Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Sound Automation of Magic Wands in a Symbolic-Execution Verifier

Bachelor Thesis

Nicola Widmer

Sunday 4th September, 2022

Advisors: Prof. Peter Müller, Dr. Malte Schwerhoff and Thibault Dardinier

Department of Computer Science, ETH Zürich

Abstract

Viper is an automatic verifier based on separation logic. Separation logic is an extension to Hoare logic, useful to reason about heap manipulating programs. In separation logic, there is an important connective, called the magic wand. The magic wand is useful, for example, to specify loop invariants of partial data structures. Due to its usefulness, the magic wand is supported in Viper. The key challenge of automating magic wands is to split a given state into two states such that in one state, called the footprint, the magic wand holds. This process is called packaging the wand. The current algorithm to package a wand in Viper sometimes computes a wrong footprint, which leads to unsound reasoning. In this thesis, we provide a sound package algorithm for a symbolic execution based backend of Viper. For the new and sound algorithm, we define and use the notion of "fixedness of heap locations" to extend the current algorithm. We reason about the soundness of the algorithm, and we give symbolic execution rules for the algorithm.

Acknowledgements

I would like to thank Dr. Malte Schwerhoff and Thibault Dardinier for the many things I could learn from them, the time they invested, and the feedback and many helpful inputs they gave me. Moreover, I would like to thank Prof. Dr. Peter Müller for giving me the opportunity to work on this project.

Contents

Chapter 1

Introduction

The world we currently live in is unimaginable without computer programs. Because of this, bugs in these computer programs can lead to massive catastrophes, which sometimes even end in people losing their lives, as the example of the Patriot missile systems shows [6]. To detect these bugs testing is often employed technique. Testing can ensure the absence of bugs for certain scenarios. However, in safety-critical areas, we want to prove the absence of bugs for "all" scenarios. In these cases, formal verification can be used. Formal verification is a way to prove that a program will not crash and will behave as expected. A widely used logic to reason about these properties is Hoare logic [3]. However, due to aliasing, it is hard to reason about concurrent or heap manipulation programs using Hoare logic. Separation logic [7] (SL hereafter) is an extension of Hoare logic to make reasoning about concurrent or heap manipulation programs easier. In SL, one important connectives the *separating conjunction*, $*$.
In SL, a program state is typically described by a heap, which is a partial map from locations to values. All locations in the domain of the heap are exclusively owned by its program state.
Intuitively, the SL assertion $A * B$ expresses that A and B hold in two disjoint portions of the program heap. More formally, $A * B$ holds in a state σ if and only if there are two states σ_A and σ_B with $\sigma = \sigma_A \uplus \sigma_B$ and A holds in σ_A and B holds in σ_B. Here, $\uplus$ denotes the combination of two compatible states. Two states are compatible, denoted by $\perp$ if the domains of their heaps are disjoint. The heap of the combination of two compatible states is just the combination of the heaps of both states.
In SL, another important connective is the *separating implication*, $-\!\!*$. The separating implication is usually called the magic wand. Intuitively, the magic wand is similar to the implication but for SL. If $A * (A -\!\!* B)$ holds in a state, then so does B. The formal definition of the magic wand is,

$$\sigma_{foot} \models A -\!\!* B \Leftrightarrow \forall \sigma_a \perp \sigma_{foot} \cdot (\sigma_a \models A \Rightarrow \sigma_a \uplus \sigma_{foot} \models B)$$

This means if a magic wand $A \mathrel{-\!\!*} B$ holds in a state σ_{foot}, then we can combine σ_{foot} with *any* state σ_a which is compatible with σ_{foot} and which satisfies A to get a state in which B holds. The magic wand is very useful, for example, to express invariants while traversing partial data structures such as lists or trees. This is the case, because $A \mathrel{-\!\!*} B$ intuitively refers to the data structure B "minus" the data structure A. An example of this is shown in Chapter 2.
As verification should ideally be used in conjunction with the development of a program, and with little effort from the programmer, there are many programs that automate the verification process. One such automatic verifier is Viper [5]. As the magic wand is such an important and useful connective it is also automated in Viper [8]. Unfortunately, the current algorithm in Viper is unsound [2]. But there is a proposal for a framework to characterize possible sound algorithms [2].
Viper [5] has two backends Carbon, which is based on verification condition generation, and Silicon [9], which is based on symbolic execution [4]. The previously introduced framework has been used to implement a sound way to automate magic wands in Carbon [2]. This thesis will present a sound way to automate magic wands in Silicon.
The structure of the thesis is the following:

- We first discuss the relevant background (Chapter 2).
- In Chapter 3, we give the intuition of our approach for automating magic wands.
- We then formalize the aforementioned intuition and justify its soundness (Chapter 4).
- In Chapter 5, we give symbolic execution rules for a possible implementation for the new algorithm in Silicon.
- Finally, we summarize our findings (Chapter 6)

Chapter 2

Background

This Chapter is concerned with the technical background of this thesis. We first introduce implicit dynamic frames [10] (IDF hereafter), a variation if SL (Section 2.1) and fractional permission [1] an extension to SL (Section 2.2). Then we introduce **inhale** and **exhale**, two important keywords in Viper (Section 2.3). Finally, we explain the usage of magic wands in an automatic verifier (Section 2.4) and how the automation is currently implemented (Section 2.5).

2.1 Implicit dynamic frames

Viper is based on IDF [10] which is a variant of SL. IDF and SL are known to be closely related [11]. The main difference between IDF and SL is that in IDF the information on the permission held for a heap location and the value of the heap location are separated. In SL, there is an important assertion called the points-to assertion. The points to assertion $x.f \rightarrow v$ means that the domain of the heap must contain the location $x.f$ and map it to v. $x.f$ represent a location where x is a reference and f is a field. In IDF the same assertion is represented by $acc(x.f)\&\&x.f == v$. The assertion $acc(x.f)$ means that the state has permission to the heap location $x.f$ and the assertion means $x.f == v$ that the heap location $x.f$ maps to v.

2.2 Fractional permission

Fractional permission [1] is a way to express heap location ownership at a more fine-grained level in both IDF and SL. With fractional permissions, the heap is not only a partial map from heap locations to values, but there is also a fraction associated with each heap location. The fractions are between zero and one. If a heap location is associated with a fraction of one it means the state has permission to read and write to that heap location. Zero permission

means neither read nor write permission and every other fraction means only read permission. In IDF, the assertion $acc(x.f, 1/2)$ can then for example be used to check whether the heap contains at least half permission to $x.f$.
When using fractional permissions two states are compatible if and only if for all heap locations contained in the heap of both states the values in both heaps are the same and the sum of both permissions amounts does not exceed one.

2.3 Inhale and Exhale

In Viper, one can **assert** an assertion to check if an assertion holds in a state. In addition, one can **inhale** or **exhale** an assertion.
Inhaling an assertion is basically adding the assumptions of the assertion to the state and adding the permission amounts specified by the assertion.
Exhaling an assertion basically corresponds to checking if the assertion holds followed by removing the permission amounts specified by the assertion.
In symbolic execution inhale and exhale are often referred to as produce and consume.

2.4 Automateing magic wands

As discussed before the assertion, $A \twoheadrightarrow B$ holds in a state σ_{foot} iff $\forall \sigma_a \perp \sigma_{foot} \cdot (\sigma_a \models A \Rightarrow \sigma_a \uplus \sigma_{foot} \models B)$. The state σ_{foot} is often called the footprint of a wand. To automate magic wands in Viper there are two important statements **package** and **apply**.

The **package** $A \twoheadrightarrow B$ statement tries to split the current state σ into two compatible states σ_{foot} and σ_{rest} where $A \twoheadrightarrow B$ holds in σ_{foot}. The verification process then continues by removing σ_{foot} from σ such that only σ_{rest} remains, but in addition, we record in the heap that the state owns an instance of the magic wand $A \twoheadrightarrow B$. If no such two states are found the verification fails. Intuitively, the statement trades a state satisfying the wand for the wand itself.

The **apply** $A \twoheadrightarrow B$ statement is used to apply the modus ponens-like rule $A * (A \twoheadrightarrow B) \Rightarrow B$. The statement is only applicable in a state holding the wand. Applying a magic wand $A \twoheadrightarrow B$ basically corresponds to exhaling A then removing the wand $A \twoheadrightarrow B$ from the state and finally inhaling B.

Listing 2.1 shows how magic wands are used in Viper and their usefulness in expressing invariants. The method `main` iterates once over the linked list starting at x. The predicate `List(x)` specifies, in a recursive fashion, permission to the whole linked list x (lines 3-6). As the pre- and postcondition of `main` is `List(x)` (lines 9 and 10), the method guarantees memory safety,

```
1  field next: Ref
2
3  predicate List(x: Ref)
4  {
5      acc(x.next)&&(x.next!=null==>List(x.next))
6  }
7
8  method main(x:Ref)
9  requires List(x)
10 ensures List(x)
11 {
12     var y:Ref:=x
13     package List(x) --* List(x)
14
15     while(y.next!=null)
16     invariant List(y) && (List(y) --* List(x))
17     {
18         y:=y.next
19         package List(y) --* List(x)
20         //{hints for package}
21     }
22
23     apply List(y) --* List(x)
24 }
```

Listing 2.1: The method main and its specification describe a memory-safe iteration over the linked list x. The predicate List expresses permission of a linked list. The invariant uses a magic wand to express permissions to the linked list x "minus" the linked list y yet to be traversed. Viper needs some advice to package some wands, called a proof script. We omit the proof script, but we made a comment about it (line 20).

which means the caller of main has to provide permission to the linked list x but after the call, it gets the permission back. The hardest part of verifying this method is finding an appropriate invariant for the loop (line 16). The loop invariant must provide permission to the sublist starting at y, in order to read the value `y.next`. This is provided by the predicate `List(y)`. In addition, the loop invariant must provide permission to the rest of the list as in the end we have to combine both to gain back permission to the whole list. Here is where the magic wand comes in handy as `List(y)--*List(x)` represents the list starting at x "minus" the list starting at y. The first package statement (line 13) is used to create the wand `List(y)--*List(x)` to fulfill the loop invariant at the start of the loop. The second package statement (line 19) is used to create the wand `List(y)--*List(x)` to fulfill the loop invariant at the end of the loop body. We have to package a new wand as y changes in the loop body. At the end (line 23) we can use the apply statement to get back all the permissions to the linked list `x`, as it trades `List(y)&&(List(y)--*List(x))` for `List(x)` in a modus ponens-like fashion.

2.5 The Footprint Inference Attempt (FIA)

An important step in automating magic wands is split a state into σ_{foot} and σ_{rest} as described in Section 2.4. We call this procedure the package algorithm. In Silicon, this is currently done by the Footprint Inference Attempt (hereafter FIA). To package a magic wand $A \mathrel{-\!\!*} B$, the FIA constructs an arbitrary state σ_A in which A is satisfied by inhaling A in an empty state. Then the FIA tries to construct a state σ_B in which B holds, by trying to take the permissions it needs from σ_A and, if σ_A does not contain them, from the current state. At the end of the FIA, the footprint is already implicitly removed from the current state.
In Figure 2.1 the process of packaging the wand **acc**`(x.f)--*`**acc**`(x.f)&&`**acc**`(y.f)` in a state σ where we have **acc**`(x.f)&&`**acc**`(y.f)` is illustrated. In the first column, we have the states σ, σ_A, and σ_B at the start of the algorithm. The state σ_A is created by inhaling **acc**`(x.f)` into an empty state, therefore the heap location $x.f$ is present in the heap. The FIA then tries to prove the assertion **acc**`(x.f)&&`**acc**`(y.f)` in σ_B. For the first conjunct **acc**`(x.f)` permission to $x.f$ can be transferred from σ_A to σ_B. The corresponding states after this transfer are illustrated in the second column. For the second conjunct **acc**`(y.f)` permission to $y.f$ has to be transferred from σ to σ_B, as σ_A does not contain permission to $y.f$. The implicit footprint computed by the FIA is **acc**`(y.f)` as this is what is removed from σ.
The problem is now that the algorithm in some cases does not compute a correct footprint as we will explain in the next Chapter.

