Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

GUC German University in Cairo

Adding Generalized Magic Wand Support to a Verification Condition Generation Based Verifier

Ahmed Gamal

Bachelor's Thesis
Chair of Programming methodology
Department of Computer Science, ETH Zürich

Supervisors
Dr. Alexander J. Summers
Dr. Malte Schwerhoff
Prof. Dr. Peter Müller

31st August 2018

Contents

Acknowledgements

My eternal gratitude goes to my supervisor Alexander J. Summers without whom a lot of the work achieved during this thesis would become impossible. Alex, Thank you for your continuous encouragement and for the inspiring discussions that we have had. You have had your door always open when I needed support.

I would like to thank Malte Schwerhoff for his discussions and ideas regarding *magic wands snapshots* and for his help with questions regarding Silicon.

I am also grateful to Prof. Peter Müller for giving me the opportunity to be part of his group and to work on an interesting problem in the field of program verification.

I would like to thank my family for their endless support, encouragement and prayers.

I am especially grateful to my friends who were there to support me during stressful times.

Last but not least, I would like to thank the programming methodology group at ETH Zürich for welcoming me and providing a friendly and supportive environment for the past six months.

Introduction

The Viper project [5] is a verification infrastructure developed by the Chair of Programming Methodology at ETH Zürich. As shown in Figure 0.1, inspired by [5], Viper currently consists of an intermediate language, which provides a flexible permission model, and two back-end verifiers: Silicon [8], a verifier that uses symbolic execution, and Carbon [3], a verification-condition-generation-based verifier.

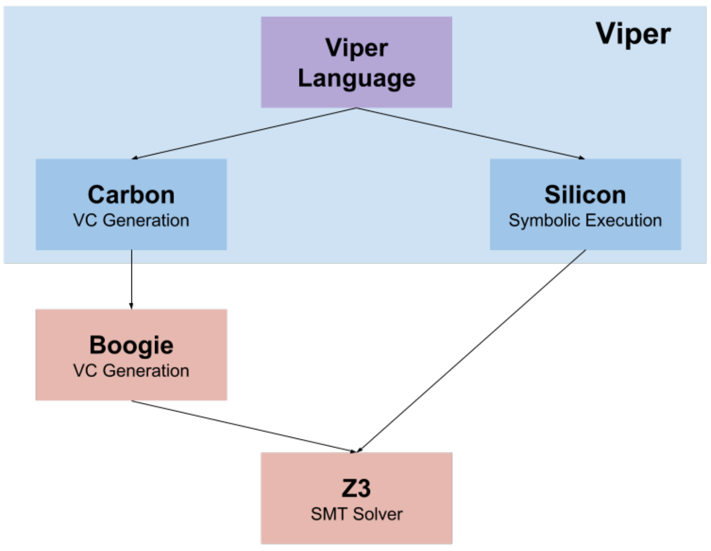

Figure 0.1: Viper Infrastructure and the underlying tools

Viper is based on separation logic [7] which is a permission-based logic that introduces two important connectives: separating conjunction and magic wands (also known as separating implication). Magic wands are useful for iterative traversal of data structures. In [9], Malte Schwerhoff and Alexander J. Summers show how to support magic wands in automatic verifiers along with an implementation for their approach in Silicon. Magic wands were then supported in Carbon in [6]. Finally, a new version of magic wand support was introduced in [1] and implemented in Silicon. The main goal of this bachelor

thesis is to support the magic wands introduced in [1] in Carbon.
We first provide the necessary information about Viper, Carbon and magic wands. Then we introduce our design for adding the new magic wand support to Carbon. Finally, we discuss an extension for our design in Chapter 3 which we had time to work on as an extension of our core goals.

Chapter 1

Background

In this chapter, we will describe the relevant Viper information from Figure 0.1 level by level. First, we will introduce the Viper language constructs that are relevant to our project. Then, we will introduce Carbon and Boogie. Finally, we will focus on the magic wands in both the Viper language and Carbon as it is essential for our work.

1.1 Viper Language

Viper language supports basic constructs such as methods, predicates (Section 1.1), conditionals and loops. Viper language also supports specifications using pre/post conditions as well as loop invariants. The following subsections describe the Viper language constructs that are necessary for understanding our project. We are going to explain the necessary Viper constructs using the example in Listing 1, which is a simple Viper program to copy the values of the field `y.f` to `x.f`. `Ref` type in Viper is similar to class in object-oriented languages.

```
field f: Int

method copy(x: Ref, y: Ref)
requires acc(x.f) && acc(y.f, 1/2)
ensures acc(x.f) && acc(y.f, 1/2) && x.f == y.f
{
        x.f := y.f
}
```

Listing 1: A method that copies the value of `y.f` to field `x.f`

Permissions

Reasoning about heap locations in Viper programs requires having permission to these locations. Field permissions specify which heap locations can be read/modified by some operation. In Listing 1, the `copy` method modifies the value of `x.f` and reads the value of `y.f`. Thus, a *write* permission for `x.f` and

a *read* permission for `y.f` are required. The *write* permission (full permission) to `x.f` is gained by having `acc(x.f)` in the precondition of the method in line 4. The *read* permission is gained by `acc(y.f, 1/2)` in the precondition.
In general, a permission to a field in Viper is represented by a fraction value. Permission to some field `x.f` is written as `acc(x.f, p)` (we call them accessibility predicates) where p is a non-negative fractional value. Writing `acc(x.f)` means having the full permission to `x.f` which is the equivalent to writing `acc(x.f, 1)` and having a full permission enables different operations to modify `x.f`. A positive permission value to `x.f` means the value of `x.f` can be read by different operations but cannot be modified unless the full permission is possessed.

Permissions are transferred between methods through method calls. Whenever a method is invoked, its preconditions should be transferred to the callee by the caller. Then, after the callee method finishes executing its body, its postconditions are transferred to the caller method. In Listing 2, the `copy` method is being called from another method (`client`). When the `copy` method is called, the caller method, `client`, gives away permission for `acc(x.f)` and `acc(y.f, 1/2)`, which are stated by the preconditions of the `copy` method, and after the call, the postconditions of the `copy` method are transferred to the `client` method. Thus, after the call the `client` method gains permission for `acc(x.f)` and `acc(y.f, 1/2)`, along with the information that `x.f` and `y.f` are now equal. The operation of losing permissions is called *exhaling* the permissions. Whenever a permission is exhaled, the current method should possess at least the required amount needed to be exhaled; otherwise, the exhale fails. The opposite operation of gaining permissions is called *inhaling* a permission.

```
1  field f: Int
2
3  method copy(x: Ref, y: Ref)
4  requires acc(x.f) && acc(y.f, 1/2)
5  ensures acc(x.f) && acc(y.f, 1/2) && x.f == y.f
6  {
7          x.f := y.f
8  }
9
10 method client(x: Ref, y: Ref)
11 requires acc(x.f) && acc(y.f)
12 {
13         copy(x, y)
14 }
```

Listing 2: The `copy` method is now invoked from a `client` method.

The concept of permissions is useful in solving the frame problem (which parts of the heap are guaranteed to be left unchanged). Framing means proving that a value or a certain assertion is not changed by modifying the heap. Permissions can be distributed among different methods such as in Listing 2, when the `copy` is called, half permission for `y.f` is transferred to the `copy` method while the other half permission remains with the `client` method. However, the total summation of permissions for a certain location distributed among different methods at one time should not exceed 1 in order to prevent data races (having a field permission with an amount greater than 1 leads to an inconsistent state in Viper). Thus, only one method can modify this location. If some method has a positive permission to the field `x.f`, it is guaranteed that this value can not be modified by any other method as no other method can gain the full permission to `x.f`. For instance, in Listing 2, the `client` method keeps half permission to `y.f` when calling `copy(x, y)`; therefore, the verifier knows that the `copy` method can only read `y.f`, and the value for `y.f` is unchanged after making the call. We say in this case that `y.f` is framed (not changed) across the method call.

Predicates

So far, we discussed permissions to single fields. In some cases, we need to express permission to data structures of an unbounded size such as linked lists and trees. As a linked list can have arbitrary length, one cannot specify permissions to all its fields by enumerating all the relevant fields. Viper supports predicates to represent permissions to such unbounded data structures. Listing 3 shows an example for a predicate, recursively defined, representing permissions to a null-terminating linked list. In Viper, the predicate and its body are not equivalent to prevent infinitely unrolling recursive predicate. For instance, having permission to predicate `list(x)` does not directly imply having permission to `x.value`. This exchange between the predicate and its body can be done explicitly through `unfold` and `fold` statements. The statement `unfold list(x)` exchanges the predicate with its body; in other words, the predicate `list(x)` gets exhaled and its body gets inhaled leading to having permission to both fields `x.value`, `x.next` and to `list(x.next)` provided that it is not null. Folding a predicate has the reverse effect: the body of the predicate gets exhaled, then the predicate itself gets inhaled.

```
field value: Int
field next: Ref

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

Listing 3: recursive predicate for linked list

Similar to single field permissions, the permissions for predicates are written `acc(list(x), p)` . We can write `list(x)` as a short for `acc(list(x), 1)`. Viper also supports the `unfolding` expressions which unfold the predicate temporarily in order to evaluate an expression and the folds the predicate again. It can be written as `unfolding list(x) in x.value == 2`.
Viper supports framing across fold-unfold pairs. This means having the field `x.value` with some value, e.g. 2, before folding the predicate `list(x)` and then unfolding `list(x)` later should preserve the value for `x.value` unchanged

(equals 2 in this example). This is shown in Listing 4. The value `x.value` is assigned to 2 in line 13. The predicate `list(x)` is then folded and any operation that does not give away the whole permission to `list(x)` does not affect the value for `x.value` as it is framed and the verifier is able to prove the information that `x.value == 2` after doing the `unfold`.

