Chair of Programming Methodology
Department of Computer Science
ETH Zurich

Verification Condition Generation for Magic Wands

Bachelor's Thesis

Author:
Gaurav Parthasarathy
gauravp@student.ethz.ch

Supervised By:
Dr. Alexander J. Summers
Prof. Dr. Peter Müller

August 17, 2015

ETH zürich

Contents

1 Introduction

Separation logic is a permission-based logic that introduces two new connectives. One of the connectives is the magic wand, which has been shown to be useful when verifying concurrent programs or when iterating over data structures in loops. Under basic assumptions it has been shown to be undecidable to reason about programs which require the magic wand to be verified without the user giving additional specifications [4]. In [16], Malte Schwerhoff and Alexander J. Summers show how to add support for magic wands in automatic verifiers and provide an implementation of their approach in Silicon [17], a verifier based on symbolic execution.

Silicon [17] is one of the two back-end verifiers developed in the Programming Methodology Group at ETH Zurich as part of the Viper project, which is described in [9]. Carbon [8], the second back-end verifier, verifies programs using verification condition generation. The goal of this bachelor's thesis is to provide a solution to support magic wands in Carbon, based on the approach outlined in [16].

We first give background information on magic wands, the Silver programming language, an overview of the most important aspects from [16] and the Carbon verifier. We then move on to discuss our implementation of the approach described in [16] in Carbon.

2 Background

2.1 Separation Logic

Separation logic [13] is a permission-based logic which has gained a lot traction in recent years, due to its ability to deal elegantly with framing or data races in programs involving shared and mutable data structures. One of the main ideas of separation logic is to provide a simple way to prove program properties using only those heap locations which could potentially have an effect on the program property itself. It achieves this with the notion of splitting a program state into partial, disjoint heaps.

Separation logic introduces two new connectives: the separating conjunction, denoted by $A * B$ and the magic wand (also known as the separating implication), denoted by $A \mathrel{-\!\!*} B$. A and B are assertions.

Before presenting these two new connectives, we introduce some notation. For simplicity we will assume that a state consists of a partial heap and a total evaluation function for local variables. A partial heap is characterized by a partial function which maps pairs of references and fields to their corresponding values in the heap. We call two states σ_1, σ_2 *compatible* in separation logic if the domains of their heaps are disjoint (i.e. the domains of their corresponding partial functions are disjoint) and they agree on the values of all local variables; we denote this by $\sigma_1 \perp \sigma_2$. If $\sigma_1 \perp \sigma_2$ holds then we can write $\sigma = \sigma_1 \uplus \sigma_2$ to denote that σ has the same local variables as σ_1 (and σ_2) and its heap consists of the (disjoint) union of the two heaps.

Let h be the partial heap of state σ. σ satisfies the points-to assertion $x.f \mapsto v$ if the $x.f$ is in the domain of the partial function f_h characterizing h and f_h maps $x.f$ to v.

2.1.1 Separating Conjunction

The formal semantics of the separating conjunction is given by:

$$\sigma \models A * B \Leftrightarrow \exists \sigma_1 \perp \sigma_2 . (\sigma = \sigma_1 \uplus \sigma_2) \wedge (\sigma_1 \models A) \wedge (\sigma_2 \models B)$$

This means if $\sigma \models A * B$, then it must be possible to split σ into two compatible states σ_1, σ_2 such that $\sigma_1 \models A$ and $\sigma_2 \models B$. Therefore if $(x.f \mapsto v) * (y.f \mapsto w)$ holds in a σ, i.e. there exists σ_1 and σ_2 such that $\sigma = \sigma_1 \uplus \sigma_2$ and σ_1 satisfies $x.f \mapsto v$, σ_2 satisfies $y.f \mapsto w$, then we can conclude that $x \neq y$. This holds since the domains of the partial functions characterizing the partial heaps of σ_1 and σ_2 are disjoint.

2.1.2 Magic Wands

The magic wand $A \mathrel{-\!\!*} B$ also known as the separating implication is the second interesting connective. Its semantics (taken from [16]) is given by:

$$\sigma \models A \mathrel{-\!\!*} B \Leftrightarrow \forall \sigma' \bot \sigma . (\sigma' \models A \Rightarrow (\sigma \uplus \sigma' \models B)$$

A magic wand describes what the state σ in which it holds could derive if any state, satisfying a specific property and compatible to σ, was added to it. In particular if a wand holds in some state and the left hand side of the wand holds in a compatible state, then one can essentially get the right hand side by combining the two states. This can be summarized by the following inference rule, which is similar to *Modus Ponens* in traditional logic:

$$\frac{\sigma \models A * (A \mathrel{-\!\!*} B)}{\sigma \models B}$$

The power of the wand lies in the fact that it draws conclusions for potentially a multitude of states, instead of just one, which may be added in the future to the current state. It has been used to verify, for example, programs iterating over data structures where at certain points there's only a partial view of the data structure. An example is given in [12] and various other examples are cited in [16].
Automatic verification of programs using magic wands is undecidable for programs with very basic features and without any help from the user [4].

2.2 Viper Project

The Viper project [9] is a verification infrastructure developed at the Programming Methodology Group at ETH Zurich as an effort to ease the development of verifiers using permission-based logics. It provides an intermediate language, called Silver, which natively supports the notion of permissions as well as two back-end verifiers Carbon and Silicon introduced already in Section 1. One of the goals is for developers who want to verify programs in certain programming languages to encode a translation into Silver which can then be verified by the already existing back-end verifiers. There are translations for languages (or subsets of a languages) available such as Chalice [10] or Scala [5]. Finally Sample is a static analyzer being developed for Silver programs. It's goal is to infer specifications using abstract interpretation.

2.3 Silver

Silver is an intermediate language supporting basic constructs such as methods, side-effect free functions (which are expressions), predicates, as well as specifications in form of pre-/postconditions or loop invariants. One can specify assumptions as well as assertions. It has a built-in heap but doesn't directly support classes. We'll mainly present the parts which are relevant for this thesis, for a more detailed discussion on Silver we refer to [9].

2.3.1 Permissions

For each heap location a permission value, represented by a fractional value in $[0, 1]$, is associated to it at each program point in a method. In Silver the accessibility predicate `acc(x.f,p)`, where $0 \leq p \leq 1$ denotes that permission amount p is held at $x.f$, where x is a reference to an object and f is a field. `acc(x.f)` is interpreted as `acc(x.f,1)` in Silver. To read a heap location $x.f$ a method must have some non-zero permission to it, to write to a heap location full permission (represented by 1) must be held at that location. In Silver if a method has full permission to a location then is guaranteed that no other method has access to the same location.

This concept of permissions mainly solves two important problems: framing and reasoning about concurrency. The heap locations which are mentioned in access predicates in the precondition of a method and where the permission value is 1 is an overapproximation of the heap locations which the method can potentially modify (and whose changes are visible to a caller). So when a method is called then an overapproximation of the "frame", i.e. the set of locations which are potentially changed, is given by the precondition of the callee. This concept of approximating the frame is called *Implicit Dynamic Frames* [18].

Since the permission value can be fractional, one may distribute the full permission among different threads running concurrently and be sure that there won't be any data races. The concept of fractional permissions is described in [2].

2.3.2 Self-Framedness

In our discussion we will call an assertion A *self-framing* if A contains enough accessibility predicates such that non-zero permission to each heap location on which A depends is held. For example `x.f==0` is not self-framing but `acc(x.f)&&x.f==0` is self-framing. Note that by our definition the following assertion would be self-framing: `acc(x.f)&&x.f==0&&x.f==0?0:z.f` since it

doesn't depend on *z.f* (where `cond?A1:A2` equals assertion *A1* if *cond* holds and equals assertion *A2* otherwise).

2.3.3 Inhale and Exhale

In Silver a pure expression is an expression which doesn't contain any accessibility predicates nor magic wands (we'll introduce magic wands in Silver later). Assume *e* is an expression only consisting of accessibility predicates and pure boolean expressions. Then **`inhale`** `e` assumes all the pure expressions in *e* and adds all the permissions mentioned in accessibility predicates in *e*. The counterpart to **`inhale`** is given by **`exhale`**. **`exhale`** `e` asserts all pure expressions in *e* and for all accessibility predicates in *e* checks if the permission specified is held and then removes the permission. If after an exhale all of the permission to a specific heap location is lost, then all the information about that heap location is lost. Note that for a pure boolean expression *e*, **`inhale`** is the same as **`assume`** and **`exhale`** is the same as **`assert`**.

Method calls can be encoded by exhaling the precondition and inhaling the postcondition.

2.3.4 Abstract Predicates

Sometimes it's not possible to enumerate all the heap locations to which a function needs permission. An example is when the function traverses a recursive data structure such as a binary tree. To still be able to specify the statically unknown heap locations, abstract predicates (introduced in [15]) are used in Silver. The body of an abstract predicate can be recursive to solve the mentioned problem and it can also contain permission to specific heap locations.

Similar to heap locations, **`acc`**`(P(x),p)` denotes that permission *p* is held for predicate *P(x)*. The main difference to heap locations is that in this case *p* may be larger than 1. Suppose the body of P is given by **`acc`**`(x.f,1/2)`. Then it makes sense to say that a method holds the predicate twice (i.e.**`acc`**`(P(x),2)` holds).

The predicate instance and its body are treated separately. Assume a predicate instance is just held once at the beginning of a method. Then none of the facts in the predicate body are known. One of the reasons is to keep the verifier from unfolding a recursive predicate an unbounded amount. To exchange a predicate instance for its body Silver provides the **`unfold`** statement and the reverse operation is given by the **`fold`** statement. These two operations are called ghost operations because they don't change the program state but are just used to guide the verifier. Verifiers may rely on

users to explicitly mention the ghost operations where needed or they can have some kind of heuristics approach that tries to infer where to fold/unfold a predicate.

Silver also supports **`unfolding`** expressions which are of the form **`unfolding`** `P(arg)` **`in`** `e` where P is a predicate and e is a pure expression. This is used when the body of P is needed to show that permission is held to each heap location in e. So the verifier unfolds P temporarily then evaluates the expression e and afterwords folds the predicate back. Since e is pure the expression has no side effects.

2.3.5 Separation Logic in Silver

As mentioned in section 2.3.1 Silver is based on the implicit dynamic frames model. It has been shown that separation logic can be encoded in the implicit dynamic frames model in [14] and that the encoded assertions are self-framing. The formal semantics for the connectives in separation logic presented in section 2.1, needs to be modified for it to make perfect sense in the implicit dynamic frames model; the details are in [14]. For us it suffices to understand what the connectives mean. There are just a few changes that we want to highlight to make the discussion later on more precise. A state in Silver consists of an evaluation function for local variables, a heap and a mask storing the permissions to all heap locations and predicates, as well as the currently held wand instances (magic wands in Silver are presented in detail in the next subsection). There are also few changes to the notation and definitions that we need to highlight.

Definition 1 *Two states σ_1, σ_2 are compatible (denoted by $\sigma_1 \perp \sigma_2$) in Silver if the sum of the permissions to each heap location doesn't exceed 1 and if they agree on the values of all heap locations to which both states have non-zero permission and on the values of all local variables.*

Definition 1 shows that we're not necessarily talking about strictly disjoint heaps anymore as in separation logic.

Definition 2 *If $\sigma_1 \perp \sigma_2$ holds in Silver then we say that $\sigma = \sigma_1 \uplus \sigma_2$ holds iff:*

- *σ agrees on the values of all local variables with σ_1 (and σ_2)*
- *σ agrees on the values of all heap locations with σ_1 to which σ_1 has non-zero permission*

- *σ agrees on the values of all heap locations with σ_2 to which σ_2 has non-zero permission*
- *the permission amount held at each heap location in σ is equal to the sum of the permission amounts held to those heap locations in σ_1 and σ_2*
- *the values of all heap locations where σ_1 and σ_2 both have no permission can be arbitrary, i.e. no assumptions can be made about these values*

We call $\sigma_1 \uplus \sigma_2$ the compatible union of states σ_1 and σ_2.

The separating conjunction presented in section 2.1.1 is implicitly already supported in Silver. We show this by example (see [14] for a formal treatment). $x.f \mapsto v * y.f \mapsto w$ in separation logic can be encoded as `acc(x.f) && x.f==w && acc(y.f) && y.f==w` in Silver. With this encoding $x \neq y$ is also implied because the conjunction in Silver is multiplicative for permissions.
By multiplicative we mean that `acc(z.f,1/4) && acc(z.f,1/4)` is the same as just specifying `acc(z.f,1/2)`. Since permission to a location is not allowed to be greater than 1 we can conclude $x \neq y$ in the earlier example.

2.4 Magic Wands in Silver

Even though we presented the main features of Silver in the last section, we dedicate an own section just for the magic wand support in Silver due to its significance for this bachelor's thesis. The meaning of magic wands was given in section 2.1.2. In this section we present the support for magic wands in Silver introduced in [16]. In Silver a wand $A \mathrel{-\!\!*} B$ can be written as `A --* B` where A and B are assertions. A and B both must be self-framing (see section 2.3.2). In the paper the authors restrict the permission amounts specified in access predicates in A and B to be 0 or 1. We remove that restriction and let the permission value be any fractional amount in $[0, 1]$. Abstract read permissions (see [7]) in wands are not supported at the moment.

A wand instance held in some program state in Silver is treated as opaque, in the sense that the verifier in general doesn't attempt to conclude facts from the wand instance's meaning without the user providing direction. This treatment is similar to the treatment of abstract predicates (see 2.3.4), where the predicate body is in general not regarded by the verifier if just the predicate instance is held and there is never an **`unfold`** statement (or **`unfolding`** expression).

2.4.1 Package

The **package** statement can be used to construct and add a wand instance `A --* B` in a state where $A \mathrel{-\!\!*} B$ holds. The **package** is not guaranteed to work for every wand that holds, but it guarantees that if a wand doesn't hold in a state then the package will fail. When **package** `A --* B` is successful then part of the the current state σ is removed, called the footprint σ_{foot} of the wand, such that $\sigma_{foot} \models A \mathrel{-\!\!*} B$ is satisfied. This ensures that changes made to the remaining state after the **package** won't affect σ_{foot} and hence the wand instance can be used soundly at any point after the package.

Formally the package tries to choose the footprint σ_{foot} to be as small as possible such that for any hypothetical state σ_A, for which $\sigma_A \bot \sigma_{foot}$ and $\sigma_A \models A$ are true, it holds that $\sigma_{foot} \uplus \sigma_A \models B$. The resulting state after the package is σ' where $\sigma = \sigma' \uplus \sigma_{foot}$. If the verifier can't find such a σ_{foot} then there will be a verification failure. An empty footprint can be a valid footprint.

We want to stress that **package** $A \mathrel{-\!\!*} B$ and **inhale** $A \mathrel{-\!\!*} B$ are very different statements. Inhaling the wand just adds the wand instance to the current state without effectively computing a footprint in which the wand holds. It just adds the assumption that the wand holds in some (unknown) state disjoint to the state right before the **inhale**, even if no such state exists.

Note the analogy to abstract predicates already hinted at right before. The idea of the package of a wand is similar to the fold of a predicate. When we successfully fold a predicate then we create a predicate instance using the current state and if the verifier can't find the necessary permissions or some heap location doesn't have the value needed then there will be a verification failure. If we inhale a predicate then we don't use anything already available in the current state, we just assume that the predicate holds.

The difference is that in the case of a **fold** we know exactly what permissions/properties we need to construct a predicate instance. In the case of a **package** it's not clear what the footprint should be statically, the footprint isn't unique in general either.

Footprint computation algorithm. We've only mentioned that the verifier picks a footprint but not how. Let σ be the state in which the **package** is executed. First the footprint computation algorithm constructs a hypothetical state σ_A which satisfies the left hand side of the wand. It then constructs a state σ_B by taking parts of states σ and σ_A such that if those parts are combined then the right hand side can be proved. The part removed from σ is exactly σ_{foot} described earlier. In particular σ is split into $\sigma' \uplus \sigma_{foot}$, σ_A is

split into $\sigma'_A \uplus \sigma''_A$ such that $\sigma_B = \sigma_{foot} \uplus \sigma''_A$ and $\sigma_B \models B$. σ' is the remaining current state after the **package** is finished.

σ_B is initialized as a state without any assumptions on heap locations, no permission and the same assumptions on the local variables as σ (and σ_A). Whenever there's an impure expression (see Section 2.3.3 for pure and impure expressions in Silver), let's say an accessibility predicate, the algorithm tries to transfer the needed permission from σ_A and if necessary also from σ along with all the heap information on that location to σ_B. For pure expressions the algorithm checks if the state σ_B it has constructed so far satisfies the expression.

As an example consider the statement

```
package acc(x.f,1/2) --* acc(x.f,3/4)
```

in a state where `acc(x.f)` holds. Then the footprint satisfies `acc(x.f,1/4)` since one half of the permission to $x.f$ can be taken from σ_A.

2.4.2 Apply

Once a wand is held in the program state (due to a precondition, a package or an inhale) then a wand can be applied using the **apply** statement. The apply statement basically employs the inference rule already presented in section 2.1.2:

$$\frac{A * (A \mathrel{-\!\!*} B)}{B}$$

apply $A \mathrel{-\!\!*} B$ exhales $A \mathrel{-\!\!*} B$ and A followed by an inhale of B. So the **apply** is successful if the wand instance is held in the current state and if the left hand side can be exhaled in the current state. We note that usually after an exhale all information is lost to all heap locations to which no permission is held. But in this case a verifier may choose to check if there's no permission and then remove the heap information only after the **inhale** of B to retain information. We'll come back to this point later when discussing the translation of **apply** in Carbon.