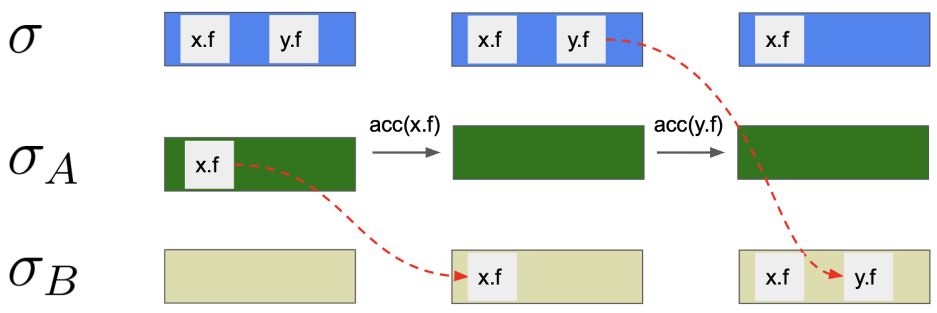


Figure 2.1: A visualization of the FIA for the magic wand **acc**`(x.f)--*`**acc**`(x.f)&&`**acc**`(y.f)` packaged in the state σ. The grey boxes represent the heap locations $x.f$ and $y.f$ respectively. If a heap location is present it means that the heap of the state contains the heap location. σ_A is the state in which A holds. We traverse **acc**`(x.f)&&`**acc**`(y.f)` and try to satisfy it in σ_B. The footprint is everything removed from σ during this process, in this case, **acc**`(y.f)`.

Chapter 3

Key Idea: Fixed Heap Locations and Expressions

3.1 Problem: Unjustified assumptions of the FIA

The current package algorithm makes some assumptions on the state in which the packaged wand will be applied. These assumptions are on the value of some heap-dependent expressions in the state. The problem is now that these assumptions may no longer hold when the wand is applied. To make these more precise, let us introduce the following notation.

Notation 3.1.1. Let σ_{assume} be the state in which the package algorithm assumes the wand will be applied in.

Notation 3.1.2. Let σ_{apply} be the state in which the wand is actually applied.

Notation 3.1.3. Let σ_{foot} be footprint computed by the current algorithm.

As an example of an assumption made by the current algorithm, let us look at how it branches over ternary operators $e ? a_1 : a_2$. It does so by creating one branch where $(\sigma_{assume} \uplus \sigma_{foot})(e)$ is assumed to be true and one where it is assumed to be false. It then computes the footprint under these assumptions for both branches. Let us call them $foot_\top$ and $foot_\bot$. Finally, the algorithm joins $foot_\top$ and $foot_\bot$ to obtain the footprint $(\sigma_{assume} \uplus \sigma_{foot})(e) ? foot_\top : foot_\bot$. But this is not always a correct footprint, because between the packaging and the application of the wand some memory locations accessed by e could be changed in such a way that $(\sigma_{apply} \uplus \sigma_{foot})(e) = \neg(\sigma_{assume} \uplus \sigma_{foot})(e)$ and $\sigma_{apply} \models A$ where A is the left-hand side of the packaged wand. The issue comes from the unjustified assumption that the value of all expressions branched over cannot change between packaging and application.
Let us look at a concrete example that shows the problem.

Example 3.1.1. For the wand `acc(x.f) --* acc(x.f) * (x.f ? acc(y.f) : acc(z.f))` the currently computed footprint is $(\sigma_{assume} \uplus \sigma_{foot})(x.f)\ ?\ acc(y.f)\ :\ acc(z.f)$. But now consider a state σ_{apply} where the heap contains full access to $x.f$ and the value of $x.f$ is $\neg(\sigma_{assume} \uplus \sigma_{foot})(x.f)$. Then we have that $\sigma_{apply} \models acc(x.f)$, as the only requirement is full access to $x.f$, therefore the wand can be applied in σ_{apply}. But $\sigma_{apply} \uplus \sigma_{foot} \not\models acc(x.f) * (x.f?acc(y.f) : acc(z.f))$ because if $(\sigma_{apply} \uplus \sigma_{foot})(x.f)$ is true then $(\sigma_{assume} \uplus \sigma_{foot})(x.f)$ is false, and therefore $acc(y.f)$ is not satisfied, and therefore σ_{foot} is not a correct footprint. If $(\sigma_{apply} \uplus \sigma_{foot})(x.f)$ is false, $acc(z.f)$ is not satisfied for similar reasons.

This example shows that some assumptions are unjustified and lead to incorrect footprints. Some assumptions, however, *are* justified. Let us explore deeper the discrepancy between σ_{assume} and σ_{apply}. Recall that σ_{apply} must satisfy A (otherwise the wand cannot be applied in σ_{apply}). Therefore, there are two factors that, combined, may make the assumption, that an expression e cannot change its value between packaging and application of the wand, justified:

Footprint Factor We do not have enough permissions to change the value of e. This is the case when the footprint contains some permission to all heap locations in e. Because these permissions have been hidden in the magic wand, and thus it is impossible to get full permission to these heap locations without applying the wand.
For example, after packaging the wand $true \twoheadrightarrow acc(x.f)$, the value of $x.f$ cannot change because all permission to $x.f$ are in the footprint

Left-Hand Side Factor The wand cannot be applied if we change the value e. This is the case when the left-hand side of the wand requires that e has a certain value.
For example, in order to apply the wand $acc(x.f) * x.f \twoheadrightarrow acc(x.f)$, $x.f$ has to be true, and therefore we cannot change the value of $x.f$ to false if we want to apply the wand.

3.2 Solution: Fixed heap locations and expressions

We have seen that the assumption that an expression e changes its value between packaging and application of the wand, can be justified by the Footprint Factor or the Left-Hand Side Factor. But in general, the assumption is unjustified. Therefore, our solution is to determine which assumptions are justified, and then do the following:
If the assumption is justified, we join branches as the current algorithm does.
If the assumption is unjustified, we join the branches by taking an upper

bound of both computed footprints. An upper bound of two footprints $foot_1, foot_2$ is a footprint which has, for any heap location at least as much permission as $foot_1$ and $foot_2$. This is sound because no matter which value the condition has, the footprint is definitely big enough because we took an upper bound of both cases.

3.3 Examples

Let us look at a few examples. In the first example, we will have a look at a wand for which the assumption is unjustified.

Example 3.3.1. `acc(x.f) && (x.f ? acc(a.f,1/2) : acc(b.f,1/2)) --* acc (a.f,1/2) && acc(b.f,1/2)`

In this case, the current algorithm would branch over $x.f$, and in the case where $x.f$ is assumed to true, the footprint would be $acc(b.f, 1/2)$ because the left-hand side provides `acc(a.f,1/2)`, so only access to $b.f$ has to be provided by the footprint. And for similar reasons in the case where $x.f$ is assumed to be false the footprint is `acc(a.f,1/2)`. So the current algorithm would compute the footprint `x.f?acc(a.f,1/2):acc(b.f,1/2)`. However, this footprint is incorrect: We can actually derive false in Viper, as shown in Listing 3.1. We do this, by setting $x.f$ to false if we lost half permission to $a.f$, and to true otherwise. We can do this because we have full permission to $x.f$. As the left-hand side of the wand is still fulfilled the wand can still be applied. But now if we apply the wand we have to give up permission to the heap location for which the footprint has already given up permission. In return, we gain permission to the heap location for which we never gave up any permission. As we now have a permission amount of more than one for a heap location we can prove false in Viper. As we started in a consistent state and derived false this is clearly an unsoundness.

In this case, neither the Footprint Factor nor the Left-Hand Side Factor from Section 3.1 is applicable for the expression $x.f$: No permission to $x.f$ is in the footprint, and the left-hand side does not enforce a value for $x.f$. Therefore, the new algorithm should detect that $x.f$ could change between packaging and applying, and therefore should compute an upper bound of both footprints which, should be `acc(a.f,1/2)&&acc(b.f,1/2)`.

The next two examples are examples in which the assumption is justified.

Example 3.3.2. `acc(x.f,1/2) && (x.f ? acc(a.f,1/2) : acc(b.f,1/2)) --* acc(a.f,1/2) && acc(b.f,1/2) && acc(x.f)`

```
1  field f: Bool
2
3  method main(x: Ref, a: Ref, b: Ref)
4   requires acc(x.f) && acc(a.f) && acc(b.f)
5   ensures false
6  {
7   package acc(x.f) && (x.f ? acc(a.f, 1/2) : acc(b.f, 1/2))
      --* acc(a.f, 1/2) && acc(b.f, 1/2)
8
9   assert (perm(a.f) == 1/1 && perm(b.f) == 1/2) || (perm(a.f
      ) == 1/2 && perm(b.f) == 1/1)
10
11  x.f := perm(a.f) != 1/1
12
13  apply acc(x.f) && (x.f ? acc(a.f, 1/2) : acc(b.f, 1/2))
      --* acc(a.f, 1/2) && acc(b.f, 1/2)
14 }
```

Listing 3.1: Viper program to illustrate the unsoundness of the current package algorithm. Line 9 is needed otherwise the verification process would fail due to incompleteness. On line 11 we set $x.f$ to the desired value using the **perm** keyword, which returns the permission amount to a heap location in the given state. This program is verified in Silicon.

In this case, `(x.f?`**`acc`**`(a.f,1/2):`**`acc`**`(b.f,1/2)) && `**`acc`**`(x.f,1/2)` is a correct footprint, because the value of $x.f$ cannot change between packaging and applying. This is the case as here the Footprint Factor is applicable, as part of the footprint permission to $x.f$ was removed and therefore there is not enough permission to change the value of $x.f$ after the packaging.

Example 3.3.3. `acc(x.f) && x.f==c && (x.f ? acc(a.f,1/2) : acc(b.f,1/2)) --* acc(a.f,1/2) && acc(b.f,1/2)`

In this case, `x.f?`**`acc`**`(a.f,1/2):`**`acc`**`(b.f,1/2)` is a correct footprint, because here the Left-Hand Side factor is applicable, as the left-hand side of the wand enforces the value of $x.f$ to be c when applying and packaging the wand.

So one goal of the sound algorithm should be finding out which branch conditions could change their value between packaging and applying a wand. In the next Section, we introduce the notion of *fixed* expressions, to formalize this idea. A heap location or expression should be considered fixed if it *must* evaluate to the same value in $\sigma_{assume} \uplus \sigma_{foot}$ and $\sigma_{apply} \uplus \sigma_{foot}$.

Chapter 4

Formalization of a Sound Approach

In this Chapter, we formalize our approach to compute valid footprints, by leveraging the current algorithm and determining which assumptions are justified and which are not. We determine which assumptions are justified using the notion of fixedness; intuitively, a heap location (or an expression) is fixed iff the value of the heap location (or expression) is the same in all states in which the wand can be applied.

We first give a formal definition of fixedness in Section 4.2, then explain how to use this notion to compute a correct footprint in 4.3. However, it is hard to compute explicitly the set of heap locations that are fixed, because it involves quantification over states. Therefore, we explain in Sections 4.4 and 4.5 how to overapproximate this set in a sound manner. Finally, we describe the precision we lose due to the overapproximation in Section 4.6.

In this Chapter, we will use the following notation: e denotes an expression, b denotes a boolean expression, and $e.f$ denotes a heap location[1]. As previously described in Chapter 1 $\sigma_1 \perp \sigma_2$ means that states σ_1 and σ_2 are compatible, and $\sigma_1 \uplus \sigma_2$ is the combination of two compatible states.

4.1 Subset of Viper

In this and the following Chapter, we will only consider a subset of the Viper language. In particular, the left-hand side and the right-hand side of our magic wands are only assertions in the following grammar:

$$A \; = \; e \mid A * A \mid acc(e.f, e) \mid e\,?A : A$$

$$e \; = \; localvar \mid op(\bar{e}) \mid e.f \mid e\,?e : e$$

[1]More precisely, we would call it a heap location expression, but for readability purposes, we will refer to it as heap location.

Here $op(\bar{e})$ stands for an operation including arbitrary many pure subexpressions, such as equality; inequality; arithmetic, and boolean operations. And $e_1 \,?e_2 : e_3$ stands for the ternary operator which has the value of e_2 if e_1 evaluates to true and the value e_3 otherwise. And $e\,?A : A$ is the ternary operator for assertions.
In the full Viper language, the left-hand side and the right-hand side of magic wands can also contain predicates, quantified expressions, quantified permissions, or even other magic wands.
Another part that we ignore is proof scripts for magic wands, which are basically hints for the package algorithm on how to package a magic wand.

4.2 Fixedness of heap locations and expressions

To formally define the concept of *fixedness*, we first need to define what an applicable state is.

Definition 4.2.1 (Applicable state). *A state σ is applicable with respect to an assertion A and a state σ_{foot} iff $\sigma \perp \sigma_{foot}$ and $\sigma \models A$.*

Note that if a state is applicable with respect to A and σ_{foot} then it is a state in which the wand $A \mathrel{-\!\!*} B$ packaged with the footprint σ_{foot} could be applied. As the notion of fixedness involves all states for which a wand is applicable we use the following definition to define this set.