```
 1  field value: Int
 2  field next: Ref
 3
 4  predicate list(x: Ref){
 5          acc(x.value) && acc(x.next) && (x.next != null ==> list(x.next))
 6  }
 7
 8  method foo(x: Ref)
 9  requires list(x)
10  {
11          unfold list(x)
12          x.value := 2
13          fold list(x)
14
15          // some operations that do not give away list(x)
16
17          unfold list(x)
18          assert x.value == 2
19  }
```

Listing 4: A Viper example that shows framing values across fold-unfold pairs.

1.2 Carbon

Carbon translates Viper code into Boogie[4] code which is then used for generating verification conditions that are passed to Z3 (SMT solver). As Boogie is lower-level language than Viper, many Viper features such as heaps, permissions and predicates need to be encoded into Boogie. We will show in this section the encoding of the main Viper features discussed in the previous section.

State

A carbon state is represented by permissions to heap locations as well as the values of these locations. We keep the permissions in a variable called `Mask` and we keep the values in another variable called `Heap`. Both Mask and Heap are represented as Boogie maps. Boogie supports updatable maps. A Boogie map entry can have multiple (possibly different type) key but only one value. The `Heap` map is represented by a map from reference and field to a value of the same type of the field. For example, the value of `x.f` is translated to `Heap[x, f]` and the returned value is of the same type as the field `f`. The `Mask` map is also a map from reference and field but it maps to a fraction value. A fraction in Boogie can be represented by type `real`. Having a reference as a location works fine when talking about a certain field but it is not clear why we need a reference key when referring to permissions to predicates. Instead of keeping permissions to predicates in a separate map, they are stored in the same `Mask` map but the reference value for them is `null` and the field, in this case, will be the Boogie translation of the predicate itself. Hence, the permission to a predicate `list(x)` is translated to `Mask[null, list(x)]`. The rest of the thesis when we refer to a Carbon state, we will use the symbol σ instead of referring to the `Mask` and `Heap` separately unless we need to specify some operation related to the `Mask` alone or the `Heap`.

```
type ref;
type Field a;
type HeapType = <a> [ref, Field a] a;
var Heap: HeapType;
```

Listing 5: Program heap encoding as a map in Boogie

Listing 5, inspired by [3], shows the encoding of the program heap in Boogie. The type `ref` and a polymorphic `Field` type are first defined. The keyword `type` in Boogie is used to define new types. Then, we define the `heapType` to be a map from a `ref` and a `Field` of type `a` to a value of type `a`. Finally, we declare the heap as a variable of type `heapType`.

Exhale

As exhale is not an operation in Boogie, it needs to be translated as a sequence of Boogie statements. Boogie supports `assert` statements. Therefore, exhaling a pure expression (an expression with no accessibility predicates) will just be translated as an assert statement. When a permission to a certain field is exhaled, the permission amount to this field needs to be subtracted from the `Mask`. After subtracting the permission amount, if the whole permission to a field is lost, the value for this field is not framed anymore, which means it could have any arbitrary value. Giving a field an arbitrary value is called to `havoc` the field. Boogie supports *havocing* a variable by using the `havoc` keyword. The way this havocing after an exhale is translated in Carbon is by having a fresh heap (with all its locations assigned to arbitrary values). Then, only values for locations to which a positive permission is still held are transferred to the new fresh heap. Finally, the new heap is assigned to the original program heap resulting in having the same values before the `exhale` for all the locations to which a permission is still held and having arbitrary values to the other locations.

```
Mask[x, f] := Mask[x, f] - p;
havoc ExhaleHeap;
assume IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask);
Heap := ExhaleHeap;
```

Listing 6: The translation of an exhale statement to Boogie

Listing 6 shows the Boogie encoding for an exhale statement. First, the needed permission is removed from the current Mask. Then, a temporary heap (ExhaleHeap) is havoced. The locations on the Heap is then assumed to be equal to the corresponding values on ExhaleHeap for all the locations to which a permission is still held in Mask. This is done via the IdenticalOnKnownLocations function. Finally, the ExhaleHeap gets assigned to Heap. Notice that the function IdenticalOnKnownLocations is a Boogie-level function, whose behavior is specified through writing Boogie axioms. We will provide a high level description of the function instead of providing the implementation detail as it is not really important for our discussion: it is a function that takes two heaps and a mask and equates all the locations, to which the mask has permission, on both heaps. We chose to introduce it here to present two Boogie features which are Boogie functions and axioms that are used to specify the behavior of different functions without implementing the body of the function.

Fold

Most Viper operations can be translated as a sequence of exhales and inhales. The fold operation, for instance, can be seen as exhaling the body of the predicate and then inhaling the predicate itself. Thus, when a fold statement is translated, Carbon will determine the body of the predicate, generates the Boogie translation of exhaling the body and then generates the Boogie translation of inhaling the predicate.
Consider a predicate p(x) whose body is acc(x.f). Listing 7 shows the Boogie encoding for folding such a predicate. The predicate body is first exhaled by removing the permissions from Mask. The predicate itself is inhaled. Finally, the locations to which permissions are lost get havoced. Notice that the havocing step here is postponed after inhaling the predicate p(x) to frame the values across fold-unfold pairs as mentioned in section 1.1.

```
Mask[x, f] := Mask[x, f] - p;

Mask[null, cell(x)] := Mask[null, cell(x)] - p;

havoc ExhaleHeap;
assume IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask);
Heap := ExhaleHeap;
```

Listing 7: The translation of an exhale statement to Boogie

1.3 Magic Wand

Magic wand (or separating implication) is a binary connective in separation logic [7]. The magic wand is written as A-*B, where both A and B are assertions. A and B are called left-hand side and right-hand side of the wand respectively. The magic wand is defined semantically as follows:

$$\sigma \vDash A \mathrel{-\!\!*} B \;\Leftrightarrow\; \forall \sigma' \perp \sigma \cdot \;(\sigma' \vDash A \;\Rightarrow\; \sigma \uplus \sigma' \vDash B)$$

Here, $\sigma \uplus \sigma'$ means that the two states σ and σ' are compatible. Two states are compatible, if and only if, for each heap location the summation for its permission in the two states does not exceed the full permission and the value for this location agrees on the heaps of both states. This definition means that for all the states σ' in which `A` holds, σ' can be combined with the magic wand to result in a state in which `B` holds. More informally, the magic wand can be thought of as the difference between the right-hand side state, in which `B` holds, and the left-hand side state, in which `A` holds.

Magic wands are supported in Viper. Although the magic wand is another construct in Viper such as predicates, we chose to present magic wands in a separate section, instead of having it in Section 1.1, due to its importance for our work. First, we will explain how magic wands are used in Viper. Then, we will present how the magic wands were implemented in the Carbon back end.

Viper

In Viper, the syntax of magic wands is `A-*B`, where both `A` and `B` are self-framed assertions. By self-framing, we mean that if some field appears in the assertion, e.g. `x.f`, the required permission to this location must appear earlier in the assertion, e.g. `acc(x.f)`. We will use the example in Listing 8 throughout the thesis to explain magic wands. It is a straightforward example for traversing a linked list. The computation done during the traversal is omitted (line 21) as it is not important for our discussion but it may be computing the sum of the values in the list for example. The predicate in line 4 represents permissions to the list and is shown in Section 1.1. The variable `cur` is the iterator which traverses the list node by node, it starts at node x (line 12) and then during the loop, it advances to the next node each time (line 24) until the next value is `null` as specified by the loop condition. Both `package` statements on lines 13, 26 and the `apply` statement on line 32 will be explained shortly. Let us now consider the specification in the pre/post conditions and the loop invariants. Line 9 specifies that the method should have permission to `list(x)` as a precondition. Similarly, line 10 specifies that when the method finishes execution, it should return permission to `list(x)` to the caller. The loop invariants on lines 16 and 17 preserve the permissions between the iterations. The first invariant represents the permissions from the current node to the end of the list (represented by `list(cur)`). The second invariant represents the rest of the list as the magic wand can be interpreted as the difference state between the right-hand side, `list(x)`, and the left-hand side, `list(cur)`, of the wand.

Packaging a wand

Packaging a magic wand in Viper means creating a new wand and adding it to the current state. This is done using a `package` statement, e.g., `package list(cur) --* list(x)`. More informally, one can think of packaging a magic wand as computing the difference state between the right-hand side and the left-hand side. Packaging the wand is the guarantee that it is sound to transform the left-hand side to the right-hand side of the wand. The state representing the

```
field value: Int
field next: Ref

predicate list(x: Ref){
	acc(x.value) && acc(x.next) && (x.next != null ==> list(x.next))
}

method traverse(x: Ref)
requires list(x)
ensures list(x)
{
	var cur: Ref := x
	package list(cur) --* list(x)

	while(unfolding list(cur) in cur.next != null)
	invariant list(cur)
	invariant list(cur) --* list(x)
	{
		unfold list(cur)

