Advancing Non-Standard Permission Utilisation in Program Verification

Bachelor's thesis

Tobias Brodmann

Supervisors: Prof. Dr. Peter Müller, Dr. Malte Schwerhoff

12.09.2018

Contents

Chapter 1

Introduction

The Viper project [11], which is developed at ETH Zurich, provides an intermediate verification language and verification tools for encoding a wide variety of concurrent verification problems. A key feature of Viper's approach to verification is the notion of *permissions* to control access to and formally reason about resources: in Viper, each heap location is such a resource (e.g. `this.val`); as are *recursive predicates* and *magic wands*, which can be used to control access to and enforce orderly modification of complex data structures [4].
Viper includes two verifiers for automated reasoning about verification problems involving such resources: Silicon [8], which is based on a technique called symbolic execution, and Carbon, which generates verification conditions.
Ultimately, both verifiers generate formulas which are discharged by an SMT solver.

Viper's support for standard utilisation of permissions, i.e. the checking and transferring of permissions between verification contexts such as a caller and a callee, is well-developed; among other things, because the theory behind this kind of utilisation has been extensively researched in the last two decades.
As more and more complicated and complex programs and systems need to be verified, this standard use of permissions is no longer enough and new features are needed.
To verify such programs, Viper additionally also supports non-standard utilisation of permissions, such as **`forperm`** expressions which serve as a quantifier over all objects for which a specified permission is currently held. Features for non-standard uses of permissions such as **`forperm`** are unique to Viper and have been shown to be immensely useful in recent projects such as [10]. However, they are not yet fully compatible with already existing Viper features. This can lead to unexpected errors for users, because some combinations of features are allowed while other, seemingly very similar, combinations are not.

An example of this problem is *permission introspection*, the querying of permissions and acting depending on their availability. Consider the following code snippet:

```
if (perm(P --* Q) == write) {
  exhale P --* Q
} else {
  exhale R
}
```

Listing 1.1: Permission introspection with a magic wand

In this example, we want to *exhale* a permission, i.e. give away the access permission to a resource, but which permission to give away depends on other, potentially currently held permissions. `P`, `Q` and `R` in the example can be any Viper assertion, including permission assertions. This is useful because one permission might be more valuable or more important to us, and therefore we want to control which permission is exhaled. The `--*` operator in this example denotes the *separating implication* or *magic wand* in separation logic. While permission introspection in general is possible in Viper, this particular combination with a magic wand is not, which can be confusing for users.

Another example of such a problem is affecting the verification state by making assumptions about the currently held permissions. In Viper, there are two statements that can be used to add assumptions to the verification state: **`inhale`** and **`assume`**. The main difference between the two statements is that **`inhale`** can modify the state (i.e. what permissions are currently held) whereas **`assume`** will never modify the state. Consider the following examples in Listing 1.2 and 1.3:

```
//acc(y.f)
inhale acc(x.f)
//acc(y.f), acc(x.f)
```

Listing 1.2: **`inhale`** with a non-pure assertion

Here, a client holds the permission to `y.f` before the **`inhale`** statement (denoted by **`acc`**`(y.f)` in the preceding comment). When we now **`inhale`** **`acc`**`(x.f)` we add the permission to `x.f` to the held permissions (as denoted in the comment).

```
//acc(y.f)
assume acc(x.f)
//acc(y.f), x = y
```

Listing 1.3: **`assume`** with a non-pure assertion

In this example, a client holds the same permission to `y.f` as before, but instead of **`inhale`**, there is an **`assume`** statement. Now we do not add any new permissions to the state, but instead we learn that `x = y`, because we assume that permission to `x.f` is already being held and the only permission held is to `y.f`. Assuming that permissions are already held is not yet supported, in contrast to assuming pure (non-permission) assertions, e.g. `x.f != 0`.

A third problem that arises related to permissions and heap location is their use as triggers in quantified expressions: Viper supports the use of *quantified expressions*, which are passed on to the underlying SMT Solver. Since SMT Solvers obviously cannot instantiate a quantified expression for every single

instance of the quantified variable(s), SMT Solvers use *triggers* to decide where to instantiate quantified expressions. Cosider the example in Listing 1.4: in this quantifier, `f(i)` and `g(j)` act as triggers: whenever terms of this form are encountered, the quantifier is instantiated with the appropriate values for `i` and `j`.

```
forall i: Int, j: Int :: {f(i), g(j)} f(i) && i < j ==> g(j)
```

Listing 1.4: Quantified expression

However, while triggers work very well if they are available, there are still expressions for which no trigger is available. If that is the case, the quantifier is essentially useless since it will not be instantiated. An example of such a quantifier with no trigger is seen in Listing 1.5.

```
assume forall x: Ref :: perm(x.f) > none ==> x.f != 0
```

Listing 1.5: Quantifier that requires a heap-dependent trigger

Here, we want to assume that for all `x` to whose field `f` we have access to, `x.f` is not 0. The only possible triggers for this quantifier are **`perm`**`(x.f)` or `x.f`, both of which are not supported.

1.1 Chapter overview

This thesis is split into 8 chapters: Chapter 2 will present some background information regarding Viper and Silicon which is helpful for understanding the rest of the thesis. The main contributions of this thesis can be found in the following three chapters. Chapter 3 covers how to extend Viper's permission introspection features to be able to deal with all resources currently available in Viper. In Chapter 4, we will present a technique to allow Viper to support impure **`assume`** statements. Chapter 5 will cover how to extend Silicon to allow heap-dependent expressions to be used as triggers in quantified expressions. In Chapter 6, we will discuss the performance impact of the new features, in particular that of heap-dependent triggers. Finally, Chapter 7 presents a conclusion to this thesis and Chapter 8 will present some future work related to this thesis.

Chapter 2

Background

2.1 Implicit Dynamic Frames

Viper is based on implicit dynamic frames [9], a program logic that is closely related to separation logic [7] [6]. The main concept of implicit dynamic frames is that of *accessibility predicates* such as `acc(x.f)` which denotes permissions to access the heap location `x.f`. There are two types of permissions that can be held: full (i.e. write) permissions and partial (i.e. read) permissions. Write permissions are exclusive whereas an arbitrary number of clients can hold permissions to read a heap location.

