ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

An Abstract Representation for Wildcard Permissions in Viper

Bachelor Thesis

Yanick Bachmann

July 31, 2021

Advisors: Prof. Dr. Peter Müller, Thibault Dardinier
Department of Computer Science, ETH Zürich

Abstract

As computer programs get ever more complex they are more prone to bugs and security vulnerabilities. Formal verification techniques can be applied to get guarantees for the correct behavior of programs for any execution and thus keep this complexity in check. Verification infrastructures that automatically apply these techniques thus get increasingly important. Viper provides such a verification infrastructure through its intermediate verification language and its two back-end verifiers. It uses separation logic to allow reasoning about heap-manipulating and concurrent programs. In particular, it uses permissions encoded as fractional values representing read and write privileges to heap locations. Wildcards are a means for representing unspecified positive permission amounts. They can be used, among other things, for encoding read permissions to heap locations for concurrent programs. In this thesis we explore an alternative representation for such wildcard permission amounts in Viper with an emphasis on performance.

Contents

Chapter 1

Introduction

We want our programs to be fast, secure and reliable. Achieving all three of these goals is increasingly difficult, as programs get larger and more complex. Such complex software is infamous for the difficulty of finding implementation bugs and critical security vulnerabilities. To prevent them from occurring in the first place, it is thus ever more important to be able to formally verify the correctness of such programs. This provides guarantees that programs behave as expected for any execution and thus allows programmers to keep the complexity in check.

The Viper verification infrastructure [6] tries to tackle this challenge by providing an intermediate verification language that specializes in the verification of heap-manipulating programs. At the heart of Viper is a permission model that allows for the reasoning about heap-manipulating programs and thread interactions in concurrent software. Various existing automatic verifiers for programming languages such as Python, Rust, Go, and Java verify programs by translation to Viper. There also exist two interchangeable verification back-ends called *Carbon* and *Silicon*, both of which use an SMT solver called *z3* [2] for the verification. For a pictorial representation of this whole tool chain see Figure 1.1.

Viper uses permissions to express ownership of heap locations. This permission system assigns heap locations a fraction between zero and one, where no permission is held if the fractional amount is zero, read permission is held if this fraction is positive and a write permission is held if the fraction is equal to one. Viper provides a feature for representing unspecified positive permission amounts called *wildcards*. They can for example be used to model read permissions. We explore an alternative representation for wildcard permissions in Viper inspired by the *VeriFast* verifier [4], which as the name suggests is optimized for fast verification speeds. We implement this feature in the Carbon back-end of Viper and our focus lies on the performance of this alternative representation.

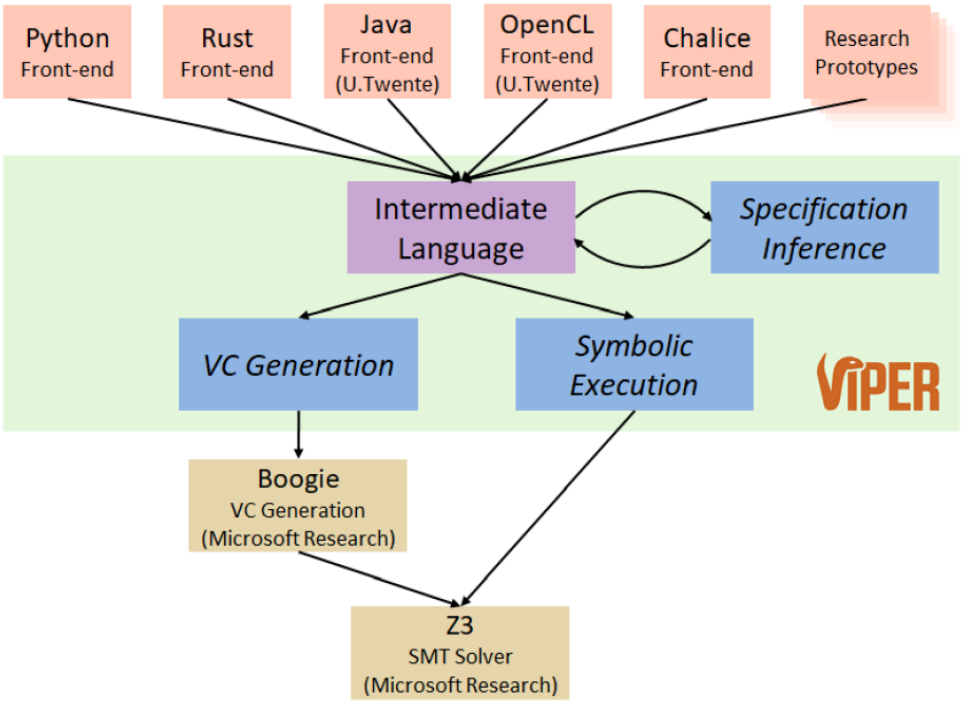

Figure 1.1: The Viper toolchain

Outline The next Chapter 2 will give some background on the verification techniques employed by Viper, wildcard permissions and their encoding in the Carbon back-end. Chapter 3 explains the proposed abstract wildcard representation we explore in this thesis and show how we implemented it in the Carbon back-end. We give a simplified proof for the soundness of the proposed abstract wildcard representation in Chapter 4 and in Chapter 5 we evaluate the new representation against the current implementation. We conclude in Chapter 6.

Chapter 2

Background

In this chapter we explain the technical background for this project. First we get into separation logic, then we focus on one particular extension of this logic, namely fractional permissions. This will lead us to an explanation of wildcard permission amounts and their encoding in Viper.

2.1 Separation Logic

Reasoning formally about programs that share mutable data structures is critical for ensuring their correctness and security. Such programs can become arbitrarily complex as any C programmer who has ever debugged pointer errors knows. Separation Logic [7] provides a means for proving formal properties about such programs while not being limited to concurrent programs. It is an extension of Hoare Logic [3] that enables local reasoning and the formalization of access rights to heap locations.

Partial Heaps Separation logic makes use of the concept of partial heaps. They represents a part of the heap and are formally modeled as partial functions, which map heap locations to values, e.g. $x.f \rightarrow 5$, where x is a reference to a heap location, f a field of this location and 5 is the value stored on this location. A partial heap and a store, which maps the names of local variables to their values are both part of the state.

Hoare Triples in Separation Logic As mentioned above, separation logic extends Hoare Logic. The meaning of Hoare triples is modified but they still have the same structure containing pre- and postconditions and a program. A Hoare triple holds if and only if the program is executed in a starting state satisfying the precondition then the final state satisfies the postcondition. Separation logic adds deduction rules that allow the description of ownership of heap locations that can be accessed and modified by the program.

Points-to Assertions An important type of assertion introduced by separation logic is the points-to assertion. We can think of it as a way of encoding ownership of heap locations. Reading and writing operations on heap locations are only permitted if the state owns the corresponding heap locations. For example, if we want to assert whether the partial heap of a state maps the heap location $x.f$ to the value v, the points-to assertion $x.f \rightarrow v$ can be used. This ownership is also transferable from the caller to callee. For example when a method holding some permission to a heap location calls a subroutine the permission amount associated with this location can be transferred to the subroutine.

Local Reasoning Separation Logic also enables local reasoning about programs which means properties can be proved on a small local state. These proofs are then never affected by enlarging this state and the additional state will also never be modified by the program. This is done by introducing a new rule, the frame rule, enabling local reasoning. Modular verification is thus possible which is important for the verification of large scale programs since the complexity of algorithms verifying the validity of the specified conditions grows linearly with the number of blocks and therefore they are reasonably scalable.

Implicit dynamic frames Implicit dynamic frames[8] is a variant of separation logic and is used by Viper. It separates the permission amount held on and the value of a heap location. Holding at least some fractional permission p to a heap location `x.f`, where `x` is a reference and `f` a field can be asserted separately form asserting that the value of `x.f` is equal to the value v. The assertion `acc(x.f, 1/2) && x.f == 0` for example is satisfied by a state if it holds at least half permission to `x.f`, its partial heap contains `x.f`, and maps the heap location `x.f` to the value `0`. For ease of use Viper allows writing `assert x.f == v` and automatically expands this assertion into the separating conjunction `assert acc(x.f) && x.f == v`.

2.2 Fractional Permissions

Fractional permissions [1] are a generalization of the aforementioned points-to assertion. They allow specifying ownership to heap locations at a more fine-grained level. Instead of either owning or not owning a particular heap location this allows the representation of different permission levels like read and write or no permission. This generalization also applies to the transfer of permission amounts. A caller can transfer a fraction of the the permission it owns to one or multiple callees by splitting the held permission amount.

These permission levels are modeled as fractions between zero and one. Where a zero represents that no permission to the given location is currently

held, a positive value means read permission and a one, also referred to as *full permission*, means that a the heap location is writable. This abstraction allows for the splitting of permission amounts to model concurrent programs that share data on the heap. The sum of all of these fractional permission amounts held on one specific heap location over the whole program must never exceed one.

Verifiers like Viper [6] and VeriFast [4] use fractional permissions to model permission levels to heap locations. We give an example of a Viper program in Listing 2.1 to further illustrate this.

Listing 2.1: Example of permission transfer

```
0  field f: Int
1
2  method check3(x: Ref)
3      requires acc(x.f, 1/2) && x.f == 3
4
5  method main(x: Ref)
6      requires acc(x.f, write)
7      requires x.f == 3
8  {
9      // 1/1 permission to x.f held
10     check3(x)
11     // 1/2 permission to x.f held
12     check3(x)
13     // 0 permission to x.f held
14
15     // following line raises verification error
16     var v := x.f
17 }
```

Pre- and postconditions are provided with the keywords `requires` and `ensures` respectively. These act as interface definitions and instruct the verifier to assume a given condition or assert that a condition holds respectively at the start and the end of the program. Both methods `main` and `check3` are verified modularly where only the interface definitions but not the internals are visible to one another.

Method `main` calls `check3` twice and then tries to read from the heap location `x.f`. The implementation of `check3` is omitted and instead abstractly defined through its pre- and postconditions. It simply asserts that the value stored in heap location `x.f` is equal to three. To be able to read, it also has to hold at least a positive permission amount on location `x.f` but `check3` specifies that it needs exactly `1/2` permission which is even stronger.