		// do something with cur.value

		var prev: Ref := cur
		cur := cur.next

		package list(cur) --* list(x){
			fold list(prev)
			apply list(prev) --* list(x)
		}
	}

	apply list(cur) --* list(x)
}
```

Listing 8: Viper example for traversing a linked list using magic wands.

difference between the two sides of the wand is called the *footprint* of the magic wand σ_{foot}.
Generally, program verification in presence of magic wands, without any user guidance, is known to be undecidable. However, In [9], Schwerhoff and Summers present an approach for automating verification in presence of magic wands by allowing ghost operations provided by the user to provide guidance to the verifier. The ghost operations guide the verifier how it could transform the left-hand side of the wand to the corresponding right-hand side (computing σ_{foot}). The allowed ghost operations in [9] are *folding*, folds a predicate, *unfolding*, unfolds a predicate, *applying*, applies a magic wand (explained in the next subsection), and *packaging*, packages a magic wand.
In Viper, the [9] version of the magic wand was supported. Then, another version [1] was supported in the Silicon back-end but not in Carbon. The

main differences between the two versions are: 1) The newer version has different syntax while packaging the wand regarding the way ghost operations are written. Figures 1.1 and 1.2, taken from [1], show the difference in syntax. 2) The newer version also adds support to more ghost operations than the four operations mentioned in [9]. It allowed more arbitrary code to be written during packaging a wand such as `assert` statements and branching via `if` statements. The new version of magic wands can be considered as a generalization of the older version. Thus, through the rest of this thesis, we will denote the new version of magic wands by *generalized magic wands*.

package_statement := **package** *assertion* $-\!\!*$ *assertion_with_ghost_operations*

Figure 1.1: Old syntax of the package statement.

package_statement := **package** *assertion* $-\!\!*$ *assertion* **{**
    statements_as_ghost_operations
}

Figure 1.2: New syntax of package statement.

The example in Listing 8 shows the syntax for the generalized magic wands. Let us see how the ghost operations guide the wand to transform the left-hand side to the right-hand side. Lines 27 and 28 show the ghost operations needed to guide the verifier to package the wand. Before packaging the wand in line 26 the program state has permission to `list(cur)` and `list(cur) -* list(x)` from the loop invariants. The first ghost operation folds `list(prev)` which results in permission to the predicate `list(prev)`. Then, the `apply` statement in line 28 gets executed resulting in the right-hand side of the wand which is `list(x)` (the right-hand side of the wand in line 26 as well). Thus, starting with the left-hand side of a package statement and following the ghost operations stated results in the right-hand side of the wand. Failing to perform a ghost operation or reach the right-hand side state causes the package statement to fail, raising a verification error.

Applying a wand

One can apply a held magic wand using `apply` statement, e.g., `apply list(cur) --* list(x)`. Once a magic wand is packaged, it is treated, in Viper, as an opaque resources. In other words, the verifier knows nothing about the footprint of the magic wand. It is seen as a promise to return the right-hand side state whenever a left-hand side state is provided. Therefore, in order for a magic wand to be successfully applied, the current state should have permission to the magic wand to be applied and the left-hand side of the wand should hold in the current state. Both the left-hand side and the magic wand are then exhaled and the right-hand side of the wand is inhaled. Consider the `apply` statement in line 32 in Listing 8, notice that the left-hand side of the wand, `list(cur)`, holds before applying the wand as it is preserved by the loop invariant. The wand is then applied resulting in the right-hand side, `list(x)`, causing the postcondition in line 10 to verify successfully.

Carbon

At the moment we started this project, only the old magic wand [9] was implemented in Carbon. Our main aim through this project is to extend Carbon to support the generalized version of magic wands. As our work is to extend the older version, one needs to understand the implementation of the older version to be able to understand our work. The older magic wand version was based on [9]. However, the design in [9] is more suitable for symbolic execution. Thus, we are going to describe a modified version of it which is more suitable for Carbon. This modified version is similar to the supported magic wands in [6]. The main differences between the [9] version and the old implemented version in Carbon are: 1) Carbon generates a Boogie code; therefore, the Carbon methods return statements instead of states. 2) The states in carbon are represented by mutable maps while the states in the [9] version are immutable. Thus, we can in Carbon add permission to some state or assign a state to some other state instead of creating a new state every time we need to modify one.

Packaging a wand

In order for a magic wand to be packaged successfully, the verifier needs to successfully compute the footprint σ_{foot} of the magic wand. Figure 1.3 shows the algorithm for computing the footprint of the magic wand. We will trace the algorithm for the package statement in line 26 in Listing 8 to explain the algorithm. Before we trace the code, note that Listing 8 uses the generalized magic wand syntax while the algorithm mentioned in Figure 1.3 is for the old syntax. Therefore, what we are really tracing is the syntax in Listing 9. First when the `package` statement is encountered, the **package** method in

```
package list(cur) --* folding list(prev) in
                          applying (list(prev) --* list(x)) in
                             list(x)
```

Listing 9: Package statement with the old magic wand syntax

Figure 1.3 is invoked on the magic wand. A fresh state, σ_{ops}, is created which will accumulate the result of executing the ghost operations as we will see. Then, a fresh state, σ_A, is created and the left-hand side, `list(cur)`, is inhaled in σ_A. Then the **exec** method is called to start executing the ghost operations. The `'++'` operator means concatenation between different Boogie statements as each method return some Boogie statements and the result is the concatenation of these statements together. As the first ghost operation is a `folding` operation, then the corresponding **exec** method is invoked. As the required permissions may come from the left-hand side state, denoted by σ_A, or the current state before the `package`, denoted by σ_{curr}, both states are put on a stack of states, denoted by $\overline{\sigma_i}$. A fresh state σ_{temp} is created, then the body of the predicate `list(prev)` is transferred from $\overline{\sigma_i}$ to σ_{temp} via calling `exhale_ext` method (by transferred we mean the permission is removed from its old state and added to the new state, σ_{temp} in this case). Thus, the `list(prev.next)` (which is `list(cur)`) is transferred from σ_A, while both

`acc(prev.value)` and `acc(prev.next)` are transferred from σ_{curr}. The σ_{temp} state now has all the required permissions to perform the fold, then the fold statement is executed in σ_{temp} and the `exec` method is called recursively on the rest of the right-hand side. Note that σ_{ops} becomes $\sigma_{ops} \uplus \sigma_{temp}$ when making a recursive call to `exec`, which means that the new σ_{ops} is the result of adding the permissions of both σ_{temp} and the old σ_{ops}.
The `exec` method is now called with the `applying` case as it is the next ghost operation. Once again, σ_{temp} is a new fresh state then both the left-hand side of the wand to be applied, and the wand itself are transferred from σ_{ops} and σ_{curr} respectively to σ_{temp}. Only then, the `apply` statement can take place in σ_{temp} and again the `exec` method is called on the rest of the right-hand side. Finally, the last `exec` method is called as there are no more ghost operations, a fresh σ_{temp} state is created, then the assertion in the right-hand side of the wand is transferred to this σ_{temp} and finally, the wand is successfully packaged and added to the current state. We consider all the permission removed from states other than the left-hand side and σ_{ops} as the footprint of the magic wand. More details on the implementation of `exhale_ext` and description of the algorithm are mentioned in [9] and [6].

State Booleans

State booleans is introduced in [6] as a part of encoding of `package` statements in Carbon. We will introduce the problem that the state booleans solves; then, we will show what is meant by state booleans and how they are used. Consider the magic wand in Listing 10. The left-hand side of the magic wand is `false`;

```
field f: Int

method foo(x: Ref)
{
        package false --* acc(x.f)
}
```

Listing 10: Packaging a wand with an inconsistent left-hand side.

therefore, this wand cannot be applied in a consistent state (`false` must hold in such a state which means it is an inconsistent state) and as a result, the wand should be trivially packaged without affecting the global state after packaging the wand. When this wand is packaged in Carbon, the left-hand side assertion is inhaled into the left-hand side state; thus, `false` should be assumed in σ_A. In Carbon, assumptions cannot be done in a certain state without affecting the global state and assuming `false` will lead to an inconsistent global state after the wand is packaged. The solution introduced in [6] is using state booleans. A state boolean is a boolean expression that carries the assumptions about some state. This way, assumptions about different states are kept separately in their state booleans. For instance, to add the assumption `x.f == 2` in some state σ_i, it is added to the state boolean b_i `:=` b_i `&& (x.f == 2)`. When we assert that `x.f == 2` in the same state, it is encoded as an implication `assert` $b_i \implies$ `(x.f == 2)`. We will denote to the state boolean of some state σ_i by σ_i`.boolvar` in the rest of this thesis. State booleans are fully explained in [6].

```
$\sigma$.package($A \mathrel{-\!\!*} G$) $\rightsquigarrow$
    $\sigma_{ops}$ := freshState()
    $\sigma_A$ := freshState()
    $\sigma_A$.inhale($A$)
    return exec($\sigma \cdot \sigma_A, \sigma_{ops}, G$) ++
        $\sigma'$.addWand($A \mathrel{-\!\!*}$ nested($G$))

exec($\overline{\sigma_i}, \sigma_{ops}$, folding $P(e)$ in $G$) $\rightsquigarrow$
    $\sigma_{temp}$ := freshState()
    $v := \sigma_{ops}$.eval($e$)
    return exhale_ext($\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{temp}$, Body($P$)[param $\mapsto v$]) ++
        $\sigma_{temp}$.fold($P, v$) ++
        exec($\overline{\sigma_i}, \sigma_{ops} \uplus \sigma_{temp}, G$)

exec($\overline{\sigma_i}, \sigma_{ops}$, unfolding $P(e)$ in $G$) $\rightsquigarrow$
    $\sigma_{temp}$ := freshState()
    $v := \sigma_{ops}$.eval($e$)
    return exhale_ext($\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{temp}, P(e)$) ++
        $\sigma_{temp}$.unfold($P, v$) ++
        exec($\overline{\sigma_i}, \sigma_{ops} \uplus \sigma_{temp}, G$)

exec($\overline{\sigma_i}, \sigma_{ops}$, applying $A \mathrel{-\!\!*} B$ in $G$) $\rightsquigarrow$
    $\sigma_{temp}$ := freshState()
    return exhale_ext($\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{temp}, A * (A \mathrel{-\!\!*} B)$) ++
        $\sigma_{temp}$.apply($A \mathrel{-\!\!*} B$) ++
        exec($\overline{\sigma_i}, \sigma_{ops} \uplus \sigma_{temp}, G$)

