun_{post}$, $\lambda\ \_, snap \cdot$
10         success())))
```

Figure 4.12: The updated function verification rule.

gument is provided by the snapshot that gets produced when the function's precondition is consumed during symbolic function evaluation.

To provide the SMT solver with the necessary knowledge about the function definition and its postcondition, two SMT axioms are emitted during the function axiomatization step: the definitional axiom and the postcondition axiom. Section 3.6 of the original description [6] contains a more detailed description.

Since function postconditions may contain access witnesses, we require knowledge about the snapshots of the witnesses that get produced during function evaluation to be able to create correct witness chunks. However, the translation of Viper functions to SMT functions prevents us from creating this snapshot from within Viper (doing so would require on the fly evaluation which comes with other issues). We solve this challenge by creating an additional SMT function for each Viper function. This additional function is called snapshot function and is a symbolic representation of the snapshot of all witnesses in the function postcondition. It takes the same arguments as the symbolic function (all arguments of the Viper function plus a snapshot). The additional *snapshot axiom* provides a definition of the snapshot function. It is generated by recording the snapshot *snap* that is produced during the function verification code (compare line 9 of Figure 4.12). *snap* is a symbolic expression in terms of the formal function arguments and therefore perfectly suited as a definition for the snapshot function. We use the same trigger for the snapshot axiom as for the definitional axiom.

4.10 Function Evaluation

Finally, the rule for function evaluation needs to be updated as well. If the function's postconditions contain witnesses, we produce them using the snapshot function that is described in Section 4.9. To guarantee minimal interference with the original function evaluation rule, we introduce a special case for such functions and use the existing function evaluation code for the de-

```
eval($\sigma_1$, $fun(\bar{e})$, $Q$) =
  eval($\sigma_1$, $\bar{e}$, $\lambda\ \sigma_2, \overline{e'}$ ·
    join'($\sigma_1\{h := \sigma_1.h\}$,
      $\lambda\ \sigma_3, Q_{join}$ ·
        consume($\sigma_3$, $fun_{pre}[\overline{x \mapsto e'}]$, $\lambda\ \sigma_4, s$ ·
          if $fun$ returns a Ref $\wedge fun_{post}$ contains accessibility predicates then
            $snap := fun\$snapshot(\overline{e'}, s)$
            produce($\sigma_4\{h := \sigma_3.h\}$, $fun_{post}[\overline{x \mapsto e'}]$, $snap$, $\lambda\ \sigma_5$ ·
              $Q_{join}(\sigma_5, fun(\overline{e'}, s)))$
          else
            $Q_{join}(\sigma_4\{h := \sigma_3.h\}, fun(\overline{e'}, s)))$
    $Q$))
```

Figure 4.13: The updated function evaluation rule

fault case. The updated code is shown in Figure 4.13. The symbolic function is denoted by fun, the snapshot function by $fun\$snapshot$.

Chapter 5

Prusti

Now that we have seen what access witnesses are and how they can be implemented in a symbolic execution based verifier, this chapters explains how to leverage them in Prusti's Rust-to-Viper encoding to achieve our goal of supporting pure functions that return shared borrows.

5.1 Background

Before we can explain our encoding changes, we first have to explain how the current encoding works in more detail. Rust types are encoded using Viper predicates. Each Rust variable is translated to a `Ref`-typed Viper variable, accompanied by the appropriate predicate instance for the Rust type. Figure 5.1 shows how Rust types are translated to Viper predicates in a small example.

All Rust functions can be annotated with preconditions and postconditions

```
struct Pair {
  fst: i32,
  snd: i32,
}
```

```
predicate i32(self: Ref) {
  // implementation omitted
}

predicate pair(self: Ref) {
  acc(self.fst) && i32(self.fst) &&
  acc(self.snd) && i32(self.snd)
}
```

Figure 5.1: Encoding of Rust types (top) to Viper predicates (bottom).

which are appropriately translated to Viper, where they are checked. Rust functions can also be annotated as *pure*, meaning that they are side-effect free and deterministic. Pure functions are encoded as Viper functions and can therefore be used in Rust specifications (preconditions and postconditions).

However, encoding pure functions to Viper functions fundamentally conflicts how Rust types are modeled. The problem is that the return type is encoded as a part of the postcondition. Since all Rust types are encoded as predicate instances, and Viper function postconditions cannot contain any accessibility predicates, this approach does not work with pure functions. By using a different type encoding for pure functions, it is possible to support boolean and integer types as return values for pure functions. This is done by using Viper's `Bool` and `Int` types instead of the corresponding predicate instances. However, this approach does not generalize to shared references, rendering the encoding of such pure functions impossible. By leveraging access witnesses, we are able to extend the type encoding for pure functions to overcome this limitation.

5.2 Type Encoding for Pure Functions

When pure Rust functions return a shared borrow, this shared borrow is always obtained by reborrowing from an argument or a field of an argument. Since the Viper encoding of Rust types causes the type predicates of reborrows to be inside the type predicate of the corresponding argument, we can use access witnesses to specify that the type predicate of the return value is somewhere inside one of the arguments' type predicates. Since access witnesses can be used in the postcondition of Viper functions, this encoding allows us to also encode pure Rust functions that return shared borrows.

However, just using all arguments as dependencies is not very precise and can be improved by leveraging the information of Rust's borrow checker. Reborrowing causes the borrowed-from location to be blocked. To determine the exact dependencies of the return value, we filter all arguments and only keep the ones that are currently blocked by the reborrowed return value.

Since pure function calls can be chained, we also have to update the type encoding in function preconditions. Previously, the permission to the type-predicate instance of each argument was passed to the Viper function as an accessibility predicate. Since access witnesses cannot be used in places where real permissions are required, this encoding prevented chaining pure function calls. By using access witnesses instead of accessibility predicates for the type-predicate instances of function arguments, we can fix this issue and allow the chaining of pure functions. Figure 5.2 shows how our new encoding translates a pure function that returns a shared borrow to a Viper function.

```rust
struct Point {
  x: i32,
  y: i32,
}

struct Line {
  start: Point,
  end: Point,
}

impl Line {
  #[pure]
  fn get_start(&self) -> &Point {
    &self.start
  }
}
```

```
predicate i32(r: Ref) { ... }
predicate Point(r: Ref) { ... }
predicate Line(r: Ref) { ... }

function line_get_start(self: Ref): Ref
  requires dep(Line(self))
  ensures dep(Point(result), Line(self))
{
  // implementation omitted
}
```

Figure 5.2: A minimal example that shows how the updated Rust-to-Viper encoding of pure functions works.

5.3 Encoding Function Calls

The changes we have seen so far allow functions to return shared borrows, as well as chaining function calls. However, to access fields of returned shared borrows, we also have to change the encoding of function calls. The reason is that field accesses require unfolding the returned access witness. To unfold the returned access witness, the result of the function call has to be available as a local variable. We therefore encode all function calls that return a shared reference using a let expression to bind the result to a fresh variable. This allows unfolding the returned access witness by using said variable and thus enables field accesses of function results. An encoded function call to the function `line_get_start` from Figure 5.2 thus looks like this:

```
let fresh_var == (line_get_start(l)) in ...
```

Before passing the generated Viper program to the verifier, the original encoding used to perform several optimizations. Because those optimizations do not work correctly in the presence of let expressions, we were forced to disable them for function calls that are encoded using a let expression. The purpose of said optimizations is to prevent a specific incompleteness in the verification backend from causing spurious verification failures. Therefore, Rust programs using pure functions that return shared borrows may experience the effects of this incompleteness as failing assertions, loop invariants or specifications. Section 6.1 will describe the incompleteness in more detail. We leave the incorporation of let expressions into those optimization algorithms to future work.

Chapter 6

Implementation and Evaluation

This chapter discusses the challenges of implementing the rules presented in Chapter 4 in Silicon, Viper's symbolic execution based backend verifier. Then, we evaluate the performance and expressiveness of our implementation.

6.1 Challenges With Implementing Snapshot Summarization

Snapshot summarization in Silicon was introduced in [6] for the implementation of quantified permissions. As discussed in Section 6.2 of [7], snapshot summarization allows Silicon to overcome certain incompletenesses of the default greedy heap lookup algorithm in the presence of disjunctive aliasing. However, the summarization approach suffers from other incompletenesses, some of which manifest in the presence of quantifiers with heap-dependent triggers.

Every time we summarise some heap chunks, we generate a fresh snapshot s_1. Without any further information (the snapshot definition), the underlying SMT solver (Z3 in the case of Silicon) does not know anything about this new snapshot. If s_1 happens to occur in the trigger of a quantified assertion (for example if the corresponding Viper trigger contains a field dereference), this often leads to incompletenesses.

The problem is that the definition of s_1 may be part of a quantifier which contains s_1 in its trigger. This happens if the summarization is done during the evaluation of the quantifier's body. The symbolic execution rule for quantifier evaluation generates an auxiliary quantifier which contains all path conditions that were recorded during the evaluation of the quantifier's body, including the snapshot definition. This auxiliary quantifier uses the same trigger expression as the original quantifier and thus contains s_1.