2.2 Separation Logic

Separation logic [7] is an extension of Hoare logic [1]. It extends Hoare logic with two important concepts which are also used by the implicit dynamic frames logic and Viper:

- *Separating Conjunction* (denoted $*$): In contrast to a regular boolean conjunction, a separating conjunction of two accessibility predicates requires that we can split the current heap into two disjoint parts in which the two parts of the conjunction hold. In the context of implicit dynamic frames and accessibility predicates, this means that the total permissions held are required to be the *sum* of the individual permissions if the accessibility predicates are concerned with the same location. For example, the separating conjunction `acc(x.f, p) * acc(y.f, q)` requires that if `x` and `y` are aliases, i.e. refer to the same location, then we need `p + q` permissions to the field `x.f` to satisfy it. If `x` and `y` are not equal, we need `p` and `q` permissions respectively to their field `f`, as expected. In particular, if a separating conjunction requires write permissions for both `x.f` and `y.f`, this implies that `x` and `y` are not aliases.
- *Magic Wand* (denoted $-\!\!*$): The magic wand or separating implication asserts that if the current heap (in which the magic wand holds) is extended with a disjoint heap in which the left-hand side of the magic wand holds, the right-hand side will hold in this combined heap. Maybe more

intuitively, it can be seen as a promise that if we gain the left-hand side assertion at some point in time, we can exchange it for the right-hand side.

2.3 Viper

2.3.1 Resources

The concept of *resources* is particularly important in my thesis, as all of the features added to Viper and Silicon should work for any resource. Viper currently supports three different types of resources: heap locations (e.g. `this.val`), *recursive predicates* and *magic wands*.
To formally represent a resource, we denote it by `res(args)`, where `args` are the arguments used for a particular resource instance. In the case of fields, there is only a single argument, the receiver. For predicates, the arguments are simply the arguments required by the predicate. However, for magic wands the arguments might no be immediately clear. Consider the following magic wand: **`acc`**`(x.f, p) --* acc(y.f, q)`. Because of the way magic wands are handled in Viper, this wand can be split in the *magic wand structure*, which is independent of the state and the subexpressions which need to be evaluated in the current state. The split wand would look like this: **`acc`**`(_.f, _) --* acc(_.f, _) (x,p,y,q)`. In this case, `(x,p,y,q)` would be the arguments for this magic wand instance. For the purpose of this thesis, magic wands can be treated the same as predicates, since we are only interested in the magic wand as a resource.
Two resource instances are considered to be equal if and only if all their arguments match. Consider for example fields: the locations denoted by two fields are equal if their receivers are aliases.

2.3.2 Permissions

In Viper, to access any resource, we require *permissions* to it. Viper permissions are fractional values `p/q` between 0 and 1. There are also the special values **`write`** and **`none`** representing `1/1` and `0/1` respectively. Permissions to a resource can be specified in the accessibility predicate: **`acc`**`(x.f, p)` represents `p` permissions to `x.f`. For ease of use, **`acc`**`(x.f)` is treated as syntactic sugar for **`acc`**`(x.f, `**`write`**`)`, i.e. to represent full permissions. To gain and lose permissions, Viper includes the statements **`inhale`** `a` and **`exhale`** `a`, which add respectively remove permissions if a is a permission assertion.

2.3.3 Quantified Permissions

A special feature of Viper are *quantified permissions*: It is possible to define permissions of the form **`forall`** `x: `**`Ref`**` :: c(x) ==> `**`acc`**`(x.f)`, where `c(x)` represents some boolean condition depending on `x`. This can be a function application, but it does not necessarily have to be, for example the quantified permission assertion **`forall`** `x: `**`Ref`**` :: x `**`in`**` xs ==> `**`acc`**`(x.f)` represents permissions to all elements of `xs`.

2.4 Silicon

2.4.1 Silicon Chunks and Snapshot Maps

Silicon represents the heap as a collection of *heap chunks*. These chunks represent the resources which are accessible in the current state. Each chunk represents access to one or multiple (in the case of quantified permissions) instances of a particular resource.
Each chunk is associated with a *snapshot map*: In the simple case of fields, this is a partial function representing the value of the field at some receiver. For example, if there is a variable `x` and the current heap provides access to `x.f`, then the corresponding chunk's snapshot map `sm` represents the value of `x.f`. If we now assign `x.f := 10`, then `sm(x)` would be equal to `10`. In the case of a predicate, the snapshot map represents the values that the predicate abstracts over. Overline notation is used to denote repetition in the following, i.e. $\overline{args}$ means one or more arguments. We represent chunks by $id(\overline{args}; sm, p)$, where $\overline{args}$ are the arguments of the resource instance the chunk corresponds to, sm is the snapshot map of this chunk and p are the permissions provided this chunk. id is the identifier of the chunk, i.e. the name of a field or predicate. For example, the chunk corresponding to the accessibility predicate **acc**`(x.f, p)` would be $f(x, sm, p)$, where sm is a new snapshot map.

2.4.2 Symbolic State

Silicon keeps a *symbolic state* during symbolic execution to include all known information up to this point. It includes the following:

- A *store* that maps local variables to their symbolic values.
- A *path condition stack* that collects all constraints that are gathered during symbolic execution (e.g. symbolically executing `x := v` would add the path condition `x == v` to the stack). The path condition stack consists of triples $(Id, V, Set[V])$ consisting of a unique scope identifier, a branch condition and a set of path conditions.
- A *heap* consisting of heap chunks representing all the resources we currently have permissions to.

2.4.3 Symbolic Execution Rules

Silicon is based on symbolic execution [2]. In symbolic execution, instead of executing a program best on real input values, the program is instead symbolically executed on some *symbolic values* and a *symbolic state*. During the symbolic execution, constraints on the symbolic values are gathered which can then be used to prove assertions about the program. To define the effect of statements on the symbolic state, Silicon defines *symbolic execution rules*. To extend Silicon with the new features introduced in this thesis, we need to define appropriate symbolic execution rules for the new features. In order to do this, we need some symbolic execution primitives that can be used in new rules. The symbolic execution rules defined here as well as the new rules defined in this thesis are all based on the rules found in [8].

The symbolic execution rules are defined in *continuation passing style*: To each function we pass a *continuation function* as an argument. This function is denoted by `Q` in the following rules and it is called after the current function is done. σ represents the current state, including the heap ($\sigma.h$).
The sets in the signatures represent the following: Σ is the set of states, A is the set of assertions, E is the set of expressions, S is the set of statements, R is the set of verification results (either success or failure) and V is the set of symbolic values. The representation $id(\overline{args})$ is used to refer to a chunk *identifier*. This identifier can be used to find matching chunks in the symbolic heap.

---

1 `produce:` $\Sigma \to A \to Snap \to (\Sigma \to R) \to R$
2 `produce(`σ`, a, s, Q)`

Produce creates a new heap chunk if we gain permissions to some new resource and adds it to the new state. If it used with a pure assertion instead, it adds the assertion to the path conditions. For example, `produce(`σ`, acc(x.f, p), s, Q)` will create the new heap chunk $f(x; s, p)$. After the creation of the chunk, `Q` is called with the state obtained by adding this new chunk to the symbolic heap.