It is important to note that after applying a wand we can't in general deduce that the resulting state satisfies A. One reason is that A may be stronger than B. Another reason is that A may express information about some state in a different way from B, for instance by using different predicates (see Section 2.3.4), which is not (automatically) equivalent in the verifier.

2.4.3 Ghost Operations

When packaging a wand we may want to rearrange permissions by folding or unfolding predicate instances or package/apply wands. With the operations we have presented up to this point, this is not possible.

Consider the following operation:

```
(acc(x.f) && x.f==2) --* acc(P(x,y))
```

where the body of P is given by `acc(x.f) && x.f==2 && acc(y.f)` and `acc(y.f)` holds in the current state. Then in an ideal case the verifier will take the full permission to *y.f* as footprint which together with the left hand side of the wand can be folded into *P(x,y)*. But as mentioned in Section 2.3.4 the verifier in general doesn't **fold** a predicate's body into a predicate instance without any direction. Therefore we need some kind of ghost operation indicating to the verifier that *P(x,y)* should be folded.

In Silver there are four such different ghost operations which can only be used on the right hand side of a package: **packaging**, **folding**, **unfolding**, **applying**. The syntax for `package A --* G` in Silver is given by (taken directly from [16]):

```
G ::= A | folding P(e) in G       | unfolding P(e) in G
        packaging A --* G in G | applying A --* A in G
```

If `package A --* G` is successful then the wand instance `A --* nested(G)` is added to the current state where `nested(G)` is the ghost-operation-free assertion nested inside `G`. For example the nested assertion of

```
packaging A --* B in (unfolding P(x) in acc(x.f)&&x.f==2))
```

is given by `acc(x.f)&&x.f==2`.

Our previously mentioned example can now be written as follows:

```
package acc(x.f)&&x.f==2 --* (folding P(x,y) in P(x,y))
```

These ghost operations in a sense indicate to the verifier how to prove the right hand side given the left hand side (and the current state).

Unfolding Ambiguity. It's important to point out that there's a difference between a **unfolding** magic wand ghost operation and an **unfolding** expression (which was briefly introduced in 2.3.4). In our discussion and also in the final implementation we always regard **unfolding** as a magic wand ghost operation in case there is an ambiguity.

For example in the following wand, the **unfolding** is clearly a magic wand ghost operation since its body is a ghost operation (where A1,A2, B are assertions and G is either an assertion or a ghost operation).

```
package A1 --* (unfolding P(x) in (applying (A2 --* B) in G)
    )
```

In contrast, in the following wand, the **unfolding** is clearly an expression since the syntax doesn't permit a ghost operation at that position (where e is a pure expression):

```
package A1 --* (acc(P(x)) && (unfolding P(x) in e))
```

But in the following wand there's a potential ambiguity (where e is a pure expression):

```
package A1 --* (unfolding P(x) in e)
```

The **unfolding** could be either a magic wand ghost operation or an **unfolding** expression (more specifically an **unfolding** assertion). We always interpret it as an **unfolding** magic wand ghost operation. One of the reasons for this decision is that `unfolding P(x) in e` can't be self-framing (see Section 2.3.2), since no permission to *P(x)* is mentioned. Therefore interpreting it as an **unfolding** expression would directly violate Silver's restriction that the right hand side of a wand must be self-framing, even though situations may arise where one is interested in just having the body e on the right hand side of the wand. In an earlier version of the tool, Silver interpreted `unfolding P(x) in e` where e is a pure expression as an **unfolding** assertion.

Footprint computation algorithm (with ghost operations)**.** The footprint computation algorithm briefly described earlier (without ghost operations) becomes a bit more involved in the presence of ghost operations. Without ghost operations the algorithm always had exactly two states to transfer information from to construct the state which satisfies the right hand side. With ghost operations there can be multiple states to transfer from because:

1. Due to the **packaging** ghost operation there can be multiple left hand sides where each is associated to a state.

2. The footprint computation algorithm carries along a state σ_{ops} (initially empty) holding information gained from the execution of the ghost operations encountered. Whenever a ghost operation is executed, first

a state σ_{used} is constructed by transferring the necessary permissions, where the corresponding locations may have to satisfy certain properties, to be able to execute the ghost operation. In a second step the ghost operation is executed in σ_{used}. The resulting state σ'_{used} is then combined with σ_{ops} to update the state holding information from the encountered ghost operations (i.e. the updated state is given by $\sigma_{ops} \uplus \sigma'_{used}$). This state can also be used by the footprint computation algorithm to transfer information from, otherwise there would be no point in having ghost operations at all, since the resulting information gained wouldn't be utilized.

It is important to note that for each **`packaging`** operation as well as for the actual **`package`** there is exactly one such state σ_{ops} and the information in σ'_{used} is always added to the state σ_{ops} corresponding to the most recently encountered **`package`**/**`packaging`** operation. The reason for this is that additional information gained inside a **`packaging`** operation is not allowed to be visible once that operation is finished. Hence the different states σ_{ops} provide a way of scoping the information.

We outline how the **`applying`** `A --* B` **`in`** `G` ghost operations works: we'll assume that σ_{ops} is the state holding information gained from the execution of ghost operations encountered after the most recently encountered **`package`**/**`packaging`** operation. Let's assume at the point when the algorithm needs to execute the **`applying`** ghost operation the input states (i.e. the left hand sides + states holding information gained from the execution of ghost operations + the current state) are stored on a stack $\overline{\sigma}$, where the more recent states are closer to the top of the stack (hence the current state is always on the bottom).

First a state σ_{used} is created just with the local variable assumptions and the algorithm attempts to transfer information from states on $\overline{\sigma}$ to σ_{used} (giving preference to states higher on the stack) to satisfy `A && (A --* B)`. Then `A --* B` is applied in state σ_{used}, the resulting state being σ'_{used}. This is guaranteed to work if the transfer succeeded in obtaining the necessary permissions. Finally the algorithm continues with `G` where the state holding the information from the executed ghost operations is updated to $\sigma_{ops} \uplus \sigma'_{used}$.

2.5 Boogie

Boogie is an intermediate verification language with the goal of making verification generation simple. It is described in [1] and the documentation is given in [11]. Generally programs and their specifications in various lan-

guages are encoded into Boogie programs from which verification conditions are generated for a solver (such as Z3 [6]) to verify.

To reason in Boogie one generally thinks of the sets of execution traces specified by the imperative constructs in the language. We present some of the relevant constructs in Boogie needed for the discussion later on.

2.5.1 Assume Statement

Assumptions in Boogie can be added to the set of execution traces using `assume e` where e is a boolean expression in Boogie. The set of traces after `assume e` are exactly those traces which existed before the statement and for which e holds. If a trace existed before the statement but e didn't hold, then this trace doesn't exist anymore after the `assume e`.

2.5.2 Assert Statement

Verification checks can be added using `assert e` where e is a boolean expression in Boogie. The set of traces after this statement are the same as those after `assume e`, but the difference is that the traces up to `assert e` where e doesn't hold are explicitly characterized as "wrong" traces. Whenever the prover is not sure that there is no "wrong" trace with respect to an `assert` statement then a verification error is output.

2.5.3 Havoc Statement

`havoc v` for any local variable v assigns an arbitrary value (of the same type as v) to v. To restrict the choices for the value assigned by `havoc` often an `assume` statement is used. For example `havoc a; assume a >= 0` where a is an integer ensures a positive integer is assigned to a.

2.5.4 Maps

Boogie supports (optionally polymorphic) map types. A map is updateable. Multiple domain types can be given but just a single range type.

The following example is inspired by [8] to show a version of how a heap is encoded in Boogie by Carbon.

Listing 1: Simplified encoding of heap in boogie

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

The type *Ref* represents Silver references and the type *Field a* represents a Silver field of type *a*, where *a* is a type parameter which makes the *HeapType* polymorphic. *HeapType* maps pairs of references and fields to values of the same type as the field.

2.5.5 Triggers

Certain SMT solvers such as Z3 use *triggers*, to limit the amount of ways a quantified formula can be instantiated. Boogie provides a way to specify triggers in quantified formulas which are then used by the SMT solver, instead of the triggers generated by the solver itself.

Consider the following syntax for a quantified formula in Boogie (example taken from [11]):

```
∀x:T :: {f(x)} g(f(x)) < 100
```

The trigger in the quantified formula is `f(x)` and directs the verifier to just instantiate the formula with values x for which the term `f(x)` showed up in the proof context.

2.6 Carbon

Carbon is one of the two back-end verifiers for Silver in the Viper project. It encodes a Silver program into Boogie from which verification conditions are generated for Z3 and it's implemented in Scala. We present the relevant parts of Carbon's Silver to Boogie translation, a more detailed documentation is available in [8].

2.6.1 State Representation

Carbon models a program state with two global maps in Boogie. One map represents the heap and maps pairs of references and fields to the corresponding heap value. The map representing the heap also stores information about abstract predicates (the details of the abstract predicate encoding which Carbon uses are given in [7]).

The second map is the mask. The mask's role is to record the permission for each heap location held, so it basically is a map from heap location/predicate to a permission value (which is represented by a *real* in Boogie).

Note that after each modification to the heap (or mask) the new heap is different than the old heap. Carbon generates various axioms which need to quantify over states. Since in Boogie this essentially is a quantification over heaps and masks, Carbon often needs to make sure that the verifier finally

only instantiates heaps and masks which together represent a program state. This is done by assuming a predicate `state(Heap,Mask)` to pair the correct heaps and masks.

The current heap and mask maps in Boogie are given by the global variables `Heap` of type `HeapType` and `Mask` of type `MaskType`. `HeapType` and `MaskType` are types defined appropriately. Carbon can also work with temporary states by using local variables for the corresponding heap and mask maps.

2.6.2 Field Types

Fields in Silver are encoded as constants in Boogie of some field type. Abstract predicates are also encoded as fields so that they can be tracked in the heap or mask. Since predicate fields are of a totally different nature as Silver fields Carbon additionally defines different types for the kind a field can have.

The type declaration for a *field type* is given by `type Field A B` (instead of just `type Field A` as in Listing 1). The parameter A denotes the kind of field it is and the parameter B denotes the type of the value stored in the field. A Silver integer field would be encoded as a constant of type `Field NormalField int`, where *NormalField* is the type used to denote that it is a Silver field.

2.6.3 Definedness Check

Before a Silver expression is translated and used in the Boogie encoding, Carbon in general first generates a definedness check in Boogie. One can roughly say that an expression e is well-defined if for each of the heap locations on which e depends, the following holds:

- for each of the heap locations on which e depends the receiver is guaranteed to be non-null and there is non-zero permission to those heap locations
- the preconditions of all function expressions in e are satisfied

The definedness check generated for the Silver statement

```
assert (b ? x.f == 2 : y.f==3)
```

is given by

```
if (b) {
  assert x != null;
  assert Mask[x,f] > 0;
} else {
  assert y != null;
  assert Mask[y,f] > 0;
}
```

Self Framing Check. In some cases Carbon must check if an assertion A is self-framing. To check this all the permissions specified in A are inhaled using a new mask, which has no permission to any location and also a heap which has no information to any location. Then a definedness check with respect to this mask and heap is performed.

3 Representing and Applying Wands in Boogie

To be able to encode the magic wand operations described in 2.4 using the Carbon verifier, we need to be able to add and track wand instances in the final Boogie encoding. In this section we show how we achieve this in our encoding. We additionally show how we encode the `apply` operation. We denote ⟦stmt⟧ as the translation of a source statement in Silver into Boogie by Carbon, where expressions are evaluated in the main state.

3.1 Wand Representation

Our approach for tracking magic wand instances is similar to the way Carbon deals with tracking predicate instances.

3.1.1 Shapes and Holes

Before we present the way we track wands in Boogie, we define what the shape of a wand is.

Definition 3 *The shape of a wand $A \mathrel{-\!\!*} B$ is the structure of the wand that remains after removing all subexpressions of the wand which are* `old` *expressions or heap independent. We call the entities of the shape of a wand which remain after removing the mentioned subexpressions of the wand the holes of the shape of the wand.*

For example the shape of the wand

```
acc(x.f,1/2)&&x.f==2 --* true
```

is given by

```
acc(_:Ref.f,_:Perm)&&_:Ref.f==_:Int --* _:Bool
```

and the holes of the shape of the wand are given by `_:Ref.f`, `_:Perm`, `_:Ref.f`, `_:Bool`, where $\_ : \tau$ denotes that the "hole" has type τ. We note that this notion of wand shapes was not developed as part of this thesis but was already used by the Silicon verifier (there's no official documentation of it known to us, so we introduced it formally here).

The point is now that if two wand instances in Silver have the same shapes and the expressions of the two wands which fill the holes of the shape evaluate to the same values, then we can be sure that the two wand instances are semantically identical.

3.1.2 Tracking Wands in Boogie

For each wand shape (as defined in Definition 3) we create a unique function `wandID` in Boogie and a unique type `Wand_ShapeID`, where the arguments of `wandID` are exactly the holes of the corresponding wand shape (see Definition 3). The returned value of the function applied to some parameters represents the wand of the given shape with the holes filled by the parameters and is of field type `Field Wand_ShapeID int` (see section 2.6.2 to understand how Carbon encodes field types).

For example for a wand with shape `acc(_:Ref.f,_:Perm) --* _:Bool` we add the following declarations in Boogie:

Listing 2: Predicate representing wand shape in Boogie

```
type Wand_Shape1;
function wand1(arg1:Ref, arg2:Perm, arg3:bool): Field
   Wand_Shape1 int;
```

Now assuming x is a local variable of type *Ref* in Boogie then the returned value of `wand1(x,1/2,true)` represents the wand `acc(x,1/2) --* true`. We use the mask (see Section 2.6.1) in Boogie to track the amount of wand instances of any wand $A \mathrel{-\!\!*} B$ that the current state holds. For the specific wand given, the amount of wand instances held are given by

```
Mask[null,wand1(x,1/2,true)]
```

After inhaling the wand $A \mathrel{-\!\!*} B$ twice in succession, two wand instances of $A \mathrel{-\!\!*} B$ will be held in the Boogie program state. We don't support a "fractional number" of wand instances in contrast to the way predicates are handled in Silver. It's not clear how associating fractional values to the amount of wand instances held would help in practice.

Consider the following Silver program:

```
package acc(x.f) --* true
y := x
b := true
exhale acc(y.f) --* b
```

After the `package` we add the wand instance `acc(x.f) --* true` to the state, which in our representation is given by `wand1(x,1,true)`. The `exhale` should work because `acc(y.f) --* b` is the same wand instance as the one we packaged, since the wand shapes of the two wands are the same and the expressions filling the holes evaluate to the same values. In our encoding the wand instance `acc(y.f) --* b` is represented by `wand1(y,1,b)`. Since `y == x` and `b == true` can be verified, the verifier concludes that

```
wand1(x,1,true) == wand1(y,,1,b)
```

Hence the verifier will notice that the two wand instances are identical.

Our current implementation stores the already-seen wand shapes into a data structure. Once a new wand instance in the Silver AST is traversed, we check if its shape has already been recorded: if not we store a new shape.

3.2 Inhaling a Wand

For the translation of **inhale** `A --* B` first a self-framedness check as described in section 2.6.3 is generated for A and B. Then we increment the value for the Mask's value at the location of the wand field corresponding to the wand:

Listing 3: Translation of **inhale** `A--*B`

```
self-framedness check for A
self-framedness check for B

//Assume that w is the wand field representing A --* B
Mask[null,w] += 1;
```

Note that in the actual Boogie output `w` will be written as a function application as described in section 3.1.2.

3.3 Exhaling a Wand

We give the pseudocode for the **exhale** of a wand.

Listing 4: Translation of **exhale** `A --* B`

```
self-framedness check for A
self-framedness check for B

//Assume that w is the wand field representing A --* B
assert Mask[null,w] >= 1;
Mask[null,w] -= 1;
```

3.4 Translation of the apply statement

We introduced the **apply** statement in section 2.4.2.
One possibility to translate **apply** `A --* B` into Boogie is as follows:

Listing 5: Potential incompleteness in apply operation

```
1 package acc(x.f) --* acc(x.f)
2 inhale acc(x.f) && x.f==2
3 apply acc(x.f) --* acc(x.f)
4 assert x.f==2
```

Listing 6: Translation of apply statement

```
var newHeap;
havoc newHeap;
⟦exhale′ A --* B⟧
⟦exhale′ A⟧
⟦inhale B⟧
newHeap ⟝^Mask Heap
Heap := newHeap;
```

$$\llbracket \texttt{apply A --* B} \rrbracket = \llbracket \texttt{exhale A --* B} \rrbracket ; \llbracket \texttt{exhale A} \rrbracket ; \llbracket \texttt{inhale B} \rrbracket ;$$

This would certainly be sound and is also one solution suggested in [16]. There is an incompleteness associated with this approach. Consider the Silver program given in Listing 5.

The wand in the **package** of Listing 5 in general doesn't have any practical use, but it allows us to show a more or less minimal example of the problem. It's clear that the operations on lines 1-3 should be successful. Now if we translate the **apply** as just outlined before, the assertion on line 4 will fail. The reason is that the translation of **exhale acc(x.f)** havocs the heap information to *x.f*. So the information `x.f==2` is lost even though we regain permission right after the exhale. To overcome this we don't havoc the information when translating **exhale**, but we perform all other steps, i.e. checking if all pure assertions hold, if for every accessibility predicate there's enough permission held and removing the permissions specified in the accessibility predicates. Let's call this "modified" version of exhale **exhale′** (defined as in [7]). Only after the **inhale** of the right hand side we havoc the heap locations to which we don't have any permission anymore.