exec($\overline{\sigma_i}, \sigma_{ops}$, packaging $A \mathrel{-\!\!*} G_1$ in $G_2$) $\rightsquigarrow$
    $\sigma_{temp}$ := freshState()
    $\sigma_A$ := freshState()
    $\sigma_{temp}$.inhale($A$)
    return exec($\overline{\sigma_i} \cdot \sigma_{ops} \cdot \sigma_A, \sigma_{temp}, G_1$) ++
        $\sigma_{ops}$.addWand($A \mathrel{-\!\!*}$ nested($G_1$)) ++
        exec($\overline{\sigma_i}, \sigma_{ops}, G_2$)

exec($\overline{\sigma_i}, \sigma_{ops}, A$) $\rightsquigarrow$
    $\sigma_{temp}$ := freshState()
    return exhale_ext($\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{temp}, A$)
```

Figure 1.3: The old implementation of magic wands in Carbon based on ECOOP15' Paper. In the `package` method, `nested(G)` means the assertion in the right-hand side without the ghost operations.

Chapter 2

Design

In this chapter, we describe our design for supporting the generalized magic wand. The final design we implemented is in Section 2.3. However, it is not easy to explain the final design alone. Therefore, we decided to present the intermediate designs that act as transitions to reach the final design. These intermediate designs are not actually implemented but they show how we reached the final design that we implemented. We will present 3 different designs, each one will be a modification on the previous design.

2.1 Design 1: Straightforward Design

The first design is the straightforward extension to Figure 1.3. Design 1 is shown in Figure 2.1. Comparing the two designs, one can find that the `exec` method in the older design is now separated into two different methods: the `execProof` method which is responsible for translating the ghost operations (denoted by *proof script*) and the `execRhs` method which translates the right-hand side assertion of the magic wand. The *proof script* of the magic wand is now translated in an iterative manner using a `for` loop which is more intuitive as the *proof script* is a sequence of statements instead of a nested assertion. Finally, the generalized magic wand supports more ghost operations than Figure 1.3, we showed one extra operation which is the `exhale` operation as an example of these extra operations. It is rather similar to older ghost operations such as the `fold`. Consider translating an `exhale` statement such as `exhale acc(x.f) && x.f == 2` as a part of the *proof script*. First, a fresh σ_{temp} state is created. Then, the permission `acc(x.f)` is transferred to σ_{temp}. Next, the assertion gets exhaled in σ_{temp} by removing permission to `acc(x.f)` and asserting that `x.f == 2`. Finally, everything in σ_{temp} is added to σ_{ops}.

Issue with Design 1

In this design, we have a separate case for every different statement that can be written inside a package statement, e.g., `fold`, `exhale`. Having different cases for every statement was acceptable in the old magic wand as there is only 4 different ghost operations. However, the generalized magic wand allows writing more arbitrary statements inside the proof script; therefore, having a separate case for each statement makes the code harder to maintain. Furthermore, consider we want to change the implementation of some operation (e.g., `fold`)

```
$\sigma$.package($A \mathrel{-\!\!*} B$, proofscript) $\leadsto$
    $\sigma_{ops}$ := freshState()
    $\sigma_A$ := freshState()
    $\sigma_A$.inhale($A$)
    return execProof($\sigma \cdot \sigma_A, \sigma_{temp}$, proofscript) ++
        execRhs($\sigma \cdot \sigma_A, \sigma_{ops}, B$) ++
        $\sigma$.addWand($A \mathrel{-\!\!*} B$)

execProof($\overline{\sigma_i}, \sigma_{ops}$, proofscript) $\leadsto$
    returnStmt  := emptyStatement()
    for(stmt:  proofscript)
        $\sigma_{temp}$ := freshState()
        if(stmt == fold P(e))
            $v$ := $\sigma_{ops}$.eval($e$)
            returnStmt := returnStmt ++ exhale_ext($\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{temp}$, Body($P$)[param $\mapsto v$])
            returnStmt := returnStmt ++ $\sigma_{temp}$.fold($P, v$)
            returnStmt := returnStmt ++ $\sigma_{ops}$ := $\sigma_{ops} \uplus \sigma_{temp}$
        else if (stmt == exhale (e))
            $v$ := $\sigma_{ops}$.eval($e$)
            returnStmt := returnStmt ++ exhale_ext($\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{temp}, e$)
            returnStmt := returnStmt ++ $\sigma_{temp}$.exhale($v$)
            returnStmt := returnStmt ++ $\sigma_{ops}$ := $\sigma_{ops} \uplus \sigma_{temp}$
        ...
    return $returnStmt$

execRhs($\overline{\sigma_i}, \sigma_{ops}, A$) $\leadsto$
    $\sigma_{temp}$ := freshState()
    return exhale_ext($\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{temp}, A$)
```

Figure 2.1: During packaging a wand we will execute the proof script in the body of package statement first, then we handle the rhs of the wand. execRhs is the same as the implementation in [9]. In execProof we iterate through the body and behave according to the type of the statement to be executed. Here only fold and exhale are shown as two examples of a modification for the old implementation, and adding a new operation.

in the future. These changes will normally take place in the method responsible for translating the `fold` statement but we also need to change the way that case `fold` is treated inside the `execProof` method. Thus, changes to some feature will require making the modifications twice. Similarly, adding a new feature to Carbon will require handling this new feature outside the package statement and handling a special case for it inside the `execProof` method as well. This special treatment for every case during a package statement makes the code more error-prone, harder to maintain and harder to scale to new operations. Therefore, we need another design that avoids treating each operation inside the *proof script* as a special case which was our motivation for developing our second design.

2.2 Design 2: More Well-engineered Design

The main idea behind the second design is that almost every Viper statement can be translated as a sequence of `exhale` and `inhale` statements (the `fold` case is explained in Subsection 1.2). We will show that it is enough to handle these inhales and exhales (along with expressions evaluation as we will see later) and the other operations will not need special treatment anymore.

Overview

We aspire to have an implementation for `execProof` method such as Figure 2.2. In Figure 2.2, all statements that are written inside the *proof script* of the magic wand are handled in the same way, by calling **translateStmt**`(stmt)` method. The **translateStmt** method is the method responsible for translating different statements outside the package statements as well. Note that this way if we want to modify the implementation of some operation in the future or support a new feature, we do not need to change the **execProof** method which solves the issue of the previous design.

```
execProof($\overline{\sigma_i}, \sigma_{ops}$, proofscript) $\rightsquigarrow$
    returnStatement := emptyStatement()
    $for(stmt : $proofscript$)$
        $\sigma_{temp}$ := freshState()
        returnStatement := returnStatement ++ $transalteStmt(stmt)$
```

Figure 2.2: The special treatment for different cases is removed from the **execProof** method.

Before moving to the implementation of **exhale**, **inhale** and **evaluate** methods, Figure 2.3 shows how a `fold` ghost operation is translated according to this design. The **execProof** method calls the **translateStmt** method, which in turn finds out that the statement being translated is a `fold` statement; therefore, it calls the **fold** method that is responsible for translating a `fold` statement in Carbon. The **fold** method then calls the methods responsible for exhaling, inhaling and evaluating expressions during a `package` statement resulting in the correct translation for the `fold` statement. Instead of having new methods **exhale_inWand**, **inhale_inWand** and **evaluate_inWand**, we decided to modify the existing **exhale**, **inhale** and **eval** methods (which are responsible for exhaling, inhaling and evaluating expressions outside a package statement respectively) to be parameterized with an extra `boolean` variable, denoted by `inWand`, to express whether the current statement is being translated during a `package` statement or not. Let us now see how to modify the **exhale**, **inhale** and **eval** methods to ensure the correct translation of the ghost operations during packaging a wand.

Exhale

How the `exhale` should behave when being translated during a `package` statement is already mentioned in Section 2.1 and its implementation is shown in the

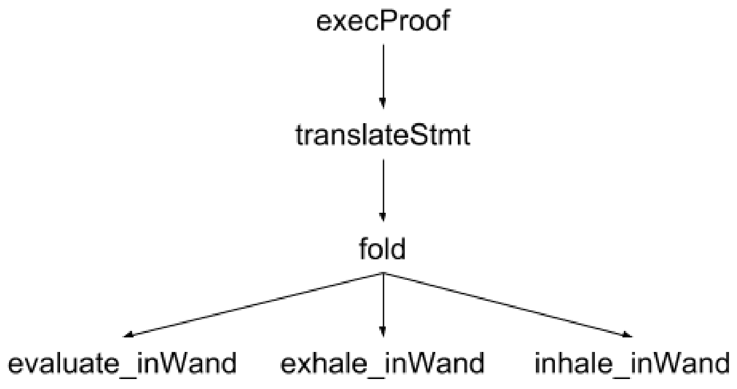

Figure 2.3: Translation of a `fold` ghost operation according to the second design

`exhale` case in Figure 2.1. We moved the implementation to the **exhale** method instead of `execProof` and added the `inWand` boolean variable as shown in Figure 2.4.

```
$\sigma$.exhale($e$, $inWand$) $\rightsquigarrow$
        $if(inWand)$
            $v := \sigma_{ops}$.eval($e$)
            returnStatement := exhale_ext($\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{temp}, e$) ++
                                   $\sigma_{temp}$.exhale($v, inWand := false$)
            $\sigma_{ops}$ := $\sigma_{ops} \uplus \sigma_{temp}$
            return $returnStatement$
        $else$
            $//$ $perform$ $exhale$ $normally$
```

Figure 2.4: Exhale method modified to handle exhaling while packaging a magic wand. If inWand is true then this means we are currently inside package statment, so we need to transfer the needed permissions from $\overline{\sigma_i}$ to σ_{temp} and then exhales it from σ_{temp}.

Inhale

The main difference between inhaling a statement during packaging a wand and outside it is how we treat assumes as mentioned in Section 1.3. We modify the inhale method as in Figure 2.5.