---

1 `consume:` $\Sigma \to A \to (\Sigma \to Snap \to R) \to R$
2 `consume(`σ`, a, Q)`

Consume exhales an assertion, i.e. removes permissions to a resource. If used with a pure assertion, it asserts this assertion in the current state. For example, `consume(`σ`, acc(x.f,p), Q)` searches the heap for a chunk matching $f(x; sm, q)$. It then replaces this chunk by a new chunk $f(x; sm, q - p)$ (or no chunk if all permissions were exhaled). Its continuation is then called with the updated state and sm, the snapshot representing the symbolic values of the exhaled permissions.

---

1 `eval:` $\Sigma \to E \to (\Sigma \to V \to R) \to R$
2 `eval(`σ`, e, Q)`

Evaluates the pure expression `e` in the state σ. This yields a symbolic expression as well as a (potentially) updated state, if new path conditions were added during the evaluation, which are then passed to the continuation `Q`.

---

1 `exec:` $\Sigma \to S \to (\Sigma \to R) \to R$
2 `exec(`σ`, s, Q)`

Executes statement `s` and updates the state accordingly. For example, `exec(`σ`, x := e, Q)` will set the variable `x` to `e` in the store and then call `Q` with the updated state.

Finally, all of these symbolic execution primitives are combined in order to build rules for the symbolic evaluation of Viper. For example, symbolically evaluating **`inhale acc`**`(x.f, p)` would first call `exec`, because we are executing a statement, which in turn would call `produce` to add a new chunk to the heap. The `produce` function would then use the `eval` function to evaluate `x` and `p` and get the correct symbolic arguments for the new chunk.

2.4.4 Auxiliary functions

In addition to the symbolic execution primitives described above, the symbolic execution rules presented in this thesis also make use of several auxiliary functions.

```
1 heap-add: H → Id → Snap → Perm → H
2 heap-add(h, id(v̄), s, p) =
3   (h ∪ {id(v̄; s, p)})
```

Listing 2.1: heap-add

`heap-add` adds a new chunk to a heap and returns the new heap.

```
1 pc-add: Π → V → Π
2 pc-add(π, v) =
3   Let(id, bc, pcs) :: suffix match π
4   (id, bc, pcs ∪ {v}) :: suffix
```

Listing 2.2: pc-add

`pc-add` is used to add new path conditions found during the symbolic execution.

```
1 check: Π → V → Bool
2 check(π, v) = pc-all(π) ⊢_SMT v
3
4 assert: Π → V → R
5 assert(π, v) =
6   if check(π, v) then success()
7   else failure
```

Listing 2.3: check and assert rules to check path conditions

The assert rule can be used to check whether the current path conditions imply some assertion. In order to do this, the SMT Solver that Silicon uses for its backend is queried, as can be seen in the check function.

Chapter 3

Permission Introspection

To reason about what permissions are currently held, Viper supports two types of permission introspection features: **perm** and **forperm**. **perm** can be used in conjunction with a resource to get the current permissions to this resource. An example of how **perm** can be used can be seen in Listing 3.1. Here, we **inhale** permissions, but which permission exactly depends on some value `b`. To **exhale** the permissions again, we can use **perm** to test whether we have permissions to `y.f`.

```
inhale b ? acc(y.f) : acc(z.f)

if (perm(y.f) >= write) {
  exhale acc(y.f)
} else {
  exhale acc(z.f)
}
```

Listing 3.1: Example use of **perm**

The other permission introspection feature of Viper is **forperm**. **forperm** is a boolean expression and serves as a quantifier over resources to which we currently hold permissions. A **forperm** expression is true if and only if for every instance of the quantified resource that we currently hold permissions to, its body holds.

```
inhale acc(x.f) && acc(y.f)
inhale acc(z.g)
assume x.f > 0 && y.f > 0

assert forperm [f] r :: r.f > 0
```

Listing 3.2: Example use of **forperm**

An example use of a **forperm** expression can be seen in Listing 3.2. Here, we have permissions to two instances of the field `f`: `x.f` and `y.f`. The assertion at the end asserts that for both of these instances, the value of the field is positive. Semantically, the expression **forperm** `[f] x :: x.f > 0` is equivalent to **forall** `x: Ref :: perm(x.f) > none ==> x.f > 0`.

At the start of this thesis, **perm** and **forperm** only supported a subset of the resources available in Viper. In particular, **perm** did not support magic wands and **forperm** only supported fields and predicates which had exactly one argument of type **Ref**, which represents references in Viper (i.e. which were as close to fields as possible).

Resource	**perm**	**forperm**
Fields	✓	✓
Predicates	✓	✗
Magic wands	✗	✗

Table 3.1: Previous support for permission introspection

Additionally, **forperm** also completely ignored quantified permissions in Silicon, which meant that an example such as Listing 3.3 would wrongly verify in Silicon, whereas Carbon correctly found the assertion to be false.

```
inhale forall x: Ref :: x in xs ==> acc(x.f)
assume forall x: Ref :: x in xs ==> x.f < 0
assert forperm [f] x :: x.f > 0
```

Listing 3.3: **forperm** in combination with quantified permissions

In this chapter, we will complete Table 3.1 and enable full support for all Viper resources with both permission introspection features.

3.1 Perm

Magic wand support of **perm** required some changes in both Viper and Silicon. In Viper, the parser and AST had to be adjusted, both of these now allow any resource to be used with **perm**. This also means that if any resources are added to Viper in the future, Viper should not need to be changed to allow **perm** to support these resources.
In Silicon, we needed an additional evaluation case that gathers all magic wand chunks and sums up all the permissions to these chunks.

The evaluation rule for perm is the same as the old rule from [8], with the difference being that we allow `id` to be a magic wand as well.

```
1 eval: Σ → E → (Σ → V → R) → R
2 eval(σ1, perm(id(ē)), Q) =
3   eval(σ1, ē, (λ σ2, ē′ ·
4     Let h_id ⊆ σ2.h contain all heap chunks for identifier id
5     sum := fold1(h_id, 0, (λ id(v̄;_,p), q ·
6       q + ite(⋀ ē′ = v̄, p, 0)))
7     Q(σ2, sum)
```

Listing 3.4: Evaluation rule for **perm**

In Listing 3.4, we find the symbolic evaluation rule for **perm**. The rule states that when evaluating a **perm** expression, first of all we evaluate the arguments of the resource ($\overline{e}$). These arguments are Viper expressions and the evaluation results in symbolic expressions ($\overline{e'}$) for the arguments that can be used by Silicon.