```
 1  predicate p(r: Ref)
 2
 3  function f(r: Ref): Int
 4    requires acc(p(r), write)
 5
 6  method foo() {
 7    var xs: Seq[Ref]
 8    inhale forall x: Ref :: { f(x) } x in xs ==> acc(p(x), write)
 9    inhale forall x: Ref :: { f(x) } x in xs ==> f(x) == 1
10    assert forall x: Ref :: { f(x) } x in xs ==> f(x) == 1
11  }
```

Figure 6.1: A minimal Viper program demonstrating the incompleteness caused by snapshot summarization. The assert statement on line 10 fails, even though no changes to the program state have been made after the completely identical inhale statement.

As a consequence, we can only learn the definition of s_1 if we are able to instantiate the quantifier. To do so, Z3 needs to be able to show that some other snapshot s_2 is semantically equal to s_1, such that the trigger is matched and the quantifier body is instantiated. But without the definition of s_1 (which only occurs within the quantifier) Z3 cannot know whether s_2 is equal to s_1, creating a chicken-and-egg problem. Figure 6.1 contains a minimal example that fails due to exactly this problem.

For performance reasons, Silicon maintains a cache of snapshot summaries, which are reused if there were no changes to the summarized heap chunks. As a side-effect this also reduces the impact of the aforementioned incompleteness. If a summary is reused, Z3 trivially knows that the reused snapshot s_1 is equal to the occurrence of the same snapshot in a trigger. However, this approach only works if summaries can be reused. Since the summarization code always uses all potentially relevant heap chunks to generate a summary for a resource, seemingly unrelated heap changes may lead to the invalidation of a snapshot summary and trigger the incompleteness.

Since access witnesses also use the summarization code, they are affected by the same incompleteness. To reduce its impact, we investigated whether the snapshot definition could be emitted independently of the auxiliary quantifier. It turns out that for witnesses, the snapshot definition itself is often independent from the quantified variables and therefore could be emitted separately from the quantifier. However, in almost all the cases, the snapshot definition occurs on the right hand side of an implication whose left hand side contains quantified variables. This renders pulling the snapshot definition out of the quantifier virtually impossible. Doing so, requires to show that there exists a choice for each variable such that the left hand side of the implication evaluates to true. Not only is existential quantification in SMT solvers poorly supported in general, but in some instances this cannot be shown at all.

Another approach would be to generate additional triggers for the quantifier containing the snapshot definition and use them to instantiate its body on demand. However, one has to be careful not to introduce matching loops.

6.2 Expressiveness of Viper Programs

In this section, we evaluate the expressiveness of access witnesses by comparing three different Viper encodings of methods that insert an element into a sorted list.

6.2.1 Scenario

The goal of each encoding is to implement and verify a Viper program that inserts a single element into a sorted list of non-primitive items. More precisely, we require the implementation of 3 Viper methods (ordered by increasing difficulty):

1. Inserting an item at the front. This method is given a list `r` that is sorted in ascending order and an item `it` that is guaranteed to be smaller or equal to all elements in `r`. The method has to return a new list that is sorted in ascending order and contains `it` at the first position, followed by the elements in `r` in the original order.
2. Inserting an item at the back. This method is given a list `r` that is sorted in ascending order and an item `it` that is greater or equal to all elements in `r`. The method has to return the updated list `r` still sorted in ascending order, containing the original elements at their original positions, followed by `it` as the last element.
3. Inserting an item at an arbitrary, but given position. This method is given a list `r` that is sorted in ascending order, an integer `pos` that indicates the position where the new element has to be inserted, and an item `it` that is smaller or equal to all elements in `r` with an index that is strictly smaller than `pos` and greater or equal to all other elements in `r`. The method has to return a new list that is sorted in ascending order and contains `it` at index `pos`. The other elements of the returned list have to be equal to the original items in `r`.

The encoding of the list and the item types follows Prusti's approach (compare Section 5.1) and is given in Figure 6.2. It is not allowed to modify this encoding. Additionally, the following utility functions are defined (a possible implementation for each of them is provided in Figure 6.3:

- A function `length(r)` which returns the length of a given list `r`.

```
field sort_value: Int

predicate item(r: Ref) { acc(r.sort_value, write) }

field element: Ref
field next: Ref

predicate list(r: Ref) {
  acc(r.element, write) && acc(r.next, write) &&
    acc(item(r.element), write) &&
    (r.next != null ==> acc(list(r.next), write)
}
```

Figure 6.2: Viper encoding of a linked list with non-primitive items.

- A function `at(r, i)` which returns the `i`-th item of `r`.
- A boolean function `lte(l, r)` which indicates whether item `l` is smaller or equal than item `r`.

Note that while it is allowed to define additional, encoding specific functions, we prohibit the definition of alternatives to the mentioned utility functions. This means that to compare two list items, only `lte` may be used. Furthermore, the insert methods have to treat the items as if they were abstract; that is, they are not allowed to directly access the fields of an item.

6.2.2 Witness-Based Encoding

An entirely witness-based encoding cannot be implemented in Viper, because there is no way to assert that an entire subset of the heap has not been modified. More precisely, there is no way to compare entire subsets of a heap. When we encode the different methods, we require write permission to the list `r` in order to be able to insert the given element. However, this also implies this method could arbitrarily modify the items. As a consequence, each method must ensure in its postcondition that no items were modified. Since methods have to treat items as if they were abstract, it is not possible to write such a postcondition, as it would require to check whether the subset of the heap that is accessible by each item is equal to its initial state.

6.2.3 Purification

It is possible to implement a solution to our problem statement entirely without access witnesses while still using function chaining. This approach involves using *domains*, a Viper feature we didn't cover in Chapter 2. Domains allow the modeling of custom types. Each domain definition may contain various *domain functions*. In contrast to normal Viper functions, they do not

```
function length(r: Ref): Int
  requires dep(list(r))
  ensures result > 0
{
  unfolding dep(list(r)) in
    r.next == null ? 1 : 1 + length(r.next)
}

function at(r: Ref, i: Int): Ref
  requires dep(list(r)) && 0 <= i && i < length(r)
  ensures dep(item(result), list(r))
{
  unfolding dep(list(r)) in
    i == 0 ? r.element : at(r.next, i - 1)
}

function lte(l: Ref, r: Ref): Bool
  requires dep(item(l)) && dep(item(r))
{
  (unfolding dep(item(l)) in l.sort_value) <=
    (unfolding dep(item(r)) in r.sort_value)
}
```

Figure 6.3: A possible implementation of the defined utility functions.

have any precondition, postcondition, or even body. Instead, one uses *axioms* to define the semantics of such functions.

We use domains to implement an alternative heap representation, which allows us to overcome the issue we encountered in the previous section. More specifically, explicitly modeling a user defined heap allows us to implement the comparison of entire heap subsets. However, it also requires the translation of all heap-dependent functions to domain functions, which is why this approach is called purification.

Implementing a Custom Heap

Our custom heap implementation uses a single domain named `Snapshot`. As mentioned earlier, we assume a type encoding similar to the one Prusti uses. In addition to the type encoding from Figure 6.2, we generate a set of domain functions and axioms for each type: The snapshot construction function takes all fields of a type `T` as arguments and returns a `Snapshot` (the type defined by our custom domain). The types of the arguments correspond to the Viper type of each field, except for `Ref`-typed fields, which are translated to a `Snapshot`-typed argument. For each field, an additional domain function is generated that acts as a lookup function and can be used to deconstruct the snapshot. Additionally, we generate axioms to specify that the snapshot construction is a bijective operation and the lookup functions are the inverse functions of the construction function. The additional domain encoding of the list and item types from the problem statement is shown in Figure 6.4. The snapshot of the `null` reference is denoted by a special domain function `snap$null` without any arguments (it is therefore equivalent to a constant). Furthermore, we generate one heap-dependent Viper function per type that is responsible to create the snapshot. The functions for the list and item type are listed in Figure 6.5.

Modeling Head-Dependent Functions

All heap-dependent functions (except the snapshot generation functions) are translated to domain functions in our `Snapshot` domain. This translation is identical to the function axiomatization implemented in Silicon, which in turn follows the approach of Heule, Kassios, Müller, *et al.* [8], and prevents matching loops in recursive functions. In Figure 6.6, we provide an example encoding for the `length` function from Figure 6.3.

Analysis

The complete code of our implementation can be found in Section C.1. Note that there are some minor differences to the encoding we described in the previous sections that were omitted for complexity reasons.

Besides the huge encoding overhead to reproduce what Silicon internally already implements, we also had to manually provide some guidance to the verifier in the form of assertions to successfully verify our solution. Furthermore, our encoding currently lacks the possibility to check the preconditions of the original heap-dependent functions. On the other hand, it allows us to both chain function applications and compare snapshots from different program states.

The manual guidance that is required to successfully verify our implementation currently prevents this approach from being a viable option for the automatic encoding of types that is, for example, used by Prusti. However, we believe that it should be possible to tweak the encoding such that many of our additional assertions become unnecessary.

6.2.4 Hybrid Approach

Our last implementation uses the same custom heap encoding as the purification approach from the previous section to model user defined snapshots. However, we do not encode the heap-dependent functions into the domain. Instead, we leverage access witnesses to be able to chain heap-dependent

```
domain Snapshot {
  function snap$null(): Snapshot

  function snap$item(sv: Int): Snapshot
  function snap$item$inv(s: Snapshot): Int

  axiom ax$snap$item$cons {
    forall s: Snapshot :: {snap$item$inv(s)}
      snap$item(snap$item$inv(s)) == s
  }
  axiom ax$snap$item$inv {
    forall i: Int :: {snap$item(i)}
      snap$item$inv(snap$item(i)) == i
  }


  function snap$list(el: Snapshot, n: Snapshot): Snapshot
  function snap$list$inv1(s: Snapshot): Snapshot
  function snap$list$inv2(s: Snapshot): Snapshot

  axiom ax$snap$list$inv1 {
    forall s1: Snapshot, s2: Snapshot :: {snap$list(s1,s2)}
      snap$list$inv1(snap$list(s1,s2)) == s1
  }
  axiom ax$snap$list$inv2 {
    forall s1: Snapshot, s2: Snapshot :: {snap$list(s1,s2)}
      snap$list$inv2(snap$list(s1,s2)) == s2
  }
  axiom ax$snap$list$cons {
    forall s: Snapshot :: {snap$list$inv1(s)}{snap$list$inv2(s)}
      snap$list(snap$list$inv1(s), snap$list$inv2(s)) == s
  }
}
```

Figure 6.4: The additional domain encoding required to model the item and list types.

```
function gensnap$item(r: Ref): Snapshot
  requires acc(item(r), write)
{
  unfolding acc(item(r), write) in snap$item(r.snapshot_value)
}

function gensnap$list(r: Ref): Snapshot
  requires acc(list(r), write)
{
  unfolding acc(list(r), write) in snap$list(
    gensnap$item(r.element),
    r.next == null ?
      snap$null() :
      gensnap$list(r.next))
}
```

Figure 6.5: The Viper functions that generate our user defined snapshots. Consult Figure 6.4 for more information about the used domain functions.

functions and thus benefit from Silicon's built-in function axiomatization. The encoding of all heap-dependent functions matches the one shown in Figure 6.3. The snapshot generation functions from Figure 6.5 are updated to require witnesses instead of normal permissions, but otherwise work identically. They are used in various method postconditions to specify that said method does not update the respective subset of the heap. Checking whether two items a and b are identical is as easy as comparing whether their snapshots are equal:

```
getsnap$item(a) == getsnap$item(b)
```

The full encoding is shown in Section C.2. Since we leverage access witnesses to allow the chaining of heap-dependent functions, we do not have to translate heap-dependent functions to domain functions, resulting in a simpler overall design. The custom snapshot representation is used to specify that the implemented methods do not update the list elements. Another advantage of this encoding over purification is that it does not require any manual guidance to verify, except for the same assertions that are required for lists of primitive types. As a result, we deem this encoding fit for use if memory equality assertions have to be encoded.

6.3 Correctness and Performance Tests

Finally, we check our implementation changes for both correctness issues and performance impacts in various setups. This includes a comparison of different configurations and versions of Silicon, as well as measuring the overall