In order for `main` to be able to call `check3` it has to have at least `1/2` permission on heap location `x.f` to satisfy the precondition of `check3`. But since `check3` does not ensure this permission amount in its postcondition `main` will not

get it back. Since `main` initially assumes full write permission it is able to call `check3` exactly twice because it transfers half of the permission to each call.

The verifier however still raises an error in line 16 since it holds no permission to `x.f` at this point.

2.3 Inhale and Exhale

Pre- and postconditions can also be further generalized. Instead of only being able to add assumptions and assertions to the start and end of a program block it can be useful to do this at an arbitrary point in the program. The keyword `inhale` is informally speaking used for adding a particular assumption to the set of assumptions and if specified adding a permission amount to the state. `exhale` is its counterpart and is used to subtract permission amounts from the state and to assert certain properties.

This can for example be used to model locks, as the following program implementing a model of a safe copy function in Listing 2.2 shows.

Listing 2.2: Modeling locks with inhale and exhale

```
0  field f: Int
1
2  method safe_copy(src: Ref, dst: Ref)
3  requires acc(src.f, 1/8) {
4      // reading value from source
5      var v = src.f
6
7      // locking destination
8      inhale acc(dst.f, write)
9
10     // critical section
11     dst.f := val
12
13     // unlocking destination
14     exhale acc(dst.f, write)
15 }
```

The method `safe_copy` assumes that `1/8` permission amount on the location of `src.f` is held initially. It is thus able to read from `src.f` in line 5. Inhaling `write` permission to `dst.f` models locking since the invariant must hold that the sum of permission amounts on a single heap location can maximally be `1/1`. Exhaling `write` permission to `dst.f` models the release of the lock. Thus the lines in between are in the critical section which means `dst.f` can safely be written since only one method call of `safe_copy` can hold write permission at once.

2.4 Wildcard Permissions

Method `safe_copy` in Listing 2.2 specifies a concrete lower bound of `1/8` on the amount of permission on `src.f` needed to be able to call it. This restricts the use of this methods to at most eight concurrent method executions which is limiting. To allow the specification of read permissions without the need of nailing down a concrete fraction and thus restricting the maximal number of concurrent executions modeled, Viper provides wildcard permissions amounts.

2.4.1 Concrete Representation

We will call the wildcard currently implemented in Viper a *concrete wildcard* to be able to distinguish it from the *abstract wildcard* examined in this thesis. Like other permission amounts, concrete wildcards too are internally represented as fractions. They represent an unspecified permission amount, known to be greater than zero. This, for example, provides a convenient way to encode read-only resources that can be shared among concurrent programs. This means a Viper program owns a wildcard permission amount to `x.f` if and only if there exists a fraction $w \in \mathbb{Q} \cap (0, 1]$ such that the program has at least permission w to $x.f$.

2.4.2 Encoding

We give the encoding of concrete wildcards in this subsection to show their working and how they interact with other Viper features.

Boogie The Viper back-end *Carbon* translates the Viper source code to its corresponding representation in *Boogie* [5]. This is an intermediate verification language like Viper that provides different abstractions. Boogie does not for example have built-in support for separation logic or fractional permissions. It is designed to provide reusable language features for representing verification requirements and encoding imperative heap-based programs. Which is why it is well suited as one of the two Viper back-end verifiers. Compiled Viper programs are passed to the Boogie back-end which generates verification conditions. These are then checked for satisfiability by an SMT solver, such as z3 [2].

In Carbon permissions held on heap locations are encoded in Boogie maps which are typed key-value stores. The key is a heap location and the value is the corresponding fractional permission amount. The type of the fractional amount is *Perm* which is an alias for the type of real numbers in Boogie. Even though conceptually we work with fractions, real values are better supported by Boogie.

We show the actual encoding by giving the translation of the simple Viper program in Listing 2.3.

Listing 2.3: Inhaling, exhaling and reading with wildcards

```
0  field f: Int
1
2  method main(x: Ref)
3  {
4      inhale acc(x.f, wildcard)
5      exhale acc(x.f, wildcard)
6
7      var v: Int := x.f
8  }
```

This program takes a reference to a heap location x as argument. It then inhales a `wildcard` permission amount to location `x.f` in line 4 and exhales a `wildcard` in the next line. Finally the heap location `x.f` is read. This program passes verification since exhaling a wildcard permission amount means that there exists some permission $w \in \mathbb{Q} \cap (0, 1]$ that is less than the currently held permission amount which is subtracted from it. Thus there exists a permission amount greater than zero after the subtraction of the wildcard which means the state owns read permission.

Listing 2.4 shows how the example program from Listing 2.3 is encoded in Boogie.

Listing 2.4: Encoding of inhaling, exhaling and reading with wildcards

```
0  procedure main(x: Ref) returns ()
1    modifies Heap, Mask;
2  {
3    var wildcard: real where wildcard > NoPerm;
4    var perm: Perm;
5    var v: int;
6
7    // -- Initializing the state
8      Mask := ZeroMask;
9      assume state(Heap, Mask);
10
11   // -- Translating statement: inhale acc(x.f, wildcard) --
12     havoc wildcard;
13     perm := wildcard;
14     assume x ≠ null;
15     Mask[x, f] := Mask[x, f] + perm;
16     assume state(Heap, Mask);
17
18   // -- Translating statement: exhale acc(x.f, wildcard) --
19     perm := NoPerm;
20     havoc wildcard;
21     perm := perm + wildcard;
```

```
22      assert {:msg "Insufficient permission to access x.f"}
23        Mask[x, f] > NoPerm;
24      assume wildcard < Mask[x, f];
25      Mask[x, f] := Mask[x, f] - perm;
26      assume state(Heap, Mask);
27
28    // -- Translating statement: v := x.f --
29
30      // -- Check definedness of x.f
31        assert {:msg "Insufficient permission to access x.f"}
32          HasDirectPerm(Mask, x, f);
33        assume state(Heap, Mask);
34      v := Heap[x, f];
35      assume state(Heap, Mask);
36  }
```

Setting up In line 3 the variable `wildcard` is defined with a positive real type which is conceptually a sub type of the `Perm` type. The variables `perm` and `v` are defined for later use. We simplified the encoding of the state setup, omitting irrelevant details for this explanation. Relevant parts include the definition of an empty `Mask`, which represents the mapping from heap locations to permission amounts. After that the assumption that the initial state is valid, i.e., that the values stored in the `Mask` lie between zero and one, is encoded.

Inhaling The encoding of the `inhale` statement starts at line 11. First the wildcard variable is havoced, i.e., from this point on it can non-deterministically hold any value permitted by its type. This value is then assigned to the previously declared `perm` variable. We also assume the given reference to the heap location to be a valid reference to the heap with the assumption statement in line 14. Then the Mask is increased on location `x.f` by the value of `perm`. The assumption in line 16 takes care of ensuring that the sum of the initial permission amount and the wildcard amount cannot exceed 1.

Exhaling The encoding of the `exhale` statement starts at line 18. First the previously declared `perm` variable is instantiated with the value `NoPerm`, which is an alias for the real value `0`. Then the `wildcard` variable is havoced which means from this point on it can hold any positive real value. The space of possible values can also be limited by assumption statements that exclude certain ranges as we see later. In the next step the temporary variable `perm` is set to be equal to `wildcard` and since `wildcard` can hold any positive real value so too can `perm`. Now to be able to change the state a check is inserted to preserve the invariant that a permission amount must never be smaller than zero. This is done through the assertion in lines 22 and 23. Without this assertion it would be possible to exhale wildcard permission amounts

without holding any permission thus resulting in negative permission. The space of possible fractional permission amounts the `wildcard` variable can have is limited by the assumption in line 24. This value has to be smaller than the currently held permission amount. This serves a similar purpose as the previous assertion since without limiting the range of values a wildcard can take on, it would be possible to exhale a wildcard that in fact is greater than the currently held permission amount. After that the `Mask` storing the permission amount for location `x.f` is decreased by the positive permission amount stored in the variable `perm` which is equal to the value of wildcard.

Reading Reading is encoded by an assertion that checks whether the Boogie function `HasDirectPerm` shown in Listing 2.5 returns true for a given `Mask`. We see in the definition that this function returns true if and only if the mask stores a positive value for a given location. This thus represents read permission.

Listing 2.5: Simplified read permission encoding

```
0
1 function HasDirectPerm(Mask: MaskType, o: Ref, f: Field):
    bool;
2 axiom (∀ Mask: MaskType, o: Ref, f: Field •
3     HasDirectPerm(Mask, o, f)
4     ⟺
5     Mask[o, f] > NoPerm
6 );
```

Chapter 3

Abstract Wildcards

Wildcards are not unique to Viper. The VeriFast verifier [4] uses a similar mechanism for representing unspecified permission amounts but implements a different internal representation. The main goal of this project is to explore the advantages and disadvantages of a such an alternative wildcard representation for Viper, where we place special emphasis on performance. As mentioned in the previous chapter, we refer to this representation as *abstract wildcards*. We aim for an overapproximation of the concrete wildcard behavior explained in Chapter 2. The encoding of abstract wildcards should be sound with respect to the current encoding of concrete wildcards, a simplified proof of which can be found in the next Chapter 4. In this chapter we explain our motivation for looking into abstract wildcards and show how they differ semantically and implementation-wise from concrete wildcards.

3.1 Motivation

The concrete wildcard representation stores information, namely concrete permission amounts for wildcards, which is not necessary to perform most of their use-cases and thus has potential to be optimized. One such optimization is used in the Verifast verifier, which represents wildcard permission amounts abstractly instead of concretely, by setting a Boolean value if there is a wildcard permission amount held. Instead of fixing a concrete fraction for wildcard permission amount non-deterministically through existential quantification, a Boolean value acts as an abstraction for some positive permission amount in addition to the already held amount. This provides the inspiration for the examined encoding of abstract wildcards. At an encoding level, such a representation seems more convenient, as the following example shows.