Our translation of **apply** is then given in Listing 6. We use similar notation as in [7] and we interpret **exhale′** as a Silver statement, even though it can't appear in actual Silver programs. Finally `newHeap` $\longmapsto^{Mask}$ `Heap` makes sure `newHeap` agrees with all values in `Heap` where `Mask` has non-zero permission (or known-folded permission, see [7] for details).

4 Encoding the package operation in Boogie

The `package A --* B` operation, described in Section 2.4.1, tries to remove a part of the current state, called the footprint σ_{foot} of the wand, such that $\sigma_{foot} \models A \mathrel{-\!\!*} B$. The footprint computation algorithm is described in a generic fashion in [16]. In this section we first take a deeper look at the **package** operation in general, which leads to an observation that is important for possible encodings for operations needed in the footprint computation algorithm. We then move on to discuss various encodings in Boogie for the operations in the algorithm and take a closer look at their limitations.

In the paper a set of basic operations (in figure 4 of the paper) along with their functionality are defined for states, with which the authors then define the algorithm in detail. We'll refer to these basic operations as *basic state operations*. So in theory it would suffice for us to just present how we encode these operations into Boogie and then the encoding of the footprint computation algorithm would be directly given by the paper's specification which only relies on those operations. We won't completely follow that route because:

- The paper restricts itself to magic wands where the permission value for each accessibility predicate that occurs must be 1 (i.e. full permission). In our discussion we'll also support fractional permissions. This means the set of basic state operations that we use is slightly different than the one in the paper and the algorithm is modified at certain points.
- We need to change the algorithm a bit to deal with states that may be inconsistent (i.e. a state which under its assumptions can't exist).

We denote $\llbracket \mathrm{e} \rrbracket$ as the Boogie code emitted to translate the Silver expression e evaluated in the main state represented by the global variables `Heap,Mask` as described in 2.6.1. We denote $\llbracket \mathrm{e} \rrbracket_\sigma$ as the Boogie code emitted to translate the Silver expression e evaluated in state σ represented by local variables storing the heap and mask. We denote $[\![\mathrm{stmt}]\!]$ as the translation of a source statement in Silver into Boogie by Carbon, where expressions are evaluated in the main state and we denote $[\![\mathrm{stmt}]\!]_\sigma$ as the translation of a source statement in Silver into Boogie by Carbon, where expressions are evaluated in σ. We use σ_{ID}, where ID may be an arbitrary identifier, to denote a state and $\overline{\sigma_i}$ to denote a stack of states.

4.1 High level view of the footprint computation algorithm

We already gave an informal view on the footprint computation algorithm in section 2.4.1. Now we want to give a brief description of the functions described in [16] which are used to describe the algorithm. The function bodies depend only on the basic state operations.

transfer. *transfer*($\overline{\sigma_i}$,σ_{used},e), where e is either an accessibility predicate or a magic wand, transfers permissions/a magic wand along with the information held from the states in $\overline{\sigma_i}$ to σ_{used} such that e is held in σ_{used}. It always tries to use states as close as possible to the top of the stack. This is the main function that we'll need to adjust. The modified states ($\overline{\sigma_i}'$,σ'_{used}) are returned.

exhale_ext. *exhale_ext*($\overline{\sigma_i}$,σ_{used},A) recursively goes through assertion A and for every pure subassertion of A checks if it holds in σ_{used} and for every impure assertion returns *transfer*($\overline{\sigma_i}$,σ_{used},A). In general *exhale_ext*($\overline{\sigma_i}$,σ_{used},A) returns ($\overline{\sigma_i}'$,σ'_{used}) which are basically the modified versions of the argument states (the states change due to the different calls to *transfer*).

exec. *exec*($\overline{\sigma_i}$,σ_{ops},G) for a ghost operation G executes it and recursively goes over the body of the ghost operation and for an assertion G returns the output of *exhale_ext* applied with appropriate arguments. σ_{ops} is the state which has accumulated the information gained from the execution of all the ghost operations encountered so far.

All these functions are recursive over the stack of states and/or the structure of the assertions involved. None of them appear explicitly in our Boogie encoding as functions: the recursions are unrolled and the appropriate Boogie code is emitted for the operations. The return values are implicit in the sense that the emitted code modifies the states through updates.

4.2 Inconsistent states and trivial wands

The paper [16] doesn't explicitly discuss how the footprint computation algorithm behaves in the presence of inconsistent states. We take a deeper look at cases in which they do occur and discuss the desired behaviour in terms of the footprint. Along the way we'll also gain a better understanding of how the footprint computation algorithm works.

Definition 4 *We call a state σ inconsistent if $\sigma \models$ false*

Recall that $\sigma \models A \mathrel{-\!\!*} B$ (see 2.1.2) holds, if for any hypothetical state σ' that is compatible (see Definition 1) to σ and satisfies A then $\sigma \uplus \sigma'$ (see Definition 2) must satisfy B. That means $A \mathrel{-\!\!*} B$ holds trivially if there is no such state σ'. In such a case we call $A \mathrel{-\!\!*} B$ trivial with respect to σ.

Definition 5 *If* $\forall\sigma'.(\sigma' \perp \sigma \Rightarrow \sigma' \not\models A)$ *holds, then we call* $A \mathrel{-\!\!*} B$ *trivial with respect to* σ*.*

From here on we'll denote the current state as σ_{cur} (i.e. the state right before the **package** in all scenarios) and an arbitrary state satisfying the left hand side A of a wand $A \mathrel{-\!\!*} B$ we denote as σ_A.

Scenario 1: Inconsistent left hand side and pure right hand side

Listing 7: Inconsistency Scenario 1

```
package false --* false
```

The wand in Listing 7 is trivial with respect to every state because there is no consistent state that can satisfy *false*. Actually it's even equivalent to the implication *false* ⇒ *false*. So when packaging such a wand we definitely want to pick an empty footprint. Let's see how the footprint computation algorithm behaves for this example.

An inconsistent state will show up in the algorithm right at the beginning when the left hand side is inhaled into some new empty state (no heap information, no permissions, no assumptions). This constructed state corresponds to σ_A. In a next step an empty state will be created in which the right hand side *false* is asserted. Since the empty state certainly doesn't satisfy *false*, the package fails. This is not the behaviour that we want. We would like the algorithm to use the information that σ_A is inconsistent and then conclude that the package works with the empty footprint.

Scenario 2: Inconsistent left hand side and impure right hand side

Listing 8: Inconsistency Scenario 2

```
package false --* acc(x.f)
```

The wand in Listing 8 is trivial with respect to every state for the same reasons as the wand before. This means when packaging such a wand we'd ideally want to pick an empty footprint. The algorithm behaves the same as the wand in Listing 7 until it reaches the right hand side. There the algorithm tries to transfer permission for *x.f* from σ_A and if needed from σ_{cur} to satisfy the accessibility predicate. Now σ_A doesn't contain any permissions but it

is inconsistent. So should it be possible to remove the required permission from σ_A? From a purely logical point of view any assertion in an inconsistent state holds. If that were the case in the algorithm then we'd exactly obtain the wanted end result, namely a succesful package with an empty footprint.

Another way to achieve this result, would be to say that if σ_A or σ_{cur} is inconsistent right before the transfer, then the transfer should just work without having to explicitly remove and add permission.

On the other hand if the query to σ_A returns that there is no permission, then the algorithm would try to remove permission from σ_{cur}. This would mean that if there is the required permission to $x.f$ in σ_{cur}, then the package would succeed with a non-empty footprint and otherwise the package would fail. Both of these results are not wanted (they are sound but incomplete).

Scenario 3: Inconsistent combination of consistent states

Listing 9: Inconsistency Scenario 3

```
//σ_cur satisfies acc(x.f) && x.f==2
package (acc(x.f,1/2) && x.f==3) --* acc(x.f)
```

The wand in Listing 9 is not trivial with respect to every state. For example if `acc(x.f,1/2) && x.f==3` holds in a state σ, then the wand is not necessarily trivial with respect to σ. So it's not clear what the ideal footprint should be. The footprint computation algorithm will transfer permission from both σ_A and σ_{cur} to the state σ_{used} that it's constructing. The transfer and hence the package will succeed. The footprint satisfies `acc(x.f,1/2) && x.f==2` (see the constraint on σ_{cur} in Listing 9). We note that every state which satisfies the left hand side can't be compatible to the footprint, since they must agree on the heap values to which both have non-zero permission. We conclude that the wand is trivial with respect to the footprint, i.e. the wand can never be applied in any consistent state after the package.

Additionally there's something interesting that happens in the transfer. In a transfer of $x.f$ from σ to σ' all the knowledge of $x.f$ known in σ is also transferred to σ'. This means that once the transfer from states σ_A and σ_{cur} to σ_{used} is finished, `acc(x.f) && x.f==2 && x.f==3` will hold in σ_{used}. Hence σ_{used} is inconsistent[1]. So as in scenarios 2 and 3 an inconsistent state shows up in the footprint computation algorithm for a trivial wand with respect to the footprint chosen.

[1]We note that in theory after each modification of a state we get a new state (i.e. states are immutable), but in our discussion we sometimes just refer to a state before and after an update with the same name for simplicity. The context should make it clear which version of the state we are referring to.

What we also learn from this example is that to make the package work, we must remove permission from the current state even though we encounter an inconsistent state, since otherwise the wand may not be trivial with respect to the footprint chosen and hence may be applied unsoundly. For instance in Listing 9 we notice an inconsistent state after transferring half permission to *x.f* from σ_{cur} to σ_{used}. Now suppose after noticing this inconsistency we would revert this transfer and continue as if the transfer was successful and hence the **package** would be successful, then the packaged wand wouldn't hold in the chosen footprint (which is empty). This is certainly unsound with regards to the specification of the **package**. To make it even clearer: this would mean that the state after the **package** still has full permission to *x.f*, which means the value 3 can be assigned to *x.f* right after the **package**. This in turn would lead to a state in which the application of the packaged wand would succeed, even though the packaged wand doesn't hold in the corresponding footprint. So it shows that the presence of an inconsistent state doesn't warrant reverting an earlier action. Hence it is important for our encoding to preserve the modifications made by the algorithm that lead to an inconsistency and once the inconsistency is reached to basically be able to assert any expression in the inconsistent state. To make this point clear consider the augmented wand in Listing 10:

Listing 10: Inconsistency Scenario 3 Augmented

```
//σ_cur satisfies acc(x.f)&&x.f==2&&acc(y.f)
package acc(x.f,1/2)&&x.f==3 --* acc(x.f)&&acc(y.f)
```

Then until `acc(y.f)` is reached the algorithm performs the exact same steps as in the scenario given in Listing 9, but now that σ_{used} is inconsistent we want the transfer of `acc(y.f)` to go through without having to remove further permissions from σ_{cur}. This is certainly sound since the algorithm has already chosen a footprint which makes the wand trivial.

Before finishing off this scenario we just want add a second basic type of package that leads to an inconsistency of a state in the algorithm through construction of a state using just consistent states.

Listing 11: Inconsistency Scenario 3 too much permission

```
//σ_cur satisfies acc(x.f)
package acc(x.f) --* acc(x.f)&&acc(x.f)
```

In Listing 11 the footprint is given by a state satisfying `acc(x.f)`. It's clear that the wand is trivial with respect to the chosen footprint. Again σ_{used} will be inconsistent at the end of the footprint computation algorithm,

but this time it's because σ_{used} holds more than full permission to *x.f*, which can't be the case for a consistent state.

Scenario 4: Inconsistency after execution of ghost operation

Scenarios 1-3 don't involve ghost operations. In this final scenario we want show how right after the execution of a ghost operation an inconsistent state is created.

Listing 12: Inconsistency when executing ghost operation

```
predicate P(x:Ref) {
    acc(x.f)
}

//σ_cur satisfies acc(x.f)
package acc(P(x),2) --*
    (unfolding P(x) in (unfolding P(x) in false))
```

In Listing 12 we don't write the **package** statement inside a method to avoid the unnecessary clutter (as in all other scenarios, but here we have a predicate). The first thing we notice is that the left hand side of the wand in Listing 12 is inconsistent in theory, since holding the predicate P twice means holding more than full permission to *x.f*. Since the verifiers in general don't treat predicate instances and their corresponding bodies interchangeably, the verifier won't notice that the left hand side is inconsistent without any direction (see Section 2.3.4).

The footprint computation algorithm carries along a state σ_{ops} which accumulates all the information which was gained through the execution of each ghost operation encountered (this is done in the *exec* function presented in Section 4.1). Whenever a ghost operation is executed, a state σ_G is created which holds the information gained solely from the execution of that ghost operation. Finally the state carried along is updated from σ_{ops} to $\sigma_{ops} \uplus \sigma_G$ (see Section 2.4.3 for a more detailed view). In Listing 12 after the execution of the first ghost operation `acc(x.f)` will hold in this state, which isn't inconsistent. After the execution of the second ghost operation `acc(x.f) && acc(x.f)` will hold in this state, which is inconsistent.

So one can say that the ghost operations make the algorithm notice that the left hand side is inconsistent and so we expect the **package** in Listing 12 to succeed with an empty footprint. If the same wand is packaged without any ghost operations, then the **package** fails.

Final Conclusions

Scenarios 1-4 show the various ways inconsistencies can occur in states and each time the algorithm records an inconsistent state it seems as though the wand is trivial with respect to the footprint picked up to that point.

We now generalize this observation to a lemma.

Lemma 1 *Let $\overline{\sigma_i}$ be the stack of states carried along in the footprint computation algorithm and let σ_{ops} be the state which is carried along to store information gained from the execution of ghost operations encountered after the most recent package or packaging operation. Then the following statements hold:*

i) If any state σ on the stack is inconsistent then it is sound for any operation to succeed without affecting any state, while σ is on the stack.
ii)If σ_{ops} is inconsistent then it is sound for any operation to succeed without affecting any state, while σ_{ops} is the state which is carried along to store information from the execution of ghost operations.
iii) If a state σ is inconsistent then it is sound for any operation applied in σ to succeed without affecting any state.

To clarify: Lemma 1 states that if, for example, σ is inconsistent and is part of the stack carried along then any transfer from the stack of states to any other state succeeds without actually removing/adding any permissions. If σ is later removed from the stack, then Lemma 1 doesn't necessarily apply any more. What Lemma 1 means by stating that an operation succeeds is not that the resulting effect of the operation is the same as in the usual case, but rather that it is sound to in a sense skip the operation and continue with the remaining part of footprint computation algorithm.

We don't provide a formal proof for Lemma 1 but we give a strong intuition as to why it holds. Let σ_{cur} be the state in the program right before the **`package`** operation. If one inspects the footprint computation algorithm closely, then one notices that all the states on the stack correspond to one of the following points:

- σ_{cur}
- one of the left hand sides of the wands (there may be multiple due to nested **`packaging`** ghost operations)
- a state that was constructed in the course of the footprint computation algorithm using only (potentially older) versions of the states on the current stack

That means theoretically the information of every state on the stack originates from the initial left hand sides and σ_{cur}, because the starting stack only consists of σ_{cur} and σ_A, where σ_A is a state satisfying the left hand side of the wand in the actual **package**. Using this knowledge we informally show Lemma 1 i).

Assume state σ' on the stack $\overline{\sigma_i}$ is inconsistent, then we can conclude that a combination of information from the initial left hand sides and σ_{cur} is inconsistent. Note that the information transferred from σ_{cur} to σ' is part of the footprint σ_{foot}.

Case 1: None of the states on the stack corresponds to a left hand side of a wand in a **packaging** ghost operation

Hence a combination of information from σ_{foot} and σ_A is inconsistent. But this directly implies that the wand is trivial with respect to the footprint chosen up to this point. Therefore we can just "abort" further computations made by the algorithm and let the **package** succeed. We conclude the Lemma 1 i) holds.

Case 2: A state on the stack corresponds to a left hand side of a wand in a **packaging** ghost operation

The situation gets a bit trickier if there are **packaging** ghost operations. The execution of **packaging** `A1--*G1` **in** `G2` for an assertion `A1` and ghost operations/assertions `G1`, `G2` first basically executes **package** `A1--*G1`. The main difference to the actual **package** operation performed in the beginning is that the stack is already filled with multiple states. Assume that the footprint computation algorithm is executing the **package** `A1--*G1` operation and this corresponds to the most recent **packaging** operation that is being executed. Further assume we just let all remaining operations in **package** `A1--*G1` succeed without removing any permission from any state (since we just encountered an inconsistent state). The point now is that to get `nested(G1)` in the state after the original **package** is over, one needs a state that satisfies all left hand sides on the stack. But the algorithm has already chosen a footprint which makes this impossible (due to the inconsistency that arises).

This argument is not quite general enough, since one may use the **applying** ghost operation somewhere in `G2`. To show the claim in this case, one would have to generalize what "footprint" means for a wand in a **packaging** operation. The "extended footprint" of such a wand potentially consists of information from σ_{cur} or any of the earlier left hand sides. Now since the

inconsistency basically shows that this "extended footprint" is not compatible with any state satisfying *A1* this means we won't be able to use the **applying** ghost operation successfully on this wand in general. We leave out the details, but our explanation should suffice for an intuition as to why Lemma 1 i) holds.