```
$\sigma$.inhale($e$, $inWand$) $\rightsquigarrow$
        $if(inWand)$
            $v$ := $\sigma_{ops}$.eval($e$)
            returnStatement := **exchangeAssumsWithBool**($\sigma_{temp}$.inhale($e$), $\sigma_{temp}.boolvar$)
            $\sigma_{ops}$ := $\sigma_{ops} \uplus \sigma_{temp}$
            return $returnStatement$
        $else$
            // $perform\ inhale\ normally$
```

Figure 2.5: While inhaling inside a package statement, we inhale to σ_{temp}, and every assume statement is added as a conjunction to σ_{temp}.boolvar. So for example *assume e* becomes σ_{temp}.boolvar `:=` σ_{temp}.boolvar && e

Evaluate

In carbon evaluating expressions means translating the Viper expression to its corresponding Boogie expression. For instance, the expression `x.f` is translated as `Heap[x, f]`. While packaging a magic wand, there are more than one state (e.g, σ_A and σ_{curr}); therefore, the **eval** method (responsible for evaluating expressions) needs to decide the correct state in which the evaluation takes place. To determine which states should be used, consider the example in Listing 11. Permissions to both `x.f` and `y.f` along with assumption about their values are gained via the `inhale` statements in lines 5 and 6. The magic wand in line 7 is then packaged, the first ghost operation will transfer the permission to `y.f` from σ_{curr} to σ_{ops} (as `assert` statements, unlike `exhale`, transfer the permission only without exhaling it). Line 9 is then executed, transferring the permission to `x.f` to σ_{temp}. By the time the verifier checks for the assertion `y.f == 2 && x.f == 3`, the permission to `x.f` is in σ_{temp} while the permission to `y.f` is in σ_{ops}; therefore, one state is not sufficient to evaluate the expressions. Instead, the expressions should be evaluated in a union state, denoted by σ_{union}, that carries the summation of the permissions of both states along with their heap values. The $\uplus$ operator here means the union of two state, which results in a state in which each location has permission equivalent to the summation of its permissions in the two input states. For each of the input states, the resulting union state agrees on the heap values of the locations which have positive permission in this input state.
Figure 2.6 shows the implementation for **eval** method, where the σ_{union} is first computed and then used to evaluate the expression *e*.

Issue with Design 2

The second design solves the issues with the previous one as the modification to different operations or adding new features need to be added in one place now. However, this second design makes heavy use of the *union* operator, $\uplus$, as it is used in **exhale**, **inhale** and **eval** methods which are used for translation of almost every statement. Thus, every statement translated inside the *proof script* uses the *union* operator at least once. The way this *union* operator

```
field f: Int

method foo(x: Ref, y: Ref)
{
        inhale acc(x.f) && x.f == 3
        inhale acc(y.f) && y.f == 2
        package true --* true {
                assert acc(y.f)
                exhale acc(x.f) && y.f == 2 && x.f == 3
        }
}
```

Listing 11: Viper example to show which states are needed while evaluating an expression during packaging a wand.

```
$eval(e, inWand) \rightsquigarrow$
        $if(inWand)$
            $\sigma_{union}$ := $\sigma_{ops} \uplus \sigma_{temp}$
            return ($\sigma_{union}$.eval($e$))
        $else$
            $//$ $perform\ eval\ normally$
```

Figure 2.6: Expression evaluation takes place in the union of σ_{temp} and σ_{ops} together.

is encoded in Boogie (Listing 12) makes the *union* an expensive operation. It calls the `sumMask` function, which sums the permission values on both masks for every heap location, then calls `IdenticalOnKnownLocations` twice and each of these calls quantifies over all the locations on both heaps which is expensive. The motivation behind the third and final design is to make less use of this *union* operator to improve the performance of Carbon.

```
assume sumMask(UnionMask, Mask1, Mask2);
assume IdenticalOnKnownLocations(Heap1, UnionHeap, Mask1);
assume IdenticalOnKnownLocations(Heap2, UnionHeap, Mask2);
```

Listing 12: The boogie code for computing the union state (HeapUnion, MaskUnion) of two states: State1 (Heap1, Mask1) and State2 (Heap2, Mask2)

2.3 Design 3: More efficient version

We are going to show how to get rid of the *union* operator from both **inhale** and **exhale** methods and making less use of it in case of **eval** method as it is not clear for us how to get rid of it completely.

Inhale

Getting rid of the union operator inside the inhale method is pretty straightforward. In the implementation discussed so far, we inhale the expressions into an empty state and then union this state with σ_{ops}. Instead of moving the permissions twice; first to σ_{temp} and then to σ_{ops}, using the *union*, we can inhale directly into σ_{ops} without the need to do the *union*.

Exhale

Let us use the code in Listing 13 to see how the previous design handles the exhales and see how to get rid of the *union*. In Listing 13, `P(x)` is a predicate with a permission to a single field `x.f`. Let us trace the first few steps of packaging the wand in line 7. Initially, a left-hand side state, σ_A, is created and the left-hand side assertion is inhaled in this state. The `fold` operation in line 8 is then executed. A new σ_{temp} will be created and then the body of the predicate will be exhaled according to the **exhale** method in Figure 2.4. The permission `acc(x.f)` will be transferred from σ_A to σ_{temp} and then exhaled from σ_{temp} and after that the union of σ_{temp} and σ_{ops} is computed and stored in σ_{ops}. However; in this case, when the union takes place the σ_{temp} state is empty as `acc(x.f)` is exhaled from it before the union. This is not a coincidence, every time the `exhale` method is called during a package statement, some expression `e` gets transferred to σ_{temp} and then exhaled from it before the union. Thus, when the *union* operation is done, σ_{temp} is always empty; which means that the *union* operation is not even necessary.

```
1  predicate P(x: Ref){
2          acc(x.f)
3  }
4
5  method foo(x: Ref)
6  {
7          package acc(x.f) && x.f == 2 --* P(x) && unfolding P(x) in
8                          x.f == 2 {
9                  fold P(x)
10         }
11 }
```

Listing 13: Viper example that uses framing across fold-unfold to verify the right-hand side assertion

Let us trace our example again but this time we will remove the *union* operation in the **exhale** method and see if it works. The first few steps of the tracing are the same as we did, the left-hand side is inhaled in σ_A, and the permission `acc(x.f)` get transferred to σ_{temp} and exhaled from it but now we do not perform the union. To continue the fold operation the predicate `P(x)` is then inhaled in σ_{ops}. The right-hand side assertion is then checked, `P(x)` is present in σ_{ops} and can be transferred but by unfolding the predicate the verifier fails to prove that `x.f == 2` as it only inhaled the predicate and never learned anything about the value of `x.f` in σ_{ops}. The missing information here is that the value `x.f` should be framed across the fold-unfolding statements because the `exhale` and `inhale` of the `fold` are done in different states (`exhale` in σ_{temp} and `inhale` in σ_{ops}). We need to transfer the values on the σ_{temp} heap to the σ_{ops} heap whenever the `exhale` is not expected to havoc the σ_{temp} heap afterwards. We achieved this problem by equating the heaps for σ_{ops} and σ_{temp} whenever we do not want to havoc the heap after the `exhale` which is the case in the translation of `fold` and `unfold`. Adding this equality at the end of translation of the `exhale` gives the verifier the information that `x.f` on σ_{ops} heap is equal to its value on σ_{temp} which equals 2. Therefore, the verifier can now successfully verify the right-hand side of the wand and package the wand. This equality solves the problem here but before adding it to our design, let us argue its soundness. Equating the two heaps means that all the fields on the two heaps have equal values. We will fields the values on σ_{temp} heap into two types: fields that have specific values such as `x.f` in our example, and fields that can have arbitrary values (about which the verifier learned nothing). For the fields that can have arbitrary values, equating these fields to the corresponding fields on σ_{ops} is sound as it will never contradict with the values on σ_{ops} and Carbon will never learn more information about them in the future (as other operations will have their own σ_{temp}) so they will not affect σ_{ops} later on. Regarding the fields that the verifier knows their value, these are the values that we do not want to havoc and we want to transfer them to σ_{ops} (`x.f` in this case); therefore, the effect of equating them to the corresponding field on σ_{ops} is the exact effect that we want. One final note, because the σ_{temp} for a certain `exhale` is not referred to again after this `exhale` is finished, then removing the σ_{temp}.`exhale(v, inWand := false)` from Figure 2.4 will not affect the program result. To understand this more, one can retrace the example in Listing 13 but as this tracing is similar to what we have done above, we chose to leave it to the reader.

Eval

As explained in Section 2.2, within the same statement some expression may need to be evaluated in σ_{temp} and others in σ_{ops}; therefore, it is not clear how to totally remove the *union* operation from the **eval** method. However, in this final design, we only used the *union* operation inside the **eval** method when it is necessary. One way to understand how we handle the *union* operations is to think of it as caching. We keep a version of the last valid state that represents the union of σ_{ops} and σ_{temp}. We denote this union state as σ_{union}. We use the σ_{union} state to evaluate expression during packaging a wand and we update it only when necessary. The question now is when it is necessary to update σ_{union}. For each statement, σ_{union} is assigned to σ_{ops} as initially the σ_{temp} state is empty. The σ_{union} is then updated whenever the σ_{temp} is updated (e.g, permissions get transferred to it) because only then the σ_{union} is changed. This way the *union* operator is not used in evaluating every expression, for instance, if an inhale statement is being translated, nothing is added to σ_{temp}; thus, the union between σ_{temp} and σ_{ops} is never computed. Figure 2.7 shows the final design that we implemented for supporting the generalized magic wands in Carbon.