It is possible to hold a fraction of a predicate in Viper, for example to model several threads concurrently reading the same data structure. When a frac-

tion of a predicate is unfolded, all permissions it contains are multiplied by the corresponding fraction. Listing 3.1 shows how unfolding a wildcard amount of the `read` predicate results in a multiplication of two wildcard permissions. Both of these wildcard values hold a concrete fractional value non-deterministically. SMT solvers are notoriously bad at non-linear arithmetic and thus Carbon implements simplification rules for cases like the multiplication of wildcard. Carbon simplifies this multiplication to a wildcard amount. Such an *ad hoc* optimization is not be necessary with abstract wildcards, where the multiplication of a wildcard with any non-zero permission results, by definition, in an abstract wildcard.

Listing 3.1: Example of permission multiplication for predicates

```
0  field f: Int
1
2  predicate read(x: Ref)
3  {
4      acc(x.f, wildcard)
5  }
6
7  method main(x: Ref)
8      requires acc(read(x), wildcard)
9  {
10     // This unfold multiplies two wildcard permissions,
11     // resulting in another wildcard permission to x.f,
12     // which allows reading the value of x.f
13     unfold acc(read(x), wildcard)
14     var x: Int := x.f
15 }
```

3.2 Formal Definitions

To more formally define what an abstract wildcard is we define some notations. Let $P = [0, 1] \cap \mathbb{Q}$ be the set of permission amounts that can be held on a heap location.[1]

To encode abstract wildcards, we introduce a tuple $(p, b) \in (P \times \{\top, \bot\}) \setminus \{(1, \top)\}$, where the first element represents the fractional permission amount held on a location and the second one represents a wildcard permission amount. The tuple $(1, T)$ is excluded from the domain since it would represent a permission amount that is greater than one. For a better understanding of this encoding, we give a few examples of represented permission amounts along with their encoding in the Table 3.1.

[1] This only applies to heap locations. For permission amounts to predicates there is no upper bound on the permission amount.

Abstract wildcard representation	Represented amount
$(0, \bot)$	0
$(0, \top)$	> 0
$(1/2, \top)$	$> 1/2$
$(1/1, \top)$	$> 1/1$ impossible

Table 3.1: Examples of abstract wildcard representation

The execution state of a Viper program thus stores the permission amount on a heap location by representing it with such a tuple $(p, b) \in (P \times \{\top, \bot\}) \setminus \{(1, \top)\}$ for each location.

Reading With this encoding, a state has permission to read a heap location if and only if the conjunct $p > 0 \bigvee b = \top$ holds in the state for this particular heap location.

Inhaling When inhaling an abstract wildcard permission amount there are two cases to consider. If the state holds $(1, \bot)$ permission on the heap location in question, inhaling a positive permission amount results in owning a permission amount greater than one and thus lead to an inconsistent state, where any statement would verify. This however is the intended behavior and thus for any permission amount the state is transformed by setting the second element of the tuple to $\top$ regardless of the current value of said element:

$(p, b) \rightarrow (p, \top)$

Exhaling Exhaling can lead to verification errors since if there is no permission held on the heap location, exhaling a positive permission amount like a wildcard would lead to negative permission. If this is not the case the state is transformed by setting the second element of the tuple to $\top$ and the first element to zero regardless of the current value of said element:

$(p, b) \rightarrow (0, \top)$

3.3 Encoding

There are several ways one could encode abstract wildcards in Boogie. We explored two of these approaches. The first was to encode the structure of a tuple as axiomatized Boogie functions. This allows to change the type of permission values in the encoding. The second approach represents the tuple as two distinct Boogie masks. This section will explain these two approaches in more detail.

3.3.1 Representing Tuples in Boogie

Our first idea was to redefine the type of permission values in the Boogie encoding of Viper programs. The `Perm` type is currently used as an alias for real numbers and there is no tuple type in Boogie. So we added a tuple type together with axiomatized functions defining the operations we needed to interact with them such as `fst(t)` and `snd(t)` that return the first and second element of the given tuple `t` respectively. This encoding can be seen in Listing 3.2.

Listing 3.2: Encoding tuples in Boogie

```
0  type Tuple A B;
1
2  function tuple<A, B>(a: A, b: B): Tuple A B;
3  function fst<A, B>(p: (Tuple A B)): A;
4  function snd<A, B>(p: (Tuple A B)): B;
5
6  axiom (∀ <A, B> a: A, b: B •
7    (fst((tuple(a, b): Tuple A B)): A) = a
8  );
9  axiom (∀ <A, B> a: A, b: B •
10   (snd((tuple(a, b): Tuple A B)): B) = b
11 );
12 axiom (∀ <A, B> p: (Tuple A B) •
13   (tuple((fst(p): A), (snd(p): B)): Tuple A B) = p
14 );
15
16 type Perm = Tuple real bool;
```

The `fst` function is defined through the axiom in line 6 which states that if we compose the tuple creation function `tuple` with the `fst` function, the result is the second input to the tuple function. `snd` is defined analogously. We then defined the type `Perm` to be such a tuple that holds a real and a Boolean value. This definition can be seen in line 16 in Listing 3.2.

For summing up permission values represented as these Boogie tuples we introduced the function `sumPerm` that is axiomatized to only return true if and only if the result parameter is equal to the tuple that holds the sum of the two first elements of the summands in its first element and the disjunction of the second elements of the two summands in the second. The encoding can be seen in Listing 3.3. This can be used for inhaling permission amounts with the abstract representation and a similar function would have to be encoded for subtracting these permission tuples from one another for exhaling.

Listing 3.3: Encoding of sum function for tuples

```
function sumPerm(result: Perm, summand1: Perm, summand2: Perm
    ): bool;
axiom (∀ result: Perm, summand1: Perm, summand2: Perm •
    sumPerm(result, summand1, summand2)
    ⟺
    result = tuple(
        fst(summand1) + fst(summand2),
        snd(summand1) ∨ snd(summand2)
    )
);
```

Because this change affects every feature that interacted with the permission system and initial performance tests indicated a worsening, we abandoned this strategy and instead focused on the idea of using a second Boolean mask for representing tuples.

3.3.2 Adding a Boolean Mask

Since the concrete wildcard encoding uses masks to store permission amounts to heap locations it seems reasonable to add a second mask that represents our abstract wildcard permission amount. This also has the benefit that the encoding of concrete wildcards is more decoupled from from the encoding of abstract wildcards.

We call the additional mask `BMask` which stands for Boolean mask and is defined as a mapping from heap locations to Boolean values.

Listing 3.4: Definition of BMask

```
type BMaskType = [Ref, Field] bool;
```

With this definition in place we are able to define what it means for the mask pair to be valid as the following listing 3.5 shows. This code snippet defines an axiomatized Boogie function that takes a `Mask` and a `BMask` as input and outputs true if it is a valid mask pair and false otherwise. A mask pair is valid if the value of `Mask` is greater or equal to `noPerm`, i.e., greater or equal to zero, and if it is no predicate field and no wand field the permission amount is not greater than one. For this we insert the implication `Mask[o, f] = FullPerm` $\Longrightarrow$ `¬BMask[o, f]` to disallow the combination of a `Mask` value of one with a `BMask` holding the value true, representing a permission amount greater than one.

Listing 3.5: Definition of valid masks

```
function GoodMask(Mask: MaskType, BMask: BMaskType): bool;

axiom (∀ Mask: MaskType, BMask: BMaskType, o: Ref, f: Field
    •
    GoodMask(Mask, BMask)
        ⟹
    Mask[o, f] ≥ NoPerm ∧ (
        (GoodMask(Mask, BMask) ∧ ¬IsPredicateField(f)) ∧
        ¬IsWandField(f)
            ⟹
        Mask[o, f] ≤ FullPerm ∧ (
            Mask[o, f] = FullPerm
                ⟹
            ¬BMask[o, f]
        )
    )
);
```

Now we can define what it means to be able to read a location in memory. This is the case if the following disjunction holds. Either `BMask` on this location is set to true or the value of `Mask` on that location is greater than `NoPerm`. Listing 3.6 shows the Boogie encoding of this notion. The changes to the encoding are highlighted.

Listing 3.6: Definition of read permission

```
function HasDirectPerm (Mask: MaskType, BMask: BMaskType,
    o: Ref, f: Field): bool;

axiom (∀ Mask: MaskType, BMask: BMaskType, o: Ref, f: Field •
  HasDirectPerm(Mask,BMask,o,f)
      ⟺
  BMask[o,f] ∨ Mask[o,f] > NoPerm
);
```

Not only do the definitions of these functions and axioms change but also the encoding of features like reading or inhaling and exhaling of permission amounts, predicates and quantified permissions. Inhaling and exhaling predicates is very similar to the corresponding operations on heap locations and so we omit their exact encoding here. The encoding of abstract wildcards used with quantified permissions is shown in the next section. We explain the encoding of inhaling and exhaling abstract wildcard permissions to heap locations by giving the translation of the simple Viper program in Listing 3.7.

Listing 3.7: Inhaling, exhaling and reading heap locations with abstract wildcards

```
0 field f: Int
1
2 method main(x: Ref)
3 {
4     inhale acc(x.f, wildcard)
5     exhale acc(x.f, wildcard)
6
7     var v: Int := x.f
8 }
```

This program takes a reference to a heap location `x` as argument. It then inhales an abstractly encoded wildcard permission amount to location `x.f` in line 4 and exhales such an abstract wildcard in the next line. Finally the heap location `x.f` is read. This program verifies in Viper since exhaling a wildcard permission amount will always leave some fractional permission amount owned by the state and so this read operation is permitted as explained in the background Chapter 2. Listing 3.8 shows how this example program is encoded into Boogie.