Showing Lemma 1 ii),iii) can be reduced to a similar argument as the one that we just made. The footprint computation algorithm just performs operations on states where the information was transferred from σ_{cur} and the left hand sides, i.e. we can use the same argument for Lemma 1 iii). σ_{ops} is just a collection of information gained through the execution of ghost operations. Since the information gained from a ghost operation is information gained from σ_{cur} and the left hand sides we can again use a similar argument for Lemma 1 ii).

4.3 Representation of states

Since the footprint computation algorithm deals with many different states and operations on those, we must show how we encode those states in Boogie. In Section 2.6.1 we already showed how Carbon represents states in Boogie. There's an issue with using that representation for our problem. Consider how one might encode adding an assumption e to a state σ_1. Using the usual state representation we would associate a heap map `Heap1` and a mask map `Mask1` to σ_1 and then we would encode adding the assumption using the statement **`assume`** `e` where e is evaluated in `Heap1`. But the problem is that these assumptions are now part of the global verification state, so if e is, for example, *false* then we assume *false* and the verification of anything that follows goes through which is certainly unsound. So we need some kind of way to collect these assumptions for each state separately. We do this by introducing a Boolean variable in Boogie for each state, which collects all assumptions about the corresponding state. This means if `b1` is the Boolean variable associated with σ_1 then we add the assumption e to σ_1 by associating a new boolean variable `bnew1` with σ_1 where `bnew1 == (b1 && e)`.

The reason why Carbon's state representation is sufficient without magic wand support is because such inconsistent assumptions which shouldn't affect the global state never occur, but in our case that doesn't hold, as we saw in Section 4.2.

So for the footprint computation algorithm we represent a state σ_{ID} in Boogie by a heap map `HeapID`, a mask map `MaskID` and a boolean variable `bID`.

4.4 Encoding of state operations

We now give an encoding for the state operations that are used in our encoding or are interesting for the discussion of the encoding. Many of the state operations are given by or are slight modifications of the *basic state operations* in [16]. In theory the states are immutable and the majority of the state operations return updated states. In our case we only create a new state explicitly at the beginning and every operation applied to that state is expressed using a statement (if a state has to be returned in theory) or an expression, instead of explicitly returning a new state. So we'll just give the Boogie statement/expression emitted for each state operation.

Let σ be a state which is encoded using a heap map `Heap`, a mask map `Mask` and Boolean variable b (`Heap` and `Mask` are not necessarily the global variables used for the actual state described in 2.6.1, σ represents an arbitrary state). σ_{ID} is a state encoded as described in Section 4.3. Let $\overline{\sigma_i}$ be a stack of states. We denote the conjunction of all boolean variables of the states in $\overline{\sigma_i}$ as $\overline{b_i}$. Let `ZeroMask` denote the mask in Boogie which has zero permission to every location. Furthermore we denote an arbitrary expression as e and an arbitrary predicate which takes a single argument as P. A, B denote (possibly impure, but self-framing) assertions. Let v be a reference type value evaluated in some state, let f be a field and p a fractional permission value also evaluated in some state.

identicalOnKnownLocs(Heap1,HeapUnion,Mask1) is a predicate defined by Carbon in Boogie. If it returns true then *HeapUnion* agrees on all values with *Heap1* where *Mask1* has non-zero permission. It also makes sure that the heaps agree on known-folded locations (see [7]).

sumMask(MaskUnion, Mask1,Mask2) is a predicate defined by Carbon in Boogie. If it returns true then it states that the permission to each location in *MaskUnion* is given by the sum of the permissions at those locations in *Mask1* and *Mask2*.

state is a predicate introduced in Section 2.6.1.

Encoding of state operations from paper

We first give the encoding of the basic state operations specified in [16]. We modified *addAcc,removeAcc, addPred, removePred* to support fractional permissions instead of just the full permission and included *getAcc*, *getPred*, *hasNoAcc*, *hasNoPred*, *hasNoWand* as well as *equate* for predicates:

σ.`getAcc`(v,f) returns the permission to $v.f$ held in σ.

σ.`hasNoAcc`(v,f) returns true iff σ has no permission to $v.f$.

σ.addAcc(v,f,p) adds permission p to location $v.f$ in σ.

σ.removeAcc(v,f,p) removes permission p from location $v.f$ in σ.

σ.getPred(P,v) returns the permission to $P(v)$ held in σ.

σ.hasNoPred(P,v) returns true iff σ has no permission to $P(v)$.

σ.addPred(P,v,p) adds permission p for predicate instance $P(v)$ by p in σ.

σ.removePred(P,v,p) removes permission p from predicate instance $P(v)$ in σ.

σ.hasNoWand($A \mathrel{-\!\!*} B$) returns true iff σ doesn't contain $A \mathrel{-\!\!*} B$

σ_1.equate($P(v)$,f,σ_2) updates σ_1 such that it contains all assumptions about $P(v)$ from σ_2 (i.e. also known-folded permissions).

The encoding of the updated set of basic state operations is then given by:

σ.eval(e) $\rightsquigarrow \llbracket \texttt{e} \rrbracket_\sigma$

σ.assume(e) $\rightsquigarrow$ b := b&&$\llbracket \texttt{e} \rrbracket_\sigma$
σ.assert(e) $\rightsquigarrow$ **assert** b ==> $\llbracket \texttt{e} \rrbracket_\sigma$

σ.getAcc(v,f) $\rightsquigarrow$ Mask[v,f]
σ.hasAcc(v,f) $\rightsquigarrow$ b ==> Mask[v,f] > 0
σ.hasNoAcc(v,f) $\rightsquigarrow$ b ==> Mask[v,f] <= 0
σ.addAcc(v,f,p) $\rightsquigarrow$ Mask[v,f] += p
σ.removeAcc(v,f,p) $\rightsquigarrow$

```
var TransferHeap: HeapType;
havoc TransferHeap;
Mask[v,f] -= p;
b := b&&identicalOnKnownLocs(Heap,TransferHeap,Mask);
Heap := TransferHeap;
```

σ.getPred(P,v) $\rightsquigarrow$ Mask[null,P(v)]
σ.hasPred(P,v) $\rightsquigarrow$ b ==> Mask[null,P(v)] > 0
σ.hasNoPred(P,v) $\rightsquigarrow$ b ==> Mask[null,P(v)] <= 0
σ.addPred(P,v,p) $\rightsquigarrow$ Mask[null,P(v)] += p
σ.removePred(P,v,p) $\rightsquigarrow$

```
var TransferHeap: HeapType;
havoc TransferHeap;
Mask[null,P(v)] -= p;
b := b&&identicalOnKnownLocs(Heap,TransferHeap,Mask);
Heap := TransferHeap;
```

Assume `wandAB` is the wand field representing the wand instance `A --* B`. See Section 3.1.2 for the details.

σ.`hasWand`($A \mathrel{-\!\!*} B$) $\rightsquigarrow$ `b ==> Mask[null,wandAB] >= 1`
σ.`hasNoWand`($A \mathrel{-\!\!*} B$) $\rightsquigarrow$ `b ==> Mask[null,wandAB] <= 0`
σ.`addWand`($A \mathrel{-\!\!*} B$) $\rightsquigarrow$ `Mask[null,wandAB] += 1`
σ.`removeWand`($A \mathrel{-\!\!*} B$) $\rightsquigarrow$ `Mask[null,wandAB] -= 1`

σ.`onlyvars()` $\rightsquigarrow$

```
var HeapNew: HeapType;
var MaskNew: MaskType;
var bNew: bool;
havoc HeapNew;
MaskNew := ZeroMask;
havoc bNew;
```

σ_1.`equate`(σ_2,v,f) $\rightsquigarrow$ `b1 := b1&&b2&&Heap1[v,f]==Heap2[v,f]`
σ_1.`equate`(σ_2,$P(v)$,f) $\rightsquigarrow$

```
var EquateMask:MaskType;
EquateMask := ZeroMask;
EquateMask[null,P(v)] := FullPerm;
b1 := b1&&b2&&identicalOnKnownLocs(Heap2, Heap1, EquateMask);
```

`if (...) ... else ...` $\rightsquigarrow$

```
if (...) {
  ...
} else {
  ...
}
```

We note that the encoding of *addAcc*/*addPred* is basically the same as an **inhale**, just that there are no definedness checks and the expressions are already evaluated. *removeAcc*/*removePred* are similar to an **exhale** just that there are no definedness checks, no check if there is enough permission and

the expressions are already evaluated. The encodings involving the predicates could easily be generalized to multiple arguments, which we have also done in our implementation. To keep the presentation simpler we just look at predicates with a single argument.

In the translation of σ_1.`equate`(v,f,σ_2) you can see that we not only add the assumption that the value $v.f$ is the same in σ_1 and σ_2, but we also add all assumptions in σ_2. To model the *equate* precisely we would have to add only those assumptions in σ_2 which contribute to the knowledge of $v.f$ in σ_2. Finding these assumptions isn't easy due to transitive reasoning, so we just add all assumptions as an overapproximation.

A state operation, which is not specified as a basic state operation, but needs to be encoded, is the compatible union ($\uplus$, see Definition 1) of two states. We encode it as follows:

$\sigma_{Union} = \sigma_1 \uplus \sigma_2 \rightsquigarrow$

```
var HeapUnion: HeapType;
var MaskUnion: MaskType;
var bUnion: bool;

bUnion := bUnion && identicalOnKownLocs(Heap1,HeapUnion,Mask1);
bUnion := bUnion && identicalOnKownLocs(Heap2,HeapUnion,Mask2);
bUnion := bUnion && sumMask(MaskUnion,Mask1,Mask2);
bUnion := bUnion && state(HeapUnion,MaskUnion);
```

Note that if σ_1 and σ_2 are not compatible then *bUnion* will be false in the end.

In the paper more complex operations such as σ.`inhale(e)` are explicitly encoded in terms of the basic state operations. We define these ourselves since Carbon already has an encoding for some of these operations and also supports fractional permissions.

σ.`inhale(e)` $\rightsquigarrow$

```
if(b) {
    ⟦inhale(e)⟧σ
}
```

σ.`exhale`($\widetilde{\sigma}$,`e`) $\rightsquigarrow$

```
if(b) {
    ⟦exhale(e)⟧σ
}
```

Analogous encodings are done for **fold** and **unfold**. The important part is that we guard the operations with the boolean variable associated with the state. The motivation for this is similar to what was already explained in Section 4.3.

Inconsistency "aware" state operations

The state operations and encodings that we have presented up to this point are independent of the footprint computation algorithm themselves. That means they are fundamental operations that make sense in most settings where states are used. We now give a set of state operations along with their encoding that we defined specifically for our encoding in Boogie (but could theoretically be used in other settings too) and we motivate their meaning.

As mentioned the footprint computation algorithm carries along a stack of states. Lemma 1 states that whenever any state on the stack is inconsistent, then it's sound to let any operation succeed without making changes to any state. Now suppose we assert expression e in a state σ_1. Using our encoding we would emit **assert** `b1 ==>` $[\![\texttt{e}]\!]_{\sigma_1}$. If σ_1 is consistent (i.e. *b1* is not known to be false) and e doesn't hold in σ_1 then the assertion will fail, which we don't want if any state on the stack is inconsistent. So it makes sense to instead emit **assert** `(b1&&` $\overline{b_i}$`)==>` $[\![\texttt{e}]\!]_{\sigma_1}$ (as a reminder: $\overline{b_i}$ is the conjunction of all boolean variables associated with a state on the stack $\overline{\sigma_i}$). It won't happen that assumptions from other states affect the truth value of $[\![\texttt{e}]\!]_{\sigma_1}$ except those which relate (directly or indirectly) the two states. This means if all states on the stack and σ_1 are consistent and e doesn't hold in σ_1 then the *assert* will fail (under the assumption just stated). This along with Lemma 1 motivates this "new" encoding.

But if we now anyway always use all assumptions to all states to prove assertions in states, then there's no point in adding all assumptions of σ_2 to σ_1 in σ_1.`equate`(v,f,σ_2) (see the discussion above on *equate*). All we need to do during the equate is add the assumption to σ_1 which relates the two values, i.e. `b1 := b1&&Heap1[v,f]==Heap2[v,f]`. This encoding doesn't completely conform to *equate*'s specification given in [16], but we get the wanted behaviour when asserting expressions and that's the main thing that we care about.

We now give the set of state operations, which are similar to the basic state operations but include this notion of inconsistency. Their names should make the meaning clear (i.e. *hasAccInc* has the same meaning as *hasAcc* except that if at least one state is inconsistent then *hasAccInc* returns true always).

σ.`assertInc`($\overline{\sigma_i}$, e) $\rightsquigarrow$ (`b&&`$\overline{b_i}$) `==>` $[\![\texttt{e}]\!]_{\sigma}$

σ.hasAccInc($\overline{\sigma_i}$,v,f) $\rightsquigarrow$ (b&&$\overline{b_i}$)==> Mask[v,f] > 0
σ.hasNoAccInc($\overline{\sigma_i}$,v,f) $\rightsquigarrow$ (b&&$\overline{b_i}$)==> Mask[v,f] <= 0

σ.hasPredInc($\overline{\sigma_i}$,P,v) $\rightsquigarrow$ (b&&$\overline{b_i}$)==> Mask[null,P(v)] > 0
σ.hasNoPredInc($\overline{\sigma_i}$,P,v) $\rightsquigarrow$ (b&&$\overline{b_i}$)==> Mask[null,P(v)] <= 0

σ.hasWandInc($\overline{\sigma_i}$,$A \mathrel{-\!\!*} B$) $\rightsquigarrow$(b&&$\overline{b_i}$)==> Mask[null,wandAB] >= 1
σ.hasNoWandInc($\overline{\sigma_i}$,$A \mathrel{-\!\!*} B$) $\rightsquigarrow$(b&&$\overline{b_i}$)==> Mask[null,wandAB] < 1

σ_1.equateInc(σ_2,v,f) $\rightsquigarrow$ b1 := b1&&Heap1[v,f]==Heap2[v,f]
σ_1.equateInc(σ_2,$P(v)$,f) $\rightsquigarrow$

```
var EquateMask:MaskType;
EquateMask := ZeroMask;
EquateMask[null,P(v)] := FullPerm;
b1 := b1&&identicalOnKnownLocs(Heap2, Heap1, EquateMask);
```

4.5 A note on *bCur*

For the state σ_{cur} which represents the state right before the **package**, it's important that the effects on σ_{cur} in the **package** become visible. For instance we need to know how much permission was removed from each heap location (i.e. implicitly what the footprint was) and to which heap locations we have any information. So after the **package** is done we assume that the corresponding boolean variable *bCur* is true using the **assume** statement in Boogie, otherwise we may lose useful information.

In the actual implementation we actually don't use *bCur* at all, whenever an assumption is added to *bCur* we just explicitly add the assumption using the **assume** statement in Boogie.

4.6 Encoding the *transfer* function for fractionals

In this section we look at three encodings for *transfer*($\overline{\sigma_i}$,σ_{used},e) (briefly presented in Section 4.1) where e is a field accessibility predicate. The case where e is a predicate accessibility predicates or a magic wand is completely analogous. The *transfer* given in [16] for field accessibility predicates is only specified if the permission value denotes the full permission. In our discussion we will regard the case where the permission value may also be a fractional value. This means the high-level view of the *transfer* we regard is slightly different compared to the version in [16].

In each of the three encodings we give the specification of the *transfer* just in terms of the state operations defined in Section 4.4 (i.e. independent of Boogie) and look at a direct translation of the specification into Boogie using the translation of the state operations into Boogie defined in Section 4.4. After discussing the limitations of an approach, we try to find a solution to the issues in the following approach (i.e. the second approach is supposed to be a direct improvement of the first approach). The reason we don't just present the third and final approach which is currently used in our implementation in Carbon is that it is easier to understand the motivation for the third approach, if one has understood the issues in the first two approaches.

In the end we look at the limitations of the final approach and sketch a solution that can solve one of the limitations, but has other limitations itself.

4.6.1 Approach 1: The Naive Approach

We give the definition of *transfer* in this first approach in Listing 13 using just the state operations defined in Section 4.4, the *min* function which returns the smaller of two arguments and *transferRec*.

transferRec($\overline{\sigma_i}$,σ_{Used},*e*,*f*,*neededPerm*) is an auxiliary function defined in the same listing, where *e* is a reference type expression, *f* is a field, and *neededPerm* is a permission value evaluated in some state.

Detailed overview

We give a detailed explanation of what the specified *transfer* in Listing 13 is doing, since [16] doesn't mention fractional permissions. In the following two approaches we won't do this, since they are iterations of this approach. In Listing 13 *transfer* evaluates the permission p in σ_{Used}, checks if the evaluated permission is positive and finally calls *transferRec* which does the main work.

transferRec iterates over the stack $\overline{\sigma_i}$ from the top state to the bottom and for each state σ that it encounters it transfers the maximum amount of permission possible for $x.f$ from σ to σ_{used}. The permission amount transferred from σ is at most the amount that is still required (this is guaranteed by transferring the minimum of how much is still needed and how much is held in σ on line 9). If a positive amount is transferred then the two values in the respective states are equated. After all states have been visited, it is asserted if p was transferred to σ_{Used} (line 23, ϵ denotes the empty stack).