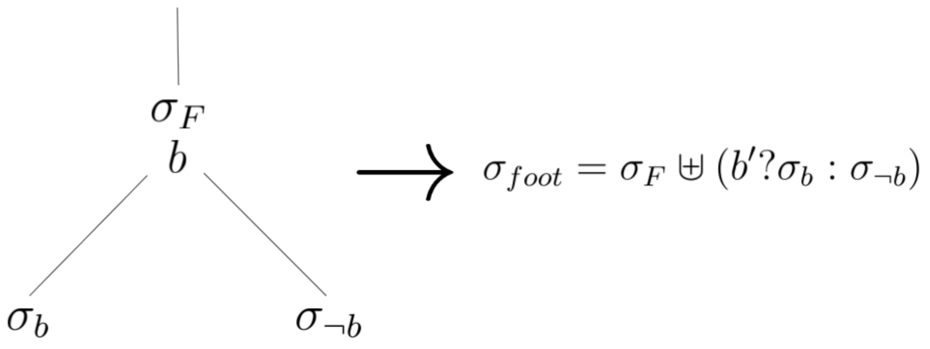
Definition 4.2.2 (Set of applicable states). *Let $SAS(A, \sigma_{foot})$ be the set of all states which are applicable with respect to the assertion A and the footprint σ_{foot}.*

Now we can proceed with the notion of fixedness. A heap location is fixed iff for all applicable states combined with σ_{foot} it evaluates to the same value. More formally the definition of fixedness for a heap location is,

Definition 4.2.3 (Fixedness of heap locations). *A heap location $e.f$ is fixed with respect to an assertion A and a state σ_{foot}, written fixed$(e.f, A, \sigma_{foot})$ iff*
$\forall \sigma_{a1}, \sigma_{a2} \in SAS(A, \sigma_{foot}) \cdot (\sigma_{a1} \uplus \sigma_{foot})(e.f) = (\sigma_{a2} \uplus \sigma_{foot})(e.f)$.

Two reasons why a heap location can be fixed are given in Section 3.1 by the Footprint Factor and the Left-Hand Side Factor. Intuitively the two reasons are:

- If σ_{foot} contains the heap location then in every applicable state the value of the heap location must be the same as in the footprint.
- If the assertion A requires the heap location to have a certain value.



σ_F
b
σ_b
$\sigma_{\neg b}$
$\sigma_{foot} = \sigma_F \uplus (b'?\sigma_b : \sigma_{\neg b})$

Figure 4.1: Illustration on how the current algorithm handles branches. b is the branch condition. σ_F is the footprint computed until the branch point. σ_b and $\sigma_{\neg b}$ are the footprints of the branches after the branch point. b' is the value $()\sigma_{assume} \uplus \sigma_F)(b)$. σ_{foot} is the final footprint.

The algorithm we will present will compute an overapproximation of which heap locations are fixed. But sometimes we also want to reason whether an expression is fixed or not; The next definition and theorem fulfill that purpose. Intuitively, an expression is fixed if its value is the same in all applicable states combined with σ_{foot}.
Or more formally,

Definition 4.2.4 (Fixedness of expressions). *An expression e is fixed with respect to an assertion A and a state σ_{foot}, written $fixed(e, A, \sigma_{foot})$ iff $\forall \sigma_{a1}, \sigma_{a2} \in SAS(A, \sigma_{foot}) \cdot (\sigma_{a1} \uplus \sigma_{foot})(e) = (\sigma_{a2} \uplus \sigma_{foot})(e)$.*

The next theorem gives us a way to detect fixed expressions given that we know which heap locations are fixed. Namely, an expression is fixed, if all heap locations it refers to are fixed.

Theorem 4.2.1. *If $\forall e_2.f \in heaplocs(e_1) \cdot fixed(e_2.f, A, \sigma_{foot})$ then $fixed(e_1, A, \sigma_{foot})$*

Proof. By structural induction on the expression e. $\square$

Here heaplocs(e_1) represents the set of all heap locations syntactically referenced by e_1.

4.3 Fix the current algorithm using fixed expressions

In this Section, we will show how we can use the current algorithm together with the information about which expressions are fixed to create a correct footprint.

Let us first have a look again at how the current algorithm handles branches. As illustrated in Figure 4.1, the current algorithm first computes the footprint until the branching point, here denoted by σ_F. Then it branches on the condition b and computes what has to be added to the footprint. If b is true, σ_b has to be added and if b is false $\sigma_{\neg b}$ has to be added. This is then combined to $\sigma_F \uplus (b'?\sigma_b : \sigma_{\neg b})$ which is the final footprint.
Let us look at a concrete example of this procedure.

Example 4.3.1. For the wand `true --* acc(x.f) && (x.f ? acc(y.f) : acc (z.f))` the branching condition is $x.f$. σ_F is `acc(x.f)` as it is the footprint recorded until the branching point. σ_b is `acc(y.f)` because if $x.f$ is true `acc( y.f)` has to be provided by the footprint. For similar reasons, $\sigma_{\neg b}$ is `acc(z.f)`. Therefore, the final footprint is `acc(x.f) && (x.f' ? acc(y.f) : acc(z.f))`. Here, $x.f'$ stands for the value of $()\sigma_{assume} \uplus \sigma_F)(x.f)$. We have seen that this is a wrong footprint as $(\sigma_{apply} \uplus \sigma_F)(x.f)$ might evaluate to $\neg x.f'$.

Even though the footprint computed by the current algorithm is incorrect some assumptions made about it and the states σ_F, σ_b and $\sigma_{\neg b}$ are still correct and useful. Two of these assumptions are stated below as they will be important to reason about the soundness of the new package algorithm. We will not prove their correctness. All the assumptions are for the above case where we branch once over b and a wand where A represents the left-hand side and B represents the right-hand side.
The first assumption is:

Assumption 4.3.1. *$\forall \sigma_A \in SAS(A, \sigma_F)$ it holds that $\sigma_A \uplus \sigma_F$ has enough permission to evaluate b*

This is the case because the wand has to be self framing.

The next assumption is that the footprint computed for each branch is a correct footprint for the corresponding branch, or more formally:

Assumption 4.3.2. *$\forall \sigma_A \in SAS(A, \sigma_F)$ where $(\sigma_A \uplus \sigma_F)(b)$ is true and $\sigma_A \bot \sigma_b$ it holds, that $\sigma_A \uplus \sigma_F \uplus \sigma_b \models B$.*
And similarly, $\forall \sigma_A \in SAS(A, \sigma_F)$ where $(\sigma_A \uplus \sigma_F)(b)$ is false and $\sigma_A \bot \sigma_{\neg b}$ it holds, that $\sigma_A \uplus \sigma_F \uplus \sigma_{\neg b} \models B$.

Using these assumptions we will now state two theorems related to the idea for a sound package algorithm given in Section 3.2. As explained in Section 3.2, if the expression over which the algorithm branched is not fixed, we compute an upper bound of the footprints computed in both branches and take it as the footprint. The following theorem shows that this is indeed a correct footprint.

Theorem 4.3.1. *$\sigma_F \uplus (\sigma_b \cup \sigma_{\neg b})$ is a correct footprint for the wand $A \mathrel{-\!\!*} B$*

Here, $\sigma_b \cup \sigma_{\neg b}$ denotes a state which is an upper bound to both σ_b and $\sigma_{\neg b}$.

Proof. From assumption 4.3.2 we know $\forall \sigma_A \in \text{SAS}(A, \sigma_F \uplus (\sigma_b \cup \sigma_{\neg b}))$ either $\sigma_A \uplus \sigma_F \uplus \sigma_b \models B$ or $\sigma_A \uplus \sigma_F \uplus \sigma_{\neg b} \models B$ holds depending on whether $(\sigma_A \uplus \sigma_F)(b)$ is true or not. And because we are in an intuitionistic logic and $\sigma_A \uplus \sigma_F \uplus (\sigma_b \cup \sigma_{\neg b})$ is bigger or equal then both $\sigma_A \uplus \sigma_F \uplus \sigma_b$ and $\sigma_A \uplus \sigma_F \uplus \sigma_{\neg b}$, we have that $\sigma_A \uplus \sigma_F \uplus (\sigma_b \cup \sigma_{\neg b}) \models B$, no matter whether $(\sigma_A \uplus \sigma_F)(b)$ is true or not. And therefore, $\sigma_F \uplus (\sigma_b \cup \sigma_{\neg b})$ is a correct footprint for the wand $A \mathrel{-\!\!*} B$ by definition. □

However, when the expression over which the algorithm branches is fixed, we can be more precise, as described by the following theorem.

Theorem 4.3.2. *If fixed(b, A, σ_F) then either $\sigma_F \uplus \sigma_b$ or $\sigma_F \uplus \sigma_{\neg b}$ is a correct footprint for the wand $A \mathrel{-\!\!*} B$.*

Proof. We prove this by case distinction.

Case 1: SAS$(A, \sigma_F) = \emptyset$
As no state that models A is compatible with σ_F, also no state that models A is compatible with $\sigma_F \uplus \sigma_b$, and therefore, $\sigma_F \uplus \sigma_b$ is a footprint for any wand with the left-hand side A.

Case 2: SAS$(A, \sigma_F) \neq \emptyset$
From our assumption that fixed(b, A, σ_F) we know that $(\sigma_A \uplus \sigma_F)(b)$ has the same value for all states $\sigma_A \in \text{SAS}(A, \sigma_F)$. Now assume without loss of generality that this value is true. Then from assumption 4.3.2 and the fact that $\text{SAS}(A, \sigma_F \uplus \sigma_b) \subseteq \text{SAS}(A, \sigma_F)$ we now know that $\forall \sigma_A \in \text{SAS}(A, \sigma_F \uplus \sigma_b) \cdot \sigma_A \uplus \sigma_F \uplus \sigma_b \models B$. And therefore, $\sigma_F \uplus \sigma_b$ is a correct footprint for the wand $A \mathrel{-\!\!*} B$. □

These two theorems lead us to an algorithm to compute a correct footprint using the states σ_F, σ_b, and $\sigma_{\neg b}$ computed by the current algorithm. Namely, if we know that fixed(b, A, σ_F) holds we can branch as we did before. This is sound, as in at least one branch we have a correct footprint. To prove correctness of the program we have to prove correctness in both branches and therefore definitely in the branch with the correct footprint. And if we cannot prove that fixed(b, A, σ_F) holds we take $\sigma_F \uplus (\sigma_b \cup \sigma_{\neg b})$ as the footprint for the wand. Because of theorem 4.3.1 we know that this is always a correct footprint. This means if we know which expressions are fixed we can augment the current algorithm to compute a correct footprint. The next Chapters are about how we can find the set of fixed expressions or at least an overapproximation of it.

4.4 Overapproximating the set of fixed heap locations

As it is hard to find the exact set of fixed expressions we will try to find the set of fixed heap locations and use theorem 4.2.1 to get an overapproximation of the set of fixed expressions. As it is also hard to find the exact set of fixed heap locations this Section will cover a way to compute an overapproximation of this set.
Given an assertion A and a footprint σ_{foot} intuitively there are two reasons why a heap location is fixed: Either because the footprint contains part of the heap location; this is what we called the Footprint Factor in Section 3.1 and is described by lemma 4.4.1. Or because the assertion requires that the heap location has a certain value; this is what we called the Left-Hand Side Factor in Section 3.1. Lemma 4.4.2 tells us that if a part of an assertion, together with the footprint already fixes a heap location then adding something to the assertion does not "unfix" the heap location.

Lemma 4.4.1. *If $e.f$ is in the heap of σ_{foot} then fixed$(e.f, A, \sigma_{foot})$*

Proof. $e.f$ is in the heap of $\sigma_{foot} \implies$
$\sigma_{foot}(e.f) = \sigma_{foot}(e.f) \implies$
$\forall \sigma_{a1}, \sigma_{a2} \in \mathrm{SAS}(A, \sigma_{foot}) \cdot (\sigma_{a1} \uplus \sigma_{foot})(e.f) = (\sigma_{a2} \uplus \sigma_{foot})(e.f)$ □

Lemma 4.4.2. *fixed$(e.f, A_1, \sigma_{foot})$ implies fixed$(e.f, A_1 * A_2, \sigma_{foot})$*

Proof. $\{\sigma | \sigma \models A_1 * A_2\} \subseteq \{\sigma | \sigma \models A_1\} \implies$
$\mathrm{SAS}(A_1 * A_2, \sigma_{foot}) \subseteq \mathrm{SAS}(A_1, \sigma_{foot})$ □

It can also happen that a sub-expression of the assertion does not directly fix a heap location, but it does it if the rest of the assertion or the footprint fixes another location. This is shown by the following example:

Example 4.4.1. `acc(x.f)&&acc(y.f,1/2)&&x.f==y.f--*acc(y.f)`
The left-hand side of this wand together with its footprint `acc(y.f,1/2)` fixes both $y.f$ and $x.f$. $y.f$ is fixed because it is part of the footprint and $x.f$ is fixed because $y.f$ is fixed and the wand's left-hand side requires that $x.f$ and $y.f$ are equal.

