Generalized Verification Support for Magic Wands

Bachelor's Thesis

Nils Becker, nbecker@student.ethz.ch

Supervisors:
Malte Schwerhoff, malte.schwerhoff@inf.ethz.ch
Alexander J. Summers, alexander.summers@inf.ethz.ch

ETH Zurich, September 2017

Contents

1 Introduction

The Viper project [5], consisting of an intermediate language and verification tools, is being developed by the Chair of Programming Methodology at ETH Zurich. Viper programs are annotated with pre- and postconditions, as well as statements, that instruct the verifier to transform the current state of the program. Moreover, Viper provides two verifiers: Silicon, which is based on symbolic execution, and Carbon, which is based on verification condition generation.

The tools are based on separation logic, which allows access permissions to fields and data structures to be expressed and is therefore particularly useful for the verification of concurrent programs. One operation within separation logic is the magic wand, which is used to exchange its left-hand-side for its right-hand-side. Furthermore, magic wands can be used to encode information about partial data structures. Magic Wands can, for example, be used to store permissions to the first few elements of a linked list and are therefore useful when looping over a linked list, where otherwise the permissions for the part of the list that has already been looped over would be lost.

Since verification in the presence of magic wands has been shown to be undecidable, one approach to automating their verification is for the verifier to require guidance, in the form of ghost code, from the user. This ghost code instructs the verifier to transform the left-hand-side of a wand and the permissions contained in the wand (called the wand's footprint) in order to get the right-hand-side. This shows that it is sound to exchange the wand's left-hand-side for its right-hand-side, i.e. that the wand holds.

The previous implementation of magic wands in Viper, which has been developed by Schwerhoff and Summers [7], allowed some select few ghost operations to be used and automatically moved permissions into the wand's footprint. This approach did, however, limit the kinds of magic wands that could be verified. It seems in particular necessary to be able to branch during the ghost code, in order to verify certain programs (e.g. [6]).

The goal of my thesis was therefore to design a language extension for Viper, that allows a more general set of ghost operations to be used. This language extension will be introduced and discussed in chapter 2. The language extension was then implemented in the symbolic execution based verifier, Silicon, which will be discussed in chapter 3. We will afterwards turn our attention to two features for magic wands that were newly added in chapters 4 and 5. Chapter 4 deals with magic wand snapshots, which allow information about the values of fields, that was previously lost, to be preserved. Furthermore, this chapter introduces a way to temporarily apply a magic wand in order to evaluate an expression, which as we will see is particularly useful in connection with magic wand snapshots. Chapter 5 deals with the integration of magic wands with quantified permissions. This chapter consists of two parts, the first of which can be understood a sets of magic wands and the second of which allows quantified permissions (i.e. sets of permissions) to be used inside of magic wands. Lastly, chapter 6 will discuss how the new syntax solves two minor problems that the old syntax had, as well as its impact on the performance of the verifier and some other statistics.

2 New Magic Wand Syntax

The Viper language allows for separation logic assertions to be used in order to verify the correctness of programs. Separation logic assertions consist of access predicates (represented in Viper as `acc(·, p)`, where p is a vale between 0, for no permission, and 1, for full (write) permission, and intermediate values are interpreted as read permission) and boolean expressions, joined by the conjunction ($*$) and magic wand ($-\!\!*$) operators (encoded in Viper as `&&` and `--*` respectively). Furthermore, Viper assertions have to be self-framing, i.e. if `x.f` is used in an expression, the corresponding access predicate, `acc(x.f, p)` has to first occur in that assertion.

The language offers `Int` (integer), `Perm` (permissions), `Bool` (booleans) and `Ref` (similar to classes in object oriented languages) types and corresponding sequence (`Seq`) and set (`Set`) types. Fields can be declared using these types and become members of any instance of the `Ref` type. Additionally, predicates can be defined to abstract over data structures. Moreover, the language allows for methods that can have side effects and side-effect-free functions to be declared. Methods and functions can have preconditions and postconditions in the form of assertions. The body of a method can furthermore contain statements, that modify the state of the program, e.g. by assigning values to fields or by folding predicates (replacing their bodies with a representation of the predicate).

Magic wands are formally defined by $\sigma \vDash A -\!\!* B \Leftrightarrow \forall \sigma' \perp \sigma \cdot (\sigma' \vDash A \Rightarrow \sigma \uplus \sigma' \vDash B)$, where σ, σ' are program states, A, B are assertions, $\perp$ indicates that two states are compatible and can be combined by $\uplus$, i.e. that they do not require access to the same heap locations and that they agree on the values of local variables [7]. This definition can intuitively be understood such that some part σ is split off from the current state and in any future state σ', in which A holds, B can be obtained by adding σ back. The permissions that were stored in the magic wand according to the intuition given in the introduction are therefore represented by σ. These permissions are called the footprint of the magic wand. Moreover splitting off σ and proving that it entails $A -\!\!* B$ is called the package operation and obtaining B from σ and A is called the apply operation.

2.1 Package Statement

Creating a magic wand in Viper is done using the `package` statement. The package statement needs to ensure that the wand actually holds. The fact that an infinite number of potential states has to be considered for σ' gives an intuition for why this problem is hard and has in fact been proven to be undecidable in general [2]. It is, however, possible to use ghost operations to instruct the verifier as to how it can get from the left-hand-side and σ to the right-hand-side, which would prove that the wand holds. Note that the σ should only be rewritten and values of variables and fields should not be changed, i.e. the ghost operations should be side-effect-free.

Previous to my thesis Viper used an implementation that allowed for ghost operations to be included in the wand's right-hand-side during a `package` statement. This approach was however limited to a select few ghost operations (`folding`, `unfolding`, `packaging`, `applying`) and notably did not allow branching during the execution of the ghost opera-

tions, which has been found to be necessary to verify some programs. The main goal of the new syntax is, therefore, to be less restrictive and allow branching inside the ghost code.

Figures 1 and 2, where *assertion* is a separation logic assertion and *statements* is a sequence of Viper statements (with a few exclusions), as defined by the Viper language [3], show the old and new syntax of package statements respectively.

package_statement := **"package"**, *assertion*, **"--*"**, *assertion*_with_ghost_operations;

Figure 1: Old syntax of package statement.

package_statement := **"package"**, *assertion*, **"--*"**, *assertion*, **"{"**,
*statements*_as_ghost_operations, **"}"**;

Figure 2: New syntax of package statement.

As a running example the wand `acc(x.val)` $-\!\!*$ `List(x)` will be used, where `List(x)` is defined as in Figure 3, `val` is a field of type `Int` and `next` is a field of type `Ref`.

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

Figure 3: Definition of the List predicate.

In order to package this wand, the verifier needs to fold the `List` predicate as a ghost operation. Figures 4 and 5 show how this is done in the old and new syntaxes.

```
package acc(x.val) --*
    folding List(x) in List(x)
```

Figure 4: Package statement in the old syntax.

```
package acc(x.val) --* List(x) {
    fold List(x)
}
```

Figure 5: Package statement in the new syntax.

With the new syntax, the restrictiveness on the ghost code of the old syntax has been mitigated by allowing arbitrary Viper statements to be used as ghost code (in the form of a proof script). Because this includes `if-else` statements, branching is also possible. There are however still some restrictions on the statements that can be used:

- **Field assignments** and **local variable assignments** to variables that were not declared in the proof script are disallowed since magic wand applications must be side-effect free.

- **New-statements** are used to create new instances of the `Ref` type in Viper. These were disallowed for a similar reason: For the wand to be side-effect free, they must not be observed from the outside. It is therefore always possible to remove them from the proof script.
- **While-loops** are not currently supported. As a workaround, they can in many cases be wrapped into a method.