```
function trigger$list(l): Bool

function length(l: Snapshot): Int
function length$limited(l: Snapshot): Int
function length$stateless(l: Snapshot): Bool

axiom length$def {
  forall l: Snapshot ::
    {length(l)}{length$stateless(l),trigger$list(l)}
    length(l) == (snap$list$inv2(l) == nullsnap() ?
      1 :
      1 + length$limited(snap$list$inv2(l)))
}

axiom length$post {
  forall l: Snapshot ::
    {length$limited(l)} length$limited(l) > 0
}

axiom length$aux1 {
  forall l: Snapshot ::
    {length(l)} length$limited(l) == length(l)
}

axiom length$aux2 {
  forall l: Snapshot ::
    {length$limited(l)} length$stateless(l)
}
```

Figure 6.6: The translation of function `length` from Figure 6.2 into a domain function.

execution time our updated version of Prusti needs to verify Rust's top 500 crates.

6.3.1 Silicon

Our implementation of access witnesses in Silicon is currently guarded by a feature flag: To enable support for access witnesses, `--enableWitnesses` has to be passed as a command line argument. This first series of tests and benchmarks compares different configurations of Silicon against the upstream version to check whether our changes introduced unsoundnesses, incompletenesses, or performance regressions. We use the testcases from the upstream Viper testsuite.

The 4 tested configurations are:

1. Silicon's default configuration. We expect neither performance nor functional regressions compared to the upstream version.
2. Silicon with enabled support for access witnesses. No functional regressions are expected; however, there might be a negative impact on the verification performance.
3. Silicon with enabled summarization of non-quantified heap chunks using the `--enableMoreCompleteExhale` command line argument. More details about summarization of non-witness chunks are provided in [7]. We include this configuration since it is the default configuration used in Prusti. It is known to suffer from incompletenesses, we therefore expect some testcases to fail. In Viper programs generated by Prusti, it has shown significant performance improvements over the default configuration; however, it is unclear whether enabling this feature will also improve the verification performance of our testset.
4. Silicon with both access witness support and summarization of non-quantified chunks. This is the intended default configuration for Prusti if our encoding of shared borrows in pure functions using access witnesses is merged into the upstream version. Again, we expect no additional functional regressions compared to configuration 3, but cannot guarantee the absence of a negative performance impact.

Test Environment

All tests were executed on a desktop computer with a 6-core (12 threads) AMD Ryzen 5 2600X 3.60GHz CPU, 32GB of RAM, and an NVMe SSD. The host operating system is Arch Linux. To ensure using a consistent and reproducible environment, all tests were executed in a Docker container which we published on Docker Hub[1]. To obtain a local copy of the benchmark environment one can use the command