Listing 3.8: Boogie encoding of inhaling, exhaling and reading heap locations with abstract wildcards

```
0  procedure main(x: Ref) returns ()
1    modifies Heap, Mask, BMask;
2  {
3    var v: int;
4
5    // -- Initializing the state
6      Mask := ZeroMask;
7      BMask := ZeroBMask;
8      assume state(Heap, Mask, BMask);
9
10   // -- Translating statement: inhale acc(x.f, wildcard) --
11     assume x ≠ null;
12     BMask[x, f] := true;
13     assume state(Heap, Mask, BMask);
14
15   // -- Translating statement: exhale acc(x.f, wildcard) --
16     assert {:msg "Insufficient␣permission␣to␣access␣x.f"}
17         BMask[x, f] ∨ Mask[x, f] > NoPerm;
18     BMask[x, f] := 0.000000000;
19     BMask[x, f] := true;
20     assume state(Heap, Mask, BMask);
21
22   // -- Translating statement: v := x.f --
23
24     // -- Check definedness of x.f
25     assert {:msg "Insufficient␣permission␣to␣access␣x.f"}
26         HasDirectPerm(Mask, BMask, x, f);
```

```
27      assume state(Heap, Mask, BMask);
28
29      v := Heap[x, f];
30      assume state(Heap, Mask, BMask);
31  }
```

Setting up In Line 3 we declare the variable v that will later hold the value of the heap location that will be read. Lines 5 to 8 set up the two masks representing our tuple. Both the `Mask` and the `BMask` are initialized to map every heap location to the value `0` and `false` respectively. Then the encoding assumes the initial state to be valid, i.e., that the combination values in the two Masks for each heap location represent permission amounts between zero and one.

Inhaling The encoding of the `inhale` statement starts on line 10. Instead of havocing a variable so it represents any value the wildcard could take on, as the concrete encoding does, here we simply assume that the reference x to the heap location is non null and set the `BMask` on the location `x.f` to true.

Exhaling From line 15 on we can see how the `exhale` statement is encoded. First there is an assertion that checks that sufficient permission is available to exhale a wildcard permission amount, i.e, that we hold a permission amount greater than zero. This amounts to checking whether the state holds a wildcard permission amount or some positive fractional permission amount in `Mask`. The exhale itself does not subtract a havoced permission amount value from the currently held amount as in the concrete encoding. Instead the concrete part of the permission, i.e. the value of `Mask` on that heap location is set to `0` and the `BMask` is set to `true`, thus this state represents some positive permission amount.

Reading The encoding for a read operation can be seen from line 24 onward. This essentially has the same structure as the encoding of a read operation in the concrete encoding with the difference that here the functions `HasDirectPerm` and `state` have been replaced with the aforementioned functions that include the `BMask` as parameters.

3.4 Quantified Permissions Encoding

In this section we describe the encoding of quantified permissions using abstract wildcards. First we have a look at the encoding of quantified permissions on the example program given in Listing 3.9.

Listing 3.9: Quantified permissions example

```
0  field f: Int
1
2  method main(s: Set[Ref])
3  {
4      inhale ∀ r:Ref • {r in s} r in s ⟹ acc(r.f, wildcard)
5
6      var x: Ref
7      var y: Ref
8
9      assume x in s
10     assume y in s
11     assume x ≠ y
12
13     assert perm(x.f) = perm(y.f)
14 }
```

Method `main` takes a set of references `s` as an argument and inhales for every reference `r` in this set a wildcard permission amount on `r`. Then two variables `x` and `y` of reference type are declared. We assume for both of them that they are included in the set `s` and that they are not equal to each other.

In line 13 we assert using permission introspection that the permission held on on the locations `x.f` and `y.f` is the same. This should not be the case since a wildcard represents not a single fractional value but non-deterministically takes on some positive fraction permitted by the assumptions. Thus there exists some fractions that are not equal to each other which means the assertion is violated.

Listing 3.10 shows a simplified version of the encoding of the inhale statement in program from Listing 3.9.

Listing 3.10: Simplified encoding of inhaling quantified permissions

```
0  const unique f: Field;
1
2  ...
3
4  function qpRange(recv:Ref): bool;
5  function invRecv(recv: Ref): Ref;
6
7  ...
8
9  var x: Ref;
10 var y: Ref;
11 var QPMask: MaskType;
12 var QPBMask: BMaskType;
13
14 ...
15
```

```
16 Mask := ZeroMask;
17 BMask := ZeroBMask;
18 assume state(Heap, Mask, BMask);
19
20 havoc QPMask;
21 havoc QPBMask;
22
23 // Define Inverse Function
24 assume (∀ r: Ref •
25     s[r] ⟹ qpRange(r) ∧ invRecv(r) = r
26 );
27 assume (∀ o: Ref •
28     s[invRecv(o)] ∧ qpRange(o) ⟹ invRecv(o) = o
29 );
30
31 // Assume set of fields is nonNull
32 assume (∀ r: Ref •
33     s[r] ⟹ r ≠ null
34 );
35
36 // Define permissions
37 assume (∀ o: Ref •
38     (s[invRecv(o)] ∧ qpRange(o)
39         ⟹
40         invRecv(o) = o ∧
41         QPMask[o, f] = Mask[o, f] ∧
42         QPBMask[o, f]
43     ) ∧
44     (¬(s[invRecv(o)] ∧ qpRange(o))
45         ⟹ QPMask[o, f] = Mask[o, f] ∧
46     QPBMask[o, f] = BMask[o, f])
47 );
48
49 assume (∀ o: Ref, f_1: Field •
50     f_1 ≠ f
51         ⟹
52     Mask[o, f_1] = QPMask[o, f_1] ∧
53     BMask[o, f_1] = QPBMask[o, f_1]
54 );
55 Mask := QPMask;
56 BMask := QPBMask;
57 assume state(Heap, Mask, BMask);
```

Setup Quantified permissions are only applicable if the applied function is injective since the inverse of these functions is needed to get the locations that are quantified over for defining the permissions to these locations. Thus line 4 declares a function returning true if the value received lies in the codomain of the quantified function. Line 5 declares the inverse function

of the quantified function. These are later instantiated. Lines 9 through 12 declare variables for later use and lines 16 to 18 set up the state as for the encoding of inhaling wildcard permission amounts on heap locations. In lines 20 and 21 the two variables `QPMask` and `QPBMask` are havoced, so they non-deterministically hold some value permitted by their type.

Inverse functions The previously declared functions `qpRange` and `invRec` get instantiated in lines 24 to 29. Lines 32 to 34 then additionally assume that every element in the domain of the quantifed function, in this case `s[r]` is not null. Similar to inhaling permission amounts to heap locations directly.

Defining the permissions Lines 36 to 47 then define how much permission is inhaled and store the corresponding resulting amount in the temporary masks `QPMask` and `QPBMask`. More concretely, in this case where we inhale a wildcard permission amount, for every reference `o`, if the origin of that value lies within the range of the quantified function `s`, the temporary mask `QPBMask[o,f]` is set to equal true and the value of the mask should not change and thus `QPMask[o,f]` should equal the value of `Mask[o,f]` before the inhale. Otherwise we simply copy the previous wildcard state stored in the two masks to the temporary masks, thus not changing the permission amount on these locations. Lines 49 to 54 then make sure that for any field that is different to the one we refer to in the quantified function the permission amount stays the same.

Finishing To finish inhaling we set the values of `Mask` and `BMask` to their corresponding temporary masks and then assume that the state is valid as in the encoding for inhaling permissions on heap locations.

Listing 3.11 shows a simplified version of the encoding of the exhale statement in program from Listing 3.9 where we omitted the setup and finishing part since it is the same one as in for inhaling.

Listing 3.11: Simplified encoding of exhaling quantified permissions

```
0  // check if sufficient permission is held
1    assert {:msg "Insufficient␣permission␣to␣access␣r.f"}
2      (∀ r: Ref •
3      s[r] ⟹ BMask[r, f] ∨ Mask[r, f] > NoPerm
4    );
5
6  // check if receiver r is injective
7    assert {:msg "Receiver␣of␣r.f␣might␣not␣be␣injective."}
8      (∀ r: Ref, r_1: Ref •
9      (r ≠ r_1 ∧ s[r]) ∧ s[r_1] ⟹ r ≠ r_1
10   );
11
12 // assumptions for inverse of receiver r
```

```
13    assume (∀ r: Ref •
14      s[r] ⟹ qpRange(r) ∧ invRecv(r) = r
15    );
16    assume (∀ o: Ref •
17      s[invRecv(o)] ∧ qpRange(o) ⟹ invRecv(o) = o
18    );
19
20  // assume permission updates for field f
21    assume (∀ o: Ref •
22      (s[invRecv(o)] ∧ qpRange(o)
23          ⟹ invRecv(o) = o ∧
24          QPMask[o,f] = 0 ∧ BMask[o,f]
25      ) ∧
26      (¬(s[invRecv(o)] ∧ qpRange(o))
27          ⟹ QPMask[o,f] = Mask[o,f] ∧
28          QPBMask[o,f] = BMask[o,f])
29    );
```

Check enough permission When exhaling permissions, the available amount on each location must at least be positive. This is what lines 0 through 4 assert. It will throw a verification error if at least one location used in the quantified function does not hold a positive permission amount.

Inverse functions Lines 12 to 18 do the same thing as in the inhale case. But when exhaling we additionally throw a verification error if the quantified function is not injective. This is encoded in lines 6 to 10.

Defining the permissions The permission updating also works the same way as in the inhale case, with the difference that instead of adding a wildcard permission, we define the temporary mask `QPMask` to equal 0 and `QPBMask` to be true.

3.5 Permission Introspection

Permission introspection is a means for reading the permission amount of a heap location held by the state at a given program location. The abstract wildcard encoding we implemented is incompatible with this permission lookup as we will explain in this section. Nevertheless we want our new implementation to be compatible with the full set of Viper statements and thus we briefly explain our approach for circumventing errors due to this incompatibility.

3.5.1 Incompatibility

To show why abstract wildcard permission are incompatible with permission introspection we need to understand how Carbon encodes it in the current implementation. The following program in Listing 3.13 shows a program that inhales a wildcard permission amount on location `x.f` in its precondition and then asserts, using permission introspection, whether the permission held on this location is positive.

Listing 3.12: Permission introspection example

```
0 method main(x: Ref)
1 requires acc(x.f, wildcard)
2 {
3     assert perm(x.f) > none
4 }
```

The encoding of permission introspection is very simple. Essentially it uses `Mask` to read the permission amount stored on the heap location `x.f` for the assertion.

Listing 3.13: Permission introspection encoding

```
0 // translating statement: assert perm(x.f) > none
1 assert {:msg "Assertion perm(x.f) > none might not hold."}
2   NoPerm < Mask[x, f];
```

We could implement a similar interface for abstract wildcards, where the programmer could specify the tuple representing the abstract permission amount. Then the encoding could be changed to read both masks and compare the two tuples instead of fractions. However, this would change the programming interface, which is not what we want to do in this project. But even this change would not solve the incompatibility of abstract wildcards with permission introspection.