Moreover, comparing these examples, one of the advantages of the new syntax is that the distinction between the wand and the ghost operations is much clearer and easier to understand. Additionally, the language is simplified since the regular `fold` statement can be used instead of a special `folding` ghost operation. One of the potential drawbacks of the new syntax is that the annotation overhead for package statements becomes larger. Because of this I considered keeping the old syntax (or something similar) as syntactic sugar, but after looking at a number of example programs, decided against it; for small examples like the one shown above the difference is negligible, whereas complex examples, while requiring more lines of code in the new syntax, quickly become unreadable in the old syntax.

2.2 Other Changes

In addition to the new package statement, there are also a number of smaller changes to the language, mostly to simplify it:

- **Ghost operation expressions** (like `folding` above) are no longer part of the language.
- **Magic wand variables** that could previously be used to store magic wands for later reference have been removed to simplify the language.
- **Let expressions** can be used to bind the value of an expression to an identifier that can later be used (similar to local variables). With the old syntax, it was possible to use them to store intermediary values during the execution of ghost operations and use them on the right-hand-side of the wand being packaged. This is no longer possible, as it is potentially unsound. This was mostly used as a workaround for the missing magic wand snapshots, which are now implemented (see chapter 4 for more information).
- An **applying-expression** has been introduced. Similar to `unfolding` for predicates `applying` can be used to temporarily and without side-effects apply a magic wand during the evaluation of a subexpression (see section 4.4).
- **Lhs-expressions** have been replaced by labeled old expressions with the `lhs` label (`old[lhs]`) to simplify the language. These expressions can be used to reference the value of an expression that it had before the wand was applied. This type of expression can probably be removed once the snapshot calculation for wands that

are inhaled is implemented since values from the left-hand-side should always be conserved (for more information see section 4.3).

- The semantics of the **assert-statement** inside of magic wand proof scripts has been defined such that it can be used to move permissions into a magic wand's footprint. This might be necessary since only permissions that are explicitly referenced are automatically moved into the footprint (e.g. evaluating a field access, `x.val`, will fail if the necessary permissions are not already present in σ, as seen in Figure 6).

```
package true --* true {

    if (x.val == 0) //cannot be
        evaluated since acc(x.val)
         is not in σ
    {}
}
```

```
package true --* true {
    assert acc(x.val)
    if (x.val == 0) //can be
        evaluated since acc(x.val)
         is in σ
    {}
}
```

Figure 6: Illustration of how `assert` can be used to transfer permissions into the footprint

3 Implementation

The Silicon verifier [8] is based on symbolic execution. In symbolic execution, the execution of a program is simulated by replacing the values of fields and variables with symbolic values and progressively adding restrictions (called path conditions) to their values as new information is learned during the execution of statements. A symbolic state in Silicon consists of local variables and their symbolic values, a representation of the program heap, that contains the symbolic values of fields and path conditions. Figure 7 indicates the symbolic state before and after adding an assumption. `loc` indicates the symbolic values of local variables, `h` indicates the heap and `pcs` indicates the path conditions. Note that the heap also stores the amount of permissions that are available for each heap location, which is indicated by `# 1`, for full write permissions here.

```
//loc = {x = v0}, h = {v0.val = v1 # 1}, pcs = ∅
assume x.val == 0
//loc = {x = v0}, h = {v0.val = v1 # 1}, pcs = {v1 = 0}
```

Figure 7: Illustration of symbolic states.

Furthermore, when `if-else` statements are encountered in symbolic execution both branches are symbolically executed and the if condition (or the negation thereof, for the else branch) is added to the path conditions. These path conditions are then called the branch conditions. Silicon moreover relies on a SMT-solver (Z3) to check whether or not the path conditions entail certain assertions.

Silicon's internal structure consists of the `executor`, which is responsible for executing statements by modifying the symbolic state accordingly, the `consumer`, which is responsible for removing permissions from the symbolic state, the `producer`, which is responsible for adding permissions to the symbolic state and the `evaluator`, which is responsible for evaluating expressions.

We will now walk through the symbolic execution of the running example (Figure 5) and give an intuition the soundness of such an execution afterwards. Note that some steps will be added to the execution of the `package` statement in chapters 4 and 5, that are not discussed here.

```
package acc(x.val) --* List(x) {
    fold List(x)}
```

Figure 5: The running example from page 5.

3.1 Walkthrough

In Viper the footprint of a wand is automatically calculated during the symbolic execution of the `package` statement. For the footprint calculation, a stack of heaps called the reserve heaps is added to the symbolic state. We will see how this is helpful during

the walkthrough. Figures 8 and 9 provide pseudo code for the symbolic execution of a `package` statement.

```
executePackage(wand, proofScript, state) {
  var hOuter = state.heap
  state.heap = new Heap
  state.reserveHeaps.push(hOuter)
  var hLhs = new Heap
  producer.produce(wand.lhs, hLhs)
  state.reserveHeaps.push(hLhs)
  var hUsed = new Heap
  var hOps = new Heap
  state.reserveHeaps.push(hUsed)
  state.reserveHeaps.push(hOps)
  executor.executeProofScript(proofScript, state) //slightly
      modified version of the regular execute method (see below)
  consume.exhale_ext(wand.rhs, state)
  state.reserveHeaps.drop(2)
  var hOuterPrime = state.reserveHeaps.pop() + wand
  state.heap = hOuterPrime
}
```

Figure 8: Pseudo for the execution of a `package` statement.

```
exhaleExt(assertion, state) {
  for each conjunct in assertion.topLevelConjuncts {
    if(conjunct is PermissionPredicate) transfer(conjunct, state)
    else consume normally
  }
}
transfer(assertion, state) {
  var permsNeeded = assertion.permissions
  var hOps = state.reserveHeaps.top
  for each heap in state.reserveHeaps.topToBottom.skipFirst {
    var consumed = consumeUpTo(assertion, heap, permsNeeded)
    hOps += consumed
    permsNeeded -= consumed.permissions
  }
  assert(permsNeeded == 0)
}
```

Figure 9: Pseudo for `exhale_ext` and `transfer`.

Let us now walk through the symbolic execution of the running example (Figure 5). We first push *hOuter* onto the reserve heaps (ll. 2-4, Figure 8). As we have seen in chapter 2 the footprint consists of the permissions, that are split off from *hOuter*. The footprint should contain as little permissions as possible, while still entailing the magic wand. We will see that putting *hOuter* on the bottom of the reserve heaps stack will result in permissions only being split off from *hOuter* if they cannot be obtained from anywhere else. The next step is to create *hLhs*, which is a heap used to store permissions that are provided by the left-hand-side of the wand being packaged (ll. 5-7, Figure 8). To do so an empty heap is created and the `producer` is used to add permissions from the wand's left-hand-side to that heap, such that in our example the resulting *hLhs* = {acc(x.next)}. Next *hOps* and *hUsed* are initialized as empty heaps and added to the reserve heaps (ll. 8-11, Figure 8). The meaning of these two heaps will become clear in the following steps.

Next the `executor` is used to execute the proof script. The `executor` will then execute the `fold List(x)` statement, which first tells the `consumer` to consume the body of the predicate, i.e. `acc(x.next) && acc(x.val) && (x.next != null ==> List(x.next))`. During the execution of the proof script this is done using `exhale_ext`.

The consumer will split the body of the predicate into its three top-level-conjuncts and consume those separately (l. 2, Figure 9). First, it will consume `acc(x.next)` since this is a permission predicate, transfer will be used (l. 3, Figure 9). First the amount of permissions, that are needed determined (l. 8, Figure 9). Since in this case, we need full write permissions, `permsNeeded` is set to 1. Transfer then considers the reserve heaps from top to bottom skipping the first one (which would be *hOps*), it will, therefore, consider the following heaps in this order: *hOps, hLhs, hOuter* (l. 10, Figure 9).