So one can say that the idea in the *transfer* given in Listing 13 is analogous to the idea in the *transfer* defined in [16], which is to remove as much as possible from states closer to the top of the stack. The main difference is that permission might have to be transferred from multiple states instead of

Listing 13: The naive high-level approach for the transfer encoding

```
1  transfer(σ̄_i,σ_Used,acc(e.f,p)) ⇝
2      needed := σ_Used.eval(p)
3      σ_Used.assert(needed > 0)
4      return transferRec(σ̄_i,σ_Used,e,f,needed)
5
6  transferRec(σ̄_i · σ,σ_Used,e,f,neededPerm) ⇝
7      v := σ_Used.eval(e)
8      if(neededPerm > 0 && σ.hasAcc(v,f)) {
9          take := min(neededPerm,σ.getAcc(v,f));
10         σ'_Used := σ_Used.addAcc(v,f,take)
11         σ''_Used := σ'_Used.equate(σ,v,f)
12         σ' := σ.removeAcc(v,f,take)
13         (σ̄'_i,σ'''_Used) := transferRec(σ̄_i,σ''_Used,e,f,neededPerm-take)
14         return (σ̄'_i · σ',σ'''_Used)
15     } else {
16         (σ̄'_i,σ'_Used) := transferRec(σ̄_i,σ_Used,e,f,neededPerm)
17         return (σ̄'_i · σ,σ'_Used)
18     }
19
20 transferRec(σ̄_i,ϵ,σ_Used,e,f, neededPerm) ⇝
21     σ_Used.assert(neededPerm == 0)
```

just one. If p is 1 and all states on the stack either have full or no permission to $e.f$ then the effect of *transfer* is the same as in the version given in [16].

The reason why *transfer* evaluates p beforehand and then the evaluated value is passed to *transferRec* instead of *transferRec* evaluating p every time is because we want a fixed starting value for the permission that we need to transfer and then over the course of the algorithm we reduce this amount. It wouldn't make sense if in *transferRec* assumptions added to the initial state σ_{Used} would have an effect on the value of the permission that must still be transferred later on. This doesn't mean that when we evaluate p in *transfer* that we exactly know what its value is, but we make sure that for every possible trace its starting value for the permission is given before *transferRec* is called and won't be directly affected by modifications of σ_{Used} in *transferRec* but only by the algorithm itself.

But following this argument, the question has to be asked why e isn't evaluated beforehand. By evaluating e in σ_{Used} (see line 7) in every recursive call to *transferRec* we want to make clear that σ_{Used} changes (states are

immutable in theory) and hence since assumptions are added through the equate it's possible that we learn more about the value of e in later iterations of the algorithm than before, which makes the algorithm more complete. This is also done in the version given in [16].

To specify the fractional permission version of *transfer* where e is a predicate accessibility predicate, the operations defined in Listing 13 for field access predicates must be modified to the corresponding version for predicates and the arguments must be adjusted. For instance σ.`hasAcc(v,f)` would become σ.`hasPred(P,v)`. Since Silver doesn't support the notion of fractional permission for magic wands, the transfer remains the same as in [16] in that case.

Boogie translation of naive approach

Listing 14 shows the partial Boogie translation of *transfer*($\sigma_{cur} \cdot \sigma_1$, σ_{Used}, *acc(e.f,1)*) right before the first recursive call to *transferRec* is made using the specification given in Listing 13 (we leave out the check if the permission greater 0, since it's clear for the full permission). The rest of the code would appear right after the code given, i.e. we don't have to unroll both calls to *transferRec* which are in the two branches in Listing 13 since we can basically rewrite the `if (...)... else ...` to `(if (...)...)...` as we can use local variables compared to the "theoretical", pure view in Listing 13. Note lines 4 to 8 are a possible encoding for *min*(*neededPerm*,σ_1.`getAcc(x,f)`). Recall that the states returned in Listing 13 are implicit in the Boogie translation. We present three issues for the Boogie encoding given in Listing 14. Whenever we mention a line number without explicitly stating the Listing, then we implicitly are referring to Listing 14.

Issue 1: Not enough information used

One of the main issues with the translation given in Listing 14 is the condition `b1 ==> Mask1[v,f] > 0` given on line 3. It is possible that the assumptions accumulated in *b1* do not suffice to conclude that `Mask1[v,f] > 0` even if σ_1.`hasAcc(v,f)` is true at that point. The reason is that to learn all the information on the value v after the assignment `v :=`$\llbracket$e$\rrbracket_{\sigma_{Used}}$ on line 2 one needs to use the assumptions in σ_{Used}, which are accumulated in *bUsed*. Consider the following concrete example:

```
package (acc(x.f) && acc(x.f.f))--*(acc(x.f) && acc(x.f.f))
```

Listing 14: Partial Boogie translation of transfer, where σ_1 is on top of the stack, given by Listing 13 (Approach 1)

```
1  neededPerm := 1
2  v := ⟦e⟧_σUsed
3  if (neededPerm > 0 && (b1 ==> Mask1[v,f] > 0) ) {
4    if (b1 ==> neededPerm <= Mask1[v,f]) {
5      take := neededPerm;
6    } else {
7      take := Mask1[v,f];
8    }
9    neededPerm := neededPerm - take;
10   MaskUsed[v, f] := MaskUsed[v,f] + take;
11   bUsed := bUsed && state(HeapUsed, MaskUsed);
12   bUsed := bUsed && b1 && HeapUsed[v,f] == Heap1[v,f];
13   havoc TransferHeap;
14   Mask1[v,f] := Mask1[v,f] - take;
15   b1 := b1 && IdenticalOnKnownLocs(Heap1,TransferHeap,Mask1);
16   Heap1 := TransferHeap;
17   b1 := b1 && state(Heap1, Mask1);
18 }
```

This wand should clearly be packaged with an empty footprint. Let σ_1 be the state created by the footprint computation algorithm which satisfies the left hand side. Hence right before the *transfer* we know that the following holds:

```
b1 ==> Mask1[x,f] == 1 and b1 ==> Mask1[Heap1[x,f],f] == 1
```

The first *transfer* is called for `acc(x.f)`, where σ_1 is on top of the stack. Since σ_1 has non-zero permission to *x.f*, the condition on line 3 is true. After the equate on line 12 we'll know that

```
bUsed ==> HeapUsed[x,f] == Heap1[x,f]
```

and after line 16 information regarding `Heap1[x,f]` will be havoced. Let's refer to the heaps and masks corresponding to σ_1 and σ_{Used} right before this *transfer* as `Heap1*`,`Mask1*` and `HeapUsed*`,`MaskUsed*`. So after line 12 we learn that

```
bUsed ==> HeapUsed*[x,f] == Heap1*[x,f]
```

Next *transfer* is called for `acc(x.f.f)`, where again σ_1 is on the top of the stack. We expect that the algorithm transfers full permission for *x.f.f* from σ_1 to σ_{Used}. It turns out that this encoding doesn't lead to the expected result. On line 2

```
v := HeapUsed[x,f]
```

will be emitted. Since `HeapUsed[x,f]` itself doesn't change during the transfers we conclude that

```
HeapUsed[x,f] == HeapUsed*[x,f]
```

holds in the Boogie encoding. The Boogie translation of σ_1`.hasAcc(v,f)` which is given on line 3 is then emitted as

```
b1 ==> Mask1[HeapUsed[x,f],f] > 0
```

which according to the observation above is semantically equivalent to

```
b1 ==> Mask1[HeapUsed*[x,f],f] > 0
```

Now note that since we actually never change the permission to *v.f* in σ_1 (where *v* is given by *x.f* evaluated in σ_{Used}) before the *transfer* of *x.f.f* and due to the insight regarding `HeapUsed*[x,f]` from before we can conclude that before the *transfer* of *x.f.f* the following holds:

```
b1 ==> Mask1[HeapUsed*[x,f],f] == Mask1*[HeapUsed*[x,f],f]
```

Therefore the Boogie translation of σ_1`.hasAcc(v,f)` on line 3 is also semantically equivalent to

```
b1 ==> Mask1*[HeapUsed*[x,f],f] > 0
```

Now we see that this condition won't be verified since *b1* doesn't contain the assumption that

```
HeapUsed*[x,f] == Heap1*[x,f]
```

which would suffice to verify the condition. This assumption was added to *bUsed* in the transfer of *x.f*. The condition would be verified if the assumptions were part of *b1* otherwise because we know that

```
b1 ==> Mask1*[Heap1*[x,f],f] > 0
```

due to the way σ_1 is constructed. So we conclude our Boogie encoding doesn't precisely encode the *eval* state operation. One could think that this means that the *equate* must add the assumptions to both states instead of just σ_{Used}. But this would be unsound, since once σ_{Used} is not any more carried

along in the algorithm then facts in σ_{Used} shouldn't affect the algorithm. This means we would like to translate σ_1.`hasAcc(v,f)` on line 3 as

```
(b1&&bUsed) ==> Mask1[HeapUsed[x,f],f] > 0
```

which would evaluate to true.

This example also shows that the receiver e in a transfer of $e.f$ is one of the main reasons why we need assumptions of potentially more than one state to express σ_1.`hasAcc(v,f)` in our Boogie encoding.

Issue 2: Undesirable footprint chosen in presence of inconsistent states

There's another issue regarding inconsistent states with respect to the encoding in Listing 14. Let us regard the **package** in Listing 10. Recall that the example is given by

```
//σ_cur satisfies acc(x.f)&&x.f==2&&acc(y.f)
package acc(x.f,1/2)&&x.f==3 --* acc(x.f)&&acc(y.f)
```

As described in Section 4.2 in this case the states on the stack when the transfer for $y.f$ executes are all consistent but σ_{Used} is inconsistent. We argue in that section that no permission to $y.f$ has to be removed from the current state. In our encoding this isn't guaranteed because the inconsistency of σ_{Used} doesn't affect the control flow at all.

Issue 3: Lemma 1 not taken into account

A final issue is that if the necessary permission wasn't transferred (i.e. $neededPerm > 0$ at the end of the *transfer*), then in general the translation of the assertion on line 22 of Listing 13 into Boogie won't be satisfied, even if a state on the stack $\overline{\sigma_i}$ is inconsistent. This is not the desired behaviour as can be seen in Lemma 1, which states that if a state on the stack $\overline{\sigma_i}$ is inconsistent, then for instance the translation of the assertion on line 22 in of Listing 13 should be verified. Note that issue 2 is also related to Lemma 1.

4.6.2 Approach 2: Using all assumptions

Using the insights gained from the first approach, we now present the modified *transfer* defined in terms of the state operations in Listing 15. The main difference in Listing 15 compared to the first approach is that we now use the inconsistency "aware" state operations introduced in Section 4.4. Note that we carry along the original stack of states at the beginning of the *transfer* such that we can use all assumptions of those states, even if the algorithm has

Listing 15: Transfer definition of approach 2

```
1  transfer(σ̄_i,σ_Used,acc(e.f,p)) ⇝
2      needed := σ_Used.eval(p)
3      σ_Used.assertInc(σ̄_i,needed > 0)
4      return transferRec(σ̄_i,σ̄_i,σ_Used,v,f,needed)
5
6  transferRec(σ̄_a,σ̄_i · σ,σ_Used,e,f,neededPerm) ⇝
7      v := σ_Used.eval(e)
8      if(neededPerm > 0 && σ.hasAccInc(σ̄_a · σ_Used,v,f) {
9          take := min(neededPerm,σ.getAcc(v,f));
10         σ'_Used := σ_Used.addAcc(v,f,take)
11         σ''_Used := σ'_Used.equateInc(σ,v,f)
12         σ' := σ.removeAcc(v,f,take)
13         (σ̄'_i,σ'''_Used) :=
14             transferRec(σ̄_a,σ̄_i,σ''_Used,e,f,neededPerm-take)
15         return (σ̄'_i · σ',σ'''_Used)
16     } else {
17         (σ̄'_i,σ'_Used) := transferRec(σ̄_a,σ̄_i,σ''_Used,e,f,neededPerm)
18         return (σ̄'_i · σ,σ'_Used)
19     }
20
21 transferRec(σ̄_a,ϵ,σ_Used,v,f), neededPerm) ⇝
22     σ_Used.assertInc(σ̄_a,neededPerm == 0)
```

already visited a few states. We denote the stack of states at the beginning of the *transfer* by $\overline{\sigma_a}$ and the corresponding conjunction of boolean variables in Boogie as $\overline{b_a}$. Finally since we use all assumptions anyway whenever we need to formulate an assertion on any given state, we can replace the translation to σ'_{Used}.equate(σ,v,f) by the translation to σ'_{Used}.equateInc(σ,v,f) which just adds the relation that *v.f* is the same in σ'_{Used} and σ without adding any other assumptions. . This has the effect that the Boogie variable *bUsed* associated to σ_{Used} itself won't contain all assumptions needed to make assertions on σ_{Used}, but since we always use all assumptions of the states on the stack + the assumptions carried along in *bUsed* to check assertions, it doesn't matter.

The corresponding partial Boogie translation is shown in Listing 16 for *transfer*($\sigma_{cur} \cdot \sigma_1$, σ_{Used}, *acc(e.f,1)*).

Let's see how the encoding in Listing 16 deals with the three issues presented for the first approach described in Section 4.6.1. From now on when-

Listing 16: Partial Boogie translation of transfer, where σ_1 is on top of the stack, using the transfer specification in Listing 16 (Approach 2)

```
1  neededPerm := ⟦p⟧_σUsed
2  v := ⟦e⟧_σUsed
3  if (neededPerm > 0 && ( (b1&&b̄_i&&bUsed) ==> Mask1[v,f]>0)) {
4    if ((b1&&b̄_i&&bUsed) ==> neededPerm <= Mask1[v,f]) {
5      take := neededPerm;
6    } else {
7      take := Mask1[v,f];
8    }
9    neededPerm := neededPerm - take;
10   MaskUsed[v,f] := MaskUsed[v,f] + take;
11   bUsed := bUsed && state(HeapUsed, MaskUsed);
12   bUsed := bUsed && HeapUsed[v,f] == Heap1[v,f];
13   havoc TransferHeap;
14   Mask1[v,f] := Mask1[v,f] - take;
15   b1 := b1 && IdenticalOnKnownLocs(Heap1,TransferHeap,Mask1);
16   Heap1 := TransferHeap;
17   b1 := b1 && state(Heap1, Mask1);
18 }
```

ever we don't explicitly mention a Listing, we are referring to the Boogie translation of the approach in Listing 16.

Analysing issue 1

Assume σ_1 is at the top of the stack when the *transfer* is initially called. We notice that *issue 1* isn't a problem in this encoding. The reason is that we use all Boolean variables on the stack $\overline{\sigma_a}$ along with σ_{Used} to check if σ_1 has permission to a specific location on line 3, which as explained in the previous approach solves the specific issue. Actually we even argued that it suffices to just use the Boolean variable corresponding to σ_1 and σ_{Used}. The reason we use all Boolean variables in this encoding is that it is possible that there is transitive reasoning. This means for Boolean variables *b1*,*b2*,*b3* accumulating assumptions for σ_1, σ_2, σ_3 it is possible that *b1* contains an assumption relating *x.f* in σ_1 and σ_2 and *b2* contains assumptions relating *x.f* in σ_2 and σ_3. Then the only way to get the relation of *x.f* in σ_1 and σ_3 is by using all boolean variables (instead of just using *b1* and *b3*).

Analysing issue 3

Issue 3 also isn't a problem any more, since we replace the translation to

σ_{Used}.`assert`(neededPerm == 0)

with

σ_{Used}.`assertInc`($\overline{\sigma_a}$,neededPerm == 0)

in Listing 15. This means whenever any state on the stack or σ_{Used} is inconsistent the transfer will definitely succeed in the corresponding Boogie encoding. So in this case it actually partly utilizes Lemma 1.

Analysing issue 2

The specific example (which is given by Listing 10) used in Issue 2 isn't a problem in the new encoding either. Recall that the mentioned example is given by:

```
//σ_cur satisfies acc(x.f)&&x.f==2&&acc(y.f)
package acc(x.f,1/2)&&x.f==3 --* acc(x.f)&&acc(y.f)
```

and the goal is that just full permission to *x.f* is removed from the current state σ_{cur} and no permission to *y.f* is removed from σ_{cur}. When the *transfer* for *x.f* is finished then also in this modified encoding we'll notice an inconsistency. Note that in the previous approach we noticed the inconsistency in σ_{Used} directly by noticing that *bUsed* could be shown to be *false*. In our new encoding this won't be the case because we don't add all assumptions from the state from which permission is removed to σ_{Used} but just the assumption relating the heap values in the respective states. But the verifier will be able to show $\overline{b_a}$ `&& bUsed` is *false* after the *transfer* of *x.f*, since this conjunction holds assumptions that if true imply that *x.f* is 2 and 3 in some state. Now let's see what happens in the *transfer* of *y.f*, where we assume σ_1 is at the top of the stack:

The condition on line 3 will hold since we just showed that the implication holds trivially due to left hand side being *false* and *neededPerm == 1*, hence the branch is taken. Then for the same reason the branch on line 4 is taken too, i.e. *take* is set to *neededPerm*, i.e. to 1. Finally *neededPerm* is set to 0, full permission is removed from σ_1 and added to σ_{Used}. Therefore the *transfer* succeeds without removing any permission to *y.f* from σ_{cur}. So in the end the result of the **package** is exactly what we want. Notice that the encoding removed permission from σ_1 even though σ_1 doesn't even have any permission to *y.f*. In this specific example this doesn't turn out to be a problem. Actually the encoding forces the needed permission to be removed if the combination of states on the stack and σ_{Used} is inconsistent, as could also be seen in this example. In a next step we show that this behaviour makes the encoding unsound.