```
execProof($\overline{\sigma_i}, \sigma_{ops}$, proofscript) $\rightsquigarrow$
    returnStatement := emptyStatement()
    $for(stmt :$ proofscript)
            $\sigma_{temp}$ := freshState()
            $\sigma_{union}$ := $sops$
            returnStatement := returnStatement ++ $transalteStmt(stmt, inWand$ := $true)$
    return $returnStatement$

inhale($\sigma, e,$ $inWand$) $\rightsquigarrow$
        $if(inWand)$
            $v := e$.eval($inWand$ := $true$)
            return $exchangeAssumsWithBool(\sigma_{ops}$.inhale($e$)$, \sigma_{ops}.boolvar)$
        $else$
            $//$ $perform$ $inhale$ $normally$

$\sigma$.exhale($e$, $inWand$) $\rightsquigarrow$
        $if(inWand)$
            $returnStatement := emptyStatement()$
            $v := \sigma_{ops}$.eval($e$)
            $returnStatement := returnStatement ++$ exhale_ext($\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{temp}, e$)
            $if(!havoc_{heap})$
                $returnStatement := returnStatement ++$
                    exchangeAssumsWithBool(assume ($\sigma_{ops}.heap$ == $\sigma_{temp}.heap$) , $\sigma_{ops}$.boolvar)
            return $returnStatement$
        $else$
            $//perform$ $exhale$ $normally$
$eval(e,$ $inWand)$ $\rightsquigarrow$
        $if(inWand)$
            return ($\sigma_{union}$.eval($e$))
        $else$
            $//$ $perform$ $eval$ $normally$
```

Figure 2.7: The final design for supporting generalized magic wands in Carbon.

Chapter 3

Framing

We were concerned so far with handling permissions and how to compute the footprint of a magic wand. In this section, we are more concerned with the values of the heap location rather than the permissions to these locations. One way to preserve information about fields in the footprint of the magic wand is to add this information to the assertion on the right-hand side. Consider the example in Listing 14.

```
1  field f: Int;
2
3  method wand_framing(x: Ref)
4  requires acc(x.f) && x.f == 3
5  {
6          package true --* acc(x.f) && x.f == 3
7          apply true --* acc(x.f) && x.f == 3
8          assert x.f == 3
9  }
```

Listing 14: Viper example that asserts the value for `x.f` on the right-hand side of the wand

When the wand is being packaged, the permission to `acc(x.f)` is successfully added to the footprint of the wand and the right-hand side assertion holds; thus, the wand is successfully packaged. When the wand is applied in line 7 the left-hand side of the wand is exhaled and then the right-hand side is inhaled. Therefore, the current state gains permission to `acc(x.f)` along with the information that `x.f == 3`, which makes the verifier able to prove the assertion in line 8. By adding the assertion `x.f == 3` to the right-hand side, we were able to preserve the information about the value of `x.f` when it is added to the footprint of the wand. Suppose we now modify this example as shown in Listing 15 Similar to the example in Listing 14, the permission to `acc(x.f)` is removed from the current state and added to the footprint of the magic wand. The right-hand side assertion is then checked and it holds as `acc(x.f)` is now in the footprint of the magic wand and the wand is successfully packaged. When the wand is applied in line 6, the left-hand side is exhaled and the right-hand side is inhaled but this time only `acc(x.f)` is inhaled (with no assumptions about `x.f`). As the wands is treated as opaque resources once

```
1 field f: Int;
2 method wand_framing(x: Ref)
3 requires acc(x.f) && x.f == 3
4 {
5         package true --* acc(x.f)
6         apply true --* acc(x.f)
7         assert x.f == 3
8 }
```

Listing 15: Viper example that needs framing the *footprint* of the magic wand.

they are packaged (the verifier knows no information about the footprint as discussed in Section 1.3), the verifier has no way to learn that `x.f` came from a state where its value equals 3 and that it should have the same value when the wand is applied. From the point of view of the state before packaging the wand, the permission to `acc(x.f)` is lost to the footprint of the magic wand; therefore, value to the field `x.f` gets havoced. Consequently, the assertion in line 7 fails.
We explored many different options to support framing in magic wands. We decided to separate framing support in magic wands into two different cases: when the magic wand is packaged and applied in the same state (Section 3.1) and when the magic wand is inhaled in a different state other than the state in which it is packaged (Section 3.2). The reason handle each case differently is that each approach is useful for one case but hard to implement in the other. Finally, in Section 3.3, we discuss the limitations of these two approaches and provide some suggestions that might act as solutions but we did not have enough time to explore thoroughly.

3.1 Packaging a wand

We will use the example in Listing 15 to explain our approach to support framing when the magic wand is packaged and applied in the same state. In this case, the verifier should keep the values on the heap for these locations whose permissions belong to the footprint unchanged until the wand is given away (through an `exhale` or `apply` statements for instance). In order to keep these fields unchanged, we should modify the verifier to be able to: 1) know which locations on the heap are framed by the wands currently held. 2) keep these framed values unchanged when havocing the heap. In order to identify the locations that are framed by a certain wand, we have a map for each magic wand instance that maps different heap locations to boolean values. This is implemented as a Boogie map which maps a certain reference and field (similar to the state `Mask`) to `true` if this field belongs to the footprint of the magic wand and to `false` otherwise. This map is called *secondary mask* and the concept of secondary masks is explained in [2]. We will denote to the secondary mask for a certain magic wand instance, `w`, by `w_sm`. The secondary masks for different magic wand instances are kept on the `Heap`, the secondary mask `w_sm` is kept in `Heap[null, w_sm]`. The `null` means that this is not a field of some reference (as the case with predicates), the `w_sm` here means the Boogie translation of the secondary mask and `Heap[null, w_sm]` returns a Boogie map from a reference and a field to a boolean value, which represents the secondary mask for wand `w`. The value of a certain field is then assigned to `true` on the secondary mask whenever it is added to the footprint of the magic wand. Let us see how this works on the code in Listing 15. We will denote to the wand in line 5 by `w` and its secondary mask by `w_sm`. When the wand `w` is packaged, the permission `acc(x.f)` is transferred from σ_{curr} to the footprint of the magic wand. Thus, `Heap[null, w_sm][x,f]` is assigned to `true`. Notice that `Heap[null, w_sm]` represents the secondary mask of the wand which is a map; therefore, `Heap[null, w_sm][x,f]` is the value of the key `[x,f]` on this map. The wand is then successfully packaged and the verifier knows that the values `x.f` should remain unchanged as long as it holds a permission to the magic wand `w`. What remains is how this information in the secondary mask can be used to prevent havocing the framed values in the secondary mask. The way havocing the heap is shown in Listing 6 by havocing a temporary heap `ExhaleHeap` and then transfer the values of locations to which the `Mask` still have permission through the `IdenticalOnKnownLocations` function. We added the suitable axioms to the definition of `IdenticalOnKnownLocations` to ensure that it copies the value of locations inside the secondary masks of the held magic wands as well. We chose not to provide the implementation details for these axioms as they contain much technical details and are not essential for our discussion. In Listing 15 after the wand is packaged, the value for `x.f` is then framed until the wand is applied in line 6 and the permission to `x.f` is inhaled; therefore, the assertion in line 7 holds.

3.2 Inhaling a wand

In this section, we will discuss the case where the wand is inhaled in a state other than the state in which it is packaged. When a wand is inhaled, it is not clear how to construct its secondary mask as the construction for the secondary mask is done during packaging the wand (which is never done in the current state in which the wand is inhaled). Thus, the previous approach that uses the secondary mask is not useful in this case. First, we will explain how can one refer to the locations whose permissions are kept in the footprint of a magic wand. Then, we will explain how to keep the assumption made about these locations to be used later when the wand is applied. Finally, we will give an example to explain the discussed approach.

Applying

To be able to refer to fields whose permissions are kept in the footprint of the magic wand, one can use `applying` expression. The `applying` (similar to `unfolding` to predicates) expression applies the magic wand temporarily to then takes hold of the wand again. When the wand is applied temporarily, the right-hand side of the wand (including its permissions) is inhaled temporarily and one can add assumption or read the heap values of the locations that are stored in the footprint of the magic wand. For instance, assume we inhaled a magic wand w, where w is `acc(x.f) --* acc(x.f) && acc(x.g)`. Without applying the wand the permission for `x.g` is in the footprint of th magic wand and cannot be referred to in the current state. If one wants to add an assumption about the value of `x.g` such as assuming that x.g equals 2, one can write `assume applying (acc(x.f) --* acc(x.f) && acc(x.g)) in (x.g == 2)`.

Snapshots

`Applying` expressions allows us to refer to heap values that belong to the footprint of a magic wand. We need to be able to use these assumptions when the same magic wand is applied again. For instance, in Listing 16, lines 7 and 8 inhale the wand and add the assumption about `x.g` as discussed in the previous subsection. When the wand is then applied in line 10, the verifier should be able to prove that the value of `x.g` equals 2 from the assumption in line 8 and consequently the assertion in line 11 should hold. In order to keep the