```
docker pull nicolastethz/thesis-benchmark:1.0.0
```

The upstream version of Silicon that is used as a baseline is built from the following commits:

- `1fd8693a0e72d88c04b7a3de4a2a2ddff23a9117` for Silicon itself
- `09341c4be21e69f93e7b3c0d94624b57708bdda8` for Silver, which contains code that is shared between different verifiers, for example the Viper parser.

Our new Silicon version that is being evaluated is built from the following commits:

[1] https://hub.docker.com/r/nicolastethz/thesis-benchmark

- `f86278412b5c9db947466e6adf87a51815da3dfb` for Silicon
- `1af9b5bd0f257d1bedc8bd98036d5af00487a25e` for Silver

Benchmark Method

As we already mentioned, we used the testcases from the Viper testsuite for our evaluation. More precisely, those testcases were taken from the aforementioned upstream versions of Silicon and Silver. Our testset contains over 900 Viper programs covering all language features. We provide an archive containing the entire testset online[2]. Note that none of those testcases actively use access witnesses, as this would prevent a comparison between the different configurations.

To determine the total verification time, every test was executed 5 times. We removed both the fastest and the slowest result and then took the average of the remaining 3 results to determine the overall verification time. All testcases are verified in the same JVM process. Additionally, we also record whether the actual verification result matched the expected result to be able to determine functional regressions.

Results

Our test results show no newly failing testcases in Silicon's default configuration compared to the upstream version. Enabling witnesses introduces 2 additional failing testcases. Analyzing those failures shows that enabling witnesses is expected to make those testcases fail due to the changed semantics of fold statements and field lookups. In configuration 3 (enabling snapshot summarization), 9 additional testcases fail. We analyzed the testcases in more detail and came to the conclusion that 5 of the failing testcases occur due to incompletenesses, 3 testcases are expected to fail as they check for an incompleteness of the default heap lookup algorithm which no longer occurs when summarization is used, and one testcase fails because of a known unsoundness in the current summarization code. Note that this unsoundness does not affect Prusti, as the Viper code generated by Prusti is known not to contain any constructs that trigger said unsoundness. When both witnesses and summarization are enabled, there are no testcases that fail in addition to the ones that already fail when each feature is enabled individually.

An overview over the observed performance changes is shown in Table 6.1. Note that we excluded all testcases that verify within less than 100ms from this analysis as they suffer from from fluctuation. With an increase in verification time over all remaining 823 testcases of only 0.47%, the default configuration of our Silicon implementation shows no performance regressions. The worst-case overhead in total verification time for a single testcase is 21% or 142ms.

[2] `http://doi.org/10.5281/zenodo.3385185`

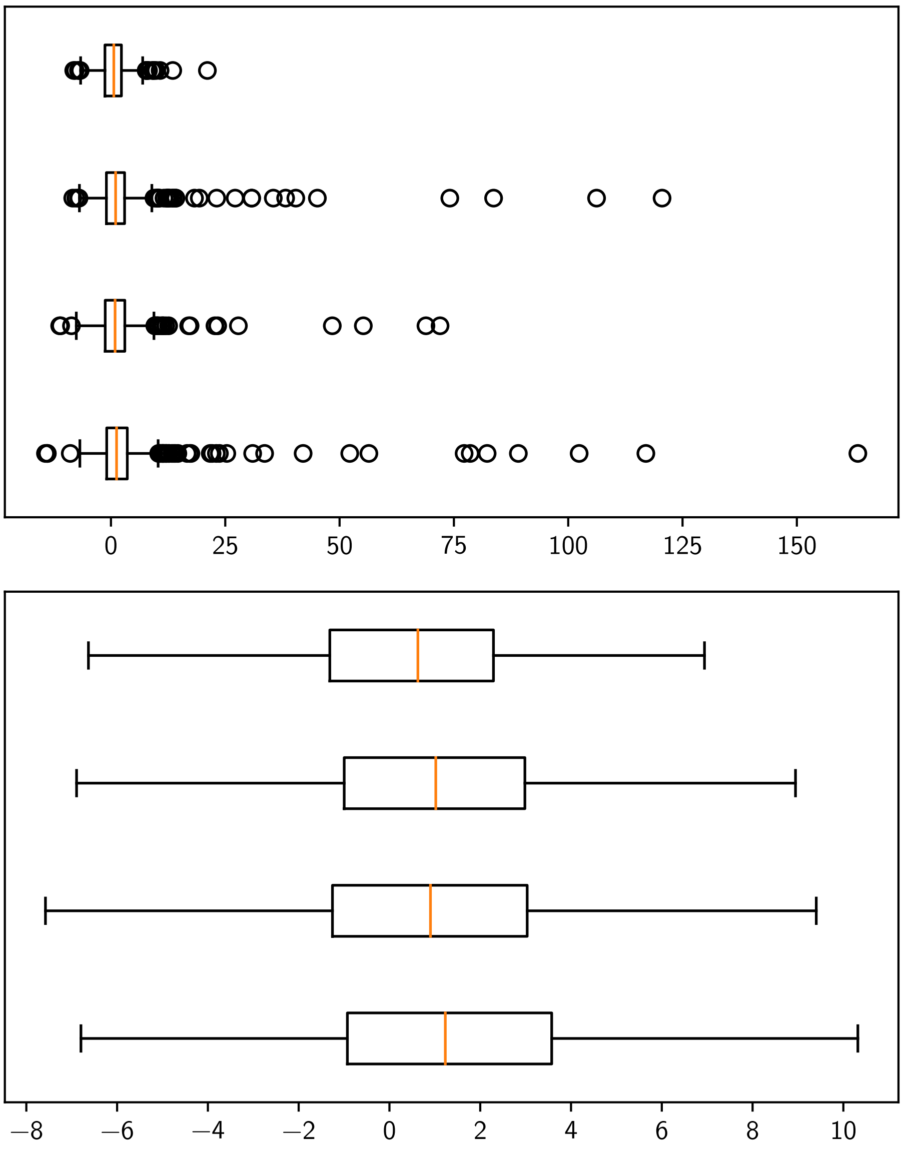

Figure 6.7: Box plots of the relative performance overhead (horizontal, in percent) of each configuration (from top to bottom: default configuration, enabling access witnesses, enabling snapshot summarization, enabling both access witnesses and snapshot summarization). While the first graph includes outliers, they are ignored in the second graph. The box of each dataset ranges from the first quartile to the third quartile, with the orange line marking the median. The whiskers range from the lowest value that is still within 1.5 times the interquartile range of the first quartile to the highest value that is still within 1.5 times the interquartile range of the third quartile.

Configuration	1	2	3	4
Average Increase in Verification Time	0.47%	4.68%	5.53%	9.09%
Median Increase in Verification Time	0.63%	1.02%	0.91%	1.24%
Maximum Increase in Verification Time	21%	121%	72%	163%

Table 6.1: Relative changes in verification time for various configurations of Silicon compared to the upstream version. Only tests with identical verification results in both versions are considered. Additionally, only tests with a total verification time of at least 100ms are considered to determine the maximum increase in verification time.

Configuration	2	3	4
Average Increase in Verification Time	3.07%	5.15%	6.02%
Median Increase in Verification Time	0.97%	0.77%	0.99%
Maximum Increase in Verification Time	36%	72%	82%

Table 6.2: Relative changes in verification time for various configurations of Silicon compared to the upstream version when verifying buggy programs (programs for which verification fails). Only passing testcases are included in the evaluation. Furthermore, only tests with a total verification time of at least 100ms are considered to determine the maximum increase in verification time.

Configuration	2	3	4
Average Increase in Verification Time	4.68%	5.85%	11.53%
Median Increase in Verification Time	1.06%	1.05%	1.43%
Maximum Increase in Verification Time	121%	69%	163%

Table 6.3: Relative changes in verification time for various configurations of Silicon compared to the upstream version when verifying correct programs (programs for which verification succeeds). Only passing testcases are included in the evaluation. Furthermore, only tests with a total verification time of at least 100ms are considered to determine the maximum increase in verification time.

The other configurations come with larger average and maximum performance increases. However, the median performance overhead is very low, which hints at a small number of testcases being responsible for the majority of the overall overhead. A more detailed breakdown reveals that this is indeed the case. Figure 6.7 shows that all configurations have barely any performance overhead except for a few outliers. Note that no configuration has more than 40 outliers in over 800 testcases.

Since we assumed that programs for which verification succeeds may suffer from more overhead, we split all testcases into two categories: correct programs and buggy programs. A program is categorized as correct if its verification succeeds and buggy otherwise. Table 6.2 and Table 6.3 show that the average overhead for correct programs is indeed slightly larger. Further-

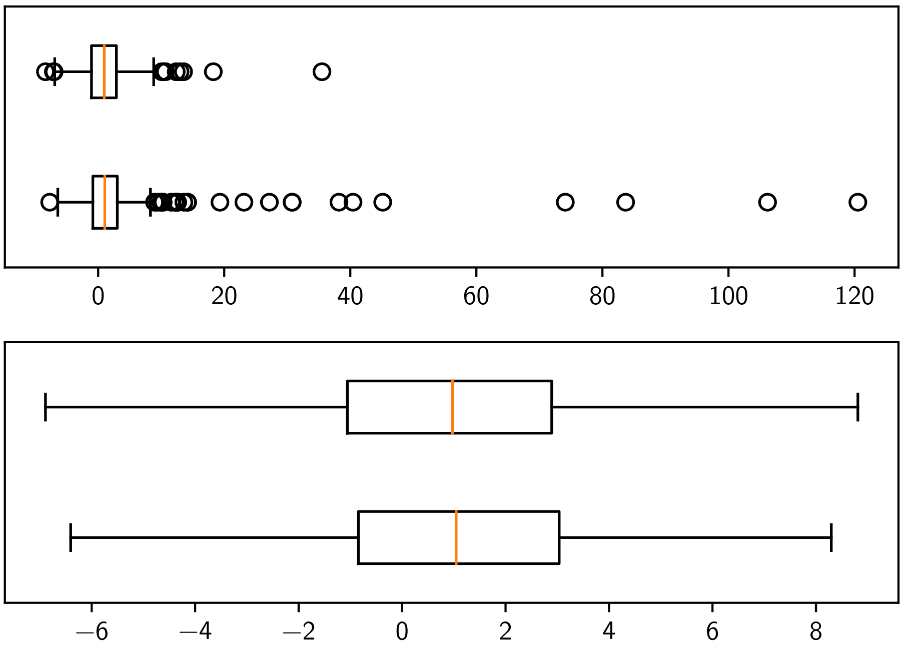

Figure 6.8: Box plots of the relative performance overhead (horizontal, in percent) of configuration 2 (enabling access witnesses) for buggy programs (top dataset) and correct programs (bottom dataset). While the first graph includes outliers, they are ignored in the second graph. A detailed description of the elements of each box plot is provided in Figure 6.7.

more, the maximum performance overhead for configurations with enabled support for access witnesses always occurs in correct programs. However, the median overhead barely changes and remains below 2% in all testcases. Analyzing the box plots of the performance overheads of both testsets in Figure 6.8, Figure 6.9, and Figure 6.10 again shows that only few testcases are responsible for the majority of the observed overhead. Furthermore, the verification of correct programs is only minimally slower than the verification of buggy programs. However, in configurations that enable witness support, it seems that outliers occur more often for correct programs than for buggy programs.

We omit a detailed breakdown of the default configuration here, as its performance is very similar to the baseline. Figure D.1 in Appendix D provides the corresponding box plot for the default configuration.

We also tested whether performance overhead is linked to the lines of code of each testcase (sloc) or the sum of all assignments, fold statements, and unfold statements. We chose that second metric because our modifications of Silicon significantly updated the symbolic execution rules for those statements. Figure 6.11 displays the performance overhead versus those two properties for

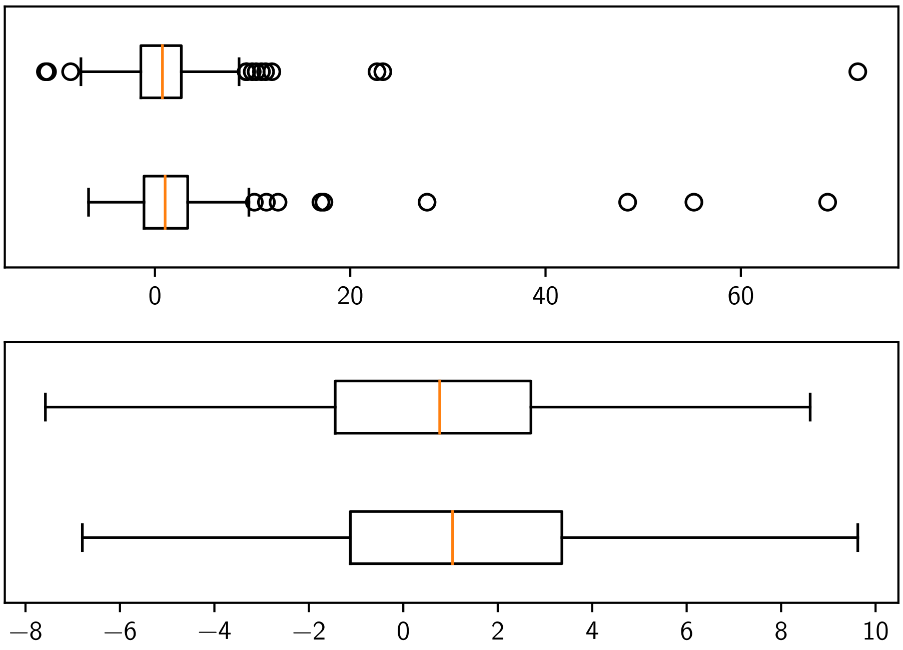

Figure 6.9: Box plots of the relative performance overhead (horizontal, in percent) of configuration 3 (enabling snapshot summarization) for buggy programs (top dataset) and correct programs (bottom dataset). While the first graph includes outliers, they are ignored in the second graph. A detailed description of the elements of each box plot is provided in Figure 6.7.

configuration 4. Both metrics seem to be rather unreliable to predict the performance overhead of a testcase. However, there are too few testcases with a large performance overhead to draw any conclusions. The same plots for configuration 2 and 3 are equally inconclusive and can be found in Figures D.2 and D.3 in Appendix D.

6.3.2 Prusti

Encoding

Since we changed the Rust-to-Viper encoding in Prusti, we also analyze the impact of those changes on verification performance. To do so, we let both the upstream version and our updated version of Prusti encode all Rust testcases from Prusti's testsuite to Viper programs. Those two new testsets of Viper programs (one for the upstream version of Prusti and one for our updated version) were then verified using our updated Silicon implementation, measuring the overall verification time. We used the identical test environment as for the Silicon evaluation. Again, all testcases were executed 5 times, measuring the the overall execution time and he fastest and slowest results were then

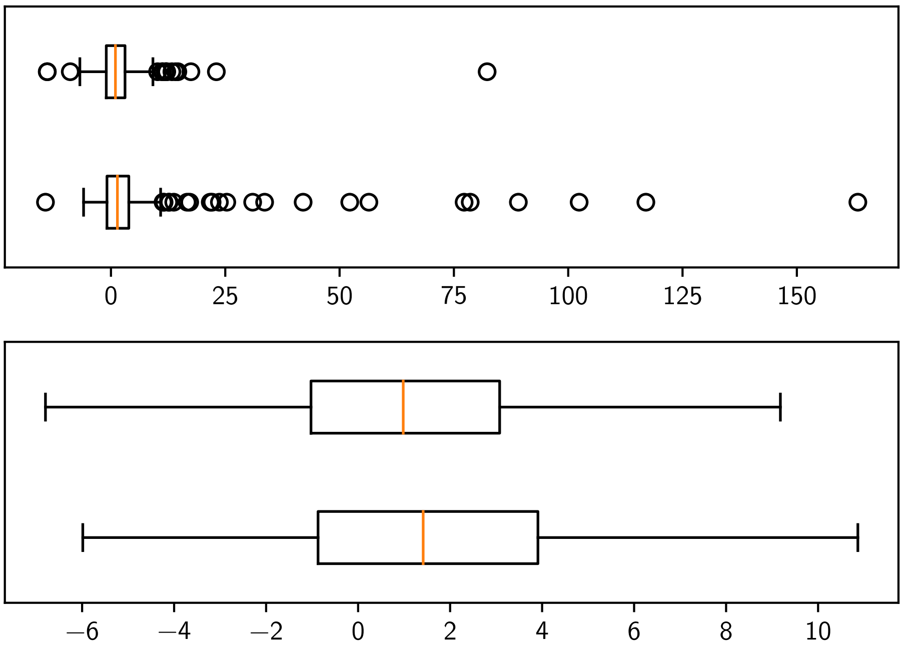

Figure 6.10: Box plots of the relative performance overhead (horizontal, in percent) of configuration 4 (enabling access witnesses and snapshot summarization) for buggy programs (top dataset) and correct programs (bottom dataset). While the first graph includes outliers, they are ignored in the second graph. A detailed description of the elements of each box plot is provided in Figure 6.7.

removed before taking the average of the remaining three measurements. The evaluated commit identifiers of Prusti are:

- `b65d22f2893279c1a9794b615de2cf06cb5bbaec` for the upstream version of Prusti.
- `35a1516b3b0551b7761657a123c0cd125847d4e8` for our updated version of Prusti.

The commit identifiers for our updated Silicon version are identical to the ones we already stated in the previous section. Note while the used Silicon version is identical for both versions of Prusti, the used configuration is different: to verify the testsuite generated by the upstream version we use configuration 3 (the default configuration plus snapshot summarization), while we use configuration 4 (default configuration with enabled snapshot summarization and access witnesses) to verify the testsuite generated by our updated version of Prusti.

The evaluation of the observed overheads shows that the verification of Viper programs generated by our updated version of Prusti has an overall perfor-

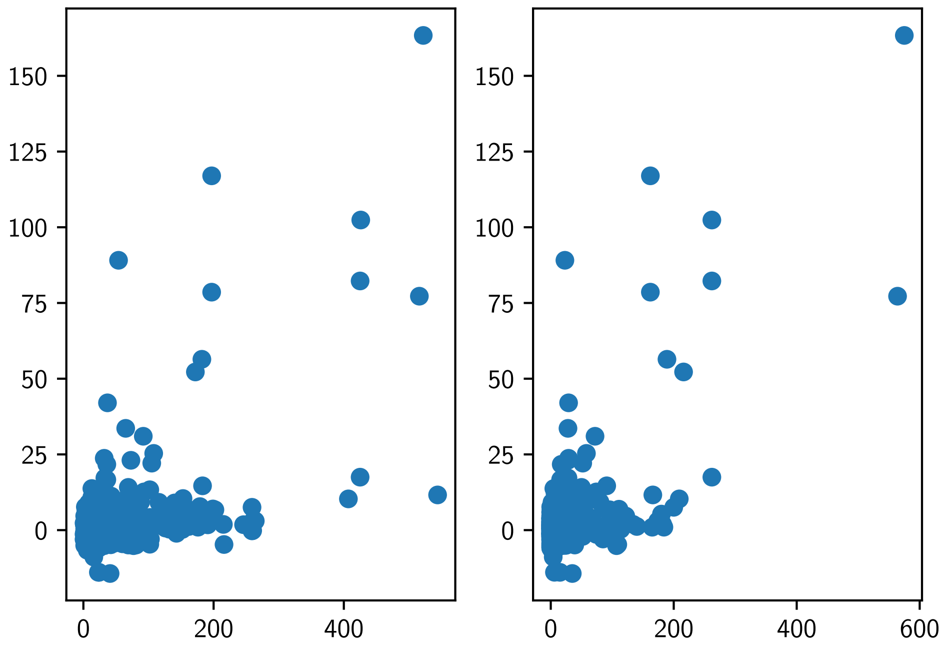