Unsoundness in the second approach

To show a minimal example of the unsoundness in the second approach, we need to have three states on the stack to show the problem. Consider the following operation, where we assume that the current state σ_{cur} has no permission to *x.f*:

```
package acc(x.f,1/4)&&x.f==2 --*
    (packaging acc(x.f,1/4)&&x.f==3 --* acc(x.f) in true)
```

The footprint computation algorithm will call *transfer*($\sigma_{cur} \cdot \sigma_1 \cdot \sigma_2$, σ_{Used}, *acc(x.f,1)*) (actually in theory there is another empty state on the stack, but since it doesn't affect the outcome we omit it). σ_1 is the state corresponding to the left hand side of the "outer" **package**, i.e. it satisfies `acc(x.f,1/4)&&x.f==2` and σ_2 is the state corresponding to the left hand side of the "inner" **package**, i.e. it satisfies `acc(x.f,1/4)&&x.f==3`. In terms of the Boogie encoding this means we know that the following holds

```
b1 ==> Heap1[x,f] == 2 and b2 ==> Heap2[x,f]==3
```

At the beginning of the *transfer* the conjunction of $\overline{b_a}$ and *bUsed* (note $\overline{b_a}$ equals `bCur&&b1&&b2`) can't be shown to be *false*, since no inconsistencies can arise from all the assumptions stored in the Boolean variables.

First the algorithm transfers 1/4 permission to *x.f* from σ_2 to σ_{Used} and then 1/4 permission to *x.f* from σ_1 to σ_{Used}. After these two iterations of the *transfer* algorithm it holds that

```
bUsed ==> Heap1[x,f] == Heap2[x,f]
```

Hence it must hold that

```
(b1 && b2 && bUsed) ==> 2 == 3
```

which implies that `b1 && b2 && bUsed` is *false*. We also know that still 1/2 permission is needed (i.e. *neededPerm == 1/2*). At this point the algorithm reaches the code block in the Boogie encoding shown in Listing 17 (note that *Mask* and *Heap* are the global variables associated with the heap and mask of σ_{cur}).

The condition on line 1 will be satisfied since *neededPerm == 1/2* and the left hand side of the implication is *false*. The condition on line 2 will also be satisfied for the same reason. Hence *take == 1/2* at line 7. Therefore we remove half permission to *x.f* on line 8, which means `Mask[x,f]` stores a negative amount of permission afterwards. Then finally on line 11 `state(Heap,Mask)` is added to *bCur*. This leads to *bCur* being *false* because if the *state* predicate would be true, then it would imply that all permissions

Listing 17: Unsoundness in the Boogie encoding for the second *transfer* approach

```
1  if (neededPerm>0 && (b1&&b2&&bCur&&bUsed ==> Mask[x,f]>0)) {
2    if (b1&&b2&&bCur&&bUsed ==> neededPerm <= Mask[x,f]) {
3      take := neededPerm;
4    } else {
5      take := Mask[x,f];
6    }
7    [changes made to σ_Used omitted]
8    Mask[x,f] := Mask[x,f] - take;
9    bCur := bCur && IdenticalOnKnownLocs(Heap,TransferHeap,Mask
        );
10   Heap := TransferHeap;
11   bCur := bCur && state(Heap, Mask);
12 }
```

in *Mask* are positive which is not true in this case. This is problematic since after the **package** is over, we assume that *bCur* is true (see Section 4.5). Hence we'll basically lose the trace which corresponds to the actual footprint computation algorithm and this leads to unsound behaviour.

4.6.3 Approach 3: The Final Approach

In the second approach we noticed that when inconsistencies arise and one starts removing permission from states even though these states may not have any permission it becomes hard to reason about the *transfer*. We also showed that the second approach is unsound. In this final approach which we are currently using in our implementation we construct an encoding which directly uses Lemma 1. Therefore the motivation of the encoding is to essentially let the *transfer* succeed without changing any state as soon as any state on the stack or σ_{Used} is inconsistent.

A way to achieve this requirement is to modify the second conjunct on line 3 of the second approach, which is given in Listing 16 (note this is the Boogie translation for σ_1.`hasAccInc`($\overline{\sigma_a} \cdot \sigma_{Used}$,*v*,*f*)). We modify the conjunct to

```
(b_i&&bUsed)&&Mask1[v,f] > 0
```

From a technical standpoint we changed the implication to a conjunction. This guarantees that if any of the regarded states is inconsistent then the

Listing 18: Transfer definition of final approach

```
1  transfer(σ_i,σ_Used,acc(e.f,p)) ⇝
2      needed := σ_Used.eval(p)
3      σ_Used.assertInc(σ_i,needed > 0)
4      return transferRec(σ_i,σ_i,σ_Used,v,f,needed)
5
6  transferRec(σ_a,σ_i · σ,σ_Used,e,f,neededPerm) ⇝
7      v := σ_Used.eval(e)
8      if(neededPerm > 0 && !σ.hasNoAccInc(σ_a · σ_Used,v,f) {
9          take := min(neededPerm,σ.getAcc(v,f));
10         σ'_Used  := σ_Used.addAcc(v,f,take)
11         σ''_Used  := σ'_Used.equateInc(σ,v,f)
12         σ'  := σ.removeAcc(v,f,take)
13         (σ'_i,σ'''_Used) :=
14             transferRec(σ_a,σ_i,σ''_Used,e,f,neededPerm-take)
15         return (σ'_i · σ',σ'''_Used)
16     } else {
17         (σ'_i,σ'_Used) := transferRec(σ_a,σ_i,σ''_Used,e,f,neededPerm)
18         return (σ'_i · σ,σ'_Used)
19     }
20
21 transferRec(σ_a,ϵ,σ_Used,v,f), neededPerm) ⇝
22     σ_Used.assertInc(σ_a,neededPerm == 0)
```

condition won't hold, where as in the earlier approach it made the condition trivially true. Note that this condition we just presented is equivalent to

```
!( (b_i&&bUsed) ==> Mask1[v,f] <= 0 )
```

which is exactly the Boogie encoding for $!\sigma_1$.hasNoAccInc($\overline{\sigma_a} \cdot \sigma_{Used}$,*v*,*f*). So we can interpret this as asking if it doesn't hold that in σ_1 there is no permission to *v.f* using the assumptions of all states on the stack and σ_{Used}. Note that in the Boogie encoding this is not the same as asking if in σ_1 there is permission to *v.f* since that assertion would be true if σ_1 is inconsistent.

The final encoding expressed with state operations defined in Section 4.4 is given in Listing 18. The corresponding partial Boogie translation for *transfer*($\overline{\sigma_i} \cdot \sigma_1$, σ_{Used}, *acc(e.f,p)*) is given in Listing 19.

Recall the example which exhibited the unsoundness in the second approach:

Listing 19: Partial Boogie translation of transfer, where σ_1 is on top of the stack, using the transfer specification in Listing 18 (Final approach)

```
1  neededPerm := ⟦p⟧_σUsed
2  v := ⟦e⟧_σUsed
3  if (neededPerm > 0 && (b1&&b̄_i&&bUsed&&Mask1[v,f]>0)) {
4    if (neededPerm <= Mask1[v,f]) {
5      take := neededPerm;
6    } else {
7      take := Mask1[v,f];
8    }
9    neededPerm := neededPerm - take;
10   MaskUsed[v,f] := MaskUsed[v, f] + take;
11   bUsed := bUsed && state(HeapUsed, MaskUsed);
12   bUsed := bUsed && HeapUsed[v,f] == Heap1[v,f];
13   havoc TransferHeap;
14   Mask1[v,f] := Mask1[v,f] - take;
15   b1 := b1 && IdenticalOnKnownLocs(Heap1,TransferHeap,Mask1);
16   Heap1 := TransferHeap;
17   b1 := b1 && state(Heap1, Mask1);
18 }
```

```
package acc(x.f,1/4)&&x.f==2 --*
    (packaging acc(x.f,1/4)&&x.f==3 --* acc(x.f) in true)
```

Note that the encoding of the minimum function doesn't explicitly use any assumptions of any state, this is because if the condition on line 3 in Listing 18 is true, then we automatically get the assumptions of all states inside the branch where the minimum of two values is computed.

As in the second approach *transfer*($\sigma_{cur} \cdot \sigma_1 \cdot \sigma_2$, σ_{Used}, *acc(x.f,1)*) is called and after 1/4 is transferred from both σ_2 and σ_1 to σ_{Used} the verifier learns that `b1&&b2&&bCur&&bUsed` is false. But in this final approach no permission will be removed from σ_{cur} since the condition on line 3 in Listing 19 won't hold (where in contrast to the Listing of course not σ_1 but σ_{cur} is the top of the stack). The transfer also succeeds since to check if the *transfer* succeeds we use the translation of σ_{Used}.`assertInc`($\overline{\sigma_a}$,`neededPerm == 0`) as in approach 2. Finally the **`package`** succeeds with an empty footprint which is the desired behaviour.

The issues presented for the first approach also are solved for this encoding. Issue 1 and 3 are solved for the same reason that approach 2 solves it,

namely by using all assumptions on the stack and σ_{Used}. Let's take a quick look at issue 2 which discussed the following example:

```
//σ_cur satisfies acc(x.f)&&x.f==2&&acc(y.f)
package acc(x.f,1/2)&&x.f==3 --* acc(x.f)&&acc(y.f)
```

In this case after the transfer of *x.f* is finished, the verifier learns that the conjunction of the Boolean variables corresponding to the states on the stack and *bUsed* is false and hence no permission will be removed from *y.f*. This is again exactly the behaviour that we desire.

4.6.4 Incompleteness in the final approach

We now look at a general problem for the final approach for which the partial Boogie encoding is given in Listing 19. Whenever we refer to an encoding or some line number without explicitly mentioning a Listing we implicitly refer to Listing 19.

The Boolean variables, which accumulate all assumptions about a certain state, lead to the main problem here. We motivated in Section 4.3 that we use such Boolean variables instead of explicit assumptions in Boogie because we don't want assumptions from various states affecting the global verification state after the **package** is over. The reason why the Boolean variable approach solves this problem is because the verifier in general never learns that the Boolean variables are true. If the state is inconsistent then then corresponding Boolean will be known to be false by the verifier (in general), but this isn't a problem as it doesn't add assumptions we don't want to the global verification state. But the fact that we generally don't know if Booleans are true or not makes the *transfer* encoding incomplete, since intuitively it affects the control flow.

Consider the following simple **package** operation:

```
//σ_cur satisfies acc(x.f)
package true --* acc(x.f)
```

It's clear this operation should succeed with a footprint that satisfies `acc(x.f)`. Hence we expect that in the state after the **package** is finished verifier concludes that the there's no permission to *x.f*. Unfortunately this is not guaranteed by our final encoding.

Let's look at what our encoding does in detail. *transfer*($\sigma_{cur} \cdot \sigma_1$,$\sigma_{Used}$, `acc(x.f)`) is called where σ_1 is the state satisfying the left hand side of the wand, in this case σ_1 is essentially an empty state and σ_{cur} is the state right before the **package**. It's clear that the condition on line 3 when σ_1 is on top of the stack will be *false* since there's no permission to *x.f* in σ_1. Now

the algorithm moves on to potentially remove permission from σ_{cur}. The condition on line 3 then can be written as (note *Mask* is the global variable representing the mask of σ_{cur}):

```
neededPerm > 0 && bCur && b1 && bUsed && Mask[x,f] > 0
```

The verifier can verify that `neededPerm > 0` and that `Mask[x,f] > 0`, but the verifier can't say anything about the value of `b1` and `bUsed`. It also doesn't know the value of `bCur` but since we assume `bCur` to be true at the end of the **package**, the verifier will learn the true value of `bCur` at the end (see Section 4.5). This means that the verifier can't be sure if the above condition is true and hence doesn't know if the branch on line 3 is taken. Let's now assume that `bCur` is true (which as just outlined is justified). So the verifier essentially considers the traces where `b1` and `bUsed` are true and the traces where at least one of them is false. In the traces where `b1` and `bUsed` are true the full permission is removed from σ_{cur} and the *transfer* then succeeds the way we expected. In the traces where `b1` or `bUsed` is false nothing is removed from σ_{cur} and the *transfer* also succeeds,since the assertion in the end is given by

```
assert bCur && b1 && bUsed ==> neededPerm ==0
```

This means once the **package** is finished the verifier can't prove that `Mask[x,f]==0`, because it knows that there is a potential trace which didn't remove anything permission from σ_{cur}. This means the verifier can just prove

```
Mask[x,f] == 0 || Mask[x,f] == 1
```

What is important to note is that after the **package** in this case there is at least one trace which leads to the correct footprint, i.e. which removed full permission to $x.f$ from σ_{cur}.

This means after the **package** all operations will have to take this trace into account, which may suggest that the encoding is sound. The question that must be answered is: Is any trace which led to the correct footprint, the trace we actually want? We just informally argued here that there is a trace that leads to the correct footprint, but this doesn't suffice for a soundness argument.

4.6.5 Potential unsoundness in the final approach

In the last section on the completeness of our final *transfer* encoding we noticed that there are multiple traces which our *transfer* encoding permits instead of just the one that leads to the correct footprint. There's an issue with the trace that leads to the correct footprint. Assume that the verifier

Listing 20: Example showing potential unsoundness

```
1  inhale acc(x.f)
2  inhale acc(y.f)
3
4  package acc(x.f,1/2)&&x.f==2 --*
5                  acc(x.f,3/4)&&acc(y.f,1/2)
6  inhale acc(y.f,1/2)
7  assert x.f == 2
```

learns this "correct" trace, i.e. all other traces are discarded by the verifier. This means that in each *transfer* in a **package** the verifier knows exactly which branches were taken and hence at the end we'll know exactly what the correct permission values should be for the state right after the **package**. The problem is that if we know a branch is taken, then we know that the condition of the branch must have been true and in the case of *transfer* this means we learn assumptions that are made during the **package** but should not affect the state after the **package** is done. A concrete example where the verifier learns part of the "correct" trace is given in Listing 20.

Let's analyse the example. The footprint of the **package** has 1/4 permission to *x.f* and 1/2 permission to *y.f*. Now due to our incompleteness issue our encoding doesn't know how much permission is exactly left for *x.f* and *y.f* in the state right after the **package**. Also no assumptions made inside the **package** are known (since the value of the boolean variables are unknown).

Something interesting happens after **inhale** `acc(y.f,1/2)`. Now the verifier learns that the footprint computation algorithm must have removed half permission from *y.f*, because if it hadn't removed it then after **inhale** `acc(y.f,1/2)` then there would be more than full permission to *y.f* hence the state would be inconsistent. That means it knows the condition $!\sigma_{cur}$.`hasNoAccInc`$(\sigma_1 \cdot \sigma_{Used}, y, f)$ holds, where σ_1 is the state representing the left hand side of the wand in the **package**. This corresponds to the second conjunct of line 3 in Listing 19 (of course in the Listing the state from which permission is potentially removed from is σ_1 and in our case it's σ_{cur}). So the verifier learns that `bCur&&b1&&bUsed` is true.

Since before the *transfer* of *y.f* there is a transfer of *x.f* where permission is transferred from σ_1 and σ_{cur} the verifier knows right after the *transfer* of *x.f* that `bUsed` has accumulated the fact that *x.f* is the same in σ_1 and σ_{cur}. So this means at the point where the verifier learns that `bCur&&b1&&bUsed` is true it learns that *x.f* has the value 2 in σ_{cur}. Therefore the assertion at the end is verified, even though we expect it not to be verified.

Listing 21: Another example showing potential unsoundness

```
1  inhale acc(x.f)
2  inhale acc(y.f)
3
4  package acc(x.f,1/2)&&x.f==2 --*
5                  acc(x.f,3/4)&&acc(y.f,1/2)
6  exhale acc(x.f,1/2)&&x.f==2 --*
7                  acc(x.f,3/4)&&acc(y.f,1/2)
8  inhale acc(y.f,1/2)
9  assert x.f == 2
```

But what is interesting about Listing 20 is that the only way the current state could have full permission to *y.f* after the **package** is if the wand which is packaged on line 4 is applied. But since we never applied the wand explicitly nor exhaled the wand we know that the wand instance is still part of the program state. This is a contradiction, since half permission to *y.f* is associated to the footprint of the wand instance, which is still part of the program state and therefore we can only have at most half permission to *y.f* and certainly not full permission to *y.f*. With this reasoning we conclude that we are in an inconsistent state and anything can be asserted.

The issue is that the verifier won't be able to deduce *false*, which now makes our verifier "inconsistent" in the sense that if it can verify *x.f==2* then it should also be able to verify *false*, but it obviously can't. So in some sense in this case we are not unsound, but we are certainly unpredictable.

Let's look at a slightly example given in Listing 21. The only difference to the example before is that we **exhale** the wand right after packaging it. Now from a technical standpoint nothing changes, in our encoding the assertion that *x.f* is 2 will still be verified. We can't use the same reasoning as before to justify this behaviour any more. The reason is that since we don't have the wand instance at the point where we get half permission, we are not necessarily in an inconsistent state, so it isn't sound to verify any assertion. It turns out there still is a way to reason why letting the assertion go through is possibly sound.

In Silver the sum of all permissions held by different methods to a specific location is at most 1. So the method *m* executing the program in Listing 21 holds all the permission to *y.f* in the beginning. After the **package** *m* holds half of the permission to *y.f* and the other half is stored in the footprint of the wand. We then give the wand instance away, which can be interpreted

as giving the footprint away. Now after inhaling half permission to $y.f$ the question to be asked is: Who gave us this permission?