4.4.1 Algorithm to syntactically overapproximate the set of fixed heap locations

The findings of the last Section give rise to an algorithm that syntactically overapproximates the set of heap locations that are fixed. We first go over the structure of the left-hand side of the wand we want to package and compute a dependency graph for fixed heap locations. We represent this dependency graph with a set of pairs with type $(Set(heaplocation), heaplocation)$ and call it *dependent fixed graph* (dfg). If $(s, e.f)$ is in the dependent fixed graph of A then it means that A and σ_{foot} fix $e.f$ if all heap locations in s are fixed by A and σ_{foot}. We then instrument the current algorithm to compute the footprint of the wand to get the information of which heap locations are fixed by the footprint. It is sound to use the current algorithm because the footprint we will compute in the end is bigger or equal to the one computed by the current algorithm. And therefore, the footprint, in the end, will at least contain all heap locations that the footprint computed by the current algorithm contains. Then, we use the dfg and information from the footprint to compute the set of fixed heap locations in the following way: We keep track of the *set of fixed heap locations* (sfhl). Initially, sfhl is the set of heap locations contained in the footprint. Then, we go over the pairs $(s, e.f)$ in the dfg and add $e.f$ to the sfhl if $s \subseteq$ sfhl. We repeat the process until we reach a fix point. The sfhl at this fix point is the final set of fixed heap locations. Let sfhl(σ_{foot},dfg) denote the set of fixed heap locations computed this way given the footprint σ_{foot} and the dependent fixed set dfg.

Let us have a look at an example where we compute the sfhl given a dfg and a footprint.

Example 4.4.2. dfg $= \{(\{y.g, x.g, b.g\}, z.g), (\{a.g\}, y.g), (\{\}, b.g), (\{x.g\}, a.g)\}$ and footprint **acc**(x.g)

Figure 4.2 is a visual representation of the dfg of example 4.4.2. Let us denote

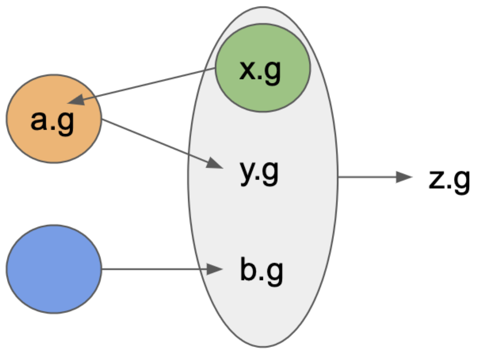


Figure 4.2: One area, together with an arrow pointing to a heap location, represents one tuple of the dfg. For example, the green circle, together with the arrow pointing to $a.g$, represents the tuple $(\{x.g\}, a.g)$. If all heap locations in one area are fixed then we can conclude that all heap locations the area points to (with an arrow) are also fixed.

$shfl_i$ as the $sfhl$ after the ith iteration of the algorithm to compute the sfhl.

As the computed footprint is **acc**(x.g), the initial $sfhl_0$ is $\{x.g\}$. After one iteration $sfhl_1$ will be $\{x.g, a.g, b.g\}$. $a.g$ is in $shfl_1$, because $(\{x.g\}, a.g) \in dfg$ and $\{x.g\} \subseteq sfhl_0$ and $b.g$ is in $shfl_1$, because $(\{\}, b.g) \in dfg$ and $\{\} \subseteq sfhl_0$. After one more iteration $sfhl_2$ will be $\{x.g, a.g, b.g, y.g\}$. $sfhl_3$ is $\{x.g, a.g, b.g, y.g, z.g\}$ and as $sfhl_4$ is also $\{x.g, a.g, b.g, y.g, z.g\}$ we reached a fix point and therefore the final $sfhl$ will be $\{x.g, a.g, b.g, y.g, z.g\}$.
We now know how we can compute the set of fixed heap locations given the footprint computed by the current algorithm and the dependent fixed graph. The next sections cover how we can compute the dfg given a wand's left-hand side.

4.4.2 Computation of the dependent fix graph

The way the dfg for an assertion is computed is given by the following rules.

dfg$(A_1 * A_2)$ = dfg$(A_1)\cup$dfg(A_2)

This is because of Theorem 4.4.2.
If for example dfg$(A_1) = \{(\{a.g\}, x.f)\}$ and dfg$(A_2) = \{(\{b.g\}, x.f)\}$ then dfg$(A_1 * A_2)$ is $\{(\{a.g\}, x.f), (\{b.g\}, x.f)\}$. Note that this means that if either $a.g$ or $b.g$ is fixed then we know that $x.f$ is fixed. This is, because A_1 gives us the information that if $a.g$ is fixed then $x.f$ is fixed and A_2 gives us the information that if $b.g$ is fixed then $x.f$. And theorem 4.4.2 tells us that the $*$ connective does not "delete" information about fixedness.

dfg$(e.f) = \{(\{\}, e.f)\}$ and dfg$(not(e.f)) = \{(\{\}, e.f)\}$

This is because in every state σ_a with $\sigma_a \models e.f$, $e.f$ has the same value, namely true. A similar reason holds for $not(e.f)$.

dfg$(e_1 == e_2.f) = \{(\{$all heap locations contained in $e_1\}, e_2.f)\}$

This is because if all heap locations contained in e_1 are fixed then so is e_1 and because $e_1 == e_2.f$ this also fixes $e_2.f$

dfg$(e_1.f == e_2) = \{(\{$all heap locations contained in $e_2\}, e_1.f)\}$
dfg$(e_1.f == e_2.f) = \{(\{e_2.f\}, e_1.f), (\{e_1.f\}, e_2.f)\}$

These rules hold similar reasons as the one above.

dfg(all other cases) $= \{\}$

4.4.3 Examples

Let us look at two concrete examples for the computation of the dfg given an assertion. For the example 4.4.1 we have dfg(`acc(x.f)&&acc(y.f,1/2)&&x.f ==y.f`) $= \{(\{x.f\}, y.f), (\{y.f\}, x.f)\}$. The fact that $(\{y.f\}, x.f)$ is in the dfg together with the fact that $y.f$ is in the footprint fixes $x.f$.
Let us look at another example:

Example 4.4.3. dfg(`acc(x.f)&&acc(y.f)&&x.f==y.f&&x.f`) $= \{(\{x.f\}, y.f), (\{y.f\}, x.f), (\{\}, x.f)\}$. The fact that $(\{\}, x.f)$ is in the dfg already fixes $x.f$ because all heap locations in $\{\}$ are fixed. And then we can use the fact that $x.f$ is fixed and that $(\{x.f\}, y.f)$ is in the dfg to conclude that $y.f$ is also fixed.

4.4.4 Overapproximation theorem

The next theorem states that the way to compute the set of fixed heap locations described in this Section is indeed a sound overapproximation. This means that all heap location in the computed set of fixed heap locations are indeed fixed.

Theorem 4.4.3. *Given A and σ_{foot} $\forall e.f \in sfhl(dfg(A), \sigma_{foot}) \cdot fixed(e.f, A, \sigma_{foot})$.*

This theorem is justified by the way dfg and sfhl are computed, and it is crucial to guarantee that the footprint that the new algorithm computes is valid.

4.5 Handling conditionals

Until now, we did not look at how ternary expressions in an assertion should be handled to overapproximate the set of fixed heap locations.

To formalize this, we first introduce the definition of conditional fixedness.

Definition 4.5.1 (Conditional fixedness of heap locations)**.** *The heap location $e.f$ is conditionally fixed with respect to an assertion A, a state σ_{foot} and a condition b, written $cfixed(e.f, A, \sigma_{foot}, b)$ iff $\forall \sigma_{a1}, \sigma_{a2} \in SAS(A, \sigma_{foot}) \cdot$
$(\sigma_{a1} \uplus \sigma_{foot})(b) \land (\sigma_{a2} \uplus \sigma_{foot})(b) \implies (\sigma_{a1} \uplus \sigma_{foot})(e) = (\sigma_{a2} \uplus \sigma_{foot})(e)$.*

This definition basically means that $e.f$ is fixed under the assumption that b is true. The notion of conditional fixedness is closely connected to the notion of fixedness. More precisely:

Lemma 4.5.1. *$fixed(e.f, A, \sigma_{foot})$ iff $cfixed(e.f, A, \sigma_{foot}, true)$*

Conditional fixedness is also closed under the negation:

Lemma 4.5.2. *$cfixed(b_2, A, \sigma_{foot}, b_1)$ iff $cfixed(not(b_2), A, \sigma_{foot}, b_1)$*

With the next theorem, we justify that if we branch over a fixed condition and both branches fix a heap location $e.f$ then $e.f$ is fixed with respect to the whole wand. This intuitively makes sense, because if we know that for all applicable states the branch condition has the same value. And if we additionally know that given the value of the branch condition the value of $e.f$ is fixed, then we know that $e.f$ is fixed.

Theorem 4.5.3. *If $fixed(b, A, \sigma_{foot})$ and*
$cfixed(e.f, A, \sigma_{foot}, b)$ and $cfixed(e.f, A, \sigma_{foot}, not(b))$
then $fixed(e.f, A, \sigma_{foot})$

Proof. Let $\sigma_{a1}, \sigma_{a2} \in \mathrm{SAS}(A, \sigma_{foot})$ be arbitrary.

b is fixed with respect to A and σ_{foot} $\Longrightarrow$
$(\sigma_{a1} \uplus \sigma_{foot})(b) = (\sigma_{a2} \uplus \sigma_{foot})(b) \overset{*}{\Longrightarrow}$
$(\sigma_{a1} \uplus \sigma_{foot})(b) \wedge (\sigma_{a2} \uplus \sigma_{foot})(b) \overset{**}{\Longrightarrow}$
$(\sigma_{a1} \uplus \sigma_{foot})(e.f) = (\sigma_{a2} \uplus \sigma_{foot})(e.f)$ □

$*$ w.l.o.g. assume $(\sigma_{a1} \uplus \sigma_{foot})(b)$ = true
$**$ cfixed$(e.f, A, \sigma_{foot}, b)$

The next theorem extends theorem 4.5.3 in a way such that it can be used for nested branches.

Theorem 4.5.4. *If $cfixed(b_2, A, \sigma_{foot}, b_1)$ and*
$cfixed(e.f, A, \sigma_{foot}, b_1 \wedge b_2)$ and $cfixed(e.f, A, \sigma_{foot}, b_1 \wedge not(b_2)))$
then $cfixed(e.f, A, \sigma_{foot}, b_1)$

Proof. Let $\sigma_{a1}, \sigma_{a2} \in \mathrm{SAS}(A, \sigma_{foot})$ be arbitrary.