Figure 6.11: Performance overhead (vertical, in percent) of configuration 4 vs sloc of the testcase (horizontal, left) and its total number of assignments, fold statements, and unfold statements (horizontal, right) for all 812 testcases that take more than 100ms to verify. 11 failing testcases are not shown.

mance overhead of 43% compared to the Viper programs that were generated using the upstream version of Prusti. However, the median overhead is only 2.70%, while the maximum overhead is 254%. This again hints at few outliers causing most of the overall performance overhead, which is confirmed by the plot in Figure 6.12. Even tough the outliers account for a large part of the performance overhead, the verification of Viper programs in this testset suffers from a noticeable but non-critical slowdown.

Distinguishing between correct and buggy programs shows that the testcases with the largest overhead are again correct programs, while buggy programs have a maximum performance overhead of 98%. The median overhead for programs that fail during verification is 2.11%, compared to 3.22% for correct programs, drawing a similar picture to the one we observed during the analysis of the Silicon testcases. However, the box plots for the two groups of testcases in Figure 6.13 reveals that correct programs in fact do suffer from higher slowdown than buggy programs.

We assume that the higher overhead for programs with succeeding verification may be caused an accumulation of many access witness chunks over the course of the verification. A higher amount of access witnesses increases the

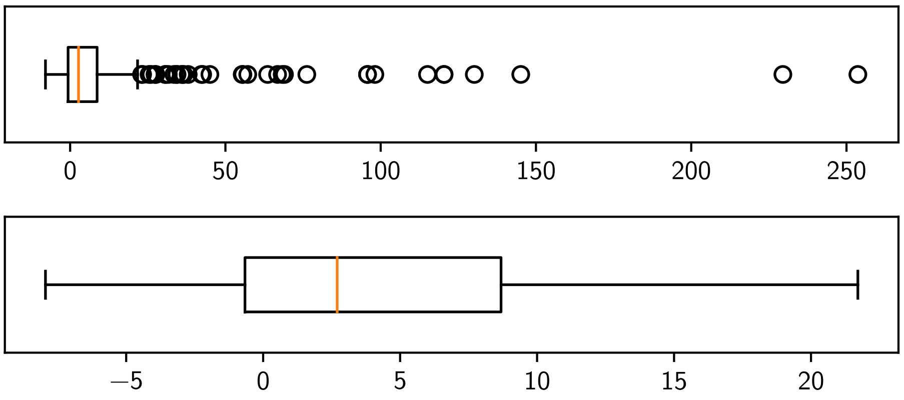

Figure 6.12: Box plots of the relative verification performance overhead (horizontal, in percent) of the updated Rust-to-Viper encoding over the upstream encoding. While the first graph includes outliers, they are ignored in the second graph. A detailed description of the elements of each box plot is provided in Figure 6.7.

cost of the expiration check that is performed after each heap update. Since verification failures abort the verification process, they prevent the accumulation of access witness chunks. On the other hand, programs that successfully verify cannot prevent this and thus may suffer from an increased overhead for the expiration check. However, further investigation is required to confirm this hypothesis.

Top 500

In order to check how our updated version of Prusti performs in a real world scenario, we used it to verify the top 500 Rust crates on `https://crates.io`. This evaluation is identical to the one conducted in part 1 of Section 7.2 of [1]. In the tested scenario we only suffer from roughly 17% performance overhead (8h 10min instead of 7h), which is noticeable but definitely not critical. No verification tasks failed or timed out.

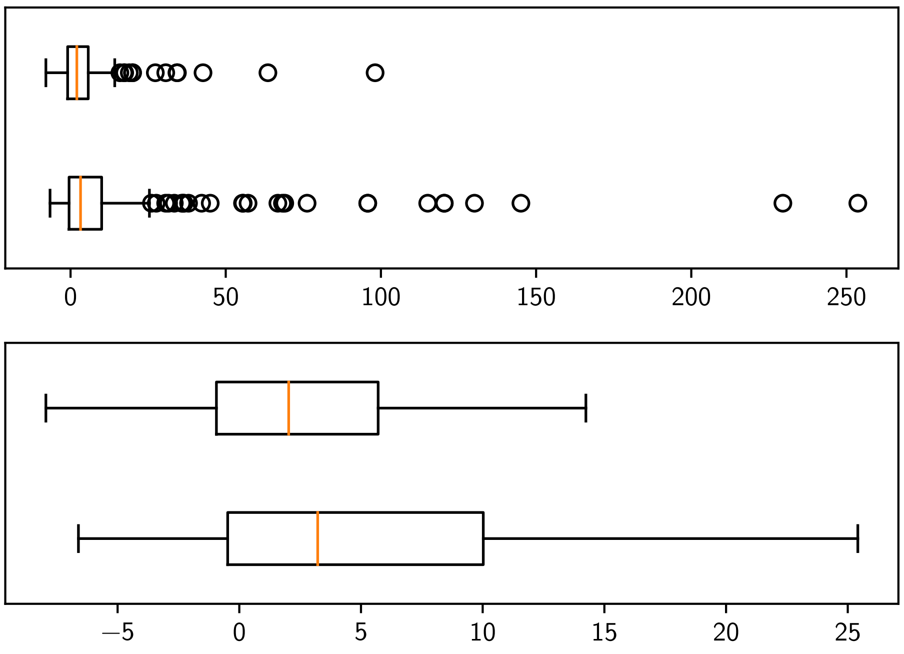

Figure 6.13: Box plots of the relative verification performance overhead (horizontal, in percent) of the updated Rust-to-Viper encoding over the upstream encoding for buggy programs (top dataset) and correct programs (bottom dataset). While the first graph includes outliers, they are ignored in the second graph. A detailed description of the elements of each box plot is provided in Figure 6.7.

Chapter 7

Future Work

Our first design and implementation of access witnesses opens various opportunities for future work. There is room for improvements and extensions of the underlying design of access witnesses, complementary language features, as well as implementation optimizations in both Silicon and Prusti. This chapter provides an overview about some of those opportunities.

7.1 Supporting Quantified Permissions

Section 3.4 mentions that access witnesses do not support quantified permissions. We have seen how the current implementation suffers from this limitation in Section 4.6. Supporting quantified permissions not only requires developing semantics on the Viper level, but also involves solving many open questions for the implementation. For example, one has to find a new way to model access witnesses that allows using quantified permissions as dependencies. As we already mentioned in Section 3.4, another challenge may be designing an algorithm to allow partial expiration of quantified access witnesses.

7.2 Incorporating Magic Wands

Besides fields and predicate instances, Viper also supports another resource type called *magic wands*. One could extend witnesses to support magic wands as dependencies, expressing that some permission is inside a magic wand's *footprint* (more details on magic wands in Viper can be found in [2] and [4], [9] describes magic wands in Silicon).

Since magic wands may occur within the body of predicates, one could also explore witnesses for magic wands (a witness that expresses that some magic wand is obtainable in the current state). Note that this most likely also includes defining semantics for temporarily applying magic wands, as well as maybe defining semantics for fractional magic wands.

7.3 Foldable Predicates

In our design, access witnesses can only express that a permission is present in the current program state; that is, it can be obtained by unfolding some predicate instance. One could extend the definition to also include predicate instances that can be folded in the current program state. This would have the beneficial effect of simplifying some aspects of the design (for example unfold operations would not need to update the dependencies of witnesses anymore). However, it also increases the overall complexity of access witnesses, and introduces the possibility of cycles in the dependency relation, so the drawbacks of the increased complexity must be compared to the benefits before proceeding with this idea.

7.4 Snapshot and Memory Equality

In Section 6.2.2 we mentioned that Viper currently provides no way to check whether two heap subsets are equal. As we have seen, this is a major issue when writing method specifications, especially for more complex Viper programs. Moreover, one may want to be able to state that an object was not modified, but may have been moved to a new heap location. Developing an extension that allows stating such properties would significantly simplify using access witnesses to their full potential.

7.5 Supporting Access Witnesses in Carbon

We only implemented access witness support in Silicon. Viper also features a second verifier based on verification condition generation named Carbon. Implementing support for access witnesses in Carbon is both important to provide a consistent user experience across the Viper ecosystem and for access witnesses to become a more mature language feature.

7.6 Addressing Summarization Incompletenesses

The summarization incompleteness we described in Section 6.1 is one of the major issues that kept reappearing as unexpected verification failures during the course of this thesis. It also forces Prusti to perform several optimizations of the generated Viper program before passing it to the verification backend. Developing a new, more complete approach to snapshot summarization would therefore not only resolve a major incompleteness in Silicon, but also has the potential to greatly simplify verification frontends that leverage Viper. Some starting points are provided in Section 6.1.

7.7 Field Lookups and Unfold Statements

The actual implementation of access witnesses is incomplete in the presence of a field lookup `r.f` when the heap contains a quantified chunk for field `f`. While the implementation of such field lookups is quite simple in the framework for symbolic execution (compare Figure 4.11), implementing it in the context of Silicon proves to be quite a bit more complex. Since Prusti does not use quantified permissions, the implementation of a field lookup rule that summarizes witness chunks and quantified chunks was deferred to future work.

Furthermore, unfold statements for access witnesses were not implemented as they can be simulated by unfolding expressions. Completing the implementation should be straightforward.

7.8 Selective Witness Generation

The current implementation creates many access witnesses that remain unused during verification. Our evaluation in Section 6.3.1 showed that the performance impact of enabling access witnesses is barely noticeable for many Viper programs. However, for few programs this overhead is as large as 120%, even tough those programs do not even use access witnesses. Hence, implementing an optimization that only generates the witnesses that are required to verify each function or method could significantly improve the verification performance for those programs.

7.9 Calling Pure Functions That Return Shared Borrows from Non-Pure Code

While our changes to the Rust-to-Viper encoding allow calling pure functions that return shared borrows in other pure functions as well as specifications, it is currently impossible to call them in non-pure code. The reason is that Prusti's internal bookkeeping for permissions and their state does not know how to deal with witnesses yet. To enable using such pure functions everywhere, one needs to incorporate access witnesses in the permission bookkeeping code, such that it can emit the required fold and unfold operations where they are required.

Chapter 8

Conclusion

We designed access witnesses—a new language feature for Viper—which extend the permission model to allow reasoning about permissions which are present in the current program state, but not directly available. Moreover, we also implemented them in a symbolic execution based backend verifier. We demonstrated the suitability of our language feature to improve the support for Rust programs in Prusti by extending its Rust-to-Viper encoding to include support for pure functions returning shared borrows. Our evaluation shows that our implementation significantly improves the expressiveness of Viper at the cost of an acceptable performance loss. Furthermore, we list several opportunities for future work to evolve and improve upon our work. This thesis is a first step in understanding the new permission design access witnesses enable.

Appendix A

Why Access Witnesses Must Not Provide Write Access

Access witnesses only provide read access. In Chapter 3, we mentioned that allowing write access would make the overall design unsound, as it would allow creating circular, recursively defined structures. Figure A.1 demonstrates this in a minimal example. It creates a circular `list` composed of a single element. Lines 18 to 24 create a list with a single element `a`. On line 26 we use `get_field_witnesses` to obtain access witnesses for all fields of `a`, which is aliased by a new variable `c`. Assuming that access witnesses provide write access, we reassign the `next` field of `a` to point to itself on line 27, thereby creating a circular list.

The program state after this assignment is not only inconsistent, but also violates one of the assumptions behind Viper's permission model, which is that every predicate instance can only be unfolded finitely many times. It is inconsistent, since the actual permission amount to `a.next` and `a.val` is not just greater than 1, but infinite. To obtain a permission amount of $n \in \mathbb{N}$, one just has to unfold `acc(list(a), write)` n times.