The only way this permission could have been extracted by another method/entity is by extracting it from the footprint of the wand which we gave away. The only way to do that is by applying the wand, hence some entity must have applied the wand which means $x.f$ had the value 2 for that entity. Since m always held permission to $x.f$, no entity could have changed the value of $x.f$. So we conclude that $x.f$ has the value 2 in m which would mean that letting the assertion go through isn't unsound.

The issue with this reasoning is that we're reasoning about other "entities", which isn't really defined in Silver. Also our reasoning seems to assume that there is some kind of "execution" tied to a Silver program, something that isn't defined either. So from a Silver perspective it's not clear if the behaviour we just explained is desired, so it's not clear if it's sound.

Finally the example we showed is quite simple, there's no guarantee that for every example where such behaviour occurs that we can reason the way we did. This is the main reason why we classify this behaviour as a potential unsoundness issue in our encoding.

4.6.6 Trading completeness for soundness

We now show a simple way of modifying our *transfer* encoding described in Section 4.6.3 to solve the unsoundness issue presented in Section 4.6.5, but the modified encoding is much less complete. It's not part of our current implementation.

The main issue as we noticed was that the assumptions on the states affected the control flow of the *transfer*. If we look closely the only assumptions on states that we don't want to learn outside of the **`package`** are assumptions that relate different states.

For example if we know outside the **`package`** that $x.f$ in σ_1 has value 4, then this won't matter, since we'll never use that state anyway after the **`package`**. If we additionally learn that $x.f$ has the same value in the state after the **`package`** then we potentially have problem. These assumptions which relate different states are only added by the `equateInc` state operation.

A solution to now solve the unsoundness issue is to have two boolean variables for each σ_{Used} in a transfer. One boolean variable $bUsedEquate$ is used to accumulate all assumptions learned from the `equate` and the other boolean variable $bUsed$ is used to accumulate all other assumptions. Now we only let the boolean variables which don't accumulate *equate* assumptions affect the control flow. Like this we guarantee that whenever we do learn a branch is taken, then all assumptions that we learn won't lead us to

Listing 22: Definition used for executing unfolding ghost operation

```
1  exec(σ̅ᵢ, σ_ops, unfolding P(e) in G) ⇝
2      σ_emp := σ_ops.onlyvars()
3      v := σ_ops.eval(e)
4      (σ̅ᵢ' · σ'_ops, σ'_used) := exhale_ext(σ̅ᵢ · σ_ops, σ_emp, P(v))
5      σ''_used := σ'_used.unfold(P,v)
```

verify something we shouldn't after the **package**. But whenever we **assert** some expression then we use all assumptions (i.e. also those known from the equate), since this shouldn't affect the knowledge of the boolean variables.

Using this modified encoding we would for instance get the expected behaviour in Listing 20. With this modified encoding we lose a lot of useful information during the *transfer*. For instance the modified *transfer* won't notice inconsistencies generated by the *equate* constraints and hence this will result in more permissions in the footprint than needed. Also if we have a transfer where only through an earlier *equate* we know that the receivers of a heap location in σ_{Used} and σ_1 are the same then we also won't notice this during the *transfer* (i.e. it doesn't solve issue 1 presented in Section 4.6.1).

4.7 Boogie encoding of *exhale_ext*

exhale_ext was briefly presented in Section 4.1. We use the same definition for *exhale_ext* as [16] but in the case where *exhale_ext* uses *transfer* we use the Boogie encoding corresponding to the definition given in Listing 18 and we also in general just use our defined encodings for the state operations given in Section 4.4. Also we modify *exhale_ext* for a pure expression e:

exhale_ext($\overline{\sigma_i}$,σ_{used},e) $\rightsquigarrow$ σ_{used}.`assertInc`($\overline{\sigma_i}$,e)

We basically replaced *assert* by *assertInc* which is justified by Lemma 1 and also the discussion on *assertInc* in Section 4.4.

4.8 Boogie encoding of *exec*

exec was briefly presented in Section 4.1. We keep the definitions of *exec* the same as in [16] except for some changes. In the *exec* for the **unfolding** ghost operation we call *exhale_ext* with *P(v)* as expression argument instead of *P(e)*.

Listing 22 shows our definition of executing the **unfolding** operating in terms of state operations. Intuitively it makes sense to input *P(v)* where v

Listing 23: Showcasing that argument of predicate should be evaluated in particular state during unfolding ghost operation

```
field f:Ref

predicate P(x:Ref) {
  acc(x.f,1/4)
}

method t01(x:Ref) {
  package acc(x.f,1/4)&&acc(P(x.f)) --* (folding P(x) in (
     unfolding P(x) in (unfolding P(x.f) in true)))
}
```

is evaluated in σ_{ops} to *exhale_ext* since we later on actually also unfold that predicate instance on line 5. Assume we were to input *P(e)* to *exhale_ext*. Then *exhale_ext* would call *transfer* which evaluates e in its own empty state which won't give any information, i.e. e may even be undefined in that state which would lead to a failure.

A concrete example which shows this is given in Listing 23. The **package** in Listing 23 should succeed with an empty footprint. The effect of **folding** and then **unfolding** the same predicate instance *P(x)* is that non-zero permission to *x.f* will be held in σ_{ops}. Hence when **unfolding** `P(x.f) ...` is executed then *x.f* will be well-defined in σ_{ops}. But if we pass *P(x.f)* instead of *P(v)* where v is given σ_{ops}.`eval(e)` then the *transfer* for *P(x.f)* will evaluate *x.f* in an empty state. This will lead to a verification failure since *x.f* is not defined in a state which has no permission to *x.f*. The only issue with our modification in Listing 23 is that *transfer*'s signature is not defined for already evaluated values, so in theory we would have to specifically use another *transfer* which gets already evaluated values (in this case for the arguments of a predicate instance).

Another change is that to make sure that the ghost operations succeed if some state on the stack is inconsistent, we for example use σ.`fold`($\overline{\sigma_i}$,P,v) instead of σ.`fold(P,v)`. Where

σ_{Used}.`fold`($\overline{\sigma_i}$,P,v) $\rightsquigarrow$

```
if(bUsed&&b̄ᵢ) {
    ⟦unfold⟧σ
}
```

Listing 24: Information learned later should influence a decision made earlier. This example is taken from the original silver test suite.

```
field next : Ref;

predicate P(x:Ref) { acc(x.next) }

 method m(l:Ref)
  requires acc(l.next) && acc(l.next.next)
  {
    var x:Ref := l.next
    package acc(l.next,1/2) && acc(l.next.next,1/2) --*
       folding acc(P(x)) in acc(P(x)) && acc(l.next)
    assert acc(l.next,1/2) && acc(l.next.next,1/2)
  }
```

Using all assumptions in this case is justified since v is already evaluated in some state. Analogously we define σ_{Used}.`unfold`($\overline{\sigma_i}$,P,v) and σ_{Used}.`apply`($A \mathrel{-\!\!*} B$). But we don't change anything in the *exec* of the **`packaging`** ghost operation.

As a side remark, if the encoding described in Section 4.6.6 were used for the *transfer* then we could only use the assumptions which don't relate states in the **`if`**-conditions for the state operations just mentioned.

4.9 Completeness issues in current package encoding

In Section 4.6.4 we looked at a completeness issue with respect to the current *transfer* encoding. In this Section we take a closer look at other completeness which don't just involve the *transfer* encoding. In the following discussion we'll assume that the *transfer* definition given in Listing 18 and the encoding of state operations given in Section 4.4 is used. A partial Boogie translation for *transfer*($\overline{\sigma_i} \cdot \sigma_1$, σ_{Used}, *acc(e.f,p)*) is given in Listing 19. Our current Boogie encoding of the footprint computation algorithm has a few issues with respect to completeness. We list the most important points.

Knowledge learned later does not affect an earlier decision
For the completeness issue here, consider the example (this is an example from the Silver test suite) in Listing 24.

The example in Listing 24 should succeed. Let's look at the package in detail. Assume σ_1 represents the state satisfying the left hand side and σ_{cur}

is the current state right before the package. So first the **folding** ghost operation will be executed for `P(x)`, where `x` = σ_{cur}`.eval(l.next)`. So full permission for `l.next.next` must be transferred where the receiver `l.next` is evaluated in σ_{cur}. This means with the current information known the footprint computation algorithm removes all the permission to `l.next.next` from the current state. It won't remove anything from σ_1 since it's not known if σ_{cur}`.eval(l.next)` = σ_1`.eval(l.next)`. Now after the ghost operation is executed, there's a *transfer* for *P(x)*, followed by a *transfer* for the full permission to `l.next`. In the *transfer* of `l.next` half permission is removed from σ_1 and half is removed from σ_{cur} to `l.next`. But after this *transfer*, we learn that σ_{Used}`.eval(l.next)` equals σ_1`.eval(l.next)` and σ_{Used}`.eval(l.next)` equals σ_{cur}`.eval(l.next)` holds in σ_{Used}. σ_{Used} is the state into which the permissions are transferred. Hence we conclude that σ_1`.eval(l.next)` equals σ_{cur}`.eval(l.next)` holds in σ_{Used}. The question is now, should this information be made available to the ghost operation executed earlier? According to the footprint computation algorithm that we presented, the answer is not clear. Logically it might make sense, since as long as we're in a single package all ghost operations (except **packaging**) just use information from σ_1 and σ_{cur}, hence the fact that we later on learn that `l.next` is the same in both states means that we can use that fact in the entire **package**. This would then mean that during the **folding** ghost operation it would only be needed to remove half permission from to `l.next.next` from σ_{cur} which then makes the final assertion hold.

Now the reason why in our current encoding we won't learn this is because we explicitly add the *equate* assumption to σ_{Used} and since σ_{Used} doesn't exist at the moment when **folding** is executed, the assumptions in σ_{Used} won't play a role in **folding**.

Heap values in the footprint should be stored

Consider the example in Listing 25. The **package** succeeds and the footprint is the smallest state satisfying **acc**`(x.g,1/2)&&x.g==1`. The **apply** also succeeds and due to our encoding of the **apply** given in Section 3.4 `x.f==1` can be verified. The problem is that `x.g==1` won't be verified in our encoding. In theory it's sound if it is verified because the footprint for the packaged wand satisfied `x.g==1`. The reason our encoding in Boogie doesn't verify this, is because we don't associate the footprint's heap with every wand instance. It's also not that clear how to do that due to the incompleteness problem presented in Section 4.6.4(where the there are many traces which actually shouldn't be there).

Listing 25: Verifies if values in footprint are stored

```
inhale acc(x.g)&&acc(x.f)
x.f := 1
x.g := 1
package acc(x.f,1/2)&&x.f==1 --* acc(x.f,1/2)&&acc(x.g)
apply acc(x.f,1/2)&&x.f==1 --* acc(x.f,1/2)&&acc(x.g)
assert x.f==1
assert x.g==1
```

5 Evaluation

In this section we evaluate the encoding we described in Section 4 which uses the *transfer* encoding presented in Section 4.6.3. First we compare the performance between our implementation in Carbon and the current magic wand implementation in Silicon, which as described in Section 1 is a back-end verifier for Silver programs based on symbolic execution. The implementation also uses the high-level approach for the magic wand support introduced in [16].

We ran Carbon and Silicon on the majority of the test cases available in the Silicon magic wand test suite. We made a few changes to some of the test cases such that our implementation of Carbon can run them. For instance Silver provides a wand variable to store a wand at a certain program point, but which isn't added to the program state. This wand can be then used in the **package** operation. The following example shows what is meant:

```
method t01(x:Ref,y:Ref)
requires acc(x.f)
requires acc(y.f)
{
    var z:Ref
    z := x
    wand w := true --* acc(z.f)
    z := y

    package w
}
```

For testing purposes we would then rewrite this program to the equivalent Silver program:

```
method t01(x:Ref,y:Ref)
requires acc(x.f)
requires acc(y.f)
{
    var z:Ref
    var zTemp: Ref

    z := x
    wand w := true --* acc(z.f)
    zTemp := z

    z := y

    package true --* acc(zTemp.f)
}
```

We didn't implement this transformation automatically in Carbon, since there were very few test cases using the wand variable and it's not clear if the support for the wand variable will change in the future.

We left out the test cases in the test suite which use the *lhs* operator. This is an operator that can be used in wands to get a value evaluated in the state satisfying the left hand side of the wand. The reason for not implementing this operator is that the operator is probably going to be replaced in the future and also due to time constraints.

5.1 Performance: Silicon vs. Carbon

The subset of test cases and the time it took to execute them for Carbon and Silicon can be seen in Figure 1. The figure shows that Silicon outperforms Carbon on every test case. This was also seen in [8]. The reasons for this aren't completely clear but we suspect the reasons being the same as already presented in the evaluation section of [8], even though Silicon and Carbon have changed quite a bit since the time when [8] was written. One of the reasons is in test cases that don't verify the program properties specified Silicon may stop after finding the first verification error, while Carbon always is forced to look at the complete program. We notice that the differences in execution times in Figure 1 is less than two seconds in every example, except in one example where the difference is more than 10 seconds. A reason for this outlier could be that sequences are used in the example which in Carbon are handled a lot differently than in Silicon.

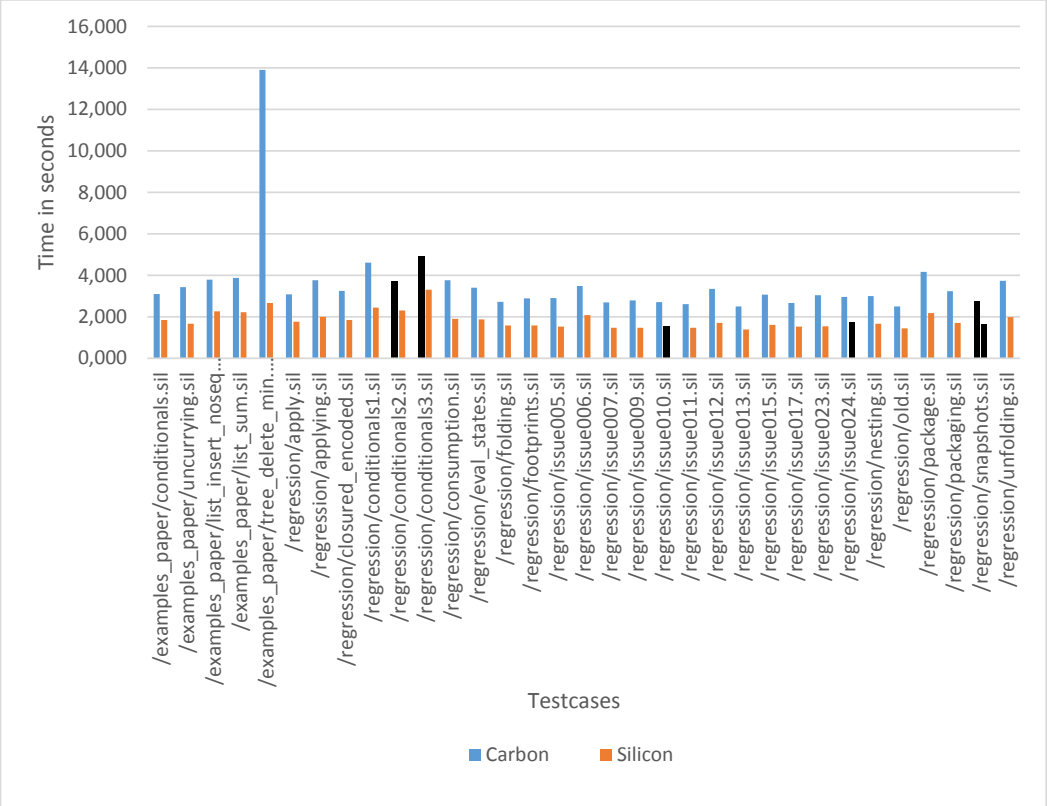

Figure 1: Execution time of a subset of test cases in the Silicon magic wand test suite when using Carbon and Silicon. For each test case the left column corresponds to Carbon and the right column to Silicon. If a column is black then it means that the corresponding verifier finished running the example in the shown time but it didn't return the expected verification results.

We also note that the potential soundness issue in Carbon described in Section 4.6.5 doesn't appear in any of the test cases which we compared Carbon and Silicon with. There are no known soundness issues in Silicon's encoding of magic wands.

5.2 Completeness: Silicon vs. Carbon

In Figure 1 the black columns indicate that the corresponding verifier (Carbon, if it's the left column for a test case or Silicon, if it's the right column for a test case) didn't return the expected verification results. We see that there are two test cases where Carbon doesn't verify certain properties but Silicon does and two test cases where Silicon doesn't verify certain properties

but Carbon does. Finally there's one test case which both don't verify.

The two test cases which Carbon doesn't directly verify are directly related to the completeness issue described in Section 4.6.4. A minimal example is given by:

```
inhale acc(x.f)
package true --* acc(x.f)
assert perm(x.f)==0
```

The `perm` operator can be used in Silver to get the permission to a specific heap location. As explained in Section 4.6.4 Carbon won't know for sure that there's no permission left. But note that Carbon would return a verification error for the following program due to the **`exhale`** (as will Silicon), which is the desired behaviour:

```
inhale acc(x.f)
package true --* acc(x.f)
exhale acc(x.f)
```

We point out that usually the `perm` operator is used for debugging purposes and not to verify practically relevant examples.