```
1  field f: Int
2  field g: Int
3
4  method foo(x: Ref)
5  requires acc(x.f)
6  {
7          inhale acc(x.f) --* acc(x.f) && acc(x.g)
8          assume applying (acc(x.f) --* acc(x.f) && acc(x.g)) in
9                  (x.g == 2)
10         apply acc(x.f) --* acc(x.f) && acc(x.g)
11         assert x.g == 2
12 }
```

Listing 16: Viper code where `applying` expression is used to add assumptions about the right-hand side of the magic wand.

assumptions about the right-hand side of the wand, we supported *snapshots* [10] for magic wands in Carbon. Snapshots can be thought of as a summary for the heap values kept by a field permission, a predicate or a magic wand. The snapshot of a single field is represented by a singleton with its value, e.g., snapshot for `x.f` whose value equals 3 is `(3)`. The snapshot for a predicate is represented by a tuple depending on the permission in its body: snapshot for predicate `P(x)` whose body is `acc(x.f) && acc(x.g)` and the values for `x.f` and `x.g` are 2 and 3 respectively is the pair `(2, 3)`. If the values are not known it can be represented by an unknown constant; for example, if in the previous example, the value of `x.f` is 2 and the value of `x.g` is unknown, the snapshot for the predicate `P(x)` is `(2, K)`, where `K` is a constant representing the value for `x.g` at the moment the predicate was folded. The snapshot of a magic wand is a function rather than a tuple. It is a function that maps left-hand side snapshot, denoted by $\mathtt{S}_{lhs}$ to the right-hand side, denoted by $\mathtt{S}_{rhs}$ snapshot. Both the left-hand and the right-hand sides of the wand are represented by tuples as their snapshots. However, the right-hand side snapshot itself is a function of the left-hand side snapshot. Let us compute the snapshot for the wand in Listing 16, `acc(x.f) --* acc(x.f) && acc(x.g)`, while being packaged. As the left-hand side of the wand is not known until the wand is applied (a future state), the value to `x.f` is unknown during packaging the wand. The snapshot for the left-hand side is then `(s)`. The right-hand side snapshot is a pair as it has permission to `x.f` and `x.g`, the value for `x.f` comes from the left-hand side of the wand; therefore, it has the same snapshot as the snapshot for `x.f` in the left-hand side (which is `(s)`). At the moment of packaging the wand the value for `x.g` is also unknown but it is some constant value that is not dependent on the left-hand side; thus, the snapshot for the right-hand side is `(s, K)`. It is important to understand that `s` is a variable that can take any value that will be in the left-hand side state at the moment of applying the wand, and that `K` is a constant that represents the value of `x.g` when the wand is packaged and is not changed by applying different left-hand sides. The snapshot for the whole wand is the function from the left-hand side to the right-hand side: λ `s.(s, 2)`. When the assumption in line 8 is added, the verifier learns the information that `K == 2`. When the wand is actually applied in line 10, the verifiers knows that `x.g == K` and also that `K == 2`; therefore, it can now prove the assertion `x.g == 2` in line 11. In Listing 16, the wand is packaged in the same state and not inhaled but the argument is rather similar when the wand is inhaled. The snapshot for the left-hand side is some variable $\mathtt{s}_{lhs}$ and the snapshot for the right-hand side is some pair $\mathtt{s}_1$`,` $\mathtt{s}_2$. Note that the correspondence between the `x.f` in the left-hand side and `x.f` in the right-hand side was clear when the ghost operations for packaging the wand was present (by tracing the ghost operations it is easy to find which permission comes from the left-hand side) but that is not the case when the wand is inhaled without having the ghost operations used to package the wand; therefore, the snapshot of the right-hand side is unknown variables that may or may not be dependent on the left-hand side. Line 8 adds the assumption that s_2 `== 2` and when the wand is applied in line 10 the verifier knows that `x.g ==` $\mathtt{s}_2$ and $\mathtt{s}_2$ `== 2` and can now prove the assertion in line 11.

More realistic example

The example we have given in the previous subsection is a simple example to explain the concept of snapshots of magic wands. Let us now consider a more realistic and complex example in Listing 17 where the `applying` expression and the magic wands snapshots are useful. This example is rather similar to the one in Listing 8 for traversing a linked list. Notice that the function `sorted` is also omitted as the implementation itself is not important for our discussion: it checks the sortedness of the linked list. The difference between this example and that in figure 17 is that the method traverse requires the input list to be sorted and ensures that after the method finishes, the list will remain sorted. The only line we need to introduce to the code in order to achieve this is line 18, which shows how `applying` is useful in simplifying the encoding of complex examples. If we want to verify this example without the use of `applying`, we need to add `sorted(x)` to the right-hand side of the wand, which will require adding more specifications to the left-hand side of the wand. A similar example where `applying` is not used is shown in Appendix A. To explain how our implementation handles this example, we need to explain three parts: 1) how to establish the loop invariant in line 18 before the first iteration. 2) how to reestablish this loop invariant after each iteration. 3) how this loop invariant proves the postcondition in line 10. Note that the loop

```
 1 field value: Int
 2 field next: Ref
 3
 4 predicate list(x: Ref){
 5         acc(x.value) && acc(x.next) && (x.next != null ==> list(x.next))
 6 }
 7
 8 method traverse(x: Ref)
 9 requires list(x) && sorted(x)
10 ensures list(x) && sorted(x)
11 {
12         var cur: Ref := x
13         package list(cur) --* list(x)
14
15         while(unfolding list(cur) in cur.next != null)
16         invariant list(cur)
17         invariant list(cur) --* list(x)
18         invariant applying (list(cur) --* list(x)) in sorted(x)
19         {
20                 unfold list(cur)
21
22                 // do something with cur.value
23
24                 var prev: Ref := cur
25                 cur := cur.next
26
27                 package list(cur) --* list(x){
28                         fold list(prev)
29                         apply list(prev) --* list(x)
30                 }
31         }
32
33         apply list(cur) --* list(x)
34 }
```

Listing 17: Viper example for traversing a sorted linked list and needs framing the footprint of the magic wand to verify.

invariant of interest is that in line 18, so in our argument whenever we refer to a loop invariant, it is the invariant in line 18 unless otherwise stated. The establishment of the loop invariant before the first iteration is because of how applying a wand is implemented in Carbon: when applying a wand, the left-hand side is exhaled first, the right-hand side of the wand is inhaled and then the locations to which we do not have permissions on the heap are havoced. Thus, when the wand in line 13 is applied, `list(x)` is exhaled, then, `list(x)` is inhaled again, only then, the unknown locations are havoced. Therefore, the value for `list(x)` (represented by the predicate snapshot) is never havoced, keeping the information that it is sorted and establishing the loop invariant. The second part is to show how this loop invariant is reestablished after every iteration. Assume the execution of some arbitrary iteration, i, we need to show that the loop invariant is established for the next iteration. Note that the wand used to verify the loop invariants in lines 17 and 18 for the next iteration is the wand packaged in line 27 in the current iteration. Thus, what we need to prove is that applying the wand in line 27 for iteration i results a sorted `list(x)`. During this iteration i, we hold a magic wand $\texttt{w}_i$ inhaled from the loop invariant in line 17. As the wand is inhaled, we get fresh snapshots for both left-hand and right-hand sides of the wand. We will call the snapshot for the left-hand side and the right-hand side of this magic wand S_{lhs} and S_{rhs} respectively. Consequently, the snapshot for the magic wand $\texttt{w}_i$ is $\lambda\texttt{S}_{lhs}\texttt{.S}_{rhs}$, where we know that applying the wand with the current left-hand side snapshot results in a sorted list because of the loop invariant in line 18. Notice that the left-hand side of the wand is a predicate with 3 permissions; therefore its snapshot is some triplet, let us denote to it by $\texttt{(s}_1\texttt{, s}_2\texttt{, s}_3\texttt{)}$. The wand in line 27, denoted by $\texttt{w}_{i+1}$, is then packaged. In order to package $\texttt{w}_{i+1}$, we fold predicate `list(prev)`. As `acc(prev.value)` and `acc(prev.next)` comes from σ_{curr}, they have the same snapshots as in σ_{curr}, which are $\texttt{s}_1$ and $\texttt{s}_2$ respectively. However, as the `list(prev.next)` comes from the left-hand side state, it takes a variable snapshot `(S)`. We then apply wand $\texttt{w}_i$ in line 29 to get a right-hand side snapshot which equals to applying the snapshot function on $\texttt{(s}_1\texttt{, s}_2\texttt{, S)}$. When the loop invariant in 18 is then checked, the wand is applied with a left-hand side snapshot $\texttt{s}_3$, which means we end up with the right-hand side `list(x)` with snapshot equals to applying the snapshot function on $\texttt{(s}_1\texttt{, s}_2\texttt{, s}_3\texttt{)}$, which results in a sorted list. Thus, `sorted(x)` holds and as a result the loop invariant in line 18 holds. The final part to prove postcondition using the loop invariants is simple, as the loop invariants states that applying the magic wand stated in line 17 with the same left-hand side snapshot will result a `list(x)` where `sorted(x)` holds and that is exactly what is done in line 33. Thus, the whole example verifies with the help of snapshots.

3.3 Limitations for applying

While `applying` is useful in cases such as listing 17, the way that the snapshots is implemented limits the use of `applying` in other cases causing incompleteness such as Listing 18. In Listing 18, a magic wand is inhaled in line 7, the assumption `x.g == 2` is then added via the `applying` expression in line 9, the value for the left-hand side of the wand (`x.f`) is changed and finally, the assertion `x.g == 2` is checked. When the wand is inhaled in line 7, the snapshot for the magic wand is $\lambda\texttt{S}_{lhs}\texttt{.(S}_{rhs1}\texttt{, S}_{rhs2}\texttt{)}$. The connection between the `x.f` in both sides of the wand is not made as the ghost operations used for packaging the wand is missing; as a result, they have different snapshots (as discussed in the previous section). The value of `x.f` is then assigned to 3. When the wand is applied in line 9, the left-hand side snapshot is `(3)`; thus, the snapshot of the right-hand side of the wand at the point where $\texttt{S}_{lhs}$ `== 3` is $\texttt{(S}'_{rhs1}\texttt{, S}'_{rhs2}\texttt{)}$ and the assumption $\texttt{S}'_{rhs2}$ `== 2` is added. The left-hand side of the wand `x.f` is then assigned to 4. When the wand is applied again in line 11, the snapshot of the right-hand side is the result of the snapshot function of the wand at the point where $\texttt{S}_{lhs}$ `== 4`, which is unknown because the assumption added in line 9 was at $\texttt{S}_{lhs}$ `== 3`. Consequently, the snapshot of the right-hand side