The issue lies in the overapproximation of abstract wildcards. We can come up with examples as the following in Listing 3.14 that show how abstract wildcards when used in conjunction with permission introspection are not sound with respect to concrete wildcards.

Listing 3.14: Example of incompatibility of abstract wildcards and permission introspection

```
0 method main(x: Ref, y: Ref)
1 requires acc(x.f, 1/2)
2 {
3     exhale acc(x.f, wildcard)
4
5     assert perm(x.f) < 1/2
6 }
```

This program inhales half permission to heap location `x.f` in its precondition and then exhales a wildcard permission amount. Using concrete wildcards the assertion in line 5 passes since we can be sure that the owned permission amount is less than half. Using abstract wildcards on the other hand leads to a verification error since after exhaling a wildcard we get the abstract permission amount $(0, \top)$ which means the state could hold a permission amount greater than $1/2$ for all we know.

3.5.2 Possible Implementation Options

There are several ways one could implement a compatibility layer between abstract and concrete wildcard permissions.

Implicit Conversion One approach would be to implicitly convert any abstract wildcard to a concrete wildcard before looking up the permission value in the mask. It would allow for very fine grained intervention in incompatible cases but it has the drawback that since abstract wildcard permissions overapproximate concrete wildcards, there are cases where the encoded permission amount would not be losslessly convertible to a concrete permission amount. The previously shown example in Listing 3.14 also works as a counter example in this case. This approach is thus not a complete solution and would still rely on some other technique to ensure full compatibility of abstract wildcards and permission introspection.

Syntactic Analysis Another solution is to use static syntactic analysis to find every field and predicate which is used with a `perm` and with a `wildcard` keyword. For every such field or predicate we use the concrete wildcard, which is compatible with permission introspection. For all others it is safe to use abstract wildcards. This approach is much more coarse grained and is likely to overapproximate the cases where such a conversion actually is necessary. The reason for using fields for the syntactic analysis and not heap locations like `x.f` is that static syntactic analysis cannot handle variable aliases which means a naive approach like this will not consider aliases of heap locations that are used in an incompatible way.

We chose to implement the syntactic analysis approach for this project since it is easier to implement than a more fully fledged solution. For this we modified the Viper parser by adding the functionality for checking and choosing the appropriate encoding strategy per field and predicate.

3.6 Optimizing Concrete Wildcards

While implementing the abstract wildcard we came across some errors and inefficiencies of the concrete wildcard implementation. There was an error in

the encoding of concrete wildcards, allowing the program in Listing 3.9 to be verified without errors or warnings. We fixed this error not just for the encoding of abstract wildcards, but also for concrete wildcards.

Furthermore, there were a lot of cases where the encoding for inhaling or exhaling wildcard permission amounts would check whether a wildcard permission amount is greater than zero, which is true by its definition and is thus unnecessary. We removed all of these checks for both encodings.

The Carbon back-end implementation also inserted assertions for ensuring correct behavior of wildcards which overlapped with other checks for whether sufficient permission is held. We simplified all of these cases, eliminating duplication of these checks.

Chapter 4

Simplified Proof of Soundness

We give a proof sketch to show why abstract wildcards correctly overapproximate the concrete wildcard encoding. For this we consider a simplified Viper semantics that only includes one heap location that we call $x.f$, where x is a reference and f is a field. We also simplify the set of Viper statements V to not include the `perm` keyword that is used for permission introspection since it is incompatible with the abstract wildcard encoding as mentioned in the previous Chapter 3. We believe this captures the core ideas of the proof and it should be straightforward to adapt it to the general setting.

We use induction over Viper statements $s \in V$ for this proof. The notations we use are given in the next section and the after that we introduce some helping lemmas.

4.1 Notations

Permission amounts Let $P = [0, 1] \cap \mathbf{Q}$ be the set of permission amounts that can be held by the state on the heap location $x.f$.

Viper execution states Let $S_c \in \mathbb{P}(P)$ and $S_a \in \mathbb{P}((P \times \{\top, \bot\}) \setminus \{(1, True)\})$ be the set of execution states of a Viper program using the concrete and abstract encoding respectively.[1] For the tuple $p \in S_a$ we refer to the first element of p as $p[0]$ and to the second as $p[1]$.

Semantics We define the function $sem_x : \mathbb{P}(S_x) \times V \mapsto \mathbb{P}(S_x) | failure$ where S_x is the set of execution states using encoding $x \in \{a, c\}$ which stands for abstract and concrete respectively. V is the set of Viper statements and $failure$ is the designated state for failed verification.

[1] Even though in our simplified state model we do not have non-determinism, we still need to account for it for the general case (where heap locations have values and thus the value of x.f is sometimes havoced), hence S_a is a set of states.

$sem_x(\Phi_x, s) = failure$ if the execution of statement s in some state $\varphi_x \in \Phi_x \subseteq S_x$ leads to a verification error. Otherwise, $sem_x(\Phi_x, s)$ is the set of states that we reach after the execution of statement s. For $sem_c(\Phi_c, s)$ we can write the output set of states as ranges in P. We partially define sem_x as follows.

The semantics of sequential composition of Viper statements $s_1, s_2 \in V$ is defined as follows.

$$sem_x(\Phi_x, s_1; s_2) = \begin{cases} failure & \text{if } failure = sem_x(\Phi_x, s_1) \\ sem_x(sem_x(\Phi_x, s_1), s_2) & \text{otherwise} \end{cases}$$

The semantics of `inhale acc(x.f, wildcard)` is defined as follows.

$$sem_a(\{\varphi_a\}, \texttt{inhale acc(x.f, wildcard)}) = \begin{cases} \{\} & \text{if } \varphi_a = (1, \top) \\ \{(\varphi_a[0], \top)\} & \text{otherwise} \end{cases}$$

$$\begin{aligned} &sem_c(\{\varphi_c\}, \texttt{inhale acc(x.f, wildcard)}) \\ &= \{p'.p' = \varphi_c + p \land p' \leq 1 \land 0 < p \land p \in P\} \\ &= (\varphi_c, 1] \end{aligned}$$

The semantics of `exhale acc(x.f, wildcard)` is defined as follows.

$$sem_a(\{\varphi_a\}, \texttt{exhale acc(x.f, wildcard)}) = \begin{cases} failure & \text{if } \varphi_a = (0, \bot) \\ \{(0, \top)\} & \text{otherwise} \end{cases}$$

$$\begin{aligned} &\text{sem}_c(\{\varphi_c\}, \texttt{exhale acc(x.f, wildcard)}) \\ &= \begin{cases} failure & \text{if } \varphi_c = 0 \\ \{p'.p' = \varphi_c - p \land 0 < p' \land 0 < p \land p \in P\} & \text{otherwise} \end{cases} \\ &= \begin{cases} failure & \text{if } \varphi_c = 0 \\ (0, \varphi_c) & \text{otherwise} \end{cases} \end{aligned}$$

We define $sem_x(\Phi_x, s)$ for sets of states $\Phi_x \subseteq S_x$ as follows.

$$sem_x(\Phi_x, s) = \begin{cases} failure & \text{if } \exists \varphi_x \in \Phi_x . sem_x(\{\varphi_x\}, s) = failure \\ \bigcup_{\varphi_x \in \Phi_x} sem_x(\{\varphi_x\}, s) & \text{otherwise} \end{cases}$$

Permission to read `x.f` To capture a state owning read permission we define the function $canRead_x : S_x \mapsto \{\top, \bot\}$ for $x \in \{a, c\}$.

$$canRead_a(\{\varphi_a\}) \Leftrightarrow 0 < \varphi_a[0] \bigvee \varphi_a[1] \tag{4.1}$$

$$\text{canRead}_c(\{\varphi_c\}) \Leftrightarrow 0 < \varphi_c \tag{4.2}$$

$$\text{canRead}_x(\Phi_x) = \bigwedge_{\varphi_x \in \Phi_x} canRead_x(\{\varphi_x\}) \tag{4.3}$$

Coupling invariant Let the coupling invariant $R(\Phi) : S_a \mapsto S_c$ we use for the induction proof be defined as follows. This function essentially converts a set of states in the abstract wildcard encoding to a set of states in the concrete wildcard encoding. Since the output is a set of concrete states we can also describe this set as a range.

$$\begin{aligned} R(\{\varphi_a\}) &= \begin{cases} \{\varphi_a[0]\} & \text{if } \varphi[1] = \bot \\ \{p'.p' = \varphi_a[0] + p \wedge p' \leq 1 \wedge 0 < p \wedge p \in P\} & \text{if } \varphi[1] = \top \end{cases} \\ &= \begin{cases} \{\varphi_a[0]\} & \text{if } \varphi[1] = \bot \\ (\varphi_a[0], 1] & \text{if } \varphi[1] = \top \end{cases} \end{aligned}$$

$$R(\Phi_a) = \bigcup_{\varphi_a \in \Phi_a} R(\{\varphi_a\}) \tag{4.4}$$

4.2 Preparation

We prove that any Viper statement $s \in V$ that is encoded with our abstract wildcard encoding and verifies also verifies statement s in any state over-approximated by the abstract wildcard encoding in the concrete wildcard encoding. So we prove the following statement where $sem_a(\Phi_x, s) = \Phi'_x$ means that statement s in any state in Φ_x will not fail since $failure \not\subseteq S_a$.

$$\forall s \in V. \forall \Phi_a, \Phi'_a \subseteq S_a. sem_a(\Phi_a, s) = \Phi'_a \Rightarrow sem_c(R(\Phi_a), s) \subseteq R(\Phi'_a) \tag{4.5}$$

As mentioned above, we prove the statement by induction over Viper statements $s \in V$. For this we use the following induction hypothesis.

$$\begin{aligned} IH(s) :=& \forall \Phi_a, \Phi'_a \in S_a. sem_a(\Phi_a, s) = \Phi'_a \\ & \Rightarrow sem_c(R(\Phi_a), s) \subseteq R(\Phi'_a) \end{aligned} \tag{4.6}$$

To simplify the induction proof we prove the following lemma that allows us to prove the induction cases for abstract states and then generalize it to sets of abstract states.