Since at this point *hOps* is empty, `consumed` contains no permissions and `permsNeeded` stays 1 (ll. 11-13, Figure 9). The same happens for *hLhs*, so *hOuter* is reached. The necessary permissions can be found on *hOuter* and by removing them they become part of the wand's footprint, as described before. The consumed permissions are then temporarily stored in *hUsed* (l. 12, Figure 9).

Next `acc(x.val)` is consumed. This again happens via transfer, which in this case finds the necessary permissions in *hLhs*, so it does not take any permissions from *hOuter* and does not contribute to the footprint. Lastly `x.next != null ==> List(x.next)` will be consumed. Since this is not a permission predicate, the assertion will be consumed normally: the left-hand-side, i.e. `x.next != null` is evaluated by the `evaluator`, which uses $hUsed \cup hOps \cup hLhs$ to evaluate heap dependent expressions. For now let us assume that `x.next == null`. In this case, the result of the `evaluator` is simply a false literal, so the consumer has no more work to do. Now that the body of the predicate has been consumed the execution of the `fold` statement continues.

The next step is to is to add the `List` predicate to the current state's heap, which concludes the regular execution of the `fold` statement. At this point we have the current state's heap = {`List(x) # 1`}, *hUsed* = {`x.next # 1, x.val # 1`} (the body of `List`), $hOps = hLhs = \emptyset$ and $hOuter' = hOuter \setminus footprint$, where *footprint* = {`x.next # 1`} are the permissions of the wand's footprint, as calculated by the transferring consume algorithm.

We are now in a state, where the `List(x)` is on the state's main heap and *hUsed* contains the predicate's body. Exchanging the permissions on *hUsed* for the ones on the state's heap is therefore equivalent to executing the `fold` statement. executeProofScript does this by replacing *hUsed* with an empty heap and moving the permissions contained in the state's heap onto *hOps*. The meaning of *hOps* can, therefore, be understood such that it stores any permissions, that are available form the execution of previous ghost operations.

Since there is no other statement in the proof script to execute, `executePackage` continues in line 13. The right-hand-side of the wand, which in our case is `List(x)`, is now consumed using `exhale_ext`, to ensure that the proof script actually ended in a state that entails the wand's right hand side. This happens analogously to what we have seen before, where this time the necessary permissions are found in *hOps*. After that, the magic wand is added to the current state and the reserve heaps are removed (ll. 14-16, Figure 8).

3.2 Generalized Transfer

We have seen in the previous section how transfer is used during the execution of the package statement to consume permissions and automatically calculate a wand's footprint.The previous implementation of transfer used a special implementation for `consumeUpTo` (l.11, Figure 9). The downside of this approach was that any changes to the consume algorithm had to be implemented in both the regular consume method and the special one used by transfer, thus resulting in code duplication and programming overhead. Furthermore different resource types, like quantified permissions (see chapter 5) and other types that might be added in the future, require different consume algorithms to be used. To reduce these problems and make the consume implementations more uniform the implementation of transfer has been changed as part of my thesis and the consume algorithm has been changed as part of Sierra's Bachelor's thesis [9].

The basic idea is to extract parts of the original consume algorithm into consume functions. These can then be passed to transfer or used directly. Consume functions take the amount of permissions, that is needed and a heap from which they should consume those permissions and return the permissions they consumed. When they are used directly it is checked, whether they consumed a sufficient amount of permissions and the verification fails if that is not the case. For the case where they are used by transfer, it is easy to see, that they can replace `consumeUpTo` on line 11 of Figure 9. One difference between `consumeUpTo` and consume functions exists, however. Consume functions should already know the assertion, they are supposed to consume, in many cases this is done using partial application, to set the corresponding arguments of the consume function before passing it to transfer.

As part of this thesis, its implementation has been changed, so it takes a consume function that is called on each relevant heap. The consume function takes as arguments a symbolic state, the heap from which permissions should be consumed and the amount of permissions that still need to be consumed and returns a consumption result that can either indicate that all requested permissions have been consumed or the amount of

permissions that still need to be consumed, after consuming as many permissions from the provided heap as possible. Furthermore, it returns the new symbolic state, the heap, from which the consumed permissions have been removed and the chunk that has been consumed (if any).

As we have seen earlier the heap contains both the values of heap locations and the amount of permissions, that are available for that heap location. This combination of location, value and permissions is called a chunk in Silicon. If the required permissions cannot be found during a consume operation outside of a package statement, a state consolidation is triggered. The state consolidation, among other things, tries to find heap locations, that are equal and unifies their chunks. As an example consider the program shown in Figure 10.

```
//before state consolidation
//loc = {x = v0}, h = {v0.val = v2 # 1/2, v1.val = v2 # 1/2}, pcs =
    {v0 = v1}
//after state consolidation
//loc = {x = v0}, h = {v0.val = v2 # 1}, pcs = {v0 = v1}
exhale acc(x.val)
```

Figure 10: Example of a program where a state consolidation is triggered.

Silicon can only take permissions from a single chunk when trying to consume permissions. Consuming `acc(x.val)` will, therefore, fail since either one of the two chunks only contains `1/2` permissions. Since Silicon can, however, prove that $v_0 = v_1$ it can unify the two chunks, such that the consume operation succeeds on a second attempt. Since each execution of the consume function can also only consume permissions form a single chunk, i.e. only a single chunk can be considered per reserve heap, similar situations also arise inside of package statements. Previously the state consolidation was not triggered, when transfer failed, leading to incompletenesses of the verification. Since Sierra did, however, modify the state consolidation such that it also works for the reserve heaps, transfer has also been modified, such that it can trigger state consolidations.

3.3 Soundness

In their paper [7] Schwerhoff and Summers have sketched a soundness proof of the original algorithm. `transfer` and `exhale_ext` have not changed in their functionality. Therefore, the corresponding Theorems (1 and 2 in the aforementioned paper) still hold. According to the authors, Theorem 1 states that the permissions that are added to *hUsed*, by `transfer`, are exactly the ones removed from the reserve heaps and that those permissions entail the ones that should be consumed. Theorem 2 states the same about `exhale_ext`, with the difference that this time the entire assertion that was exhaled should be entailed.

Theorem 3 deals with the soundness of the symbolic execution of the `package` statement, with respect to the four ghost operations that were previously supported. The theorem again states that the permissions that are added to *hUsed* are exactly the ones

removed from the reserve heaps and that the execution of the ghost operations ends up in a state equivalent to the one that would be obtained from executing the statements, corresponding to those ghost operations. The proof of Theorem 3 utilizes the fact that the heap effects of the execution of the ghost operations can be expressed as `exhale_ext` followed by a transformation of *hOps*. Intuitively, Theorem 3 should, therefore, hold for the algorithm shown in the example above and the algorithm should be sound if the execution of a proof script statement is equivalent to such a decomposition. We argue that such a decomposition does indeed exist for any Viper statement as follows: the heap effects of any Viper statement can only consist of the removal and addition of permissions. As we have seen `exhale_ext` results in permissions being removed. Since added permissions are moved to *hOps*, the addition of permissions can be understood as a transformation of *hOps*. The effects of any Viper statement can, therefore, be expressed as an `exhale_ext` operation followed by a transformation of *hOps*[1].

Note that we consider `assert` statements to remove and add-back the same assertion). In terms of the implementation assert statements simply copy all contents of *hUsed* to the current state's heap. The effect of this is that the semantics of `assert` are to move permissions, the wand doesn't have, yet, into the footprint, as was stated in 2.2.