We then gather all relevant chunks of the heap (i.e. all the chunks providing permissions to the resource we are interested in). Then we sum up all the permissions provided by these chunks, given the arguments of the chunk match the arguments given by the symbolic expressions $\overline{e'}$. The underscore notation used in line 5 for $id(\overline{v};\_,p)$ denotes an arbitrary value, it is irrelevant for the rest of the rule and therefore does not need a name.
Here, `foldl` represents a left fold as can be found in functional programming languages such as Haskell. It takes a collection, a starting value and a combination function as arguments and then iteratively applies the combination function to the next element of the collection and the result of the previous computation, starting with the starting value. `ite` is used to the denote a conditional expression, equivalent to the ternary `?:` operator that is found in Viper as well as in many programming languages. If the first argument holds, the value of the expression is the second argument, otherwise it is the third argument. Finally, the value returned by this evaluation is the sum of all the chunk permissions.
For example, assume the current heap contains two magic wand chunks corresponding to the wands **acc**`(x.f)` `--*` **acc**`(y.f)` and **acc**`(a.f)` `--*` **acc**`(b.f)`. These chunks would be of the form $wand(x', y'; sm, 1)$ and $wand(a', b'; sm', 1)$. Now we want to symbolically evaluate the expression **perm**(**acc**`(c.f)` `--*` **acc**`(d.f)`). According to the rule, we first evaluate the arguments, in this case `c` and `d`. Assume that these are evaluated to c' and d' respectively. Next we gather all chunks with the same identifier, in this case this would be the two chunks mentioned before. Then we sum up the permissions provided by these chunks, this would result in the following term: $(\texttt{ite}(a' = c' \wedge b' = d', 1, 0)) + (\texttt{ite}(x' = c' \wedge y' = d', 1, 0))$. Finally, this term would be passed on to the continuation function.

3.2 Forperm

To support the use of any resource with **forperm**, some additional changes were needed. First of all, the syntax of **forperm** had several restrictions that posed a problem to extending it to the yet unsupported resources.

Old `forperm` syntax:
forperm [(*predicate* | *field*)] *variable* :: *expression*

Shortcomings of the old `forperm` syntax:

- The old **forperm** syntax only allowed a single variable. This is a problem for resources with more than one argument as it makes it impossible to mention a specific argument in the body fo the **forperm** expression.
- The variable's type could not be declared, it was always of type **Ref**.
- Resources were specified by their name only, not their arguments. This is a problem because it requires that all arguments to a resource are quantified. We could not specify things such as `P(x,3)` if we were interested only in the predicates whose second argument is 3.

To combat these shortcomings, the **forperm** syntax was changed to allow easier extension to support all resources.

New `forperm` syntax:
`forperm` *variable*: *Type* [*resourceAccess*] :: *expression*

In the new syntax, there are multiple quantified variables, such that is possible to quantify all arguments of an arbitrary predicate or magic wand. Additionally, the type of any quantified variable can now be specified by the user, which is necessary for predicates or magic wands with non-**`Ref`** type arguments. The resource that we consider for **`forperm`** is now specified together with its arguments, some of which can also be non-quantified, e.g.: **`forperm`** `x: Ref [P(x, 3)] :: x != null`, where `P(x: Ref, i: Int)` is a predicate, is supported and would only consider predicate instances whose second argument is 3. An example of this can be seen in Listing 3.5. As an additional simplification of the language, **`forperm`** also only allows a single resource under the new syntax. This is however not a reduction of the language: If multiple resources were specified, this meant that **`forperm`** should consider permissions to instances of either resource. Semantically, this is equivalent to the conjunction of multiple **`forperm`** expressions with the same body, one for each resource. Therefore, anything that could be specified before can now still be specified.

```
inhale P(null, 5)
inhale P(z, 3)
assume z != null

assert forperm x: Ref [P(x,3)] :: x != null //verifies
assert forperm x: Ref, i: Int [P(x,i)] :: x != null //fails
```

Listing 3.5: Using non-quantified arguments with **`forperm`**. The first assertion verifies, because the second argument of `P(null, 5)` is not 3, therefore we only consider the instance `P(z, 3)`. In the second assertion however, we consider both instances and therefore the assertion fails.

The symbolic evaluation rule for **`forperm`** found in Listing 3.6 has the following changes from the original found in [8]: The biggest change is that we need to accommodate the fact that some of the arguments of the resource might not be quantified, therefore we need to first evaluate all non-quantified arguments. For easier notation, the non-quantified arguments are mentioned last in the rule, however, there is no requirement for that, the quantified and non-quantified arguments can be interleaved as well. The body is denoted as $b(\overline{x})$. This does not mean that the body is a function application, but rather that the body is a (boolean) expression potentially depending on $\overline{x}$. Also, as in the **`perm`** evaluation rule, we allow *id* to be any resource, so a field, predicate or magic wand. Then to evaluate **`forperm`**, we iterate over all relevant chunks and check that if 1) we do have permissions to the chunk and 2) the non-quantified arguments match, then the body of the **`forperm`** holds. Additionally, to support the use of quantified permissions with **`forperm`**, the permission amount of the chunk can now depend on the arguments of the chunk. This is necessary for quantified permissions, since the permission amount might depend on the quantified variable(s).

```
1  eval: Σ → E → (Σ → V → R) → R
2  eval(σ1, forperm x̄ : T̄ [id(x̄, ē)] :: b(x̄), Q) =
3    eval(σ1, ē, (λ σ2, ē′ ·
4      Let h_id ⊆ σ2.h contain all heap chunks with identifier id
5      Let z̄ : T̄ be fresh program variables s.t. |z̄| = |x̄|,
6      and let z̄′ be corresponding fresh symbolic values
7      cnj(z̄) := foldl(h_id, true, (λ id(v̄;_, p(v̄)), c ·
8        Let (v̄1 :: v̄2) = v̄ be the arguments corresponding
9        to x̄ and ē respectively
10       c ∧ (p(z̄′, ē′) > none && ⋀ z̄′ = v1 && ⋀ ē′ = v2 ==> b(z̄)
11     eval(σ2 {γ := σ2.γ[z̄ ↦ z̄′]}, cnj(z̄), (λ σ3, cnj(z̄)′ ·
12       Q(σ3{γ := σ2.γ}, ∀ x̄ : T̄ · cnj(x̄)′))
```

Listing 3.6: Evaluation rule for **forperm**, the main additions are highlighted

Chapter 4

Assume

The Viper language supports a variety of statements, of these, four of these are similarly concerned with assertions: **`inhale, exhale, assume, assert`**. However, as can be seen in table 4.1, in contrast to the other statements, **`assume`** can only be used in conjunction with pure assertions.

Support for assertions	
`exhale` `a`	✓
`inhale` `a`	✓
`assert` `a`	✓
`assume` `a`	✗ (if not pure)

Table 4.1: Support for non-pure assertions

As part of this thesis, I extended the assume semantics to also support non-pure assertions (i.e. resource access assertions). The semantics for **`assume`** `a` (as opposed to **`inhale`** `a`) are that it is assumed that the assertion `a` holds in the current state without modifying the state. On the other hand, **`inhale`** `a` will change the state if `a` is an impure assertion. In this case it will add the permissions to the resources in `a` to the state (i.e. create corresponding chunks in Silicon).
For impure assertions such as **`assume acc`**`(x.f, p)`, we assume that the current heap provides `p` permissions to the field `x.f`. Therefore we learn that `x` has to be an alias of some other variable to whose field `f` we have at least `p` permissions. For example, if we had at least `p` permissions to some field `y.f` before the **`assume`** statement and nothing else provides permissions to the field `f`, then we would learn that `x == y`.