Lemma 4.1 *If we can show that for any statement $s \in V$ and states $\varphi_a, \varphi'_a \in S_a$ the abstract wildcard encoding is sound with respect to the concrete encoding then it is also sound for any set of states $\Phi_a, \Phi'_a \subseteq S_a$, i.e.,*

$$\begin{aligned}&\forall s \in V.\\&(\forall \varphi'_a \in S_a.\Phi''_a \subseteq S_a.sem_a(\{\varphi_a\}, s) = \Phi''_a \Rightarrow sem_c(R(\{\varphi_a\}), s) \subseteq R(\Phi''_a))\\&\Rightarrow \forall \Phi_a, \Phi'_a \subseteq S_a.sem_a(\Phi_a, s) = \Phi'_a \Rightarrow sem_c(R(\Phi_a), s) \subseteq R(\Phi'_a)\end{aligned}$$

holds.

Proof Let $s \in V$ be an aribtrary Viper statement. We assume the left hand side of the implication, i.e., $\forall \varphi'_a \in S_a.\forall \Phi''_a \subseteq S_a.sem_a(\{\varphi_a\}, s) = \Phi''_a \Rightarrow sem_c(R(\{\varphi_a\}), s) \subseteq R(\Phi''_a)$. Then we assume that $sem_a(\Phi_a, s) = \Phi'_a$ holds. From this we can deduce the following.

$$\begin{aligned}&(\forall \varphi'_a \in S_a.\forall \Phi''_a \subseteq S_a.sem_a(\{\varphi_a\}, s) = \Phi''_a\\&\Rightarrow sem_c(R(\{\varphi_a\}), s) \subseteq R(sem_a(\{\varphi_a\}, s))) && (4.7)\\&\Rightarrow \bigcup_{\varphi_a \in \Phi_a} sem_c(R(\{\varphi_a\}), s) \subseteq \bigcup_{\varphi_a \in \Phi_a} R(sem_a(\{\varphi_a\}, s)) && (4.8)\\&\Leftrightarrow sem_c(\bigcup_{\varphi_a \in \Phi_a} R(\{\varphi_a\}), s) \subseteq R(\bigcup_{\varphi_a \in \Phi_a} sem_a(\{\varphi_a\}, s)) && (4.9)\\&\Leftrightarrow sem_c(R(\bigcup_{\varphi_a \in \Phi_a} \{\varphi_a\}), s) \subseteq R(sem_a(\bigcup_{\varphi_a \in \Phi_a} \{\varphi_a\}, s)) && (4.10)\\&\Leftrightarrow sem_c(R(\Phi_a), s) \subseteq R(sem_a(\Phi_a, s)) && (4.11)\\&\Leftrightarrow sem_c(R(\Phi_a), s) \subseteq R(\Phi'_a) && (4.12)\end{aligned}$$

For Implication 4.7 we expanded the definition of Φ''_a on the right hand side. Implication 4.8 follows from set theory. From the definition of sem_x and R the equivalences 4.9 and 4.10 follow. Step 4.11 rewrites Φ_a and Step 4.12 follows from the assumption $sem_a(\Phi_a, s) = \Phi'_a$.

This proves the lemma. □

We give the prove for the soundness of reading permissions here as a lemma since this operation does not fit the induction proof structure properly.

Lemma 4.2 *Reading permissions in the abstract encoding are sound with respect to the concrete encoding, i.e.,*

$$\forall \Phi_a \subseteq S_a.canRead_a(\Phi_a) \Rightarrow canRead_c(R(\Phi_a))$$

Proof For the correct verification of s using the concrete wildcard encoding, every concrete state represented by the abstract encoding must hold a positive

value. We first show that the right hand side holds for any singleton state $\Phi_a = \{\varphi_a\}$ by considering two cases. We assume the left hand side of the implication. Thus we know that $\varphi_a[0] > 0 \bigvee \varphi_a[1] = \top$ holds. To prove that the right hand side follows we use Equation 4.1.

1. Case: $\varphi_a[1] = \bot$
   $\varphi_a[0] > 0$ must hold since otherwise $canRead_a(\{\varphi_a\})$ would not hold. With the definition of R we get $canRead_c(R(\varphi_a)) = canRead_c(\{\varphi_a[0]\})$. Expanding the definiton of $canRead_c$ we further get $canRead_c(\{\varphi_a[0]\}) = \varphi_a[0] > 0 = \top$ and thus the right hand side holds which concludes this case.

2. Case: $\varphi_a[1] = \top$
   Any $\varphi_c \in R(\varphi_a) = (\varphi_a[0], 1]$ must be positive since $0 \leq \varphi_a[0]$ holds. This means $canRead_c(\varphi_c) = \top$ holds for any such φ_a and thus $canRead_c(R(\varphi_a)) = \top$ holds by using the definition of R.
   This concludes the second case. □

Let $\Phi'_a \subseteq S_a$ be some arbitrary set of abstract states using the abstract wildcard encoding. We assume the left hand side of the implication and with Equation 4.1 we get $canRead_a(\Phi'_a) = \bigwedge_{\varphi'_a \in \Phi'_a} canRead_a(\{\varphi'_a\})$. We already proved that for any state $\varphi_a \in S_a$ the implication holds and thus we know $\bigwedge_{\varphi'_a \in \Phi'_a} canRead_a(\{\varphi'_a\}) \Rightarrow \bigwedge_{\varphi'_a \in \Phi'_a} canRead_c(R(\{\varphi'_a\}))$ holds. The right hand side of this implication is equivalent to $\bigwedge_{\varphi'_c \in R(\Phi'_a)} canRead_c(\{\varphi'_c\})$ which can then be transformed into $canRead_c(R(\Phi'_a))$ using Equation 4.1 which proves the lemma.

4.3 Proof

We show $IH($`s1 ; s2`$)$, $IH($`inhale acc(x.f, wildcard)`$)$
and $IH($`exhale acc(x.f, wildcard)`$)$ only.

1. Case $s = s_1; s_2$
   We fix $s = s_1; s_2$ for some arbitrary statements $s_1, s_2 \in V$. Let $\Phi_a, \Phi'_a \subseteq S_a$ be some arbitrary sets of abstract states. We assume the left hand side of $IH(s_1; s_2)$, i.e., $sem_a(\Phi_a, s_1; s_2) = \Phi'_a$ holds and prove the right hand side $sem_c(R(\Phi_a), s_1; s_2) \subseteq R(\Phi'_a)$.

   From $sem_a(\Phi_a, s_1; s_2) = \Phi'_a$ we know that s_1 in state Φ_a does not fail since otherwise $sem_a(\Phi_a, s_1; s_2) = failure$ would hold. We can thus deduce that $sem_c(R(\Phi_a), s_1; s_2) = sem_c(sem_c(R(\Phi_a), s_1), s_2)$.

   Using $IH(s_1)$ we know that $sem_c(R(\Phi_a), s_1) \subseteq R(\Phi''_a)$ where $\Phi''_a = sem_a(\Phi_a, s_1)$. Plugging this into the equation gives us $sem_c(R(\Phi''), s_2)$ which when applying $IH(s_2)$ yields $sem_c(R(\Phi''), s_2) \subseteq R(\Phi'''_a)$ where $\Phi'''_a = sem_a(\Phi''_a, s_2)$.

To conclude this case we need to show that $\Phi_a''' = \Phi'$ holds. This is the case since $\Phi_a''' = sem_a(\Phi_a'', s_2) = sem_a(sem_a(\Phi_a, s_1), s_2) = sem_a(\Phi_a, s_1; s_2) = \Phi'$ and thus $sem_c(R(\Phi_a), s_1; s_2) \subseteq R(\Phi_a')$ holds.

2. Case: $s =$ `inhale acc(x.f, wildcard)`
We fix $s =$ `inhale acc(x.f, wildcard)`.
First we prove that $sem_a(\Phi_a, s) = \Phi_a' \Rightarrow sem_c(R(\Phi_a), s) \subseteq R(\Phi_a')$ holds for some arbitrary singleton abstract state $\varphi_a \in S_a$ and then generalize to arbitrary set of states $\Phi_a \subseteq S_a$ from this.

   We distinguish three cases.

   - $\varphi_a[0] = 1$
   $\varphi_a[1] = \bot$ holds because otherwise φ_a would not be a valid state in S_a. Thus $\Phi_a' = sem_a(\{\varphi_a\}, s) = \{(\varphi_a[0], \top)\} = \{\}$ holds, as well as $R(\{\varphi_a\}) = \{\varphi_a[0]\} = \{1\}$. From this $sem_c(R(\{\varphi_a\}), s) = sem_c(\{1\}, s) = \{\}$ follows from the definition of sem_c which proves $IH(s)$.

   - $\varphi_a[0] < 1 \wedge \varphi_a[1] = \top$
   From the left hand side of the induction hypothesis we know $\Phi_a' = sem_a(\{\varphi_a\}, s) = \{(\varphi_a[0], \top)\}$ holds and thus we get $R(\Phi_a') = (\varphi_a[0], 1]$.

     For any $\varphi_c \in R(\{\varphi_a\})$ we have that $sem_c(\{\varphi_c\}, s) = (\varphi_a[0], 1]$ and thus for every such φ_c $sem_c(\{\varphi_c\}, s) \subseteq R(\Phi_a')$ holds. The union over all φ_cs of $sem_c(\{\varphi_c\}, s)$ is thus also a subset of $R(\Phi_a')$ which means $sem_c(R(\Phi_a), s) \subseteq R(\Phi_a')$ holds which concludes this case.

   - $\varphi_a[0] < 1 \wedge \varphi_a[1] = \bot$
   From the left hand side of the induction hypothesis we know $\Phi_a' = sem_a(\{\varphi_a\}, s) = \{(\varphi_a[0], \top)\}$ holds and thus we get $R(\Phi_a') = (\varphi_a[0], 1]$.

     Since we have $R(\{\varphi_a\}) = \{\varphi_a[0]\}$ we get $sem_c(R(\{\varphi_a\}), s) = (\varphi_a[0], 1]$ and thus $sem_c(R(\Phi_a), s) \subseteq R(\Phi_a')$ holds.