3.4 Branching

Let us now turn our attention to branching-behavior during the execution of the proof script and let us again start by looking at the running example. For simplicity, we previously assumed that `x.next == null`. Without this information, Silicon will branch at this point. For illustrative purposes, we will make this branching explicit by using an if-else-statement as shown in Figure 11.

```
package acc(x.val) --* List(x) {
    assert acc(x.next)
    if (x.next == null) {
        fold List(x)
    } else {
        assert List(x.next)
        fold List(x)
    }
}
```

Figure 11: Modified version of the package statement using if-else.

In this modified version of the running example assert first transfers permissions to `x.next` into the wand's footprint. This is necessary since if-conditions must be framed by the state they are evaluated in. We then encounter the if-else-statement. First, the then-branch is executed by assuming `x.next == null` and executing `fold List(x)`, as shown before. Then the end of the proof script is reached and the execution of

[1]Note that the removed/added assertions may be empty.

`executePackage` continues. As before the wand's right-hand-side is consumed and the magic wand is added to the state. In the end, the resulting state and the current branch conditions (in this case `x.next == null`) are recorded.

Next, the algorithm backtracks to the point where the if-else-statement was encountered and now executes the else-branch. To do so `!(x.next == null)` is assumed and `assert List(x.next)` is executed, resulting in `acc(List(x.next))` being added to the wand's footprint. Next `fold List(x)` is executed, this is done the same way as before, except this time the left-hand-side of the implication is true. The consumer, therefore, consumes the right-hand-side, i.e. `List(x.next)`, as previously described. After completing the fold-statement the end of the proof script is reached. Again the right-hand-side of the wand is consumed, the magic wand chunk produced and the results recorded as before (this time the branch conditions are `!(x.next == null)`).

As there are no more points to backtrack to the `packageWand` method now executes the rest of the method, in which the `package` statement occurred. Silicon does this by executing the rest of the method inside of each branch. To continue it, therefore, uses the state it recorded for each branch and restores the corresponding path conditions.

Note that the footprint of the wand is different, depending on which branch was chosen. If after a package statement the verifier can find out that one of the paths taken by the package statement is infeasible, the execution of that branch can be stopped. This allows the permissions that were added to the footprint on that branch to be used again in assertions, that would fail if they were consumed on the infeasible branch.

Now that we have introduced the new syntax and explained how `package` statements are executed, we will turn our attention to some improvements with respect to magic wands in the next two chapters.

4 Snapshots and Usage in Expressions

Silicon can only keep track of values of fields while permissions to that field exist. Therefore, as seen in Figure 12, when folding the `List` predicate from Figure 3, any information about the values of `x.next` and `x.val` would be lost. Consequently, after unfolding the predicate again it would not be possible to prove that the values of `x.next` and `x.val` have not changed.

```
//loc = {x = v0}, h = {v0.next = v1 # 1, v0.val = 0 # 1, List(v1) #
    1}, pcs = ∅
fold List(x)
//loc = {x = v0}, h = {List(v0) # 1}, pcs = ∅
unfold List(x)
//loc = {x = v0}, h = {v0.next = v2 # 1, v0.val = v3 # 1, List(v2) #
    1}, pcs = ∅
assert x.val == 0 //fails
```

Figure 12: Result of symbolic execution without snapshots.

Silicon solves this issue by using snapshots. The idea of snapshots is to store additional information inside the predicate's chunk during the fold operation that can be used to restore the original values of the fields that the predicate abstracts over. Snapshots can furthermore contain other snapshots. For example the snapshot for a linked list, as shown in Figure 3, with nodes v_0, v_1 and v_0`.val` $= 0$ and v_1`.val` $= 1$, would be `(v_1, 0, (null, 1))`. The way to interpret such a snapshot is that each element corresponds to a top-level-conjunct of the body of the predicate: v_1 is the value of `x.next`, `0` is the value of `x.val` and `(null, 1)` is the snapshot for `List(x.next)`. Note that the same predicate can have snapshots with different shapes and that if no information is available a *fresh* snapshot is used. A *fresh* snapshot is simply a symbolic value for which no information is available.

4.1 Magic Wand Snapshots

A similar issue occurs for magic wands: permissions from the footprint are given up during the package operation and permissions to the wand's left-hand-side are given up during the apply operation. Similar to how snapshots are calculated for predicates during the `fold` operation, snapshots should therefore also be calculated for magic wands during the `package` operation. The problem is however that the values occurring in the left-hand-side of the wand are not known at the time at which the package operation is executed. The intuition for a magic wand snapshot should, therefore, be a function that, when given the values from the left-hand-side, provides a corresponding snapshot for the right-hand-side. This can be done by first calculating a snapshot with placeholders for the values from the left-hand-side, which are later filled in.

In sections 5.2.2 and 5.2.3 of his Ph.D. thesis [8], Schwerhoff describes a possible strategy for calculating such snapshots. This strategy relies on the regular snapshots, as

shown before, for heap locations in the footprint and uses a *fresh* snapshot (called the *root*) during the production of the left-hand-side. It furthermore sets a flag to indicate that the generated snapshot should be defined relative to the provided root. A resulting snapshot might look like (`first:`*`root`*`, second:`*`root`*`)`, indicating that whenever additional information about *root* becomes available, the value of the first top-level-conjunct will be that of the first element of *root*. The proof script is then executed and the right-hand-side is consumed, during which a snapshot is collected, as it would be during the consumption of the body of a predicate. The collected snapshot will then include the values from the footprint and the placeholder values from the left-hand-side.

During the application of a wand, this snapshot is used to produce the right-hand-side. To replace the placeholders it then suffices to equate *root* with the snapshot of the actual left-hand-side obtained during the apply operation.

Figure 13 adds a few statements to the running example (Figure 5). We will now go through the steps of the snapshot generation using this example.

```
package acc(x.val) --* List(x) {
    fold List(x)
}
x.val := 3
apply acc(x.val) --* List(x)
unfold List(x)
assert x.val == 3
```

Figure 13: Example program for snapshot generation.

Let us call the value of `acc(x.next)` and the snapshot of `List(x.next)` before the `package` statement is executed, v_{next} and $Snap_{List}$ respectively. During the execution of the `package` statement, Silicon will now generate *root* and use it to produce the wand's left-hand-side, as described above. Since the left-hand-side consists of only one conjunct (`acc(x.val)`), the resulting snapshot for the left-hand-side that is used during the execution of the proof script will simply be (*`root`*). Next the fold statement of the proof script is executed, which will calculate a snapshot for `List(x)`, using the snapshot from the left-hand-side for `acc(x.val)` and the snapshots from *hOuter* for permissions that become part of the footprint, i.e. `acc(x.next)` and `List(x.next)`. The resulting snapshot is (v_{next}, *`root`*, $Snap_{List}$).

When the wand's right-hand-side is then consumed this snapshot obtained. Next, a magic wand snapshot is generated that combines the *root* with the snapshot of the right-hand-side, resulting in (*`root`*, (v_{next}, *`root`*, $Snap_{List}$)), which we will call $Snap_{wand}$.

Now that the package statement has been executed, 3 is assigned to `x.val`, so the value of `acc(x.val)` becomes 3. During the execution of the `apply` statement first, the magic wand is consumed and $Snap_{wand}$ is obtained. Next, the left-hand-side of the wand is consumed and 3 is obtained. Now the left-hand-side's snapshot, provided by $Snap_{wand}$, with the snapshot obtained form the actual left-hand-side, i.e. *`root`* `= 3` is added to the path conditions. The next step is to produce the wand's right-hand-side using the

right-hand-side snapshot provided by $Snap_{wand}$, resulting in a predicate chunk for `List` with the snapshot (`Snap`$_{next}$, *root*,`Snap_List`).