There were several options to add support for non-pure assertions: One possibility was a map-based approach that would encode locations to which we hold permissions as a map in the SMT Solver underlying Silicon. In this approach, whenever we **`assume`** a certain permission amount to a resource, we would increment the value of the map at the resource by this permission amount. This way possible aliasing would not need to be handled by Viper, but would instead be handled by Z3 [13], which is the SMT Solver that Silicon uses as its backend. However, an implementation of this approach would only work for Silicon (for now) and would have to be implemented separately for Carbon.

An alternative approach can be found when we realize that **assume acc**(x.f) is semantically equivalent to **assume perm**(x.f) == **write**, the latter of which is already supported as it is a pure assertion. If any impure **assume** statement could be rewritten in this way and turned into a pure statement, no additional support would be needed. A big advantage of this approach is that it also supports Carbon directly without any changes, because the translation would be a Viper to Viper translation resulting in a currently supported program.

This chapter will explore how to rewrite arbitrary impure assertions into pure assertions in order to complete Table 4.1.

4.1 Naive Rewriting

This section will describe the first approach used to rewrite impure **assume** statements into semantically equivalent pure **assume** statements. In order to simplify the explanation, we will first consider only fields and non-quantified permissions and later show how to extend the same approach to be used with the other resources as well as quantified permissions.

The main challenge when rewriting assume statements is the difference between a separating conjunction and a standard boolean conjunction: The separating conjunction requires the heap to be separable such that in one part of the heap, we have the permissions required by the left conjunct, and in the other part, we have the permissions required by the right conjunct. In particular, if there is aliasing between the receivers of two fields, we require the sum of the permissions required by the two conjuncts. Consider the following assume statement:

```
assume acc(x.f, p_x) && acc(y.f, p_y) && acc(z.f, p_z)
```

Listing 4.1: Impure **assume** with a separating conjunction

Since && denotes a separating conjunction when used with impure assertions, this means that if x, y and z happened to be aliases, we would need p_x + p_y + p_z permissions to x.f/y.f/z.f to be able to split the heap as required. Similarly, if any two of the variables are aliases, we would need the sum of the two respective permissions to the field f.

The first approach to rewrite non-pure assumes into pure assumes was to essentially test all possible combinations of *aliasing conditions* and assign permissions accordingly. The rewriting of the example in Listing 4.1 would look like the following:

```
assume perm(x.f) >= p_x &&
  perm(y.f) >= (y == x ? p_x + p_y :
                         p_y) &&
  perm(z.f) >= (z == x && z == y ? p_x + p_y + p_z :
                          z == x ? p_x + p_z :
                          z == y ? p_y + p_z :
                                   p_z)
```

Listing 4.2: Rewriting of Listing 4.1

However, the big disadvantage of this rewriting is its growth rate. If we have $n + 1$ conjuncts, then to get the largest term we have to consider all

possible combinations of any length of the remaining n conjuncts. For each of those combinations we create an additional conditional expression. This results in $\sum_{k=0}^{n} \binom{n}{k} = 2^n$ conditional subexpressions. Therefore, the terms grow exponentially in the number of separating conjunctions and the runtime also grows accordingly. This meant that only **assume** statements with less than 10 conjuncts could feasibly be verified.

4.2 Fast Rewriting

A much better approach to rewriting is instead of defining the permission amount as one huge conditional expression to define it as a sum of conditional expressions that each represent one aliasing condition. Consider again the example from Listing 4.1. The rewriting of this can be seen in Listing 4.3.

```
assume perm(x.f) >= p_x &&
  perm(y.f) >= p_y + (y == x ? p_x : none) &&
  perm(z.f) >= p_z + (z == x ? p_x : none)
                   + (z == y ? p_y : none)
```

Listing 4.3: Better **assume** rewriting approach for Listing 4.1

Using this approach to rewrite **assume** statements, the number of terms only grows linearly with the number of conjuncts. With this approach, even large conjunctions could be verified very fast.
This approach gives rise to the following rewriting rule for fields:

assume $\ast_{i=0}^{n}$ **acc**(x_i.f, p_i) $\rightsquigarrow$
  assume $\bigwedge_{i=0}^{n}$ **perm**(x_i.f) >= p_i + $\sum_{j=0, j\neq i}^{n}$ (x_i == x_j ? p_j : **none**)

Listing 4.4: Rewriting rule for **assume** with fields

Here, $\ast$ and $\bigwedge$ are used to illustrate the difference between a separating and boolean conjunction, even though in Viper they would both be denoted as `&&`.

4.2.1 Predicates and Magic Wands

In order to generalize the rule from above to predicates and magic wands, we need to consider when two instances of these resources refer to the same resource. This is the case if and only if their arguments are equal, so instead of comparing receivers in the case of fields, we need to compare all arguments for equality.
A more generalized form of the rewriting rule can be found in Listing 4.5, where `args(res)` is used to denote the arguments of a given resource, i.e. the receiver of a field or the arguments of a predicate or magic wand. So `args(x.f)` would refer to `x` and `args(p(x,y,z))`, where `p` refers to a predicate, would refer to `x,y,z`.

assume $\ast_{i=0}^{n}$ **acc**(res_i, p_i) =
  assume $\bigwedge_{i=0}^{n}$ **perm**(res_i) >= p_i + $\sum_{j=0, j\neq i}^{n}$ (args(res_i) ==
    args(res_j) ? p_j : **none**)

Listing 4.5: Generalized rewriting rule for **assume**

4.3 Quantified Permissions

Rewriting assumptions that assume quantified permissions is not as straightforward as for non-quantified permissions: the main issue is that given a resource, determining if a quantified permission assertion gives permission to this resource cannot be easily checked. For example, consider Listing 4.6: Here, the quantified variable is not a reference, but an integer and we gain permissions to the fields of the elements of the sequence `xs`.

```
assume (forall i: Int :: i in [0..|xs|) ==> acc(xs[i].f, p)
    && acc(x.f, q)
```

Listing 4.6: Quantified permission example

In this example, we have no easy way to check if `x` is an alias of an element of `xs`. A possible solution to this problem would be to use existential quantifiers. This way to check if the permissions to `x` are provided by the above quantified permissions we would check **exists** `i: Int :: x == xs[i]`. The rewritten **assume** using existential quantifiers can be seen in Listing 4.7. However, even though existential quantifiers are supported by Viper, we would like to avoid them, because then the verifier needs to find a witness for this quantifier, which tends to be much harder than using universal quantifiers.

```
assume (forall i: Int :: i in [0..|xs|) ==> perm(xs[i].f)
    >= p) &&
  perm(x.f) >= q + (exists i: Int :: x == xs[i] ? p :
      none)
```