   Applying Lemma 4.1 generalizes this proof to arbitrary sets of abstract states $\Phi_a \subseteq S_a$.

3. Case: $s =$ `exhale acc(x.f, wildcard)`
We fix $s =$ `exhale acc(x.f, wildcard)`.
First we prove that $sem_a(\Phi_a, s) = \Phi_a' \Rightarrow sem_c(R(\Phi_a), s) \subseteq R(\Phi_a')$ holds for some arbitrary singleton abstract states $\varphi_a \in S_a$ where we fix $\Phi_a = \{\varphi_a\}$ and then generalize to arbitrary states $\Phi_a \subseteq S_a$ from this.

   For exhaling wildcard permission amounts there are three cases to consider.

- Case: $\varphi_a = (0, \bot)$
  The left hand side of $IH(\texttt{exhale acc(x.f, wildcard)})$ does not hold since $sem_a(\{\varphi_a\}, \texttt{exhale acc(x.f, wildcard)}) = failure$. Thus the implication holds trivially.
- Case: $\varphi_a[0] < 1 \land \varphi_a[1] = \top$
  We assume the left hand side of the induction hypothesis which means $\Phi'_a = sem_a(\{\varphi_a\}, s) = \{(0, \top)\}$ holds. We get $R(\{\varphi_a\}) = (\varphi_a[0], 1]$ and with the definition of sem_c we get $sem_c(R(\{\varphi_a\}), s) = \bigcup_{\varphi_c \in (\varphi_a[0],1]} sem_c(\{\varphi_c\}, s)$. The union of all of these ranges is $(0, 1]$ and thus $sem_c(R(\{\varphi_a\}), s) = (0, 1]$.

  For $R(\Phi'_a)$ we get $R(\Phi'_a) = R(\{(0, \top)\}) = (0, 1]$

  To conclude, we have $sem_c(R(\{\varphi_a\}), s) = (0, 1] \subseteq R(\Phi'_a)$ which proves $IH(\texttt{exhale acc(x.f, wildcard)})$ for this case.
- Case: $\varphi_a[0] > 0 \land \varphi_a[1] = \bot$
  We assume the left hand side of the induction hypothesis which means $\Phi'_a = sem_a(\{\varphi_a\}, s) = \{(0, \top)\}$ holds and thus $R(\Phi'_a) = R(\{(0, \top)\}) = (0, 1]$ holds too. We get $R(\{\varphi_a\}) = \{\varphi_a[0]\}$ and with the definition of sem_c, $sem_c(R(\{\varphi_a\}), s) = sem_c(\{\varphi_a[0]\}, s) = (0, \varphi_a[0])$. Concluding, the following holds $sem_c(R(\{\varphi_a\}) = (0, \varphi_a[0]) \subseteq (0, 1] \subseteq R(\Phi'_a)$, which concludes this case.

Applying Lemma 4.1 generalizes this proof to arbitrary abstract states $\Phi_a \subseteq S_a$.

Chapter 5

Evaluation

Our focus during this project was on the performance of the abstract compared concrete wildcard encoding and of course the correctness of the implementation. For this reason we did benchmark testing throughout the whole project to guide our next steps and used the Viper test suite to find mistakes. In this chapter we show our approach for evaluating the different encodings. First we go over our methodology and explain where the data for our benchmarks came from and then we show our most interesting and characteristic results.

5.1 Methodology

5.1.1 Implementation

For the implementation of the abstract wildcard we added the keyword *sWildcard* to the Viper language. This allowed us to use the concrete wildcard representation in tandem with our new representation and let us compare the semantics of two encodings more easily and thus to find errors more quickly. Because these two representations should be compatible with each other, this of course added some overhead to the concrete wildcard encoding. This overhead comes from the fact that for comparing two wildcards, the Boolean mask also has to be taken into account. This results in an additional disjunction for such checks which enlarges the state space the SMT solver has to explore.

5.1.2 Performance Evaluation

To evaluate the performance we measured the execution time of the Boogie command line tool for input files generated from Viper files by the Carbon back-end. To automate the task of generating the Boogie input files we wrote a python script that, when given a folder with Viper input files, automatically

File name	Content
wildcard.vpr	Copy of input file
sWildcard.vpr	Copy of input file where every `wildcard` is replaced by `sWildcard`
old_wildcard.bpl	Output for unmodified Viper implementation
new_wildcard.bpl	Output for new Viper implementation using the `wildcard` keyword
new_sWildcard.bpl	Output for new Viper implementation using the `sWildcard` keyword

Table 5.1: Generated files for benchmarking

generates a directory with the files listed in Table 5.1. The three output Boogie files hold the encoding of the input Viper program each using either the unmodified version of Viper or the new implementation that enables the abstract wildcard encoding through the use of the keyword `sWildcard`.

These generated files are then also automatically benchmarked with a python script that uses the command line tool *hyperfine* to measure the execution time of the Boogie command line tool for each of these files. Hyperfine is parameterized to run three warmup runs for each command and then runs it 10 times, measuring the mean, average and standard deviation of the execution time samples. The machine we ran the benchmarks on has 16GB of memory, an Intel i7-8550U CPU running at 1.80GHz and runs *Manjaro Linux*.

5.1.3 Testing

To make sure our implementation works in the intended way, we used the existing Viper testing infrastructure. For this we extracted all test cases that used wildcards and syntactically replaced every occurrence of `wildcard` keyword with a `sWildcard`. In addition to that we also introduced our own test cases. This gave us a quick way to find implementation errors and even unintended semantic differences.

5.2 Example Programs

To see if there is a noticeable difference in performance between the unmodified Viper implementation and our new one we constructed example programs that inhaled and exhaled wildcard permission amounts hundreds

of times. The results were promising and so we decided to automatically generate and benchmark them.

These algorithmically generated programs however were not enough to tell whether there is a significant performance increase for real Viper programs and thus we needed to get a broader and more meaningful set of samples. We used the test cases provided by the Viper verification infrastructure as well as test cases from the test suites of front-end Viper verifiers. The results of the evaluation of these programs are shown in the next section.

5.2.1 Algorithmically Generated Programs

We automatically generated programs of the form shown in Listing 5.1. The parameters for the generation were the number of references the method main takes as parameters and the number of times the program inhales and exhales for each reference.

Listing 5.1: Algorithmically generated program with three references and n inhale-exhale-blocks

```
0  field f: Int
1
2  method main(ref0: Ref, ref1: Ref, ref2: Ref)
3  {
4      inhale acc(ref0.f, wildcard)
5      inhale acc(ref1.f, wildcard)
6      inhale acc(ref2.f, wildcard)
7      exhale acc(ref0.f, wildcard)
8      exhale acc(ref1.f, wildcard)
9      exhale acc(ref2.f, wildcard)
10
11     ...
12
13     inhale acc(ref0.f, wildcard)
14     inhale acc(ref1.f, wildcard)
15     inhale acc(ref2.f, wildcard)
16     exhale acc(ref0.f, wildcard)
17     exhale acc(ref1.f, wildcard)
18     exhale acc(ref2.f, wildcard)
19 }
```

5.2.2 Sample Programs

The Viper test suite includes many test cases of different lengths and complexities and thus provides a good basis for profiling the perfomance of the abstract wildcard implementation. In addition to that we also used the *Nagini*, *Gobra* and *Vercors* test suites to generate sample Viper programs. The only restriction we had for these programs was that they were parsable. Whether they raised verification errors or not does not matter for us since it is as

important for a verifier to rapidly find verification errors as it is to conclude that the program is correct.

5.3 Results

In this section we show some exemplary results we got from benchmarking the different programs we had at our disposal.

5.3.1 Algorithmically Generated Programs

We ran benchmarks on the generated programs using our new implementation of abstract wildcards and the unmodified Viper implementation. To compute the speed increase of the new implementation with respect to the unmodified one we divided the execution time of the abstract wildcard implementation by the execution time of the unmodified concrete version for each of the sample programs. The results can be seen in Figure 5.1 on a linear and in Figure 5.2 on a logarithmic scale. There we can see that the ratio of the execution times increases drastically with the number of references used.

5.3.2 Sample Programs

We used the previously presented benchmark infrastructure to measure the run-times of all of these programs for three cases.

- The unmodified Viper implementation using the `wildcard` keyword
- The new implementation using the `wildcard` keyword
- The new implementation using the `sWildcard` keyword

The new implementation using the `wildcard` keyword is interesting for us to test since it works with the overhead of a having a second mask to check in every permission amount lookup. On the other hand it also implements some performance optimizations to the concrete wildcard encoding as mentioned in Chapter 3. Our other programs did not show such a stark performance improvement as the artificial examples. For these programs we will not explain in detail what they prove since this would be out of the scope of this project.

In Figure 5.3 we see several examples with moderate execution time.

Viper program 0082 In the test suite of Viper there are a lot of small programs that test specific features of Viper. A typical one of these is the program *0082*. It consist of 14 lines of code that declare one method and one predicate that is used within the method. For such small programs we could not measure a noticeable difference in verification speeds using either encoding.