$(\sigma_{a1} \uplus \sigma_{foot})(b_1) \wedge (\sigma_{a2} \uplus \sigma_{foot})(b_1) \overset{*}{\Longrightarrow}$
$(\sigma_{a1} \uplus \sigma_{foot})(b_1) \wedge (\sigma_{a2} \uplus \sigma_{foot})(b_1) \wedge (\sigma_{a1} \uplus \sigma_{foot})(b_2) = (\sigma_{a2} \uplus \sigma_{foot})(b_2) \overset{**}{\Longrightarrow}$
$(\sigma_{a1} \uplus \sigma_{foot})(b_1 \wedge b_2) \wedge (\sigma_{a2} \uplus \sigma_{foot})(b_1 \wedge b_2) \overset{***}{\Longrightarrow}$
$(\sigma_{a1} \uplus \sigma_{foot})(e.f) = (\sigma_{a2} \uplus \sigma_{foot})(e.f)$ □

$*$ cfixed$(b_2, A, \sigma_{foot}, b_1)$
$**$ w.l.o.g. assume $(\sigma_{a1} \uplus \sigma_{foot})(b_2)$ = true
$***$ cfixed$(e.f, A, \sigma_{foot}, b_1 \wedge b_2)$

Theorems 4.5.3 and 4.5.4 can be used to prove overall fixedness using conditional fixedness. Assume we branch over b_1 and in the true branch we once

again branch over b_2 and in the false branch over b_3. If $e.f$ is now conditionally fixed under the condition $b_1 \land b_2$, that means in the branch where we assume b_1 and b_2 to be true, and also under the condition $b_1 \land not(b_2)$. And if additionally b_2 is conditionally fixed under the condition b_1 then by theorem 4.5.4 $e.f$ is conditionally fixed under the conditions b_1. If we now can show in a similar fashion that $e.f$ is also conditionally fixed under the condition $not(b_1)$. And if b_1 is fixed then by theorem 4.5.3 $e.f$ is fixed.
This now gives rise to a rule to extend the dependent fixed graph for ternary operators.

dfg$(b?A_1 : A_2) = \{(\{$all heap locations contained in $b\}, e.f)|$
$e.f \in ($sfhl$($dfg$(A_1))\cap$sfhl$($dfg$(A_2)))\}$

Here, sfhl(dfg) denotes the set of fixed heap locations computed using the algorithm from Section 4.4.1 but with an empty footprint. If $(s, e.f)$ is in dfg$(b?A_1 : A_2)$ then if everything in s is fixed $e.f$ is also fixed. This is the case, because on the one hand if everything in s is fixed then by theorem 4.2.1 b is fixed. On the other hand $e.f$ has to be in the sfhl of both A_1 and A_2, and therefore we know that $e.f$ is conditionally fixed on both b and $not(b)$. From theorem 4.5.3 we then know that $e.f$ has to be fixed.
Next, we will look at a concrete example, where we want to compute the dfg of the following assertion.

Example 4.5.1. `acc(y.f) && acc(x.f) && (x.f ? y.f : acc(z.f) && y.f== z.f && !z.f)`

Figure 4.5.1 shows an illustration of this example. By the rules in Section 4.4.2, we have dfg(`y.f`) $= \{(\{\}, y.f)\}$ and
dfg(`acc(z.f) && y.f==z.f && !z.f`) $= \{(\{y.f\}, z.f), (\{z.f\}, y.f), (\{\}, z.f)\}$ and therefore sfhl(dfg(`y.f`)) $= \{y.f\}$ and sfhl(dfg(`acc(z.f) && y.f==z.f && !z.f`)) $= \{y.f, z.f\}$. Using the rule for dfg$(b?A_1 : A_2)$, we can conclude that dfg(`acc(y.f) && acc(x.f) && (x.f ? y.f : acc(z.f) && y.f==z.f && !z.f)`) $= \{(\{x.f\}, y.f)\}$. This means that the assertion in the example fixes $y.f$ if $x.f$ is fixed.

As Silicon branches over ternary expressions, we cannot use the above approach directly. What we will do instead is we will record a dfg for each branch then compute the sfhl for each branch and then take the union of all these sfhl's as the final set of fixed heap locations.
So for $b?A_1 : A_2$ two branches are created: One where b is assumed to be true and one where it is assumed to be false. In the true branch, the dfg will be

$$\{(\{\text{all heap locations contained in } b\}, e.f)|e.f \in \text{sfhl}(\text{dfg}(A_1))\}$$

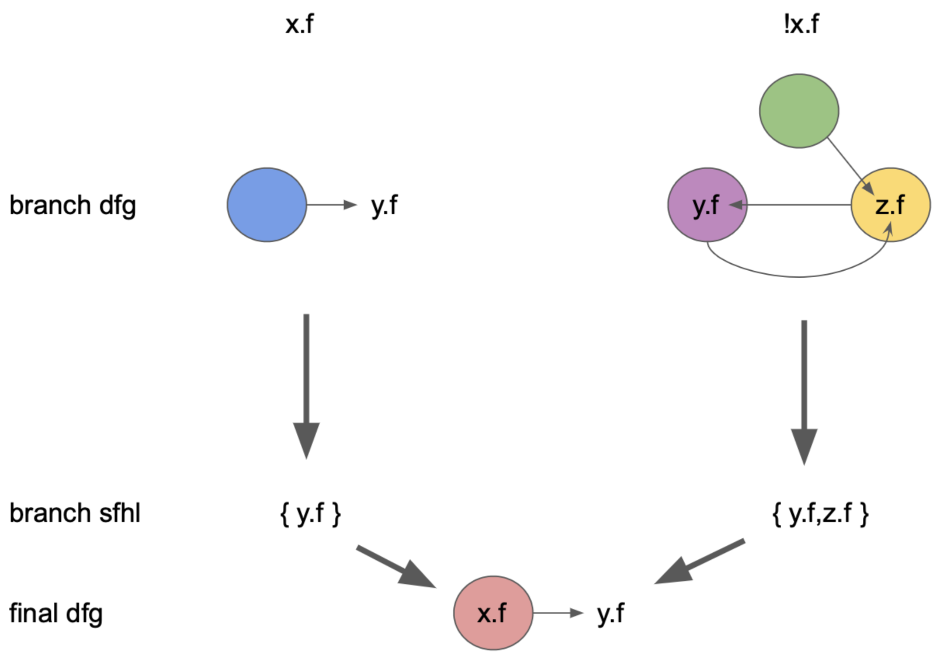


Figure 4.3: Visual representation of example 4.5.1. There is one column for each branch: One branch where $x.f$ is assumed to be true and one where it is assumed to be false. The top row is the dfg computed for the assertions $y.f$ and $acc(z.f)\&\&y.f == z.f\&\&!z.f)$. The second row is the sfhl of both assertions. The last row is the combined dfg as described by the rule for $\text{dfg}(b?A_1 : A_2)$.

and in the false branch, it will be

$$\{(\{\text{all heap locations contained in } b\}, e.f) | e.f \in \text{sfhl}(\text{dfg}(A_2))\}$$

Now $e.f$ can only be in the final sfhl of both branches if all heap locations of b are in both sfhl's and therefore fixed.

4.6 Completeness gap

As we syntactically overapproximate the set of fixed heap locations the algorithm might fail to detect that a certain heap location is fixed. Which could then lead to a case where the algorithm doesn't compute the smallest footprint.
One such case is shown with the following example.

Example 4.6.1. **acc**`(x.g) && x.g>a && x.g<=a --* `**acc**`(x.g) && (x.g==0 ? `**acc**`(y.g) : `**acc**`(z.g))`

For this wand our algorithm would not detect that the assertion `x.g>a && x.g <=a` would fix the value of $x.g$ to a, because for the assertion `x.g>a && x.g<=a` nothing is recorded in the dfg. As the algorithm cannot be certain that the branch condition `x.g==0` is fixed it computes an upper bound of the footprints of both branches, which in this case could be **acc**`(y.g) &&` **acc**`(z.g)`. But more precise would be to detect that the branch condition `x.g==0` is fixed and therefore `x.g==0 ?` **acc**`(y.g) :` **acc**`(z.g)` would also be a correct footprint which is even smaller.

Chapter 5

Symbolic Execution Rules

This Chapter provides symbolic execution rules for a possible implementation of a new and sound package algorithm based on the findings of the previous Chapters. As the new algorithm uses many parts of the current algorithm we first recap how the current algorithm works. For a more detailed explanation, we refer to Chapter 5 of the PhD thesis of Dr. Malte Schwerhoff [9]. In the rest of the Chapter, we present symbolic execution rules for the new algorithm. To save space we will use many notations and functions from the PhD thesis of Dr. Malte Schwerhoff [9], without reintroducing them. Therefore, Chapters 3 and 5 of his PhD thesis should be seen as a prerequisite for this Chapter.

5.1 Current Algorithm

Listing 5.1 shows the symbolic execution rule of the current algorithm. To package a magic wand, the current algorithm first creates a state representing the left-hand side (σ_{lhs}) by inhaling (produce) the left-hand side in an empty state (line 7). It then tries to exhale (consume) the right-hand side (line 8). If more permissions are needed than provided by the left-hand side heap ($\sigma_{lhs}.h$), they are taken from the heap of the state in which we package the wand ($\sigma_1.h$). This way the footprint is computed implicitly, and it corresponded to the permissions removed from $\sigma_1.h$ during this process.
As `produce` and `consume-ext` could branch line 9 could be reached multiple times. As discussed before making assumptions on the value of some of the branch conditions is unsound because they may change between packaging and applying the wand.

```
exec($\sigma_1$,package $a_1 - - * a_2$, $Q$) =
    $s_{lhs}$ := fresh
    Let $id_{wand}(e')$ be a magic wand chunk identifier
    corresponding to $a_1 - - * a_2$
    $wandChunk$ := $id_{wand}(e')$
    $\sigma_{empty}$ := $\sigma_1\{h$ := $\emptyset\}$
    produce($\sigma_{empty}, a_1, s_{lhs}, (\lambda\sigma_{lhs}\cdot$
        consume-ext($[\sigma_{lhs}.h, \sigma_1.h], \sigma_{lhs}\{h$ := $\emptyset\}, a_2, (\lambda[\_, h'], \sigma_{used}, \_\cdot$
            $Q(\sigma_{used}\{h$ := heap-add($h', wandChunk$)$\}$)))
```

Listing 5.1: The wand $a_1 - - * a_2$ is packaged in the state σ_1. Q is the continuation that has to be executed after the packaging is completed. σ_{lhs} is the state which represents the left-hand side of the wand. h' is the heap of σ_1 after the FIA, which will also be the heap of the state given to the continuation. heap-add is used to add the chunk representing the magic wand to the heap given to the continuation.

5.2 Overview

Before we have a look at each part of the new algorithm in more detail we will give a brief overview of what the new algorithm will look like based on the findings of Chapter 4. Figure 5.1 illustrates the new package algorithm. It takes as input a wand $A \twoheadrightarrow B$ and outputs a footprint of the wand. For this

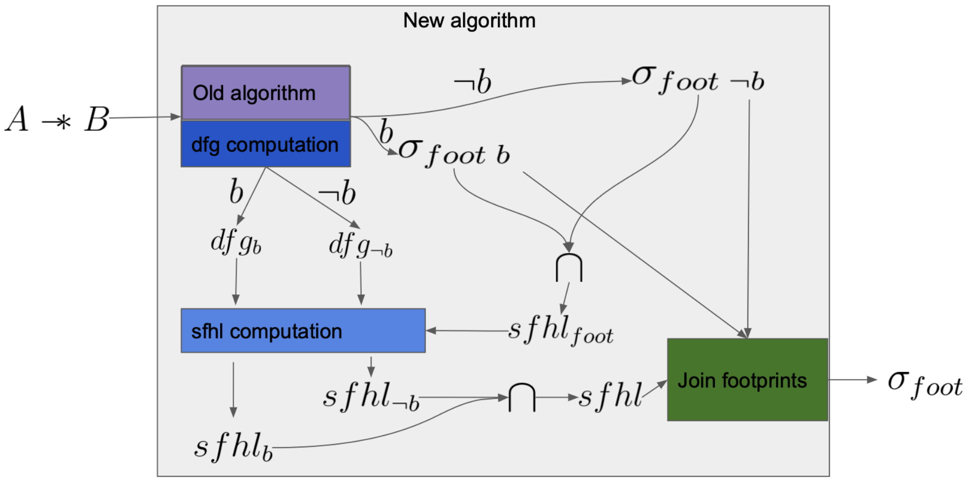


Figure 5.1: The gray box represents the new algorithm. $A \twoheadrightarrow B$ is the wand we want to package. σ_{foot} is the computed footprint. We assume the current algorithm branches exactly once. b is the branch condition. $\sigma_{foot\ b}$ and $\sigma_{foot\ \neg b}$ are the footprints of the corresponding branches computed by the current algorithm. dfg_b and $dfg_{\neg b}$ are the dependent fixed graphs computed for each branch. $sfhl_{foot}$ contains all heap location contained in both $\sigma_{foot\ b}$ and $\sigma_{foot\ \neg b}$. $sfhl_b$ and $sfhl_{\neg b}$ are the set of fixed heap locations computed for each branch. $sfhl$ is the set of fixed heap locations for the whole wand.

description, we will have a look at a wand for which the current algorithm branches exactly once on the condition b. If there were more branches the description could be generalized for more branches. The first step of the new algorithm is to use the current algorithm to compute the footprint on each branch. The current algorithm is also extended with the dfg computation part, which simultaneously with the footprint computation also computes the dfg per branch. As we know that all heap locations that occur in both $\sigma_{foot\ b}$ and $\sigma_{foot\ \neg b}$ will be in the footprint and therefore fixed, we take $sfhl_{foot}$, as the initial set of fixed heap locations for the sfhl computation. We then use dfg_b, $dfg_{\neg b}$ and $sfhl_{foot}$ to compute $sfhl_b$ and $sfhl_{\neg b}$. $sfhl_b$ and $sfhl_{\neg b}$ are then combined to $sfhl$, by taking all heap locations which are fixed by both $sfhl_b$ and $sfhl_{\neg b}$. We then join the footprints $\sigma_{foot\ b}$ and $\sigma_{foot\ \neg b}$ with the information of the $sfhl$, which concretely means if all heap locations of b are in $sfhl$ then $\sigma_{foot} = b?\sigma_b : \sigma_{\neg b}$ and otherwise $\sigma_{foot} = \sigma_b \cup \sigma_{\neg b}$ as described in Chapter 4.3. This joined footprint is then the footprint computed by the new algorithm.