Listing 4.7: Rewriting of Listing 4.6 using existential quantifier

A better alternative that avoids using existential quantifiers is to define *inverse functions*. In the above example, the inverse function would be a mapping from references to integers. The rewriting of Listing 4.6 using inverse functions can be seen in Listing 4.8. In this example, `inv` is a fresh function only used for this purpose. In order to define `inv`, we also need to add the two axioms found at the top.

```
// axioms defining the inverse function
assume forall r: Ref :: inv(r) in [0..|xs|) ==> r ==
    xs[inv(r)]
assume forall i: Int :: i in [0..|xs|) ==> inv(xs[i]) == x
// the actual rewritten assume
assume (forall r: Ref :: inv(r) in [0..|xs|) ==>
    perm(r.f) >= p) &&
  perm(x.f) >= q + (inv(r) in [0..|xs|) ? p : none)
```

Listing 4.8: Rewriting of Listing 4.6 using inverse functions

As a matter of fact, both Silicon and Carbon currently instantiate such inverse functions for all quantified permissions, since they are also needed in many other cases, however, these cannot be used for a Viper to Viper translation, they only become available later and therefore new inverse functions were required. The inverse functions can now already be instantiated as Viper functions, allowing for full compatibility with the new **assume** rewriting. An example of the axiomatizing of inverse functions can be seen in listings 4.9 and 4.10. The

definitions for the inverse functions are the same found in [3]. In this example, `c(x)`, `e(x)` and `p(x)` do not necessarily denote function applications, but they can be any expression of type **`Bool`**, **`Ref`** and **`Perm`** respectively, and can potentially depend on `x`. There can also be multiple quantified variables, the quantified permission assertion is not restricted to a single variable. If there are multiple quantified variables, there is one inverse function per quantified variable in order to specify to which quantified variable we are mapping.

```
forall x: T :: c(x) ==> acc(e(x).f, p(x))
```

Listing 4.9: General quantified permission example

```
// definitional axioms
forall r: Ref :: c(inv(r)) ==> r == e(inv(r))
forall x: T :: c(x) ==> inv(e(x)) == x
// quantified permission assertion using inverse function
forall r: Ref :: c(inv(r)) ==> acc(r.f, p(inv(r)))
```

Listing 4.10: Inverse function definitions for 4.9, where `inv` is a fresh function

All quantified permissions are rewritten using the inverse functions, this way they are always quantified over references and the quantified variable can easily be replaced by whatever we want to check. Since now there is a lot of duplicate functionality across silver and the verifiers, ideally in the future inverse functions could be generated in Silver directly and both Carbon and Silicon could use those.

For predicates and magic wands, the same type of inverse functions can also be built, as can be seen in Listings 4.11 and 4.12. Once again, `c(`x`)`, $\overline{e(x)}$ and `p(`x`)` are not necessarily function application, but expressions that potentially depend on the quantified variable. Also, multiple quantified variables are possible here as well. $\overline{y}$ represent the possibility for the predicate to have some non-quantified arguments which of course do not have to be the last arguments, but for readability they are placed last.

`forall` $x : T$ `::` `c(`$\overline{x}$`)` `==>` **`acc`**`(pred(`$\overline{e(x)}$`,` $\overline{y}$`),` `p(`$\overline{x}$`))`

Listing 4.11: Predicate quantified permission example

```
// definitional axioms for inverse function
forall r̄ : S :: c(inv(r̄)) ==> r̄ == e(inv(r))‾
forall x : T :: c(x) ==> inv(e(x)‾, ȳ) == x
// rewritten quantified permission assertion
forall r̄ : S :: c(inv(r̄)) ==> acc(pred(r̄, p(inv(r̄))‾))
```

Listing 4.12: Inverse function definitions for 4.11

Currently, the axiomatization of inverse functions does not include any injectivity checks. However, for the inverse functions to really exist, the original function (i.e. `e(x)` in listing 4.9) has to be injective. This is not a problem in the case of **`assume`** statements, because both verifiers currently only check injectivity on **`exhale`** and **`assert`** statements, but importantly not on **`assume`**'s duality **`inhale`**, so it is consistent with the current behaviour. However, if the Viper inverses would be used in other cases as well, an injectivity check would need to be added.

4.4 Full translation example

Both the inverse functions and the assume translation are implemented as Viper domain functions which are equal to mathematical functions. They are domain functions because regular Viper functions always take the state into account (they can have preconditions concerning the current heap etc.). However, both the inverses and the assume translation should be independent of the current state. The assume translation is implemented using helper functions. Helper functions are generated as needed for the size of the largest separating conjunction in an **assume**. The arguments of the helper functions are the *aliasing conditions* as well as the corresponding permissions. They are defined as `assume_helper_n(`$\overline{c}$`, `$\overline{p}$`, q) = q + `$\sum_{i=0}^{n}$` (c_i ? p_i : none)`. The reason for using these helper functions (instead of translating in place) is the better performance. As mentioned before, the main problem with rewriting was the performance for large separating conjunctions. The main cause of the bad performance was the rapid growth of the SMT code generated by Silicon. Using helper functions helps to keep the generated code small, because they can be reused with consecutive assumes.
An example of the final translation can be seen in Listings 4.13 and 4.14. To simplify the example, triggers for all the quantifiers are omitted.

```
field f: Int

method m(xs0: Seq[Ref], xs1: Seq[Ref], xs2: Seq[Ref], xs3:
    Seq[Ref], p: Perm) {
  inhale forall i0: Int :: i0 in [0..|xs0|) ==> acc(xs0[i0].f)

  assume (forall i1: Int :: i1 in [0..|xs1|) ==> acc(xs1[i1].f, p))
    && (forall i2: Int :: i2 in [0..|xs2|) ==> acc(xs2[i2].f, p))
    && (forall i3: Int :: i3 in [0..|xs3|) ==> acc(xs3[i3].f, p))
}
```