Lastly the `unfold` statement causes `acc(x.val)` to be produced with the snapshot *root* and `x.val == 3` can be proven during the execution of the `assert` statement since it knows that `x.val =` *root* and the path conditions contain *root* `= 3`.

4.2 Additional Details

The algorithm as described in the Implementation chapter has to be modified a bit further to make sure the strategy described above always works:

- `transfer` might cause a state consolidation to happen if it does not find the necessary permissions. If this happens the snapshots are unified by equating them to each other and using one of them, or in some situations creating a *fresh* snapshot and equating it to both of the original snapshots.

  However, any information regarding snapshots would be added to the path conditions and therefore be lost, since the path conditions are restored to what they were before the execution of the package statement started, when the package statement finishes. To address this issue a path condition mark is set during the execution of `transfer`. Path condition marks allow it to collect all path conditions, that were added after it was set. In this case it is later used to collect all path conditions that were added during the execution of `transfer`. The collected path conditions are then added to a stack of path conditions that is part of the state and later used to restore those path conditions when the `package` statement has finished execution. Furthermore, for nested package statements these path conditions are copied to the next lower entry on the stack, so they can propagate all the way to the outer-most state.

- The same issue occurs, when permissions are consumed from multiple reserve heaps: Here, too, the snapshots of several chunks (one from each reserveHeap, from which permissions were consumed) are unified into one (that is returned as part of the result of `transfer`). As the corresponding assumptions are added during the execution of `transfer`, they are automatically captured by the approach above. Note that the unification of snapshots was already necessary to ensure that the usage of snapshots during the execution of ghost operations is sound and consequently was already part of the original algorithm.

4.3 Open Problems

It should furthermore be noted that, as Schwerhoff mentions in section 5.4.2 of his PhD thesis [8], this snapshot implementation would no longer be sound, if fractional wands were to be implemented in the future. Fractional wands would allow a fraction of the permissions required by the left-hand-side to be exchanged for a fraction of the permissions obtained from the right hand side. For example, if in the example above half of the wand should be applied, `acc(x.val, 1/2)` would be given up and `acc(List(x), 1/2)` would

be obtained. This could be done twice using the same magic wand chunk. Consequently, the same magic wand snapshot would be used twice, such that the actual left-hand-sides, obtained during the two `apply` statements, are equated to each other by transitivity. A program, such as the one shown in Figure 14, would therefore result in an inconsistent state, which is unsound.

```
//loc = {x = v0}, h = {v0.next = null # 1, v0.val = v1 # 1}, pcs = ∅
package acc(x.val) --* List(x) {fold List(x)}
//loc = {x = v0}, h = {v0.val = v1 # 1}, pcs = ∅
x.val := 3
//loc = {x = v0}, h = {v0.val = 3 # 1}, pcs = ∅
apply acc(acc(x.val) --* List(x), 1/2)
//loc = {x = v0}, h = {v0.val = 3 # 1/2, List(v0) # 1/2}, pcs = {
    root = 3}
unfold acc(List(x), 1/2)
//loc = {x = v0}, h = {v0.val = 3 # 1/2, v0.val = root # 1/2}, pcs =
     {root = 3}
x.val := 2
//loc = {x = v0}, h = {v0.val = 2 # 1}, pcs = {root = 3}
apply acc(acc(x.val) --* List(x), 1/2)
//loc = {x = v0}, h = {v0.val = 2 # 1/2, List(v0) # 1/2}, pcs = {
    root = 3, root = 2}
assert false //succeeds (root = 3 ∧ root = 2 ⟹ false)
```

Figure 14: Example showing unsoundness of snapshot algorithm in connection with fractional magic wands.

Besides obtaining them through `package` statements, magic wands can also be obtained by inhaling them (either as part of a precondition or using an `inhale` statement). The current implementation is not capable of calculating snapshots for magic wands in that case. Note that this incompleteness can be circumvented by strengthening the right-hand-side of the magic wand to include terms like `x.f == old[lhs](x.f)`, for any field `f` and receiver `x` that express that the value of `x.f` is the one `x.f` had on the wand's left-hand-side.

For information from the wand's footprint, calculating a snapshot is impossible, since the necessary information is not available. Since, as we have seen earlier, magic wands are defined to be side-effect-free, i.e. values of subexpressions that occur on the left-hand-side of a wand still have the same value when occurring on the right-hand-side, it should, however, be theoretically possible to construct a snapshot that preserves values from the left-hand-side of the magic wand. This could be done by analyzing the right-hand-side and finding such subexpressions that occur on the left-hand-side of the wand. Since such an analysis would have to also check bodies of recursive predicates, this analysis would have to be infinitely deep and would, in general, be undecidable. Such an analysis would thus have to introduce some incompletenesses (e.g. by limiting the depth to which predicates are analyzed). An alternative approach would be to try generating a proof

script, which is also undecidable, but might be possible, to some extend, using heuristics.

Since the problem lies in the fact that the current snapshot implementation requires knowledge of the exact location of the occurrence of a value in an assertion, using a different snapshot implementation might also solve the problem. Constructing snapshots lazily, by providing enough information within the outer snapshot, such that the inner snapshot can be calculated, if needed, might be a solution to this issue. A simple way such an implementation might be realized would be to represent snapshots using a map that maps field accesses (symbolic values as receivers in combination with a field that should be accessed) to their symbolic values and that is stored as part of the symbolic state. This allows the structure of the wand's right-hand-side to be explored only as far as the verification requires it, while providing a snapshot for any field access that is actually reached.

Note that this implementation would also solve the issue with fractional wands described before, as the map would be directly accessed whenever necessary, thus eliminating the need to equate any snapshots.

4.4 Applying

It is in some cases helpful or even necessary to be able to use a wand temporarily to obtain the permissions, that are needed to evaluate an expression. This might be the case if it is hard or impossible to `package` the wand again or if the expression, that should be evaluated using permissions obtained from the wand occurs in a precondition, postcondition or loop invariant since statements cannot be used in those places. This becomes particularly interesting, since we have just seen, how snapshots are implemented for magic wands: the information contained in the snapshot can then be used during the evaluation of an expression. This would allow assertions about the snapshot of the snapshot of a wand to be encoded, such that notably snapshots for wands that were inhaled, which, as we have just seen, cannot currently be calculated automatically, could be added manually.

In order to use magic wands in expressions, they have to be applied temporarily and side-effect-free, meaning that the symbolic state after evaluating the expression should be indistinguishable from the symbolic state before evaluating the expression (with the exception that information about the value of the expression is learned). With the new syntax, this is possible using the `applying` keyword. The syntax of such an expression is shown in Figure 15.

applying_ expr := **"applying"**, **"("**, *magic_ wand_ assertion*, **")"**, **"in"**, *expression*;

Figure 15: Syntax of an applying expression

The value of an applying expression, `applying (A --* B) in C`, is defined to be the value of `C` if `A --* B` is applied.

The evaluation of such an expression in Silicon is done by first applying the wand normally, then evaluating `C` in the resulting state and continuing execution in the original

state, before the wand was applied. The value of `C` is furthermore obtained and given as the value of the `applying`-expression.