5.3 Computing the dependent fixed graph

The rules for how to compute the dependent fixed graph (dfg) are already given in Chapter 4, and it is dependent on the structure of the wand's left-hand side. Therefore, it is natural to compute the dfg simultaneously with producing the left-hand side of the wand. Therefore, we will extend the produce rule such that it also takes a dfg as an input and the continuation also takes the extended dfg as an input. In Silicon, a heap location is represented as a tuple of the symbolic value of the reference and the field id. Because dfg is of type $Set[(Set[(heaplocation)], heaplocation)]$ the type in Silicon will be $Set[(Set[(Symbol, Id)], (Symbol, Id)))]$. Which leads to the new `produce` rule called `produce'` with the type
`produce'` : $\Sigma \rightarrow A \rightarrow Snap \rightarrow Set[(Set[(Symbol, Id)], (Symbol, Id))] \rightarrow (\Sigma \rightarrow List[Set[Symbol, Id)], (Symbol, Id)] \rightarrow R) \rightarrow R$

Listing 5.2 shows the new produce′ rule but only the cases where we do not branch. For the first case (lines 1 to 4) where we produce a pure expression, just like normal produce, we add e', which is e evaluated in σ_1, to the path conditions. The more interesting part is what we add to the dfg. `dependentfix` is a function which calculates exactly that. If e is of the form $e_1.f$ then we can add the corresponding heap location, which is described by the tuple $(\sigma.\texttt{eval}(e_1), f)$, without any dependency because we know that $e_1.f$ has to have the value true. $\sigma.\texttt{eval}(e_1)$ is e_1 evaluated in σ which is the symbolic value of the reference e_1. If we have something of the form $e_1.f == e_2$ then we add $(\sigma.\texttt{eval}(e_1), f)$ under the dependency that e_2 is fixed which is, as described by Theorem 4.2.4, the case if all heap locations in e_2

```
1  produce'($\sigma_1$,$e$,$s$,$dfg$,$Q$)) =
2      eval($\sigma_1$,$e$,($\lambda\sigma_2$,$e'\cdot$
3          $\pi_1$ =pc-add($\sigma_2.\pi$,$\{e', s = unit\}$)
4          $Q(\sigma_2\{\pi = \pi_1\}$,$dfg\cup$dependentfix($\sigma_2$,$e$)))
5
6
7  dependentfix: $\Sigma \to E \to Set[(Set[(Symbol, Id)], (Symbol, Id))]$
8
9  dependentfix($\sigma$,$e.f$) = $\{(\{\}$, ($\sigma$.eval($e$),$f$))}
10 dependentfix($\sigma$,$e_1.f == e_2.g$) =
11     $\{(\{(\sigma$.eval($e_1$),$f$)},($\sigma$.eval($e_2$),$g$)),{($\sigma$.eval($e_2$),$g$)},($\sigma$.eval($e_1$),$f$))}
12 dependentfix($\sigma$,$e_1.f == e_2$) = {(neededtofix($\sigma$,$e_2$),$\sigma$.eval($e_1$),$f$))}
13 dependentfix($\sigma$,$e_1 == e_2.f$) = {(neededtofix($\sigma$,$e_1$),($\sigma$.eval($e_2$),$f$))}
14 dependentfix($\sigma$,$e$) = {}
15
16 neededtofix: $\Sigma \to E \to Set[(Term, Id)]$
17 neededtofix($\sigma$,$e$) = {($\sigma$.eval($e_1$),$f$) | $e_1.f$ is a sub expression of $e$}
18
19 produce'($\sigma_1$,$a_1$&&$a_2$,$s$,$dfg_1$,$Q$) =
20     produce'($\sigma_1$,$a_1$,$first(s)$,$dfg_1$,($\lambda\sigma_2$,$dfg_2\cdot$
21         produce'($\sigma_2$,$a_2$,$second(s)$,$dfg_2$,$Q$)
```

Listing 5.2: The new produce' rule. Similar to the old produce but it also computes the dfg. dependentfix is a function that computes the dfg of a pure expression. σ.eval(e_1) is e_1 evaluated in σ. neededtofix computes the heap location needed to be fixed in order to fix the expression given as input.

are fixed. The set of all heap locations of an expression is computed by the function neededtofix.
Now we look at lines 19 to 21, where we handle the separating conjunction. For the separating conjunction, we first produce the first part and give the resulting state and dfg as input to produce the second part.
Let us look at a few examples. In the following, x' stands for the symbolic value of x and so does y' for y and z' for z. The simplest case is if we produce $x.f$ then after the produce the dfg is $\{(\{\}, (x', f))\}$. Another example is producing $x.g == (y.g + z.g)$, which leads to the dfg $\{(\{(y', g), (z', g)\}, (x', g))\}$. And therefore the computed dfg of $x.f \&\& x.g == (y.g + z.g)$ is $\{(\{\}, (x', f))$, $(\{(y', g), (z', g)\}, (x', g))\}$ as it is by the rules, just the union of the dfg of $x.f$ and $x.g == (y.g + z.g)$.

Listing 5.3 shows the dfg computation for the branch case. We branch over e' and in both branches we use produce' to produce a_1 or a_2 and also compute the dfg for both branches. The dfg we give to the continuation is the dfg we had before the production of a_1 or a_2 plus the set of fixed heap

```
produce'($\sigma_1$,$e?a_1 : a_2$,$s$,$dfg_1$,$Q$) =
    eval($\sigma_1$,$e$($\lambda\sigma_2$,$e'\cdot$
        branche($\sigma_2$,$e'$,
            ($\lambda\sigma_3\cdot$ produce'($\sigma_3$,$a_1$,$s$,$\emptyset$,
                ($\lambda\sigma_4$,$dfg_2 \cdot Q$($\sigma_4$,$dfg_1\cup$addDependency($\sigma_4$,$e$,$dfg_2$)))))
            ($\lambda\sigma_3\cdot$ produce'($\sigma_3$,$a_2$,$s$,$\emptyset$,
                ($\lambda\sigma_4$,$dfg_2 \cdot Q$($\sigma_4$,$dfg_1\cup$addDependency($\sigma_4$,$e$,$dfg_2$)))))))

addDependency: $\Sigma \to E \to Set[(Set[(Term, Id)], (Term, Id))] \to$
$Set[(Set[(Term, Id)], (Term, Id))]$

addDependency($\sigma$,$e$,$dfg$) =
    {(neededtofix($\sigma$,$e$),$heaploction$)
      | $heaplocation \in$ dfgToFixSet($\sigma.\pi$,$\emptyset$,$dfg$)}
```

Listing 5.3: produce' rule for the branch case. addDependency computes the dfg of the branches as explained at the end of Section 4.5. dfgToFixSet computes the $sfhl$ given a dfg as explained in Section 4.4.1. dfgToFixSet will be explained in more detail later on in Listing 5.7.

```
produce'($\sigma_1$,$a$,$s$,$dfg$,$Q$)) =
    produce($\sigma_1$,$a$,$s$,($\lambda\sigma_2 \cdot Q$($\sigma_2$,$dfg$))
```

Listing 5.4: produce' for remaining cases. For this cases, it is just the normal produce and we do not augment the dfg.

locations given the dfg produced by a_1 or a_2. But there we have to add the dependency that the condition we branch over is also fixed.

Listing 5.4 shows the produce' rule for the remaining cased. As we do not gain information about the fixedness of heap locations we do not augment the dfg.

5.4 Computing the explicit footprint

The next step is to augment the consume-ext function such that it also explicitly computes which chunks are removed from the outside heap, as part of the footprint. We call this new consume-ext function consume-ext'. We need to have an explicit footprint per branch because we want to join them afterward and the explicit footprint also tells us which heap locations are fixed by the footprint. The type of the footprint in Silicon is a list of chunks. First, we have to augment the transfer function such that it also records which chunks it had removed from each heap. The new type of this augmented transfer, called transfer' is,
$List[H] \to H \to \Pi \to Id \to Perm \to Option[(List[H], List[chunck], H, Snap)]$

```
consume-ext': $List[H] \rightarrow List[chunk] \rightarrow \Sigma \rightarrow A$
$\rightarrow (List[H] \rightarrow List[chunk] \rightarrow \Sigma \rightarrow Snap \rightarrow R) \rightarrow R$

consume-ext'$(\bar{h}_1, footprint, \sigma_1, acc(e.f, p), Q) =$
    eval$(\sigma_1, p :: e, (\lambda \sigma_2, p' :: e' \cdot$
        tranfer'$(\bar{h}, \sigma_2.h, \sigma_2.\pi, e'.f, p')$ matches
        $Some(\bar{h}_2, \overline{ch}, h_2, s) :$
            $Q(\bar{h}_2, footprint +: \overline{ch}[-1], \sigma_1\{h := h_2\}, s)$
        $None :$
            $failure()))$
```

Listing 5.5: Rule for consume-ext'. Similar to consume-ext but the footprint is additionally recorded.

`transfer'` is the same as `transfer`, but it also returns the chunks it has removed from the corresponding heaps,
e.g. if `transfer'`$([h_1, h_2], h, \sigma, e.f, p) = Some([h'_1, h'_2], [ch_1, ch_2], h', s)$ then the chunk ch_1 was removed from h_1 to create h'_1 and ch_2 was removed from h_2 to create h'_2. Listing 5.5 shows the rule of `consume-ext'` for the $acc(e.f, p)$ case. The main difference to `consume-ext` is that it and the continuation have an additional input of type $List[chunk]$, which is the explicitly computed and recorded footprint. Here $\overline{ch}[-1]$ means the last element of the list $\overline{ch}$ which is the chunk we removed from the heap of the state in which we want to package the wand. $\overline{ch}[-1]$ is part of the footprint and must therefore be recorded. `consume-ext'` for all other assertions is the same as `consume-ext` but we give the footprint unchanged to the continuation.

5.5 Combining footprint and dfg to fixed set

Now that we have the dfg and the footprint per branch, we can compute which heap locations are fixed in this branch. First, we know every heap location which is contained in the footprint of all branches is fixed as of theorem 4.4.1. Listing 5.6 shows the function to compute this set of heap locations. The first input is the path condition which we need to check the equality of the symbolic references e' and e'_2 (line 5). The second input is a list of footprints which will contain the footprints of all branches. The function `fixedByFootprints` returns all heap locations for which all footprints have a corresponding chunk.
Now we can combine the information from the dfg and the heap locations fixed by the footprint to compute the overall fixed set of heap locations of a branch. Listing 5.7 shows the implementation of the algorithm to compute this fixed set of heap locations. The first input is the path condition which we need to check the equality of two symbols of a reference. The second is

```
fixedByFootprints: $\Pi \to List[List[chunck]] \to Set[(Symbol, Id)]$

fixedByFootprints($\pi, footprints$) =
    $\{(e', f) \mid \exists f(e'; s, p) \in footprints[0]$ where $\forall fp \in footprints$
        $\exists f(e'_2; s_2, p_2) \in fp\cdot$ check($\pi, e'_2 = e' \wedge p_2 > 0)\}$
```

Listing 5.6: Function to compute the heap locations fixed by the footprints. If the chunk $f(t'; s, p)$ is an element of a footprint this means that at heap location $t'.f$ a value is stored with symbolic value s and with permission amount p . Check is a call to the SMT solver, which takes two inputs path conditions and a condition to be checked. It returns true if the path conditions implie that the condition to be checked is true.

```
dfgToFixSet: $\Pi \to Set[(Symbol, Id)] \to List[(Set[(Symbol, Id)], (Symbol, Id)]$
$\to Set[(Symbol, Id)]$

dfgToFixSet($\pi, fixedset_1, dfg_1$) =
    $dfg_2 := \{$
        $(\{(y', g) \mid (y', g) \in s_1$ where $\nexists(z', g) \in fixedset_1\cdot$check$(\pi, y' = z')\}, (x', f))$
          $\mid (s_1, (x', f)) \in dfg_1\}$
    $fixedset_2 := \{(x', f) \mid (s_1, (x', f)) \in dfg_2$ where $s_1 = \emptyset\}$
    $dfg_3 := \{(s_1, (x', f)) \mid (s_1, (x', f)) \in dfg_2$ where $s_1 \neq \emptyset\}$
    if $fixedset_2 == \{\}$
    then $fixedset_1$
    else dfgToFixSet($\pi, fixedset_2, dfg_3) + fixedset_1$
```