```
1  field f: Int;
2  field g: Int
3
4  method wand_framing(x: Ref)
5  requires acc(x.f)
6  {
7          inhale acc(x.f) --* acc(x.f) && acc(x.g)
8          x.f := 3
9          assume applying (acc(x.f) --* acc(x.f) && acc(x.g)) in x.g == 2
10         x.f := 4
11         assert applying (acc(x.f) --* acc(x.f) && acc(x.g)) in x.g == 2
12 }
```

Listing 18: A Viper example that shows the incompleteness of the snapshot representation of magic wands using in Section 3.2.

is some unknown values ($\texttt{S}''_{rhs1}$, $\texttt{S}''_{rhs2}$). As a result, the assertion in line 11 fails because the verifier knows nothing about the value of ($\texttt{S}''_{rhs2}$) which is the value of `x.g` when the wand is applied with left-hand side `x.f == 4`. However, the permission `acc(x.g)` belongs to the footprint of the magic wand because it cannot come from the left-hand side state as the permissions in the left-hand side have already been fully transferred to the right-hand side in the form of `acc(x.f)`; as a result, the value for `x.g` is a constant and changing the left-hand side of the wand should not affect its value. Consequently, assuming that `x.g == 2` with any left-hand side of the wand should be sufficient to prove that for any left-hand side `x.g` is not changed. This example shows the incompleteness in this representation for snapshots for magic wands as a function.
This shows that `applying` expression is not powerful when the wand is inhaled and the left-hand side changes because it only adds assumption one point at a time (for a specific magic wand with a specific left-hand side snapshot) and changing the left-hand side gives a totally different snapshot for the right-hand side. This way of representing snapshots that we discussed cannot know what permissions belong to the footprint of the magic wand to keep the values to these fields unchanged even when applying different left-hand sides. This representation is helpful as long as the left-hand side of the wand is not changed such as in data structure traversal (Listing 17) but when we begin to modify the data structure and change the left-hand side values, the `applying` becomes less powerful.
We explored different ideas to be able to solve this incompleteness issue and verify the example in listing 18 but we did not have time to explore these ideas fully. One option is to accept that the `applying` is not useful in cases we change the left-hand side and we can introduce a new expression that can add assumptions about certain locations regardless of the left-hand side applied. This expression has the same syntax the `applying` but is more general that it can be used to add assumptions about the right-hand side snapshot regardless of the left-hand side snapshot. It is not yet clear whether this general is going to be useful in many cases or not and it is also not clear whether it is going to be more useful (such as make the code simpler) in some cases than adding these information about the right-hand side values as assertions to the right-

hand side of the wand. Another suggestion is to have a different representation for snapshots. One representation that we explored is that if we assume that we have a function that tells what goes into the footprint of the magic wand (similar to the secondary mask), we can use this to add an assumption that for all left-hand sides applied to the magic wand, the values for these locations are the same. Thus, we can now connect these locations among different `applying` statements even when the left-hand side differs. It is not clear for us how this function that tells what is in the footprint can be implemented.

Chapter 4

Conclusion

We present in this chapter the current status of implementation, other achievements that are done through the project and future work.

4.1 Status of Implementation

Currently, Carbon supports the generalized magic wands. The majority of the tests in the Silicon test suites passed. Some of the tests fail because we have left some features unsupported during a package statement such as quantified permissions as we decided to spend the time on other directions such as magic wands snapshots. Other tests fail as the current encoding for magic wands is incomplete in some cases. More information about this incompleteness can be found in [6]. Finally some tests that shows unsound behavior in Silicon passes in Carbon successfully by our implementation. These tests are related to wands snapshots as the current implementation of wands snapshots in Silicon shows unsound behavior in some cases.

4.2 Other achievements

During our project, we also made some other achievements that are not mentioned in the previous sections. We have written many Viper tests for magic wands. We also encoded more complex Viper examples for data structures that uses magic wands such as priority queue in Appendix A. We also explored different snapshots encoding designs other than the mentioned designs in Chapter 3 before deciding on the implemented design. We fixed some bugs and filed some issues. We explored some Carbon optimizations that are not wand-related such as optimizing the Boogie output code during asserting an accessibility predicate.

4.3 Future Work

We have left some features such as quantified permissions during packaging a wand unsupported. Other than supporting quantified permissions inside package statements, another extension is to support writing quantified permissions inside both sides of the magic wands and to support writing magic wands inside quantified permissions as well.

The limitation for the `applying` expression discussed in Section 3.3 is an interesting area to explore. As we mentioned in Section 3.3, one of the solutions we explored is adding a more general expression that reasons about any magic wand, regardless its left-hand side snapshot, but we did not have enough time to explore. It is not clear whether having this general expression will be useful or not and also if it provides more power over adding assertions to the right-hand side of the wand. The other solution that we mentioned is having a different representation for magic wand snapshots which is also worth more exploration.

Appendix A

Priority queue encoding without the use of snapshots

```
field val: Int
field next: Ref

predicate priorityQueue(q: Ref){
        acc(q.val) && acc(q.next) && ((q.next != null) ==> priorityQueue(q.next))
}

function sorted(q: Ref): Bool
requires priorityQueue(q)
ensures (result == true) ==> (forall i:Int :: {greaterThanOrEqual(q, i)}
         i<= (unfolding priorityQueue(q) in q.val) ==> greaterThanOrEqual(q, i))
{
        unfolding priorityQueue(q) in
        q.next == null?true:((unfolding priorityQueue(q.next) in q.val <= q.next.val)
                && sorted(q.next))
}


function greaterThanOrEqual(q: Ref, x: Int): Bool
requires priorityQueue(q)
{
        unfolding priorityQueue(q) in
        (q.val >= x && (q.next != null?greaterThanOrEqual(q.next, x):true))
}

method peek(q: Ref) returns(front: Int)
requires priorityQueue(q) && sorted(q)
ensures priorityQueue(q) && sorted(q)
ensures greaterThanOrEqual(q, front)
{
        unfold priorityQueue(q)
        front := q.val
        fold priorityQueue(q)
        }

        method poll(q: Ref) returns(front: Int, qprime: Ref)
        requires priorityQueue(q) && sorted(q)
        ensures qprime != null ==> priorityQueue(qprime)
        ensures qprime != null ==> greaterThanOrEqual(qprime, front)
        {
                front := peek(q)
                unfold priorityQueue(q)
                qprime := q.next
        }

        function getFirst(x: Ref): Int
        requires priorityQueue(x)
        {
                unfolding priorityQueue(x) in x.val
        }

        method insert(q: Ref, x: Int) returns(ret: Ref)
        requires priorityQueue(q) && sorted(q)
        ensures priorityQueue(ret) && sorted(ret)
        {
                if(x <= getFirst(q)){
                        ret := new(*)
                        ret.val := x
                        ret.next := q
                        fold priorityQueue(ret)
                }else{
                // x > getFirst(cur)
                var cur: Ref := q
                var oldCur: Int := unfolding priorityQueue(q) in q.val;
                package (priorityQueue(cur) && sorted(cur)
                        && getFirst(cur) >= oldCur) --*
                        priorityQueue(q) && sorted(q)
                while(unfolding priorityQueue(cur) in cur.next != null
                        && unfolding priorityQueue(cur.next) in cur.next.val<x)
                invariant priorityQueue(cur) && sorted(cur)
                invariant getFirst(cur) >= oldCur
                invariant x >= getFirst(cur)
                invariant priorityQueue(cur) && sorted(cur) &&
                        getFirst(cur) >= oldCur --* priorityQueue(q) && sorted(q)
                {
                        var p: Ref := cur
                        var oldP: Int := oldCur
                        unfold priorityQueue(cur)
                        cur := cur.next
                        oldCur := unfolding priorityQueue(cur) in cur.val
                        package priorityQueue(cur) && sorted(cur) &&
                                getFirst(cur) >= oldCur --*
                                priorityQueue(q) && sorted(q){
```

```
                                fold priorityQueue(p)
                                apply priorityQueue(p) && sorted(p) &&
                                        getFirst(p) >= oldP --*
                                        priorityQueue(q) && sorted(q)
                        }
                }

                var node: Ref
                node := new(val, next)

                unfold priorityQueue(cur)
                node.val := x
                assert x >= cur.val
                node.next := cur.next
                cur.next := node
                fold priorityQueue(node)
                fold priorityQueue(cur)
                apply priorityQueue(cur) && sorted(cur) &&
                        getFirst(cur) >= oldCur --* priorityQueue(q) && sorted(q)
                ret := q
        }
}
```

List of Figures

List of Listings

Bibliography

[1] Nils Becker, Alexander J Summers, and Malte Schwerhoff. Generalized verification support for magic wands.

[2] Stefan Heule, Ioannis T Kassios, Peter Müller, and Alexander J Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In *European Conference on Object-Oriented Programming*, pages 451–476. Springer, 2013.

[3] Stefan Heule and Peter Müller. *Verification condition generation for the intermediate verification language SIL*. PhD thesis, Masters thesis, Dept. of Computer Science, ETH Zurich, 2013.

[4] K Rustan M Leino. This is boogie 2. *Manuscript KRML*, 178(131), 2008.

[5] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 41–62. Springer, 2016.

[6] Gaurav Parthasarathy, Alexander J Summers, and Peter Müller. Verification condition generation for magic wands. 2015.

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

[8] Malte Schwerhoff. Symbolic execution for chalice. Master's thesis, Eidgenössische Technische Hochschule Zürich, Departement of Computer Science, Chair of Programming Methodology, 2011.

[9] Malte Schwerhoff and Alexander J Summers. *Lightweight support for magic wands in an automatic verifier*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[10] Malte H Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. PhD thesis, ETH Zurich, 2016.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

Title of work (in block letters):

Adding Generalized Magic Wand Support to a Verification Condition Generation Based Verifier

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

Name(s):	**First name(s):**
Khedr	Ahmed Gamal

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)**
Zurich, 29/08/2018	Ahmed Gamal

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