Listing 4.13: **assume** containing two conjuncts

In Listing 4.13, we have three quantified permission assertions, all of which need to be transformed to use inverse functions. This is the first thing that happens: as we can see, in the translation there is a domain added for the inverse functions, with one function per quantified permission assertion (in a more complicated example with predicates or magic wands and multiple quantified variables, there would be multiple functions per quantified permission assertion). The axioms to specify the inverse functions are not added as domain axioms. The reason for this is that they depend on variables which are not available to domain axioms, for example, the inverse to `forall i1: Int :: i1 in [0..|xs1|) ==> acc(xs1[i1].f, p)` is only defined with regards to `xs1` and therefore depends on it. Because of this, the defintions of the (partial) inverse functions are assumed before the actual transformed **assume** as additional axioms.
The assume functions, on the other hand, use domain axioms to define the value of the functions. This is possible because they are defined independently of any local variables, they are also the same for every **assume** statement.

```
// inverse function declaration
domain Inverse {
  function inv_0(r: Ref): Int
  function inv_1(r: Ref): Int
  function inv_2(r: Ref): Int
}

// assume helper functions including definitions
domain Assume {
  function assume_helper_1(c_1: Bool, p_1: Perm, p_0: Perm): Perm
  function assume_helper_2(c_2: Bool, c_1: Bool, p_2: Perm, p_1:
      Perm, p_0: Perm): Perm
  axiom assume_helper_1_axiom {
    (forall c_1: Bool, p_1: Perm, p_0: Perm ::
      assume_helper_1(c_1, p_1, p_0) == p_0 + (c_1 ? p_1 : none))
  }
  axiom assume_helper_2_axiom {
    (forall c_2: Bool, c_1: Bool, p_2: Perm, p_1: Perm, p_0: Perm ::
      assume_helper_2(c_2, c_1, p_2, p_1, p_0) == p_0 + (c_1 ? p_1
          : none) + (c_2 ? p_2 : none))
  }
}

field f: Int

method m(xs0: Seq[Ref], xs1: Seq[Ref], xs2: Seq[Ref], xs3:
    Seq[Ref], p: Perm) {
  inhale (forall i0: Int :: (i0 in [0..|xs0|))
    ==> acc(xs0[i0].f, write))

  // inverse function definition
  assume (forall i1: Int :: (i1 in [0..|xs1|))
    ==> inv_0(xs1[i1]) == i1)
  assume (forall r: Ref :: (inv_0(r) in [0..|xs1|))
    ==> xs1[inv_0(r)] == r)
  // transformed assume
  assume (forall r: Ref :: (inv_0(r) in [0..|xs1|))
    ==> perm(r.f) >= p)

  // inverse function definition
  assume (forall i2: Int :: (i2 in [0..|xs2|))
    ==> inv_1(xs2[i2]) == i2)
  assume (forall r: Ref :: (inv_1(r) in [0..|xs2|))
    ==> xs2[inv_1(r)] == r)
  // transformed assume
  assume (forall r: Ref :: (inv_1(r) in [0..|xs2|))
    ==> perm(r.f) >= assume_helper_1((inv_0(r) in [0..|xs1|)), p,
        p))

  // inverse function definition
  assume (forall i3: Int :: (i3 in [0..|xs3|))
    ==> inv_2(xs3[i3]) == i3)
  assume (forall r: Ref :: (inv_2(r) in [0..|xs3|))
    ==> xs3[inv_2(r)] == r)
  // transformed assume
  assume (forall r: Ref :: (inv_2(r) in [0..|xs3|))
    ==> perm(r.f) >= assume_helper_2((inv_1(r) in [0..|xs2|)),
        (inv_0(r) in [0..|xs1|)), p, p, p))
}
```

Listing 4.14: Final translation of 4.13

Chapter 5

Heap-Dependent Triggers

Viper supports universal quantifiers of the form `forall x: T :: e(x)`. Since it is obviously not feasible to instantiate this axiom for every possible `x: T` or even for every `x: T` that occurs in the program to be verified, Viper includes *triggers*. Triggers are uninterpreted expressions that are used to instantiate the axiom: if we add the trigger `t(x)` to the above axiom, then whenever an expression of the form `t(y)` is encountered, the axiom is instantiated as `e(y)`. The requirement for the trigger to be uninterpreted concerns not Silicon but the underlying SMT Solver: any expression that has a meaning to the SMT Solver, such as arithmetic expressions, comparisons, implications, conjunctions, etc. cannot be used as triggers.
At the start of this thesis, Silicon did not support the use of heap-dependent expressions such as fields, predicates and magic wands as triggers in most cases. However, for some cases, such as Listing 5.1, there is no real practical alternative to a heap-dependent trigger such as a field. In this example, the only usable triggers would be `perm(x.f)` or `x.f`, both of which were unsupported (recall that `perm(x.f)` is not a function call and can therefore not be used as trigger).

```
assume forall x: Ref :: perm(x.f) > none ==> x.f != 0
```

Listing 5.1: Quantifier with heap-based trigger

Silicon ultimately creates formulae that are discharged by an SMT Solver (Z3) and the triggers are passed down to the SMT Solver. However, for heap-dependent triggers this is a problem, since the concept of a heap does not really exist on an SMT level. For example, the way fields are represented in the SMT code varies depending on how the field is used: If a field is read (`x.f`), in the SMT code, there will be a call of a value function with the receiver as argument. This value function represents the value of fields at certain receivers. However, if we instead had `perm(x.f)`, then the value function is not used, but instead we have a sum of permission terms representing the total permission to that field with this receiver. We would, however, like both of those to trigger quantifiers that use the field `x.f` as their trigger.
Because of this a completely new function specifically for triggering was created. The function is a *location function* $loc_f(sm(x), x)$ that depends on both the current value ($sm(x)$), where sm is a value function representing the current value, and the receiver (x) (in the case of a field). There is one location function

declared for every quantified resource in the current program. In the case of fields, the type of the function is $loc_{field} : T \times Ref \rightarrow Bool$, where T is the type of the field. For predicates and magic wands, the function is $loc_{pred} : S \times Snap \rightarrow Bool$, where S is the sort corresponding to the arguments of the predicate/magic wand. The return type in both cases is a Boolean, the reason for this is that it allows to easily add the function to the path conditions when we want to create a trigger. To make sure that the triggering function does not influence anything beyond triggering, additional axioms are added to declare the triggering function to always be true: $\forall x : T, s : S \cdot loc_{id}(x, s)$ is added to the path conditions for every resource, where T and S are the types correct types as specified by loc_{id}. This allows us, for example, to use the triggering function as follows: $loc_f(sm(x), x) \Rightarrow e(x)$, where $e(x)$ is some expression that should generate this trigger.
Wherever a heap-dependent trigger is used, it is translated to this location function which is then used as the actual trigger.
The biggest challenge now was to make sure that this function triggers axioms as expected. The intended triggering is that for any resource that is part of the heap at the time the axiom is added to the state, the axiom should trigger. However, if permissions to a new resource are added later, then mentioning this resource should not trigger the axiom.

```
assume forall x: Ref :: {x.f} perm(x.f) == write ==> x.f > 0
inhale acc(y.f)
assert y.f > 0 // Fails
```

Listing 5.2: Heap-dependent trigger not triggering as intended

In Listing 5.2, we can see how only the heap at the time the axiom is added to the state is relevant. Since the permissions to `y.f` are only added later, it does not trigger the axiom, even though we clearly mention the relevant field.

```
inhale acc(y.f)
assume forall x: Ref :: {x.f} perm(x.f) == write ==> x.f > 0
assert y.f > 0 // Succeeds
```

Listing 5.3: Successful triggering using heap-dependent trigger

In Listing 5.3 on the other hand, we see when triggering succeeds. Since `y.f` is already part of the symbolic heap at the time the axiom is added to the state, it does trigger the axiom and the assertion succeeds.