Let us now take a look at a few examples, to illustrate why such an expression might be necessary. In Figure 16, where `next` is a function giving the value of the next element in a linked list, `values` is a function that gives the values of a linked list as a sequence, `head` is the head of a linked list and `origValues` contains the original values of the linked list, with `List` defined as in Figure 3 we see that in some circumstances simply applying and packaging the wand might come with a significant overhead. The example loops over the linked list and tries to assert that the values of that list are not changed, to gain access to all values of the list it is necessary to get the permissions to the beginning of the list back using a magic wand.

```
package List(head) --* List(head)
var cur: Ref := head
while (next(cur) != null)
//some invariants omitted
invariant List(cur) --* List(head) {
    var prev: Ref := cur
    unfold List(cur)
    cur := cur.next
    package List(cur) --* List(head)
        {
        fold List(prev)
        apply List(prev) --* List(
            head)
    }
    apply List(cur) --* List(head)
    assert values(head) == origValues
    var cur2: Ref := head
    package List(cur2) -* List(head)
    while (cur2 != cur) {
        var prev2: Ref := cur2
        unfold List(prev2)
        cur2 = cur2.next
        package List(cur2) --* List(head)
            {
            fold List(prev2)
            apply List(prev2) --*
                List(head)
        }
    }
}
```

```
package List(head) --* List(
    head)
var cur: Ref := head
while (next(cur) != null)
//some invariants omitted
invariant List(cur) --* List(
    head) {
    var prev: Ref := cur
    unfold List(prev)
    cur := cur.next
    package List(cur) --*
        List(head) {
        fold List(prev)
        apply List(prev) --*
            List(head)
    }
    assert applying (List(cur) --*
        List(head)) in values(head)
        == origValues
}
```

Figure 16: Left: without `applying` expression, Right: with `applying` expression

Moreover, Viper supports abstract predicates. Abstract predicates can be used like regular predicates, but no body is provided. It is therefore not possible to fold or unfold such a predicate. For wands, that contain abstract predicates it might be necessary to use `inhale` and `exhale` statements to fully undo an apply operation, as seen in Figure 17, where `P` is an abstract predicate and `f` is a field of type `Int`. Note that during the execution of the package statement `acc(x.f)` simply becomes part of the wand's footprint.

```
apply P() --* acc(x.f)
assert x.f == 0
package P() --* acc(x.f)
inhale P()
```

```
assert applying (P() --* acc(x.f)) in
    x.f == 0
```

Figure 17: Left: without `applying` expression, Right: with `applying` expression

This solution is error prone as, in this case, any change to the left-hand-side of wand requires changes to the inhale statement and any mistakes made result in unsoundnesses rather than verification errors. In contrast, if the left-hand-side of wand used in an `applying`-expression is not changed to match the one present in the current state a verification error stating that the magic wand could not be found would be generated.

When verifying loops invariants are used in Viper. Silicon first checks whether those invariants hold on entry into the loop and then verifies that the invariants are preserved by symbolically executing the loop body once. Since only the assertions contained in the loop invariants are checked only those assertions can be used after exiting the loop. Hence it is necessary to include assertions about a magic wand's snapshot in loop invariants. This can be done using `applying`-expressions. Figure 18 is a modified version of Figure 16 that illustrates how this can be done.

```
while (next(cur) != null)
//some invariants omitted
invariant List(cur) --* List(head)
invariant applying (List(cur) --* List(head)) in values(head) == origValues {
    var prev: Ref := cur
    unfold List(prev)
    cur := cur.next
    package List(cur) --* List(head) {
        fold List(prev)
        apply List(prev) --* List(head)}}
apply List(cur) --* List(head)
assert values(head) == origValues //succeeds, because of the
   highlighted invariant
```

Figure 18: `applying` being used to encode information about a magic wand snapshot.

5 Integration with Quantified Permissions

Quantified permissions in Viper [4] can be used to express permissions to a set of locations. For example, the assertion `forall x: Ref :: x in xs ==> acc(x.f)`, where `xs` is of type `Set[Ref]` and `f` is a field, expresses that for any element of `xs` we have full permission to `f`. The intuitive understanding of such an expression is that for any `x` that satisfies the condition `x in xs`, `acc(x.f)` holds. Note that quantified permission assertions, in separation logic, are different from quantified expressions in first order logic.

While quantified chunks can be used to represent single locations by ensuring that the set the quantified permission assertion abstracts over only contains one element, Silicon internally distinguishes between chunks for quantified permission assertions and those for regular assertions. This is mostly due to performance reasons. Non-quantified chunks are moreover only created for fields and predicates, that never occur in quantified permission assertions. This ensures that the consumer only has to consider chunks of one kind (quantified or non-quantified) at a time.

As part of this thesis quantified permissions have been integrated with magic wands. This allows the usage of magic wands in quantified permissions (i.e. sets of magic wands). For example the assertion `forall x: Ref :: x in xs ==> acc(x.val) --* List(x)`, states that the wand `acc(x.val) --* List(x)` is available for any `x` that is in `xs`. Moreover quantified permissions can now be used in magic wands (i.e. quantified permissions occurring as part of the wand's left-hand-side or right-hand-side, using them in the proof script and calculating an appropriate footprint for them). A wand taking advantage of this might look like `(forall x: Ref :: x in xs ==> acc(x.val)) --* List(y)`. This is a single magic wand, that requires `acc(x.val)` for any `x` in `xs` as part of its left-hand-side and allows for those permissions to be exchanged for `List(y)`, for some `y` of type `Ref`. Magic wands inside of quantified permission assertion and quantified permission assertions inside of magic wands are mostly independent of each other and will be treated separately in the following.

5.1 Quantified Magic Wands

As explained above, quantified magic wands are sets of magic wands. For Silicon to be able to handle them correctly it needs an internal representation (the chunk) and the `producer` and `consumer` have to be modified.

The quantified magic wand chunk consists of the variables over which the assertion quantifies, a snapshot map, the permissions, and either inverse functions (will be explained later) and the condition of the quantified permission assertion or the values of the quantified variables for single magic wands.

To be able to ensure that only quantified chunks are created for magic wands that occur in quantified permission assertions, they are collected by traversing the AST of the Viper program. The heap independent expressions can be replaced by quantified variables in different quantified permission assertions, the structure of a wand is therefore defined to be the original wand with the heap independent expressions being replaced by "holes." The structures of the found wands are therefore stored globally and only quantified

chunks are created for wands with those structures.

Snapshot maps are functions that take an assignment of the "holes" in the structure of a wand and return the snapshot for the assertion with the provided values for the quantified variables. These functions are directly handled by the SMT-solver. A snapshot map has to be declared for each magic wand shape that was found.

We will now use the example shown in Figure 19, where `xs` is of type `Seq[Ref]` and `List` is defined as in Figure 3, to show how the production of a quantified magic wand assertion works.

```
forall i:  Int ::  i > 0 ==> acc(x.val) && x.val == i+1 --* List(x)
```

Figure 19: A quantified magic wand assertion.

The pseudo code for the production of a quantified magic wand assertion is given in Figure 20.

```
1  produceQuantifiedMagicWandAssertion(quantifiedVariables,
       condition,wand,state,snapshotMap) {
2    var heapIndependentExpressions = wand.
       heapIndependentExpressions
3    var formalVariables = new List
4    for each expression in heapIndependentExpressions {
5      var formalVar = new Variable
6      wand.replace(expression, formalVar)
7      formalVariables += formalVar
8    }
9
10   //calculate relationships between the formal variables and the
       quantified variables
11   var symbolicQVariables = state.newSymbolicValesForVariables(
       quantifiedVariables)
12   state.pathConditions += condition
13   var relationships = evaluator.evaluate(
       heapIndependentExpressions)
14   var inverseFunctions = new List
15   for each i in formalVariables.indices {
16     inverseFunctions += (formalVariables[i], relationships[i])
17   }
18
19   state.heap += new QuantifiedMagicWandChunk(symbolicQVariables,
       snapshotMap,WritePermissions,inverseFunctions,condition)
20 }
```

Figure 20: Pseudo code for the production of quantified magic wand assertions.