Listing 5.7: Recursive computation of the complete set of fixed heap locations given a dfg and an initial set of fixed heap locations. It is based on the ideas of Section 4.4.1.

a set of heap locations for which we know that they are fixed. The third is the dfg. Recall that if $(s, (x', f))$ is in the dfg then the heap location (x', f) is fixed if all heap locations in s are fixed. The way the algorithm works is the following: dfg_2 is created by removing from the first element of the tuples in dfg_1 all heap locations which are proven to be equal to some heap location in $fixedset_1$ as they are now known to be fixed. So only the remaining heap locations of the first element of the tuples need to be fixed to fix the second element. Then we go over the tuples in dfg_2 for all tuples where the first element is the empty set we now know that the second element is a fixed heap location as all its dependencies are fixed, and therefore we add it to the $fixedset_2$. This process is then repeated recursively until no more heap locations can be fixed.

```
1   exec($\sigma_1$,package $a_1 --* a_2, Q$) =
2       $s_{lhs}$ := fresh
3       $id$ := fresh
4       $results$ := []
5       $\pi_1$ := $\sigma_1.\pi$
6       $\sigma_{empty}$ := $\sigma_1\{h$ := $\emptyset, \pi$ := pc-push($\sigma_1.\pi, id, true$)$\}$
7       produce'($\sigma_{empty}, a_1, s_{lhs}, \emptyset, (\lambda\sigma_{lhs}, dfg\cdot$
8           consume-ext'($[\sigma_{lhs}.h, \sigma_1.h], [], \sigma_{lhs}\{h$ := $\emptyset\}, a_2, (\lambda[\_,\_], footprint, \sigma_{used}\cdot$
9               ($cnds, bcs$) := (pc-segs(pc-after($\sigma_{used}.\pi$,id)))
10              $results := results$+:($bcs, dfg, footprint$)
11              $\pi_1$ := pc-add($\pi_1, cnds$)
12              $Success$()))
13      $\wedge$(
14          $sfhl_{foot}$ := fixedByFootprints($\sigma.\pi, [fp|(\_,\_,fp) \in results]$)
15          $results_2$ := $[(bcs, dfgToSfhl(\sigma.\pi, sfhl_{foot}, dfg), footprint)|$
16              $(bcs, dfg, footprint) \in results]$
17          $Success$()
18      )
```

Listing 5.8: Intermediate state of the final algorithm. The blue parts are the parts added to the original algorithm. In $results$ the tuple of branch conditions, dfg and $footprint$ is recorded for each branch. In $results_2$ per branch the tuple of branch conditions, $sfhl$ and $footprint$ is recorded.

5.6 Combination of the previous parts

In this Section, we will combine all the parts we looked at until now. Listing 5.8 shows an intermediate step towards the finished algorithm. It uses the functions covered so far. It is important to understand that line 9 is reached multiple times once for each branch created during `produce'` and `consume-ext'`. To join them we will store, for each branch, the branch conditions (bcs), the dfg, and the footprint. The newly added lines 3, 5, and 11 are there such that we can record the branch conditions (bcs) and the branch dependent path conditions ($cnds$) of the current branch. `pc-push` pushes fresh identifier, created on line 3, onto the path conditions. `pc-after` returns the path conditions after the identifier, which are in our case all path conditions created during `produce'` and `consume-ext'`. `pc-segs` returns a tuple of branch dependent path conditions and branch conditions of its input path conditions. Since the input to `pc-segs` are the path conditions after id, bcs represent the branch condition over which the current branch branched during `produce'` and `consume-ext'`. Likewise, $conds$ are the branch-dependent path conditions during `produce'` and `consume-ext'`. It is important to record them as they hold information about the structure of s_{lhs}, which could be different on each branch, so we have to record it for each branch. The dfg

```
combineFixedSets: $\Pi \rightarrow List[Set[(Symbol, Id)]] \rightarrow Set[(Symbol, Id)]$

combineFixedSets($\pi$, $[]$) = $\{\}$
combineFixedSets($\pi$, $(fs : fixedSets)$) =
    $\{(t, f)$ | $(t, f) \in fs$ where
        $\forall fs' \in fixedSets\ \exists (t', f) \in fs' \cdot$ check($\pi, t = t'$))}
```

Listing 5.9: Function to combine the fixed sets. check is again a call to the SMT solver.

and footprint, given to `produce'` and `consume-ext'` as input, are initially empty; therefore, `produce'` and `consume-ext'` have the empty set and the empty list as additional input compared to `produce` and `consume-ext`. The computed dfg and footprint by `produce'` and `consume-ext'` are, together with the bcs, added to `results`. If all branches reach line 12 then the first part of the algorithm succeeds. Afterward lines 13-17 are executed. On line 14 the set of heap locations which is fixed by the footprints is computed, which is then used on line 15 to compute a new $results_2$ list where the tuple of bcs, sfhl, and footprint of each branch is stored.

Let us summarize what we have seen so far. We have augmented the current algorithm such that it records for each branch it creates during `produce'` and `consume-ext'` the branch conditions, the set of fixed heap locations and the chunks needed from the outside heap. The next steps are combining the fixed sets per branch to the set of all fixed heap locations for the whole wand. Then use this new fixed set to figure out which branch conditions are not fixed and have therefore to be replaced by the union of both branches.

5.7 Combining fixed sets

Now that we have the fixed set of all branches we can combine them with the function described in Listing 5.9. The function will take path conditions and a list of all the fixed sets and combines them by checking which heap locations are contained in all fixed sets. If the list of fixed sets is the empty list then the fixed set should also be the empty set (this is covered by line 3). If there is at least one set in the list then the resulting set contains all heap locations which are contained in all sets of the list. Two heap locations are equal if the field id is the same, and we can prove using the path condition that also the references are equal.

```
fixedHeapLocationsToNotFixedSymbols: $\Pi \rightarrow Heap$
$\rightarrow Set[(Symbol, Id)] \rightarrow Set[Symbol]$

fixedHeapLocationsToNotFixedSymbols($\pi, h, fixedHeapLocations$) =
  $\{s \mid \exists f(t'; s, p) \in h \not\exists (t, f) \in fixedHeapLocations \cdot check(\pi, t = t')\}$
```

Listing 5.10: Function to find the not fixed symbols given the not fixed heap locations. Remember that if the chunk $f(t'; s, p)$ is in the heap, then the heap stores the symbolic value s at location $t'.f$ with permission amount p.

5.8 Compute not fixed symbols given fixed heap locations

Now we have the set of all heap locations, but what we need are the symbolic values stored at the not fixed heap locations, as these are the ones that could change between packaging and applying. This is done by the function described in Listing 5.10. `fixedHeapLocationsToNotFixedSymbols` computes the set of not fixed symbols given path conditions, a heap, and a set of fixed heap locations. If there is a chunk $f(t'; s, p)$ in the heap and there is no tuple (t, f) in the set of fixed heap locations for which given the path conditions we can prove that $t' = t$ then the chunk corresponds to a possible not fixed heap location and therefor s has to be in the set of possible not fixed symbols. Note that check is a query to the SMT solver which might be imprecise, but this is not a soundness problem as every symbol corresponding to a not-fixed heap location is definitely in the set. The heaps given as input to this function will be the outside heap and the heap created by the `produce'` of the wand's left-hand side as these are the only heaps accessed during the algorithm.

5.9 Joining the footprints

The next step is to join the footprints of all branches into one correct footprint for the whole wand. First of all, in Silicon, there is no direct way to express a conditional heap. What is meant by that is that we cannot express in a simple way the following: If $cond_1$ holds then the heap is h_1 and if $cond_2$ holds then the heap is h_2. What we have to do instead is create a heap where we put the condition into the permission amount. An example would be to express if b is true then the heap is $(f(x; s_1, 1/2), g(y; s_2, 1/2))$ and otherwise the heap is $f(x; s_1, 1)$. This can be represented in Silicon by the heap $(f(x; s_1, b?1/2 : 1), g(y; s_2, b?1/2 : 0))$.
If we now want to join the footprints of two branches there are two cases: One where we know that the branch condition is fixed, in this case, we can create a conditional footprint as described above. The other case is when we do not know that the condition is fixed. As discussed before,

we then have to take an upper bound of both footprints. One way to do this is that for each heap location we take the maximum permission amount of both footprints. If for example, b were a not-fixed symbol, $(f(x; s_1, 1/2), g(y; s_2, 1/2))$ would be the footprint of the branch with branch condition b and $f(x; s_1, 1)$ the one with branch condition $\neg b$ then the joined footprint would be $(f(x; s_1, max(1/2, 1)), g(y; s_2, max(1/2, 0)))$.
This joining of the footprints is done by the function `joinFootprints` in Listing 5.11. The joining is done recursively. Lines 4 to 9 are for the base case where we do not have any branch conditions and only one footprint. In that case, we combine all chunks to the same heap location into one chunk by summing up all their permission amounts and return the resulting list of chunks. Now for the non-base case if there is at least one branch condition left we extract, on line 13, the top-level branch condition (tlbc). Then we split the *conditionalFootprints* into two new ones: One for the branches where tlbc is true and one where it is false. The tlbc is also removed from the bcs, such that the next recursion level splits the *conditionalFootprints* on the next branch condition. Then the joined footprints of the two new *conditionalFootprints* are computed recursively. On lines 26 to 29, we then decide whether we join the permissions amount with $tlbc ? p_1 : p_2$ or with $max(p_1, p_2)$. We have to take the latter if some symbol contained in *tlbc* is a not fixed symbol in one of the branches because then it could be that *tlbc* is a not fixed expression and therefore *tlbc* could change between packaging and applying. Otherwise, we can take $tlbc ? p_1 : p_2$ as the merged permission amount. The function *symbols*() on line 27 returns all symbols contained in the input term. The two footprints are then joined using the `joinChunks` method with the before computed join function. The `joinChunks` function joins two lists of chunks if there is a chunk with the same reference and same field identifier in both lists then the resulting list has a chunk with the same reference and field identifier and the permission amount is the permission amount of the original chunks combined using the *joinFunction*. If one list contains a chunk where there is no corresponding chunk in the other list then it is treated as if there was one just with permission amount 0.
On lines 6 and 41 we check for the existence of chunks with the exact same symbol as reference. This check does not involve the SMT solver. This is due to soundness reasons, but it might lead to incompleteness due to aliasing.