Let's now look at the two test cases Silicon doesn't verify. The first test case corresponds to a variation of Listing 5. Hence Silicon doesn't seem to solve the incompleteness that can arise in **`apply`** encodings that we discussed in Section 3.4. The completeness issue in the second test case is due to Silicon not noticing certain inconsistencies as discussed in Section 4.2. We just present two examples:

```
package acc(x.f)&&acc(x.f) --* false
```

The state satisfying the left hand side is inconsistent, so the **`package`** should succeed, but Silicon doesn't notice the inconsistency.

A second example is given by

```
inhale acc(x.f,1/2)
package acc(x.f,1/4)&&x.f==2 --*
    (packaging (acc(x.f,1/4)&&x.f==3 --* acc(x.f)) in true)
exhale acc(x.f,1/2)
```

We notice that the footprint computation algorithm transfers 1/4 permission to $x.f$ from the left hand side satisfying

```
acc(x.f,1/4)&&x.f==3
```

and then 1/4 permission to $x.f$ from the left hand side satisfying

```
acc(x.f,1/4)&&x.f==2
```

Then the footprint computation notices an inconsistency (since in the state to which these permissions are transferred $x.f$ has value 2 and 3 which isn't possible). According to Lemma 1 the **package** can then succeed without any further action, i.e. no permission needs to be removed from $x.f$ in the state before the **package**. Hence the **exhale** in the end should work. In Silicon this isn't the case, since it removes permission to $x.f$ from the state before the **package** (which can be checked by `assert perm(x.f)==0`, which is verified by Silicon).

The test case which both verifier fail to verify is given in Listing 25. Another test case which we didn't include in Figure 1 and which both verifiers fail to verify is given by Listing 24. See the discussion on both these examples in Section 4.9.

As one can see it's hard to say which verifier is more complete, they both have their strengths and weaknesses in terms of completeness. The main advantage with Silicon is that currently no potential soundness issues are known.

6 Extension: Basic Quantified Permission Support in Carbon

In this section we present work done on quantified permissions. This is quite unrelated to magic wands, which was the main topic of the thesis. Quantified permissions denote permissions for multiple locations instead of just one. Silicon (see [9] for an overview) already has support for quantified permissions (an earlier approach is described in [3]) and we provided a first step towards quantified permission support in Carbon as part of this bachelor's thesis. Some of our work was largely inspired by ideas taken from the current quantified permission implementation in Silicon.

6.1 Quantified Permissions in Silver

In Silver a quantified permission is of the following form:

```
∀ x:T :: c(x) ⇒ acc(e(x).f, p(x))
```

where T is any valid Silver type, `c(x)` returns a Boolean expression, `e(x)` is a pure expression of type *Ref* and is *injective* with respect to x and

Listing 26: Boogie encoding of inhale ∀x:T :: c(x) ⇒ acc(e(x).f, p(x))

```
1   var QPMask:MaskType;
2   havoc QPMask;
3
4   assume ∀x: T :: {e(x)} c(x) ==> inv(e(x)) = x
5   assume ∀r: Ref :: {inv(r)} c(inv(r)) ==> e(inv(r)) = r
6
7   assume ∀x1:T, x2:T :: ( x1 != x2 &&
8             c(x1)&&c(x2) ) ==> e(x1) != e(x2)
9
10  assume ∀x:T :: (c(x) && p(x) > 0) ==> e(x) != null
11
12  assume ∀r:Ref :: {Mask[r,f]}{QPMask[r,f]}{inv(r)}
13    ( c(inv(r)) ==> QPMask[r,f] == Mask[r,f]+p(inv(r)) ) &&
14    ( !c(inv(r)) ==> QPMask[r,f] == Mask[r,f] )
15
16  assume ∀r:Ref,h: (Field A B) :: {Mask[r,h]}{QPMask[r,h]}
17      h != f ==> QPMask[r,h] == Mask[r,h]
18
19  Mask := QPMask
```

`p(x)` returns a permission expression. The restriction that `e(x)` is injective makes it easier to support in verifiers. Suppose `e(x)` were not injective. Then it would be possible that there exist distinct x,y of type T such that `c(x)`,`c(y)` and `e(x)==e(y)` holds. This means when inhaling the above quantified permission we would need to add at least permission `p(x)` and permission `p(y)` to the location `e(x).f`. In general we would need to find all x' of type T, for which `e(x')==r` for an arbitrary reference r. Now adding the restriction that `e(x)` is injective means we know there is at most one such x'.

6.2 Inhaling Quantified Permissions

An encoding of **`inhale`** ∀x:T :: `c(x)` ⇒ **`acc`**`(e(x).f, p(x))` is given in Listing 26. Note that the terms in the curly brackets are triggers (see Section 2.5.5).

The identifier *inv* in Listing 26 is a fresh identifier for the corresponding quantified permission. This means for another quantified permission that we inhale, we use another identifier (i.e. technically we use another function).

The assumptions on line 4 and 5 define *inv(·)* to be the inverse function of *e(·)*, which should exist due to Silver's restriction that *e(·)* must be injective. On line 12 the permission value of the updated mask is set for all heap locations *z.f*. Using the inverse allows one to quantify over references instead of elements of type *T*. If we quantified over *T* then we would not be able to specify the permission for those heap locations *z.f* for which there is no *x:T* such that *e(x) = z*. This idea of using inverses was developed already while implementing quantified permissions in Silicon. On line 16 we specify the permission for all heap locations where the corresponding field is different from *f*.

6.2.1 Alternative Triggers

Consider the following Silver program:

```
1   inhale acc(y.g);
2   inhale acc(y.f);
3   exhale acc(y.g);
4   inhale (∀x:Ref :: x!= null ==> acc(x.f));
5   assert false;
```

Note that after the **inhale** on line 4 we have more than full permission to *y.f* and hence we reach an inconsistent state (the assertion of `false` should succeed). The issue is that the mask changes after the **exhale** of *y.g*. This means the verifier won't instantiate line 12 in Listing 26 with *y*, since *y.f* only occurs with respect to an older version of the mask. Therefore the verifier won't learn that the state is inconsistent. A possible solution would be to add the following function and corresponding axiom to the Boogie encoding:

```
function qptrigger<A, B>(o: Ref, f: (Field A B)): bool;
axiom (∀ <A, B> Mask: MaskType, r: Ref, f(Field A B) ::
  { Mask[o, f] }
  qptrigger(o, f)
);
```

Then we would replace the trigger quantified formula line 12 in Listing 26 by

```
{qptrigger[r,f]}{QPMask[r,f]}{inv(r)}
```

The verifier can then instantiate the formula for heap locations *x.f* which have been mentioned at least once in some version of the mask. But this potentially leads to many unnecessary instantiations of the axiom introduced. We don't use this alternative triggering in our implementation.

Listing 27: Boogie encoding of exhale ∀x:T :: c(x) ⇒ acc(e(x).f, p(x))

```
1   var ExhaleHeap: HeapType;
2   havoc ExhaleHeap;
3   var QPMask:MaskType;
4   havoc QPMask;
5
6   assert ∀x: T :: (c(x) && p(x) > 0) ==> e(x) != null)
7   assert ∀x: T :: c(x) ==> Mask[e(x),f] >= p(x)
8
9   assume ∀x1:T, x2:T :: x1 != x2 &&
10            c(x1)&&c(x2) ==> e(x1) != e(x2)
11
12  assume ∀x: T :: {e(x)} c(x) ==> inv(e(x)) = x
13  assume ∀r: Ref :: {inv(r)} c(inv(r)) ==> e(inv(r)) = r
14
15  assume ∀r:Ref :: {Mask[r,f]}{QPMask[r,f]}{inv(r)}
16    ( c(inv(r)) ==> QPMask[r,f] == Mask[r,f]-p(inv(r)) ) &&
17    ( !c(inv(r)) ==> QPMask[r,f] == Mask[r,f] )
18
19
20  assume ∀r:Ref,h: (Field A B) :: {Mask[r,h]}{QPMask[r,h]}
21      h != f ==> QPMask[r,h] == Mask[r,h]
22
23  Mask := QPMask;
24  assume identicalOnKnownLocs(Heap,ExhaleHeap,Mask);
25  Heap := ExhaleHeap;
```

6.3 Exhaling Quantified Permissions

An encoding of **exhale** ∀x:T :: `c(x)` ⇒ **acc**`(e(x).f, p(x))` is given in Listing 27. The encoding is analogous to the **inhale** case, just that the necessary properties are asserted before the mask is updated appropriately.

6.4 Framing Axiom

We introduced the concept of function framing in Section 2.3.1. The frame of a function is the set of heap locations on which a function depends. In Silver an overapproximation of the frame is given by the set of permission required by the precondition of a function. If the following can be shown

Listing 28: Example showcasing importance of frame of a function

```
field next: Ref
field val: Int

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

function sum(ys: Ref): Int
  requires acc(List(ys))
{ unfolding List(ys) in ys.val +
    (ys.next == null ? 0 : sum(ys.next)) }
```

with respect to two different function calls *f(a)*, *f(b)*:

1. a = b
2. the set of heap locations belonging to the (overapproximation of the) frame of *f* evaluated at the program point where *f(a)* was called evaluate to the same values as the same heap locations evaluated at the program point where *f(b)* was called

then it can be concluded that *f(a)* and *f(b)* evaluate to the same value. This can easily be generalized to functions with multiple arguments. We now briefly look at how Carbon uses this observation. Consider the Silver program in Listing 28. The frame of sum is given by *List(ys)*. Carbon now creates two functions for *sum* in Boogie:

```
function sum(Heap: HeapType, ys: Ref): int;
function sumFrame(frame: FrameType, ys: Ref): int;
```

The two functions are related as follows (for arbitrary `Heap:HeapType` and `ys:Ref`):

```
sum(Heap, ys) == sumFrame(Frame(Heap[null, List(ys)]), ys)
```

This way two functions calls to *sum* in Boogie can be seen as two function calls to *sumFrame* which evaluate to the same value exactly if the two conditions we looked at before hold. If this relation to *sumFrame* weren't present,

then if just some heap location not part of the frame of *sum* changes, the verifier would not be able to verify the equality of a function call before the heap modification and a function call after the heap modification.

Now we extend Carbon to define a correct frame if quantified permissions are part of the precondition. Consider the following function (the function body isn't important for the frame):

```
function test(y:T2):T3
requires ∀x:T :: c(x) ⇒ acc(e(x).f, p(x))
{ ... }
```

where the precondition is a valid quantified permission as defined in Section 6.1 and *T2,T3* are valid Silver types. So the frame of *test* is given by the set of heap locations *e(x).f* where $c(x) > 0$ and $p(x) > 0$.

In Boogie we then create the following functions for *test*:

```
function test(Heap:HeapType,y:T2):T3

function testFrame(frame: FrameType,y:T2):T3

function testQP1(Heap: HeapType, y:T2)
```

We then relate two function calls of *testQP1* as follows (for arbitrary `Heap1:HeapType,Heap2:HeapType,y:T2`):

```
(∀x:T :: c(x) ==> Heap1[e(x),f] == Heap2[e(x),f]) ==>
    testQP1(Heap1,y) == testQP2(Heap2,y)
```

This relation essentially states that if the set of heap locations where permission is potentially obtained in an inhale of ∀x:T :: `c(x)` ⇒ `acc(e(x).f, p(x))` evaluate to the same values in *Heap1* and *Heap2* then *testQP1(Heap1,y)* equals *testQP(Heap2,y)*. The reason why we input y is that in general the quantified permission can depend on y (it doesn't here, so it wouldn't be needed in this case).

Finally we can state the framing axiom by relating *test* with *testFrame* (for arbitrary `Heap:HeapType` and `y:T2`):

```
test(Heap, y) == testFrame(Frame(testQP1(Heap,y)), y)
```

Hence the verifier can learn that *test(Heap1, y)* equals *test(Heap2,y)* if *testQP1(Heap1,y)* equals *testQP2(Heap2,y)*.

We note that in practice we generate more functions than just presented since we need to ensure that recursive functions don't get instantiated an arbitrary number of times by the verifier. The details of the trigger encoding are given in [7].

6.5 Conclusion

In this section we make our final conclusions on the work regarding the encoding of quantified permissions in Carbon. We have implemented the encoding that we presented in a separate branch of Carbon. A variety of test cases verify but at this moment there are still certain test cases where in our encoding the verifier runs out of memory. We suspect this is due to us not having a clear *triggering* strategy. We observed that in most cases the quantified permission version of Silicon outperforms our encoding as already noticed with magic wand related tests in Section 5.

The work we did for the quantified permission support should be seen as a starting block for a more mature version. Future work could therefore certainly focus on, for example, finding a good strategy for *triggers* to improve performance.

7 Conclusion

In this section we make our final conclusions on the work regarding the encoding of magic wands in Carbon.

7.1 Status of Implementation

We have extended the Carbon verifier with support for magic wands in Silver. Our implementation passes a large majority of the magic wand related tests from the Silicon test suite as discussed in Section 5. All the tests which don't pass can be reduced to one of the examples presented in Sections 4.6.4 and 4.9. We have also written new magic wand related tests which complement the already available tests in Silicon.

As discussed in Section 4.6.5 our encoding is not entirely sound. It's not completely clear if the example we show really exhibits unsoundness, as we explain there are certain views which make the example presented not unsound. Yet the behaviour noticed is not consistent for a person who doesn't know the background of our encoding and is therefore not desired. It's not clear if the behaviour shown appears often in practical settings, for instance none of the existing test cases in the Silicon test suite led to this behaviour.

We haven't added support to the *lhs* operator and the wand variable as discussed in Section 5.

7.2 Future Work

Our current encoding of the `package` operation is incomplete in various settings and it's also not clear if it is sound. The main issue in the encoding of `package` is that we need to use assumptions on various states, as well as use assumptions that relate these states to direct the control flow of the algorithm. At the same time we don't want the verifier to learn some of these assumptions because otherwise we learn information on the state after the `package` which we shouldn't learn. These two goals are in direct conflict.

As future work one could try to extract the main abstract problem that we had when trying to encode the `package` operation and then analyse if this problem can be at all solved completely with Boogie. It might also make sense to first just find a Boogie encoding of the `package` which isn't practical at all, as long as one can show that the `package` can be encoded precisely the way it is specified. If it turns out that there is no Boogie encoding to precisely model the `package` then this would be an extremely interesting result.

Another direction to take would be to analyse the potential soundness issue in our own encoding in greater detail.

7.3 Final Conclusion

We have introduced a potential encoding for the magic wand support described in [16] for Carbon and we have implemented this encoding in separate fork of Carbon. We have also taken deeper look into the `package` operation in the presence of inconsistent states. Our work suggests that it doesn't seem trivial to find a precise encoding for the `package` operation using Boogie. Even though our encoding is not complete and is potentially unsound, it still achieves to verify a lot of examples in cases where we are pretty sure that there are no soundness issues.

7.4 Acknowledgements

First I would like to thank Alexander J. Summers, without whom many of the ideas in this thesis would not have come to fruition. Alex, I thank you for the great supervision, encouragement and for showing me how to elegantly reason about complicated problems. I would like to thank Prof. Peter Müller for giving me the opportunity to work on an extremely interesting problem in the field of software verification. I would like to thank Malte Schwerhoff for his comments on the `apply` encoding and his help with questions regarding Silicon and quantified permissions. Finally I would like to thank all people who contributed to the Viper tools, which made my own work much easier.

Listings

References

[1] Mike Barnett, Bor-YuhEvan Chang, Robert DeLine, Bart Jacobs, and K.RustanM. Leino. Boogie: A modular reusable verifier for object-oriented programs. In FrankS. de Boer, MarcelloM. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer Berlin Heidelberg, 2006.

[2] John Boyland. Checking interference with fractional permissions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag.

[3] Korbinian Breu. Quantified permissions for random access data structures. Master's thesis, ETH Zurich, TU Munich, 2014.

[4] Rémi Brochenin, Stéphane Demri, and Etienne Lozes. On the almighty wand. *Inf. Comput.*, 211:106–137, February 2012.

[5] Bernhard Brodowsky. Translating Scala to SIL. Master's thesis, ETH Zurich, 2013.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] S. Heule, I. T. Kassios, P. Müller, and A. J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In Giuseppe Castagna, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 7920 of *Lecture Notes in Computer Science*, pages 451–476. Springer, 2013.

[8] Stefan Heule. Verification condition generation for the intermediate verification language SIL. Master's thesis, ETH Zurich, 2013.

[9] U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. Technical report, ETH Zurich, 2014.

[10] C. Klauser. Translating Chalice into SIL. Bachelor's thesis, ETH Zurich, 2012.

[11] K. Rustan M. Leino. This is boogie 2, 2008.

[12] Toshiyuki Maeda, Haruki Sato, and Akinori Yonezawa. Extended alias type system using separating implication. In *Proceedings of TLDI 2011: 2011 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Austin, TX, USA, January 25, 2011*, pages 29–42, 2011.

[13] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 1–19, London, UK, UK, 2001. Springer-Verlag.

[14] M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3:01):1–54, 2012.

[15] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 247–258, New York, NY, USA, 2005. ACM.

[16] M. Schwerhoff and A. J. Summers. Lightweight support for magic wands in an automatic verifier. In J. Boyland, editor, *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer, 2015.

[17] Malte Schwerhoff. Symbolic execution for chalice. Master's thesis, ETH Zurich, 2015.

[18] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 148–172, Berlin, Heidelberg, 2009. Springer-Verlag.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

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

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

Title of work (in block letters):

Verification Condition Generation for Magic Wands

Authored by (in block letters):

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

Name(s):	**First name(s):**
Parthasarathy	Gaurav

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)**
Fislisbach, 17.08.2015	Gaurav Parthasarathy

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