The first step is to determine the heap independent expressions (the "holes" in the wand's structure), since these are the expressions over which a quantified permission assertion can quantify (l.2). In our case we get `x` three times (once in `acc(x.val)`, once in `x.val == i+1` and once in `List(x)`) and `i+1`. Note that these include `x`, even though it is not a quantified variable. Each of these expressions is replaced by a newly generated formal variable, these will be `x0`, `x1`, `x2` and `x3` in this example, such that the resulting magic wand is `acc(x0.val) && x1.val == x2 --* List(x3)` (ll. 3-8).

We now need to ensure, that the wand using the formal variables is equivalent to the original one. This is done using inverse functions, which take the values of the quantified variables as arguments and return the corresponding values of the formal variables.

To calculate these inverse functions new symbolic values are generated for every quantified variable (l. 11). In our case, we only have one quantified variable (i) and we will call the corresponding symbolic value v_i. Next, the condition of the quantified magic wand assertion is added to the path conditions since we only want to consider assignments to the quantified variables, for which the condition holds (l. 12). In this example the condition is `i > 0`, v_i `> 0` is therefore added to the path conditions. It is then possible to evaluate the values of the heap independent expression in dependence on the newly introduced symbolic values (l. 13). Assuming that the symbolic value of `x` is v_x, the result will be v_x for the 3 expressions that are `x` and v_i `+ 1` for `i+1`. Finally, the inverse functions can be generated, such that for any assignment of the quantified variables the expressions generated in the last step can be evaluated and the result assigned to the specified formal variable (ll. 14-17).

Now that all the necessary information has been calculated, the chunk can be created and added to the heap of the symbolic state, thus completing the produce operation (l. 19). During the production, a snapshot map has already been provided. If a quantified chunk should be generated based on a non-quantified magic wand assertion, a snapshot map can be generated, by evaluating the heap independent expressions of the wand and adding a path condition to ensure that the value of the snapshot map, at the point specified by those expressions, is the wand's snapshot. For example for the wand `acc(x.val) && x.val == i+1` $-\!\!*$ `List(x)`, with v_x being the symbolic value of `x`, `i = 0`, `snap` being the snapshot of the wand and `snap_map` being the snapshot map for the wand's structure, a suitable path condition would be `snap_map(`v_x`, `v_x`, 1, `v_x`) = snap`.

The consume operation for quantified magic wands works analogously to the one for quantified predicates and fields that will be explained as part of the next section.

5.2 Quantified Permissions in Magic Wands

In this section, we will take a look at how quantified permissions can be used in magic wands. As mentioned before consuming quantified permission assertions is done differently from how non-quantified assertions are consumed. We, therefore, have to make sure that this consumption works inside of `package` statements and that the footprint is calculated correctly.

We have seen in chapter 3 that the consumption and footprint calculation inside of `package` statements is done using `transfer`. As described in section 3.2 the new transfer

method offers more flexibility by accepting a consume function that is called on each heap that should be considered. In order to consume quantified permission assertions and calculate an appropriate footprint, it, therefore, suffices to implement a consume function for quantified permission assertions. This implementation will be based on the original algorithm for quantified permission assertions outside of `package` statements and will be explained in the rest of the chapter.

The signature of the consume function has been defined in section 3.2 such that it takes a heap from which permissions should be consumed and the amount of permissions that still need to be consumed as arguments and returns the permissions it actually consumed. The pseudo code for the consume function is given in Figure 21.

```
1  given: resource //the field, predicate or wand used in the
       quantified permission assertion
2  given: condition //of the quantified permission assertion
3  consumeQuantifiedPermissionAssertion(heap, permsToConsume) {
4    var relevantChunks = heap.getChunksFor(resource)
5    var permsNeeded = permsToConsume
6
7    //precompute how many permissions should be taken from each
         chunk
8    var toTake = new List
9    for each chunk in relevantChunks {
10     var permsTaken = condition ? min(chunk.permissions,
           permsNeeded) : 0
11     permsNeeded -= permsTaken
12     toTake += permsTaken
13   }
14
15   var snapshotMaps = new List
16   for each i in relevantChunks.indices {
17     relevantChunks[i].permissions -= toTake[i]
18     snapshotMaps += relevantChunks[i].snapshotMap
19   }
20
21   var totalPermissionsTaken = permsToConsume - permsNeeded
22   var consumedSnapshotMap = unifySnapshotMaps(snapshotMaps)
23   return new QuantifiedChunk(resource, consumedSnapshotMap,
         totalPermissionsTaken, condition)
24 }
```

Figure 21: Pseudo code of the consume function for quantified permission assertions.

To illustrate the steps taken by the algorithm recall the running example, as shown in

Figure 5. During the execution of the `fold` statement transfer is used to consume `x.next`, as we have seen in section 3.1. Let us now assume that there are two quantified chunks in *hOuter*, representing `forall y: Ref :: y in ys ==> acc(y.next, 1/2)` (we will denote this chunk with $\boxed{1}$) and `forall z: Ref :: z in zs ==> acc(z.next, 1/2)` (we will denote this chunk with $\boxed{2}$), where `ys` and `zs` are local variables of type `Set[Ref]` that have `x` as an element. `transfer` is therefore called with the consume function shown in Figure 21. The consume function is first called for *hOps*, with write permissions needed.

The first step of the consume function is to find any quantified chunks for the `next` field (l. 4), which in this case yields an empty list. The algorithm then precomputes some data to determine how many permissions will be taken from each chunk it found and afterward consumes those permissions, both of which we will see in more detail when we reach *hOuter* and find some chunks (ll. 7-19). Up to this point, the algorithm behaves exactly the same as the original one for quantified permission assertions outside of `package` statements. The next step of the original algorithm would now be to check whether any more permissions need to be consumed, which is the case, and fail. The new algorithm now creates a quantified chunk that represents the permissions it consumed, i.e. a chunk with no permissions and returns it (ll.21-23). `transfer` next calls the consume function for `hLhs`, with write permissions needed, for which the execution looks the same.

Next, the consume function is called for *hOuter*. This time two chunks are found during the first step (l. 4). As we have seen the next step is to precompute how many permissions should be taken from each chunk (ll. 7-13). Since we now found some chunks, we start with $\boxed{1}$. First, we create a term $permsTaken_{\boxed{1}}$ = `r == x ? min((r in ys ? 1/2 : 0), 1) : 0` to represent the permissions that should be taken from $\boxed{1}$, where `r` is the quantified variable for the receiver of the field access in chunk $\boxed{1}$ (l. 10). The condition at the beginning is used to ensure that permissions are only taken for `x` and not for any other elements of `ys`. If the condition is true we take as many permissions as possible, but never more than the chunk provides (`(y in ys ? 1/2 : none)` are the permissions provided by $\boxed{1}$) or more than we still need. This term is then subtracted from the permissions that still need to be consumed resulting in $permsNeeded_{\boxed{1}}$, which indicates the amount of permissions that will be needed after consuming permissions from $\boxed{1}$ (l. 11). Note that $permsNeeded_{\boxed{1}}$ only evaluates to `1/2` if `x == y && y in ys` holds. The same is done with $\boxed{2}$. This time we calculate $permsTaken_{\boxed{2}}$ = `r == x ? min((r in zs ? 1/2 : none),` $permsNeeded_{\boxed{1}}$`) : none` permissions and calculate $permsNeeded_{\boxed{2}}$ analogous to before.

To actually consume those permissions it now suffices to first subtract $permsTaken_{\boxed{1}}$ from $\boxed{1}$'s permissions and $permsTaken_{\boxed{2}}$ from $\boxed{2}$'s permissions (l. 17). We furthermore collect their snapshot maps (l. 18) which are used to create a chunk summarizing the permissions we just consumed (ll. 21-23). Note that, during the creation of the new snapshot map, path conditions are generated that need to be preserved (as explained in section 4.2). This happens automatically since the consume function is executed as part

of the `transfer` method, which already records any newly generated path conditions.