5.10 Unfixedness in the permission amount

Besides the branch conditions, there is one other place where the not fixed expression could appear, namely in the permission amount. For example, if the footprint computed by the current algorithm is $acc(a.f, x.f ? 1/2 : 1)$ and $x.f$ is not fixed, the new algorithm should make the footprint to $acc(a.f, max(1/2, 1))$. But we cannot just replace all occurrences of $e ? p_1 : p_2$

```
joinFootprints: $List[(List[Term], List[chunk], Set[Symbol])]$
$\rightarrow List[chunk]$

joinFootprints$(([], footprint, \_) : []) =$
    $foldl\ \ (\lambda chunks, f(x; s_1, p_1) \cdot$
        if $\exists f(x; s_2, p_2) \in chunks$
        then $chunks - f(x; s_2, p_2) + f(x; s_2, p_2 + p_1)$
        else $chunks + f(x; s_1, p_1)$
    $)\ \ []\ \ footprint$


joinFootprints$(conditionalFootprints) =$
    $tlbc := conditionalFootprints[0][0][0]$

    $conditionalFootprints_{tlbc} := [(bcs, footprint, notFixedSymbols)\ \ |$
        $((tlbc_1 : bcs), footprint, notFixedSymbols) \in$
        $conditionalFootprints$ where $tlbc_1 = tlbc]$

    $conditionalFootprints_{\neg tlbc} := [(bcs, footprint, notFixedSymbols)\ \ |$
        $((tlbc_1 : bcs), footprint, notFixedSymbols) \in$
        $conditionalFootprints$ where $tlbc_1 \neq tlbc]$

    $joinedFootprint_{tlbc} :=$joinFootprints$(conditionalFootprints_{tlbc})$
    $joinedFootprint_{\neg tlbc} :=$joinFootprints$(conditionalFootprints_{\neg tlbc})$

    $joinFunction :=$
        If $\forall s \in symbols(tlbc)\ \forall(\_, \_, notFixedSymbols) \in conditionalFootprints \cdot$
        $s \notin notFixedSymbols$ then $(\lambda t_1, t_2 \cdot tlbc ? t_1 : t_2)$
        else $(\lambda t_1, t_2 \cdot max(t_1, t_2))$

joinChunks$(joinFunction, joinedFootprint_{tlbc}, joinedFootprint_{\neg tlbc})$


joinChunks: $(Term \rightarrow Term \rightarrow Term) \rightarrow List[chunk]$
$\rightarrow List[chunk] \rightarrow List[chunk]$

joinChunks$(joinFunction, [], chunks) =$
    $[f(x; s, joinFunction(0, p)) | f(x; s, p) \in chunks]$

joinChunks$(joinFunction, (f(x; s_1, p_1) : chunks_1), chunks_2) =$
    if $\exists f(x; s_2, p_2) \in chunks_2$
    then $f(x; s, joinFunction(p_1, p_2))$+:
        joinChunks$(joinFunction, chunks_1, chunks_2 - f(x; s_2, p_2))$
    else $f(x; s, joinFunction(p_1, 0))$+:
        joinChunks$(joinFunction, chunks_1, chunks_2)$
```

Listing 5.11: Rekursive function to join the footprints. The top-level branch condition *tlbc* is the branch condition of the oldest branch. *conditionalFootprints* is spilt into two new conditional footprints: One for the footprints where *tlbc* was assumed to be true and one where it was assumed to be false. For both a non conditional footprint is computed recursively. The function joinChunks is used to join the chunks of exactly two footprints with a given join function. The join function is chosen depending on whether the expression over which is branched is fixed or not.

```
joinFootprints(([], $footprint, notFixedSymbols$) : []) =
    $foldl\ (\lambda chunks, f(x; s_1, p_1)\cdot$
        $p'_1 := maximumWhenNotFixedRemoved(p_1, notFixedSymbols)$
        if $\exists f(x; s_2, p_2) \in chunks$
        then $chunks - f(x; s_2, p_2) + f(x; s_2, p_2 + p'_1)$
        else $chunks + f(x; s_1, p'_1)$
    ) [] $footprint$
```

Listing 5.12: Augmented base case of `joinFootprints` replacing lines 4 to 9 in Listing 5.11. The function `maximumWhenNotFixedRemoved` returns the maximal possible permission amount of p_1 if the *notFixedSymbols* are removed.

with $max(p_1, p_2)$ as, for example, in $acc(a.f, -(x.f? - 1/2 : -1))$ $x.f? - 1/2 : -1$ has to be replaced by $min(-1/2, -1)$ if $x.f$ is not fixed to obtain the maximum possible value. To account for the occurrence of not fixed symbols in the permission amount the base case of the `joinFootprints` function, lines 4 to 9, has to be changed. Listing 5.12 shows the augmented base case. Here, `maximumWhenNotFixedRemoved`: $Term \rightarrow List[Symbols] \rightarrow Term$ is a function which replaces $e?p_1 : p_2$, if e is determined to be not fixed given the *notFixedSymbols*, appropriately with $max(p_1, p_2)$ and $min(p_1, p_2)$ such that it is the maximum possible value. For the above examples it would be $max(1/2, 1)$ and $-min(-1/2, -1)$ respectively.

5.11 Removing footprint

The last step that remains is removing the computed joined footprint from the initial heap. In the current algorithm, this was done implicitly in each branch. But now we have to do this explicitly. The function for that is shown in Listing 5.13.

It takes as the arguments a heap and a footprint and returns a heap where the footprint is removed. The function goes through the footprint chunk by chunk and finds a chunk in the heap with the same id and same reference. And then updates the heap chunk by removing the permissions amount equivalent to the permissions amount of the corresponding footprint chunk. If no such chunk is found, which should never happen as chunks in the footprint were originally part of the original heap, there is an error. Let us look at an example. Let the heap be $h : f(x; s_1, 1/2), g(x; s_2, 1/1)$ and the footprint $foot : g(x; \_, 1/2), f(x; \_, max(1/2, r))$. The result of `heapRemFootprint`$(h, foot)$ will be $f(x; s_1, 1/2 - max(1/2, r)), g(x; s_2, 1/2)$.

```
1 heapRemFootprint: $H \rightarrow List[chunck] \rightarrow H$
2
3 heapRemFootprint: $(h, []) = h$
4 heapRemFootprint: $(h, f(x; s_1, p_1) : chs) =$
5     $h'$ := if $\exists f(x; s_2, p_2) \in h$
6         then $h - f(x; s_2, p_2) + f(x; s_2, p_2 - p_1)$
7         else $error$
8     heapRemFootprint$(h', chs)$
```

Listing 5.13: Function to remove a footprint from a heap. On line 5 we check for the existence of chunks with the exact same symbol as reference and field id. This check does not involve the SMT solver. This is due to soundness reasons, but it might lead to incompleteness due to aliasing.

5.12 Combine everything

Now everything is ready to be combined into the final algorithm. This final algorithm is shown in Listing 5.14. Line 22 to 24 computes the per branch not fixed symbols. Line 26 then combines the branch conditions, the footprint, and the not fixed symbols set of all branches into the final footprint. The footprint is then on line 27 removed from the original heap. The continuation is then executed with a heap where the footprint is removed and a chunk for the wand is added to the original heap. Therefore, trading the footprint for the magic wand. As the path conditions, we use π_1 as they are the one from σ_1 plus the branch dependent path conditions, which are important as they store the information about the structure of s_{lhs}.

```
exec($\sigma_1$,package $a_1 --* a_2, Q$) =
    Let $id_{wand}(e')$ be a magic wand chunk identifier
    corresponding to $a_1 --* a_2$
    $wandChunk := id_{wand}(e')$
    $s_{lhs}$ := fresh
    $id$ := fresh
    $results_1$ := []
    $\pi_1 := \sigma_1.\pi$
    $\sigma_{empty}$ := $\sigma_1\{h$ := $\emptyset, \pi$ := pc-push($\sigma_1.\pi, id, true$)$\}$
    produce'($\sigma_{empty}, a_1, s_{lhs}, \emptyset, (\lambda\sigma_{lhs}, dfg\cdot$
        consume-ext'($[\sigma_{lhs}.h, \sigma_1.h], [], \sigma_{lhs}\{h$ := $\emptyset\}, a_2, (\lambda[\_,\_], footprint, \sigma_{used}, \_\cdot$
            $fixedset_1 :=$ fixedByFootprint($\sigma_{used}.\pi, footprint$)
            $fixedset_2 :=$ dfgToFixSet($\sigma_{used}.\pi, fixedset_1, dfg$)
            $(cnds, bcs)$ := (pc-segs(pc-after($\sigma_{used}.\pi$,id)))
            $results := results$+:$(bcs, fixedset_2, footprint, \sigma_1.h + \sigma_{lhs}.h, \sigma_{used})$
            $\pi_1 :=$pc-add($\pi_1, cnds$)
            $Success()$))
    $\wedge$(
        $sfhl_{foot}$ := fixedByFootprints($\sigma.\pi, [fp|(\_,\_,fp,\_,\_) \in results]$)
        $results_2$ := $[(bcs, dfgToSfhl(\sigma.\pi, sfhl_{foot}, dfg), footprint, h, \sigma_r)|$
            $(bcs, dfg, footprint, h, \sigma_r) \in results]$

        $results_3 := [(bcs_r, footprint_r,$
            fixedHeapLocationsToNotFixedSymbols($\sigma_r.\pi, h_r, fixedset_r$))
            $|(bcs_r, fixedset_r, footprint_r, h_r, \sigma_r) \in result_1]$

        $footprint := joinFootprints(results_3)$
        $h' := heapRemFootprint(\sigma_1.h, footprint)$
        $Q(\sigma_1\{\pi := \pi_1, h :=$heap-add($h', wandChunk$)$\}$))
```

Listing 5.14: Complete algorithm. The blue part is everything added since listing 5.8. In $results_3$ the tuple of branch conditions, $footprint$, and the set of not fixed symbols is recorded for each branch. `joinFootprints` is used to compute the final footprint. The $footprint$ is removed from the heap of the state in which we package the wand to create h'. The heap we give to the continuation is h' plus a chunk which represents the packaged wand.

Chapter 6

Conclusion

In this thesis, we have presented a sound package algorithm for a symbolic execution based verifier. The algorithm is based on the current package algorithm and the notion of fixed expressions. We have reasoned about its soundness, and we have given symbolic execution rules for the algorithm.

The current algorithm makes assumptions that some heap dependent expressions do not change their values between packaging and applying the wand. These assumptions are sometimes unjustified. To categorize expressions that do not change their value between packaging and applying a wand, we introduced the notion of a fixed expression. As it is hard to find the exact set of expressions that are fixed we syntactically over approximated the set of fixed expressions. The new algorithm then assumes that a heap expression does not change its value between packaging and applying the wand only if the expression is in the over approximation of the set of fixed expressions.

6.1 Future work

As this thesis only provides the symbolic execution rules for a package algorithm one next step would be to implement it in Silicon. The algorithm then needs to be evaluated with respect to completeness and performance. If the package algorithm is not precise enough for common magic wands or the newly introduced overhead in performance is too high, the algorithm might need to be refined. One bottleneck of the algorithm described in Section 4.6 is the syntactic approximation of the set of fixed heap locations. There might be a way to use a semantic overapproximation instead of a syntactical one or to refine the existing one.

As this thesis only covers a subset of the Viper language a next step should also be to use the idea of the thesis for the rest of the language. Most notable are predicates, proof scripts, nested wands, and quantified permissions. We think that the first two are easy to support, but for the latter two, the ideas of the thesis have to be extended.

References

[1] Richard Bornat et al. “Permission Accounting in Separation Logic”. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’05. Long Beach, California, USA: Association for Computing Machinery, 2005, pp. 259–270. ISBN: 158113830X. DOI: 10.1145/1040305.1040327. URL: https://doi.org/10.1145/1040305.1040327.

[2] Thibault Dardinier et al. “Sound Automation of Magic Wands”. In: *Computer Aided Verification*. Ed. by Sharon Shoham and Yakir Vizel. Cham: Springer International Publishing, 2022, pp. 130–151. ISBN: 978-3-031-13188-2.

[3] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[4] James C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252. URL: https://doi.org/10.1145/360248.360252.

[5] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. “Viper: A verification infrastructure for permission-based reasoning”. en. In: *Dependable Software Systems Engineering*. Ed. by Alexander Pretschner, Doron Peled, and Thomas Hutzelmann. Vol. 50. Amsterdam: IOS Press BV, 2017, pp. 104–125. ISBN: 978-1-61499-809-9. DOI: 10.3233/978-1-61499-810-5-104.

[6] United States. General Accounting Office et al. *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudia Arabia : Report to the Chairman, Subcommittee on Investigations and Oversight, Committee on Science, Space, and Technology, House of Representatives*. The Office, 1992. URL: https://books.google.ch/books?id=EG3stCQt5coC.

[7] J.C. Reynolds. “Separation logic: a logic for shared mutable data structures”. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[8] Malte Schwerhoff and Alexander J. Summers. “Lightweight Support for Magic Wands in an Automatic Verifier”. In: *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Ed. by John Tang Boyland. Vol. 37. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 614–638. ISBN: 978-3-939897-86-6. DOI: 10.4230/LIPIcs.ECOOP.2015.614. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5240.

[9] Malte H. Schwerhoff. “Advancing Automated, Permission-Based Program Verification Using Symbolic Execution”. en. PhD thesis. Zürich: ETH Zurich, 2016. DOI: 10.3929/ethz-a-010835519.

[10] Jan Smans, Bart Jacobs, and Frank Piessens. “Implicit Dynamic Frames”. In: *ACM Trans. Program. Lang. Syst.* 34.1 (May 2012). ISSN: 0164-0925. DOI: 10.1145/2160910.2160911. URL: https://doi.org/10.1145/2160910.2160911.

[11] Alexander J Summers and Matthew J Parkinson. “The Relationship Between Separation Logic and Implicit Dynamic Frames”. In: *Logical Methods in Computer Science* 8 (2012).

ETH
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):

Sound Automation of Magic Wands in a Symbolic-Execution Verifier

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

Name(s):	**First name(s):**
WIDMER	NICOLA

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)**
Meilen 4.9.22	[illegible]

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