```
field val: Int
field next: Ref

predicate list(r: Ref) {
  acc(r.val, write) && acc(r.next, write) &&
    (r.next != null ==> acc(list(r.next), write))
}

function get_field_witnesses(r: Ref): Ref {
  requires dep(list(r))
  ensures dep(result.val, list(r))
            && dep(result.next, list(r))
{
  unfolding dep(list(r)) in r
}

method create_circular_list() {
  var a: Ref
  inhale acc(a.val, write)
  inhale acc(a.next, write)

  a.next := null
  fold acc(list(a), write)

  var c: Ref
  c := get_field_witnesses(a)
  c.next := a
}
```

Figure A.1: A minimal Viper program that demonstrates how witnesses that allow write access enable the creation of circular structures.

Appendix B

Symbolic Execution Rules of Utility Methods

B.1 produce'

```
produce'($\sigma, v : V, s, Q$) =
  $Q(\sigma\{\pi :=$ pc-add$(\sigma.\pi, \{v, s = unit\})\}, \varnothing)$

produce'($\sigma_1, e, s, Q$) =
  eval($\sigma_1, e, \lambda\ \sigma_2, e' \cdot$
    produce'($\sigma_2\{h := \sigma_1.h\}, e', s, Q$))

produce'($\sigma_1$,acc($id(\bar{e}), p$)$, s, Q$) =
  produce $id(\bar{e}, p)$ in its quantified version as $c$
  $Q(\sigma, (c, \bar{e}))$

produce'($\sigma_1, a_1$ && $a_2, s, Q$) =
  produce'($\sigma_1, a_1, first(s), \lambda\ \sigma_2, c_1 \cdot$
    produce'($\sigma_2, a_2, second(s), \lambda\ \sigma_3, c_2 \cdot$
      $Q(\sigma_3, c_1 \cup c_2)$)))

produce'($\sigma_1, e?a_1 : a_2, s, Q$) =
  eval($\sigma_1, e, \lambda\ \sigma_2, e' \cdot$
    branch($\sigma_2, e'$,
      $\lambda\ \sigma_3 \cdot$ produce'($\sigma_3, a_1, Q$),
      $\lambda\ \sigma_3 \cdot$ produce'($\sigma_3, a_2, Q$))))

produce'($\sigma_1$,forall $x : T$ :: $c(x)$ ==> acc($id(\overline{e(x)}), p(x)$)$, s, Q$) =
  exactly like the rule from [6], and returning $\{ch, \varnothing\}$
```

B.2 produce-dependent

```
produce-dependent($\sigma_1, a, id(\overline{e}), s, Q$) =
  eval($\sigma_1, \overline{e}, \lambda\ \sigma_2, \overline{e'}\ \cdot$
    Let $parents \subseteq \sigma.h$ be the set of all chunks for $id(\overline{e'})$
    produce-dependent'($\sigma_1, a, \{(p, \overline{e'}) \mid p \in parents\}, s, Q$))

produce-dependent'($\sigma, e, s, parents, Q$) =
  produce($\sigma, e, s, Q$)

produce-dependent'($\sigma_1$, acc($id(\overline{e}), p$), $s, parents, Q$) =
  eval($\sigma_1, p :: \overline{e}, \lambda\ \sigma_2, p' ::: \overline{e'}\ \cdot$
    Let $v$ be $e' \neq null$ if $id$ denotes a field, and $true$ otherwise
    $c := id^*(\overline{e'}, parents, s, p')$
    $h_3 := \sigma_1.h \cup \{c\}$
    $Q(\sigma_2\{h := h_3\})$)

produce-dependent'($\sigma_1, a_1$ && $a_2, s, parents, Q$) =
  produce-dependent'($\sigma_1, a_1, first(s), parents, \lambda\ \sigma_2\ \cdot$
    produce-dependent'($\sigma_2, a_2, second(s), parents, Q$))

produce-dependent'($\sigma_1, e\ ?\ a_1 : a_2, s, parents, Q$) =
  eval($\sigma_1, e, \lambda\ \sigma_2, e'\ \cdot$
    branch($\sigma_2\{h := \sigma_1.h\}, e'$,
      $\lambda\ \sigma_3 \cdot$ produce-dependent'($\sigma_3, a_1, s, parents, Q$),
      $\lambda\ \sigma_3 \cdot$ produce-dependent'($\sigma_3, a_2, s, parents, Q$)))

produce-dependent'($\sigma_1$, forall $x : T$ :: $c(x)$ ==>
                              acc($id(\overline{e(x)}), p(x)$), $s, parents, Q$) =
  $Q(\sigma_1)$
```

Appendix C

Viper Programs from Expressiveness Evaluation

This chapter provides the complete Viper programs that were developed during the evaluation in Section 6.2. Note that both examples use syntax that is not described in Chapter 2. Consult the Viper tutorial [4] for the missing details.

C.1 Purified Solution

```
domain Snapshot {
  function snap$item(sv: Int): Snapshot
  function snap$list(el: Snapshot, n: Snapshot): Snapshot
  function nullsnap(): Snapshot

  function tag(s: Snapshot): Int

  axiom tagdef_nullsnap {
    tag(nullsnap()) == 0
  }
  axiom tagdef_list {
    forall s: Snapshot, n: Snapshot ::
      {snap$list(s,n)} tag(snap$list(s,n)) == 1
  }
  axiom tagdef_item {
    forall sv: Int :: {snap$item(sv)} tag(snap$item(sv)) == 2
  }

  function snap$item$inv(s: Snapshot): Int
  function snap$list$inv1(s: Snapshot): Snapshot
  function snap$list$inv2(s: Snapshot): Snapshot
```

```

axiom item1 {
  forall i: Int ::
    {snap$item(i)} snap$item$inv(snap$item(i)) == i
}
axiom item2 {
  forall s: Snapshot ::
    {snap$item$inv(s)} snap$item(snap$item$inv(s)) == s
}

axiom list1 {
  forall s1: Snapshot, s2: Snapshot ::
    {snap$list(s1,s2)} snap$list$inv1(snap$list(s1,s2)) == s1
}
axiom list2 {
  forall s1: Snapshot, s2: Snapshot ::
    {snap$list(s1,s2)} snap$list$inv2(snap$list(s1,s2)) == s2
}
axiom list3 {
  forall s: Snapshot ::
    {snap$list$inv1(s)}{snap$list$inv2(s)}
    snap$list(snap$list$inv1(s), snap$list$inv2(s)) == s
}

function trigger$list(l: Snapshot): Bool
function trigger$item(l: Snapshot): Bool

axiom trgf$list {
  forall l: Snapshot :: {trigger$list(l)} trigger$list(l)
}

axiom trgf$item {
  forall l: Snapshot :: {trigger$item(l)} trigger$item(l)
}

function lte(l: Snapshot, r: Snapshot): Bool
function lte$limited(l: Snapshot, r: Snapshot): Bool
function lte$stateless(l: Snapshot, r: Snapshot): Bool

axiom lte$def {
  forall l: Snapshot, r: Snapshot :: { lte(l, r) }
    lte(l, r) == (snap$item$inv(l) <= snap$item$inv(r))
}

```

```
axiom lte$aux1 {
  forall l: Snapshot, r: Snapshot :: { lte(l, r) }
    lte(l, r) == lte$limited(l, r)
}

axiom lte$aux2 {
  forall l: Snapshot, r: Snapshot :: { lte$limited(l, r) }
    lte$stateless(l, r)
}

function at(l: Snapshot, i: Int): Snapshot
function at$limited(l: Snapshot, i: Int): Snapshot
function at$stateless(l: Snapshot, i: Int): Bool

axiom at$def {
  forall l: Snapshot, i: Int ::
    { at(l,i) }{ at$stateless(l,i), trigger$list(l) }
    at(l,i) == (i == 0 ?
      snap$list$inv1(l) :
      at$limited(snap$list$inv2(l), i-1))
}

axiom at$aux1 {
  forall l: Snapshot, i: Int :: { at(l,i) }
    at$limited(l,i) == at(l,i)
}

axiom at$aux2 {
  forall l: Snapshot, i: Int :: { at$limited(l,i) }
    at$stateless(l,i)
}

function length(l: Snapshot): Int
function length$limited(l: Snapshot): Int
function length$stateless(l: Snapshot): Bool

axiom length$def {
  forall l: Snapshot ::
    { length(l) }{ length$stateless(l), trigger$list(l) }
    length(l) == (snap$list$inv2(l) == nullsnap() ?
      1 :
      1 + length$limited(snap$list$inv2(l)))
}

```

```
  axiom length$post {
    forall l: Snapshot :: { length$limited(l) }
      length$limited(l) > 0
  }

  axiom length$aux1 {
    forall l: Snapshot :: { length(l) }
      length$limited(l) == length(l)
  }

  axiom length$aux2 {
    forall l: Snapshot :: { length$limited(l) }
      length$stateless(l)
  }
}

field sort_value: Int

predicate item(r: Ref) {
  acc(r.sort_value)
}

function getsnap$item(r: Ref): Snapshot
  requires item(r)
{
  unfolding item(r) in snap$item(r.sort_value)
}

field element: Ref
field next: Ref

predicate list(r: Ref) {
  acc(r.element) && acc(r.next) &&
    item(r.element) && (r.next != null ==> list(r.next))
}

function getsnap$list(r: Ref): Snapshot
  requires list(r)
{
  unfolding list(r) in snap$list(
    getsnap$item(r.element),
    r.next == null ? nullsnap() : getsnap$list(r.next))
}

```