Transfer then checks that no more permissions are needed and terminates successfully. Since we removed the necessary permissions from *hOuter*, the appropriate footprint has furthermore been calculated (as described in chapter 3).

Since the `transfer` method has now been modified we need to assure ourselves that Theorem 1 of [7], still holds, which will be argued informally. The Theorem has two parts:

If transfer completes successfully

- The state is split into two parts (a part added to *hUsed* and a remainder of the reserve heaps), which together contain the same permissions as the original state.

  The consume function and `transfer` method are designed, such that the following claims hold:

  1. The consume function splits the heap it receives as an argument in a remainder heap and a quantified chunk that contains exactly the permissions that were taken from the original heap.
  2. After the successful execution of the `transfer` method *hUsed* contains the chunks returned by the executions of the consume function and the reserve heaps contain the remainder heaps of the same executions.

  It is intuitively clear that these two claims entail the original statement, made by the theorem.

- The contents of *hUsed* satisfy the assertion that was consumed, i.e. enough permissions were consumed.

  This is ensured by counting down the amount of permissions that still need to be consumed during each execution of the consume function and the fact that `transfer` terminates successfully if and only if it reaches a point where no more permissions need to be consumed.

6 Evaluation

One goal of my thesis was to improve error reporting in certain situations. To illustrate the problem Figure 22, where `List` is defined as in Figure 3, `x` is a local variable of type `Ref` and `id` is an identity function that has `List(x)` as its precondition, shows the same `package` statement in the old and the new syntax. Note that 1 and 2 are not part of the program, but serve to facilitate discussion.

```
package List(x) --* 1 unfolding
    List(id(x)) in 2 id(x) == x
```

```
package List(x) --* id(x) == x {
    1
    unfold List(id(x))
    2
}
```

Figure 22: Left: old syntax, Right: new syntax

Verifying this package statement will always fail: from the left-hand-side we get permission to `List(x)`, which is available when point 1 is reached. When evaluating `id(x)`, for the first time its precondition therefore holds. After unfolding the `List` predicate and reaching point 2, we no longer have permission to `List(x)`, thus evaluating `id(x)` for the second time, as part of the wand's right-hand-side, fails, since the precondition cannot be established. The reported error in the old syntax would state that the `package` statement failed, and give as a reason that the precondition of `id` could not be established. However, from that information, it was not possible to find out which of the two evaluations of `id` failed. With the new syntax, if the evaluation of `id` during the `unfold` statement fails, the reported error will indicate that that statement failed, rather than the entire `package` statement. If in contrast, the evaluation of `id` as part of the right-hand-side (after reaching 2) fails, the reported error indicates that the `package` statement failed. It is, therefore, possible to distinguish the two cases.

Another problem with the old approach was that it was not possible to reference the states at 1 and 2. Viper offers `label` statements that can be placed in a program to mark the state at that point. A labeled old expression can then, later on, be used to evaluate an expression in the state indicated by the provided label. The new syntax now allows `label` statements to be placed at points 1 and 2 in the example above and for the corresponding labels to be used inside the proof script (referencing states that occur inside a proof script from the outside would be unsound).

Additionally, the following achievements have been accomplished:

- The performance of the Verifier does not seem to have been impacted by my changes. Sierra ran performance tests as part of his Bachelor's thesis [9] on a Silicon version that includes my changes and did not find any significant performance differences.

- Silicon includes some heuristics that can be used to infer `package`, `apply`, `fold` and `unfold` statements. With minor adjustments to the source code, it was possible to keep these heuristics working with the new syntax.
- As part of this thesis, 131 test methods and functions have been added to the Viper test suite, representing approximately a 33% increase over the 394 test methods and functions for magic wands, the suite previously included.[2]
- 7 known issues, in Silver, Silicon, and Carbon, connected to magic wands have been solved or partially addressed.

7 Conclusion

The new syntax and the Silicon verifier allow for complex magic wands and their proof scripts to be expressed, managed, and verified. It, moreover, lifts some restrictions, that the old syntax imposed on the kinds of ghost operations that can be used. It for example now possible to branch during the execution of the proof script and an example program that could not be verified using the old syntax, but can be verified using the new syntax is available on Viper Online [6].

Furthermore, internal changes to Silicon allow assertions inside of `package` statements to be consumed more uniformly in comparison to their consumption outside of `package` statements. This reduces code duplication and programming overhead and will facilitate future additions to the types of resources that are supported by the Viper tools.

On the basis of several examples, we have seen how Silicon verifies programs in the presence of magic wands and given intuitions for the algorithms' soundness. We have furthermore seen how the proof script can be used to construct a footprint for a magic wand. In the context of quantified permissions, we have seen how the new transfer method makes it easy to extend the footprint algorithm to new resource types.

Lastly to make the transition to the new syntax easier a migration tool is available at [1].

8 Future Work

- Implement the new syntax in Carbon.
- Support snapshots for magic wands that were inhaled.

  The problem with such snapshots is to find the locations, that values from the left-hand-side should have in the snapshot of the right-hand-side, where terms need to go in the presence of recursive predicates. More details and some ideas as to how this problem could be solved have been given in section 4.3.

[2] These numbers are based on the number of occurrences of the `method` and `function` keywords in the wands and wands/new_syntax directories of the test suite that were found using the `grep` command line tool.

- Support fractional magic wands and adjust the snapshot algorithm accordingly.

  Fractional wands would allow the same wand to be partially used multiple times. Using the current snapshot implementation this would cause the left-hand-sides to be equated indirectly. For more information see section 4.3.

- Implement self-framingness checks for magic wands in Silicon.

  These checks had been disabled, due to some problems, prior to my thesis and have not been reimplemented, yet.

- Develop additional heuristics for the capabilities of the new syntax.

  Silicon's heuristics allow it to infer `package`, `apply`, `fold` and `unfold` operations in many cases, including as part of magic wand proof scripts. Since it is now possible to use additional statements as ghost operations in magic wand proof scripts, some additional heuristics could possibly be implemented. Furthermore, the existing heuristics do not handle quantified permissions and quantified magic wands.

- Develop heuristics similar to those implemented in Silicon for Carbon.

  Silicon's heuristics are based on the fact that it can inspect the symbolic state when it encounters an error. With verification condition generation in Carbon, a similar approach is not possible. There are therefore currently no heuristics implemented in Carbon.

References

[1] Nils Becker. MagicWandMigrator. `https://bitbucket.org/nilsbecker_/magicwandmigrator`. Accessed: 2017-08-15.

[2] Rémi Brochenin, Stéphane Demri, and Étienne Lozes. On the almighty wand. *Information and Computation*, 211:106–137, February 2012.

[3] Cyrill Martin Gössi. A Formal Semantics for Viper. Master's thesis, ETH Zürich, Switzerland, 2016.

[4] P. Müller, M. Schwerhoff, and A. J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification (CAV)*, volume 9779 of *LNCS*, pages 405–425. Springer-Verlag, 2016.

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

[6] Viper Online. List Iterator. `http://viper.ethz.ch/examples/list_iterator.html`. Accessed: 2017-09-03.

[7] M. Schwerhoff and A. J. Summers. Lightweight Support for Magic Wands in an Automatic Verifier. In J. T. Boyland, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 37 of *LIPIcs*, pages 614–638. Schloss Dagstuhl, 2015.

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

[9] Robin Sierra. Towards Customizability of a Symbolic-Execution-Based Program Verifier, Bachelor's thesis, ETH Zürich, Switzerland, 2017.