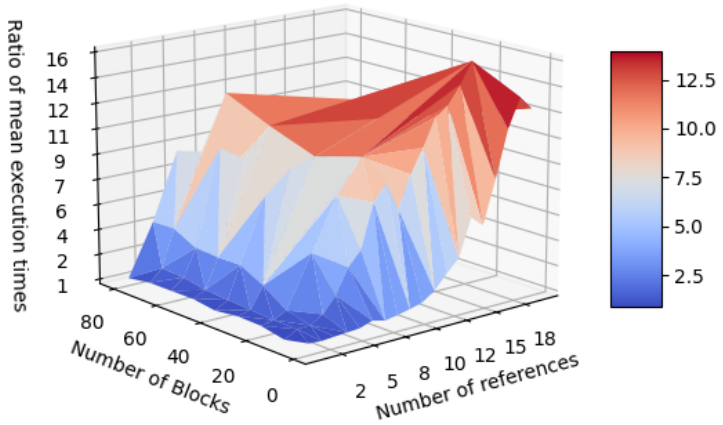

Figure 5.1: Plot of execution time ratios for algorithmically generated programs

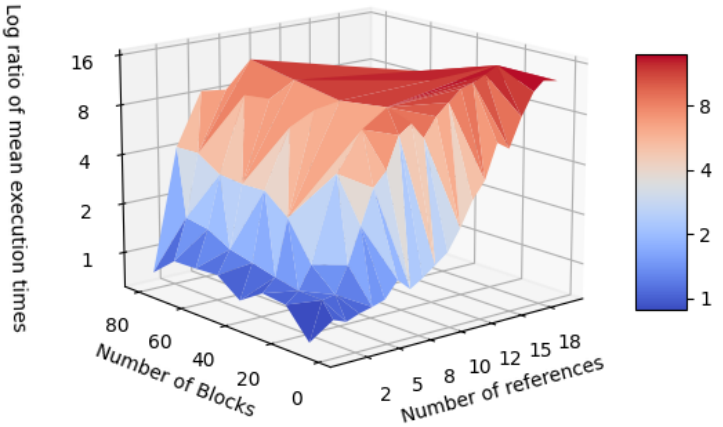

Figure 5.2: Log scale plot of execution time ratios for algorithmically generated programs

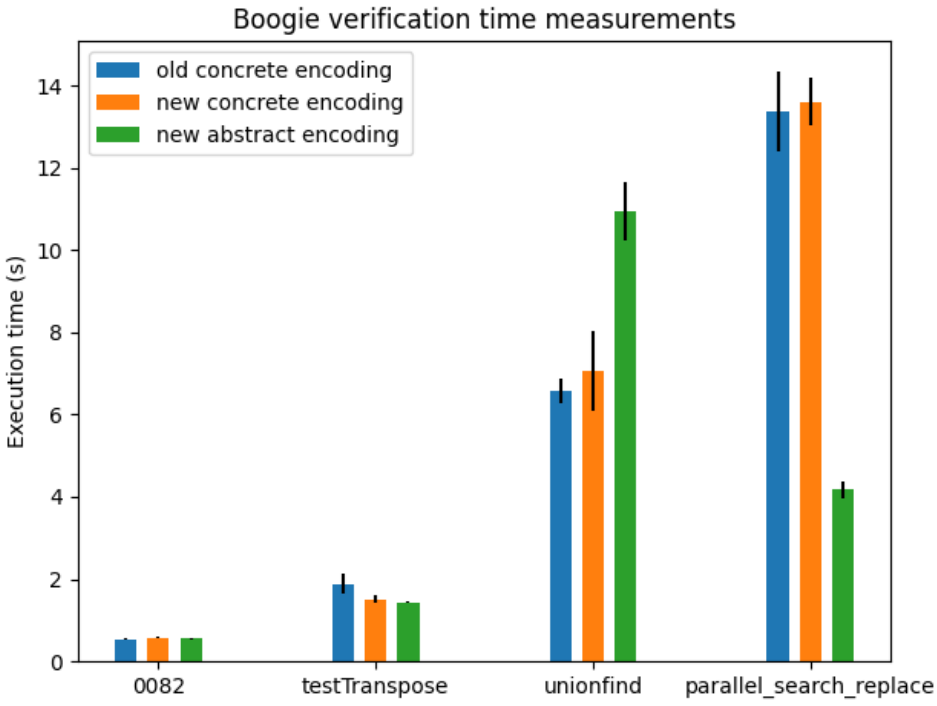

Figure 5.3: Benchmarks of programs with moderate verification time

testTranspose The program *testTranspose* also stems from the Viper test suite but shows a small performance increase. It uses wildcards in four places in method preconditions using quantified permissions.

unionfind Also from the test suite we got the program *unionfind*. This shows quite drastically that there are cases where the abstract wildcard encoding takes a performance hit. The verification time almost doubles in comparison to the old implementation as well as the new implementation that optimizes the concrete wildcard encoding optimized wildcard version. This might be caused by the additional disjunctions this encoding brings with it but we were not able to test this hypothesis.

parallel_search_replace The program *parallel_search_replace* was derived from the Gobra test suite. This is a big program with 1126 lines of code and 14 wildcard uses. This shows exactly the opposite of the unionfind example. The performance of the abstract wildcard encoded version increases dramatically with respect to the two concrete versions.

Figure 5.4 shows benchmarks of programs with slower verification speeds. With these programs the performance differences are more pronounced and so we use a logarithmic scale for this plot.

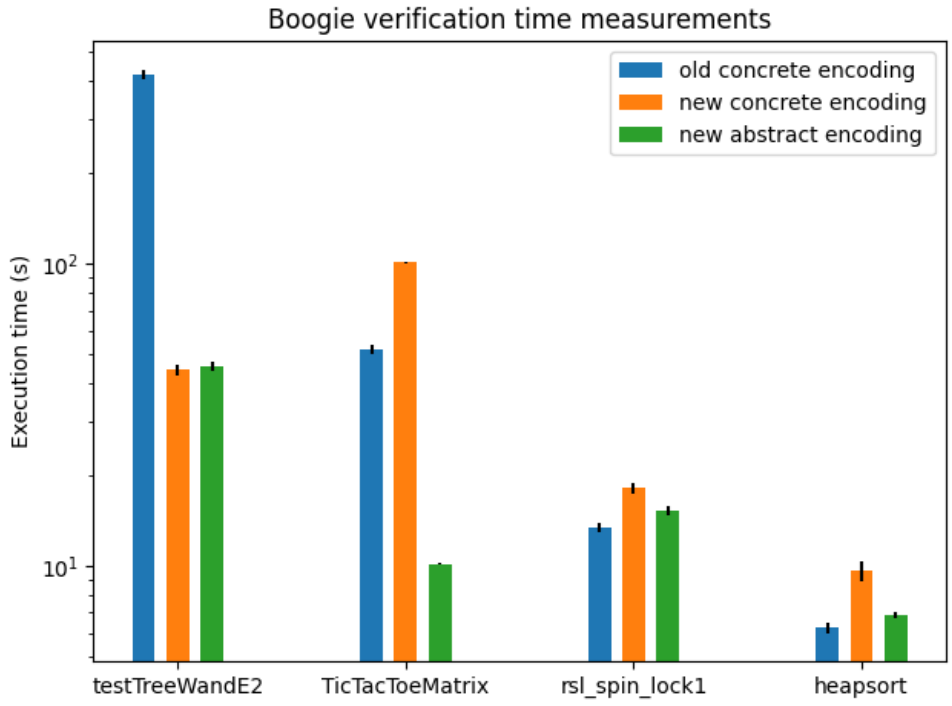

Figure 5.4: Benchmarks of programs with longer verification time

testTreeWandE2 We found this example in the test suite of Viper. It uses wildcards in 16 places and has about 500 lines of code and the verification terminates with errors. The plot shows that the new implementation gets to the result almost an order of magnitude faster than the old implementation.

TicTacToeMatrix We derived the sample program *TicTacToeMatrix* from the Vercors test suite. This program has almost 560 lines of code and uses the wildcard keyword in 56 places. Abstract wildcards seem to give this program a huge performance boost over concrete wildcards. It verifies the program more than five times faster than the old encoding. Also clearly visible here is the overhead the concrete wildcard faces in the new implementation due to the introduction of a Boolean mask, the verification of this case takes about twice as long.

rsl_spin_lock1 We found the program *rsl_spin_lock1* on the Viper encoding of RSL Logics examples website. It has about 750 lines of code and uses wildcards 11 times. It also makes extensive use of permission introspection which is why every single use of an abstract wildcard with the keyword `sWildcard` gets internally converted to a concrete wildcard. For this reason, the new implementation using either keyword shows such a similar perfor-

mance. Other than that there the old implementation still seems to be faster for verifying this program.

heapsort The program *heapsort* was derived from the Gobra test suite and serves as a good example to show the overhead the second mask in the encoding introduces. The new implementation is quite a lot slower when used with the `wildcard` keyword as opposed to the `sWildcard` keyword.

Chapter 6

Conclusion

We have explored an alternative representation for wildcard permission amounts we call abstract wildcards. For this we implemented a working version of the encoding, evaluated it against the existing concrete wildcard encoding and made it compatible with permission introspection. We also gave a simplified proof of the soundness of the abstract wildcard encoding with respect to the concrete encoding that should easily be extensible to a more general setting.

We saw that for some test cases the verification speeds for programs making use of abstract wildcards was five to ten times faster than for concrete wildcards. For other programs however the verification speed slowed down by half, but overall the performance difference was quite small. The slowdown might be explained by the introduction of an additional disjuction in the encoding of permission amount look-ups but we were not able to test this hypothesis. Our algorithmically generated examples showed that the abstract wildcard fares way better when multiple references are used in the same method. We were not able to find the exact reason for this.

It would be interesting to see how the memory usage for the verification changes with abstract wildcard permissions since this often seemed like the source for inconclusive test results. There is still some work to be done to ensure the compatibility of abstract wildcards and permission introspection in Viper. The solution we chose is too restrictive and programs that use permission introspection barely benefit from the use of abstract wildcards or even get slowed down by it.

For the implementation of abstract wildcards we introduced the `sWildcard` keyword to the Viper language, which has the same purpose as the already existing `wildcard` keyword. If abstract wildcards were to be used it would make sense to keep only one keyword for wildcards since having more keywords introduces more complexity to the programming interface. The

encoding to be used by Viper could then be chosen behind the scenes.

Not just the addition of a second keyword for wildcards leads to more complexity, but also the introduction of abstract wildcards into the Carbon back-end. The code-base gets more complex by adding such an additional feature and is thus more prone to errors and there is more maintenance necessary.

Bibliography

[1] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 259–270, New York, NY, USA, 2005. Association for Computing Machinery.

[2] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[3] Charles Antony Richard Hoare. An axiomatic basis for computer programming. In *Communications of the ACM, 12(10):576–580*, 1969.

[4] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA Formal Methods (NFM). pp. 41–55*, 2011.

[5] K. Rustan M. Leino. This is boogie 2. In *Microsoft Research, Redmond, WA, USA*, 2008.

[6] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation pp 41-62*, 2015.

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

[8] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1), May 2012.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

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

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

Title of work (in block letters):

An Abstract Representation for Wildcard Permissions in Viper

Authored by (in block letters):

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

Name(s):	**First name(s):**
Bachmann	Yanick

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)**
Winterthur, 31.7.21	Y. Bachmann

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