```
method insert_first(r: Ref, it: Ref) returns (node: Ref)
  requires r != null
  requires list(r)
  requires item(it)
  requires forall i: Int, j: Int ::
    (0 <= i && i <= j && j < length(getsnap$list(r))) ==>
      lte(at(getsnap$list(r), i), at(getsnap$list(r), j))
  requires forall i: Int ::
    (0 <= i && i < length(getsnap$list(r))) ==>
      lte(getsnap$item(it), at(getsnap$list(r), i))
  ensures node != null
  ensures list(node)
  ensures length(getsnap$list(node)) ==
    old(length(getsnap$list(r))) + 1
  ensures old(getsnap$item(it)) == at(getsnap$list(node), 0)
  ensures forall i: Int ::
    1 <= i && i < length(getsnap$list(node)) ==>
    at(getsnap$list(node), i) == old(at(getsnap$list(r), i - 1))
  ensures forall i: Int, j: Int ::
    (0 <= i && i <= j && j < length(getsnap$list(node))) ==>
      lte(at(getsnap$list(node), i), at(getsnap$list(node), j))
{
  node := new(element, next)
  node.element := it
  node.next := r
  fold list(node)
  assert trigger$list(getsnap$list(node))
}

method insert_last(r: Ref, it: Ref)
  requires r != null
  requires list(r)
  requires item(it)
  requires forall i: Int, j: Int ::
    { at(getsnap$list(r),i), at(getsnap$list(r),j) }
    (0 <= i && i <= j && j < length(getsnap$list(r))) ==>
      lte(at(getsnap$list(r), i), at(getsnap$list(r), j))
  requires forall i: Int :: { at(getsnap$list(r),i) }
    (0 <= i && i < length(getsnap$list(r))) ==>
      lte(at(getsnap$list(r), i), getsnap$item(it))
  ensures list(r)
  ensures length(getsnap$list(r)) ==
    old(length(getsnap$list(r))) + 1
  ensures at(getsnap$list(r), length(getsnap$list(r)) - 1) ==
    old(getsnap$item(it))
  ensures forall i: Int :: { at(getsnap$list(r),i) }
    (0 <= i && i < length(getsnap$list(r)) - 1) ==>
      at(getsnap$list(r), i) == old(at(getsnap$list(r), i))
  ensures forall i: Int, j: Int ::
    {at(getsnap$list(r), i), at(getsnap$list(r), j)}
    (0 <= i && i <= j && j < length(getsnap$list(r))) ==>
      lte(at(getsnap$list(r), i), at(getsnap$list(r), j))
{
  if (length(getsnap$list(r)) == 1) {
    var node: Ref
    node := new(element, next)
    node.next := null
    node.element := it
    fold list(node)
    assert trigger$list(getsnap$list(node))
    assert trigger$list(getsnap$list(r))
    unfold list(r)
    r.next := node
    fold list(r)
    assert trigger$list(getsnap$list(r))
    assert old(at(getsnap$list(r), 0)) == at(getsnap$list(r), 0)
  } else  {
    assert trigger$list(getsnap$list(r))
    unfold list(r)
    label k
    assert forall i: Int ::
      (0 <= i && i < length(getsnap$list(r.next))) ==>
        old(at(getsnap$list(r), i + 1)) ==
          at(getsnap$list(r.next), i)
    insert_last(r.next, it)
    fold list(r)
    assert trigger$list(getsnap$list(r))
    assert forall i: Int ::
      (1 <= i && i < length(getsnap$list(r))-1) ==>
        old[k](at(getsnap$list(r.next), i-1)) ==
          at(getsnap$list(r), i)
    assert old(at(getsnap$list(r), 0)) == at(getsnap$list(r), 0)
  }
}

method insert_at(r: Ref, it: Ref, pos: Int) returns (node: Ref)
  requires r != null
  requires list(r)
  requires forall i: Int, j: Int ::
    { at(getsnap$list(r),i), at(getsnap$list(r),j) }
    (0 <= i && i <= j && j < length(getsnap$list(r))) ==>
      lte(at(getsnap$list(r), i), at(getsnap$list(r), j))
  requires item(it)
  requires 0 <= pos && pos <= length(getsnap$list(r))
  requires forall i: Int :: { at(getsnap$list(r),i) }
    (0 <= i && i < pos) ==>
      lte(at(getsnap$list(r), i), getsnap$item(it))
  requires forall i: Int :: { at(getsnap$list(r),i) }
    (pos <= i && i < length(getsnap$list(r))) ==>
      lte(getsnap$item(it), at(getsnap$list(r), i))
  ensures node != null
  ensures list(node)
  ensures length(getsnap$list(node)) ==
    old(length(getsnap$list(r))) + 1
  ensures forall i: Int ::
    { at(getsnap$list(node),i) }{ at(getsnap$list(r), i) }
    (0 <= i && i < pos) ==>
      at(getsnap$list(node), i) == old(at(getsnap$list(r), i))
  ensures at(getsnap$list(node), pos) == old(getsnap$item(it))
  ensures forall i: Int ::
    { at(getsnap$list(r),i) }{ at(getsnap$list(node),i) }
    (pos < i && i < length(getsnap$list(node))) ==>
      at(getsnap$list(node), i) ==
        old(at(getsnap$list(r), i - 1))
  ensures forall i: Int, j: Int ::
    { at(getsnap$list(node),i), at(getsnap$list(node),j) }
    (0 <= i && i <= j && j < length(getsnap$list(node))) ==>
      lte(at(getsnap$list(node), i), at(getsnap$list(node), j))
{
  if (pos == 0) {
    node := insert_first(r, it)
  } else {
    if (pos == length(getsnap$list(r))) {
      insert_last(r, it)
      node := r
    } else {
      assert trigger$list(getsnap$list(r))
      unfold list(r)
      assert forall i: Int ::
        (0 <= i && i < length(getsnap$list(r.next))) ==>
          old(at(getsnap$list(r), i + 1)) ==
            at(getsnap$list(r.next), i)
      label k
      node := insert_at(r.next, it, pos - 1)
      r.next := node
      assert forall i: Int ::
        (pos - 1 < i && i < length(getsnap$list(r.next))) ==>
          at(getsnap$list(r.next), i) ==
            old(at(getsnap$list(r), i))
      label l
      fold list(r)
      assert trigger$list(getsnap$list(r))
      assert old(at(getsnap$list(r), 0)) ==
        at(getsnap$list(r), 0)
      assert forall i: Int :: (1 <= i && i < pos) ==>
        old[k](at(getsnap$list(r.next), i-1)) ==
          at(getsnap$list(r), i)
      node := r
    }
  }
}
```

C.2 Hybrid Solution

```
domain Snapshot {
  function snap$item(sv: Int): Snapshot
  function snap$list(el: Snapshot, n: Snapshot): Snapshot
  function nullsnap(): Snapshot

  function snap$item$inv(s: Snapshot): Int
  function snap$list$inv1(s: Snapshot): Snapshot
  function snap$list$inv2(s: Snapshot): Snapshot

  axiom item1 {
    forall i: Int ::
      {snap$item(i)} snap$item$inv(snap$item(i)) == i
  }
  axiom item2 {
    forall s: Snapshot ::
      {snap$item$inv(s)} snap$item(snap$item$inv(s)) == s
  }

  axiom list1 {
    forall s1: Snapshot, s2: Snapshot ::
      {snap$list(s1,s2)} snap$list$inv1(snap$list(s1,s2)) == s1
  }
  axiom list2 {
    forall s1: Snapshot, s2: Snapshot ::
      {snap$list(s1,s2)} snap$list$inv2(snap$list(s1,s2)) == s2
  }
  axiom list3 {
    forall s: Snapshot ::
      {snap$list$inv1(s)}{snap$list$inv2(s)}
      snap$list(snap$list$inv1(s), snap$list$inv2(s)) == s
  }
}

field sort_value: Int
field element: Ref
field next: Ref

predicate item(r: Ref) {
  acc(r.sort_value)
}

function getsnap$item(r: Ref): Snapshot
  requires dep(item(r))
{
  unfolding dep(item(r)) in snap$item(r.sort_value)
}

function lte(l: Ref, r: Ref): Bool
  requires dep(item(l)) && dep(item(r))
{
  (unfolding dep(item(l)) in l.sort_value) <=
    (unfolding dep(item(r)) in r.sort_value)
}

predicate list(r: Ref) {
  acc(r.element) && acc(r.next) && item(r.element) &&
    (r.next != null ==> list(r.next))
}

function getsnap$list(r: Ref): Snapshot
  requires dep(list(r))
{
  unfolding dep(list(r)) in snap$list(
    getsnap$item(r.element),
    r.next == null ?  nullsnap() : getsnap$list(r.next))
```

```
}

function length(r: Ref): Int
  requires dep(list(r))
  ensures result > 0
{
  unfolding dep(list(r)) in r.next == null ?
    1 :
    1 + length(r.next)
}

function at(r: Ref, i: Int): Ref
  requires dep(list(r)) && i in [0..length(r))
  ensures dep(item(result), list(r))
{
  unfolding dep(list(r)) in i == 0 ?
    r.element :
    at(r.next, i - 1)
}

method insert_first(r: Ref, it: Ref) returns (node: Ref)
  requires r != null
  requires list(r)
  requires item(it)
  requires forall i: Int, j: Int ::
    (0 <= i && i <= j && j < length(r)) ==>
      lte(at(r, i), at(r, j))
  requires forall i: Int ::
    (0 <= i && i < length(r)) ==> lte(it, at(r, i))
  ensures node != null
  ensures list(node)
  ensures length(node) == old(length(r)) + 1
  ensures old(getsnap$item(it)) == getsnap$item(at(node, 0))
  ensures forall i: Int ::
    (1 <= i && i < length(node)) ==>
      getsnap$item(at(node, i)) ==
        old(getsnap$item(at(r, i - 1)))
  ensures forall i: Int, j: Int ::
    (0 <= i && i <= j && j < length(node)) ==>
      lte(at(node, i), at(node, j))
{
  node := new(element, next)
  node.element := it
  node.next := r
  assert unfolding list(r) in true
  fold list(node)
}

method insert_last(r: Ref, it: Ref)
  requires r != null
  requires list(r)
  requires item(it)
  requires forall i: Int, j: Int :: { at(r,i), at(r,j) }
    (0 <= i && i <= j && j < length(r)) ==>
      lte(at(r, i), at(r, j))
  requires forall i: Int :: { at(r,i) }
    (0 <= i && i < length(r)) ==> lte(at(r, i), it)
  ensures list(r)
  ensures length(r) == old(length(r)) + 1
  ensures getsnap$item(at(r, length(r) - 1)) ==
    old(getsnap$item(it))
  ensures forall i: Int :: { at(r,i) }
    (0 <= i && i < length(r) - 1) ==>
      getsnap$item(at(r, i)) == old(getsnap$item(at(r, i)))
  ensures forall i: Int, j: Int :: {at(r, i), at(r, j)}
    (0 <= i && i <= j && j < length(r)) ==>
      lte(at(r, i), at(r, j))
{
  if (length(r) == 1) {
    var node: Ref
    node := new(element, next)
    node.next := null
    node.element := it
    fold list(node)
    unfold list(r)
    r.next := node
    fold list(r)
  } else  {
    unfold list(r)
    assert forall i: Int ::
      (0 <= i && i < length(r.next)) ==>
        old(at(r, i + 1)) == at(r.next, i)
    insert_last(r.next, it)
    fold list(r)
  }
}

method insert_at(r: Ref, it: Ref, pos: Int) returns (node: Ref)
```

```
  requires r != null
  requires list(r)
  requires forall i: Int, j: Int :: { at(r,i), at(r,j) }
    (0 <= i && i <= j && j < length(r)) ==>
      lte(at(r, i), at(r, j))
  requires item(it)
  requires 0 <= pos && pos <= length(r)
  requires forall i: Int :: { at(r,i) }
    (0 <= i && i < pos) ==> lte(at(r, i), it)
  requires forall i: Int :: { at(r,i) }
    (pos <= i && i < length(r)) ==> lte(it, at(r, i))
  ensures node != null
  ensures list(node)
  ensures length(node) == old(length(r)) + 1
  ensures forall i: Int :: { at(node,i) }{ at(r, i) }
    (0 <= i && i < pos) ==> getsnap$item(at(node, i)) ==
      old(getsnap$item(at(r, i)))
  ensures getsnap$item(at(node, pos)) == old(getsnap$item(it))
  ensures forall i: Int :: { at(r,i) }{ at(node,i) }
    (pos < i && i < length(node)) ==>
      getsnap$item(at(node, i)) ==
        old(getsnap$item(at(r, i - 1)))
  ensures forall i: Int, j: Int :: { at(node,i), at(node,j) }
    (0 <= i && i <= j && j < length(node)) ==>
      lte(at(node, i), at(node, j))
{
  if (pos == 0) {
    node := insert_first(r, it)
  } else {
    if (pos == length(r)) {
      insert_last(r, it)
      node := r
    } else {
      unfold list(r)
      assert forall i: Int ::
        (0 <= i && i < length(r.next)) ==>
          old(at(r, i + 1)) == at(r.next, i)
      node := insert_at(r.next, it, pos - 1)
      r.next := node
      fold list(r)
      node := r
    }
  }
}
```

Appendix D

Additional Plots from Performance Evaluation

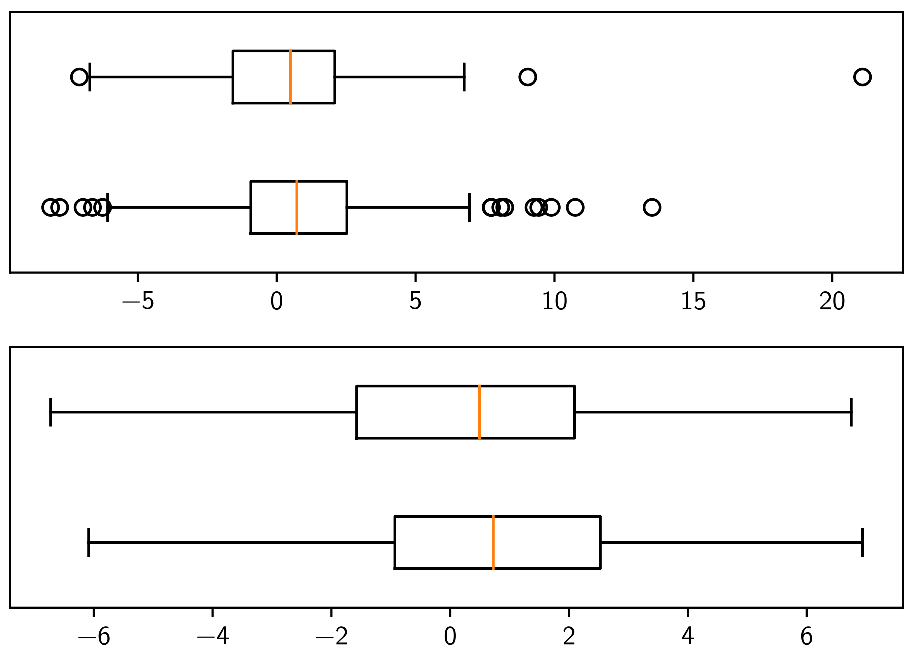

Figure D.1: Box plots of the relative performance overhead (horizontal, in percent) of configuration 1 (default configuration) for buggy programs (top dataset) and correct programs (bottom dataset). While the first graph includes outliers, they are ignored in the second graph. A detailed description of the elements of each box plot is provided in Figure 6.7.

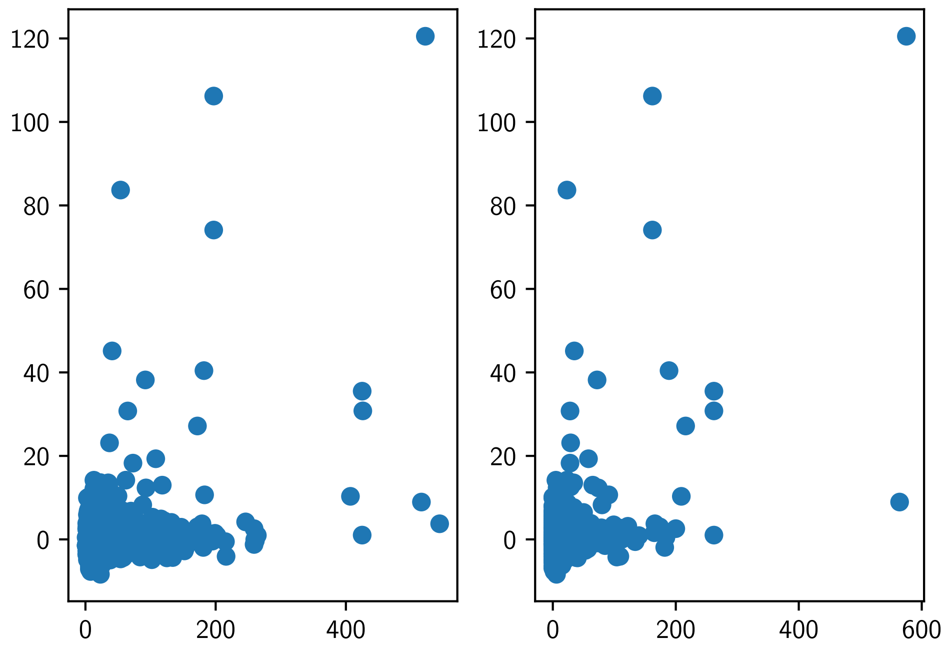

Figure D.2: Performance overhead (vertical, in percent) of configuration 2 (enabling access witnesses) vs sloc of the testcase (horizontal, left) and its total number of assignments, fold statements, and unfold statements (horizontal, right) for all 821 testcases that take more than 100ms to verify. 2 failing testcases are not shown.

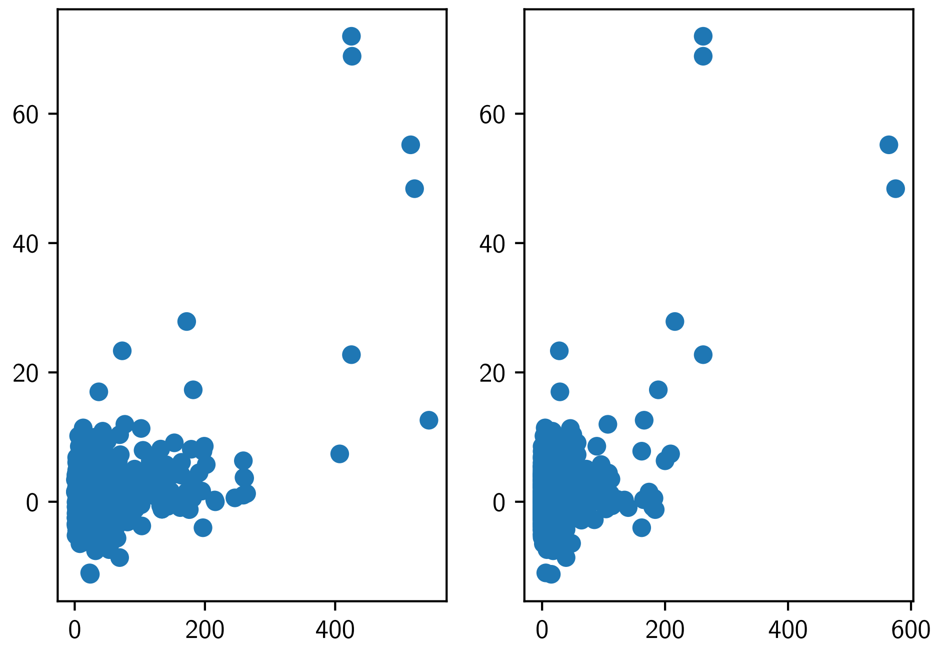

Figure D.3: Performance overhead (vertical, in percent) of configuration 3 (enabling snapshot summarization) vs sloc of the testcase (horizontal, left) and its total number of assignments, fold statements, and unfold statements (horizontal, right) for all 814 testcases that take more than 100ms to verify. 9 failing testcases are not shown.

Bibliography

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification", ETH Zurich, Tech. Rep., 2019. DOI: 10.3929/ethz-b-000311092.

[2] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning", in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, B. Jobstmann and K. R. M. Leino, Eds., ser. LNCS, vol. 9583, Springer-Verlag, 2016, pp. 41–62.

[3] J. Smans, B. Jacobs, and F. Piessens, "Implicit dynamic frames: Combining dynamic frames and separation logic", in *ECOOP 2009 – Object-Oriented Programming*, S. Drossopoulou, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 148–172, ISBN: 978-3-642-03013-0.

[4] Viper tutorial, [Online]. Available: http://viper.ethz.ch/tutorial/ (visited on 08/12/2019).

[5] M. Moskal, "Programming with triggers", in *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, ser. SMT '09, Montreal, Canada: ACM, 2009, pp. 20–29, ISBN: 978-1-60558-484-3. DOI: 10.1145/1670412.1670416. [Online]. Available: http://doi.acm.org/10.1145/1670412.1670416.

[6] M. H. Schwerhoff, "Advancing automated, permission-based program verification using symbolic execution", PhD thesis, ETH Zurich, 2016.

[7] R. Sierra, "Towards customizability of a symbolic-execution-based program verifier", Bachelor's Thesis, 2017. [Online]. Available: https://www.ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Robin_Sierra_BA_report.pdf (visited on 03/14/2019).

[8] S. Heule, I. T. Kassios, P. Müller, and A. J. Summers, "Verification condition generation for permission logics with abstract predicates and abstraction functions", in *European Conference on Object-Oriented Programming (ECOOP)*, G. Castagna, Ed., ser. Lecture Notes in Computer Science, vol. 7920, Springer, 2013, pp. 451–476.

[9] M. Schwerhoff and A. J. Summers, "Lightweight support for magic wands in an automatic verifier", ETH Zurich, Tech. Rep., 2014. DOI: 10.3929/ethz-a-010089016.

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):

Supporting Borrows in Specification Functions of 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):**
Trüssel	Nicolas

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)**
Möhlin, September 4, 2019	N. Trüssel

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