To generate triggers whenever a specific resource is mentioned, we often need to summarise the heap. A heap summary is essentially a new snapshot map that is a combination of all snapshot maps of the current heap. Since snapshot maps are partial functions, they are only defined within a certain domain, i.e. they are only defined if we have permissions to a resource. The definition of the summary sm is now that for every snapshot map sm_i, on the domain of sm_i, sm is equal to sm_i, i.e. $e \in dom(sm_i) \Rightarrow sm(e) = sm_i(e)$. For example, if the heap consisted of the two quantified permission assertions `forall x: Ref :: x in xs ==> acc(x.f)` and `forall y: Ref :: y in ys ==> acc(y.f)` when we create a summary, this would mean the symbolic heap consists of two chunks with the snapshot maps sm_1 and sm_2. The new summary snapshot map sm would then be defined as follows: $\forall r : Ref \cdot r \in xs \Rightarrow sm(r) = sm_1(r) \wedge r \in$

$ys \Rightarrow sm(r) = sm_2(r)$.
The summary gives us a snapshot map that is essentially defined over the complete current heap, instead of the individual snapshot maps that are defined for each chunk. The reason we need such a summary for the triggering is that we want any part of the current heap to be able to trigger quantifiers with heap-dependent triggers. Using such a heap summary, we also manage to trigger just for the current heap (since the summary will not be defined for chunks added later), and not on any resource that is added later to the heap.
The function to create the summary can be found in Listing 5.4. This summarising is almost equivalent to the definition found in [8]. There is however the addition of one axiom to the definitional axioms for the summary (found in the highlighted lines 9-10): $\forall \overline{r : E}\ \{loc_{id}(sm(\overline{r}), \overline{r})\}\ \bigwedge_i\ loc_{id}(sm_i(\overline{r}), \overline{r})$. This axiom specifies that whenever we mention the location that corresponds to a particular summary snapshot map, we also mention the location of every single component of the summary.

```
1  qp-summarise(h, id) =
2    Let h_id ⊆ h be all chunks for identifier id
3    Let sm be a fresh snapshot map of type Ē → Snap
4    sm_def := ∅
5    perm := λ r̄ · 0
6    foreach id(r̄; sm_i(r̄), q_i(r̄)) ∈ h_id do
7      sm_def := sm_def ∪ {∀r̄ : Ē · 0 < q_i(r̄) ⇒ sm(r̄) = sm_i(r̄)}
8      perm := λr̄ · perm(r̄) + q_i(r̄)
9    loc_ax := ⋀_{i∈h_id} loc_id(sm_i(r̄), r̄)
10   sm_def := sm_def ∪ {∀r̄ : Ē {loc_id(sm(r̄), r̄)} loc_ax}
11   (sm, sm_def, perm)
12   //where Ē are the sorts corresponding to the arguments of id,
         for example Ref if id denotes a field
```

Listing 5.4: qp-summarise, the highlighted section is the newly added axiom

In this chapter, we will present the changes needed in the symbolic execution rules to support heap-dependent triggers as described above. Until the last section of this chapter, we will assume that we only deal with quantified permissions. The reason for this is that Silicon handles quantified and non-quantified permission settings quite differently. In Section 5.5 we will address how to handle non-quantified permission settings. In a quantified permission setting, assertion such as `inhale acc(x.f)` can be treated as syntactic sugar for the assertion `inhale forall r: Ref :: r == x ==> acc(r.f)`.

5.1 Evaluating Heap-dependent Triggers

The symbolic evaluation of heap-dependent triggers required some additional changes. All we need to evaluate are the arguments of whatever resource we are using as trigger. However, consider the example in Listing 5.5: the trigger in line 3 is perfectly valid, but if we just evaluate the receiver of the field access (i.e. `ys[x.f]`), we encounter multiple problems.

```
inhale forall x: Ref :: {x.f} x in xs ==> acc(x.f)
inhale forall x: Ref :: x in xs ==> x.f in [0..|ys|)
inhale forall x: Ref :: {ys[x.f].f} x in xs ==>
    acc(ys[x.f].f)
```

Listing 5.5: An example of a nested heap-dependent trigger

Firstly, we do not know if we have permission to access `x.f` and secondly, we also do not know if `x.f` is a valid index into `ys`. Both of these are the case, however only under the condition `x in xs`. At the time of trigger evaluation there is no way to know that this condition holds and therefore we get an error for this trigger even though it should be valid. However, since the trigger is a purely syntactic construct and does not contain any semantic meaning (two quantifiers which only differ in their trigger and in nothing else are equal from a logical standpoint), errors that occur during the trigger evaluation are also not a problem for the verification. Therefore, to allow any trigger to be used, the solution is to just translate the trigger without doing any checks: whether `ys[x.f]` is a valid access of an element of a sequence does not really matter inside the trigger, we simply want to trigger the quantifier whenever an expression of this sort occurs. Any expression that could trigger the quantifier of course still includes all the checks, therefore nothing is lost really, but more triggers are allowed. In Listing 5.6, we find the symbolic evaluation rule for heap-dependent triggers. It differs from the regular evaluation rule in that it evaluates an identifier instead of a pure expression.

```
1 evalTrigger: Σ → Id → (Σ → V → R) → R
2 evalTrigger(σ1, id(e(x)), Q) =
3   evalNoCheck(σ1, e(x), (λ σ2, e(x)' ·
4     Let h_id ⊆ σ2.h contain all chunks for identifier id
5     (sm, sm_def _) = qp-summarise(h_id, id)
6     π2 := pc-add(σ2.π, sm_def)
7     Q(σ2{π := π2}, loc_id(sm(e(x)'), e(x)')))
```

Listing 5.6: Symbolic evaluation of heap-dependent triggers, `evalNoCheck` is a helper function that is equivalent to `eval`, it just does not perform checks such as whether we have permissions.

5.2 Inhaling and Exhaling Permissions

When inhaling quantified permissions, Silicon internally rewrites the quantified permission assertion using newly axiomatized inverse functions (similar to section 4.3). The trigger defined by the user is only used for one of the axioms that define the inverse functions, for the other axiom the inverse function is used as the trigger. Therefore, when inhaling quantified permissions using a heap-dependent trigger such as in Listing 5.5, we only need to change the trigger for one inverse function axiom. When inhaling permissions, we create a new chunk that represents the new heap resources that we now gained permissions to. Every chunk includes a snapshot map that represents the values of this chunk. The trigger for the inhaled permissions is now translated to $loc_{id}(sm(\overline{e}), \overline{e})$, where sm is the snapshot corresponding to the newly created chunk, id is the resource

for which we inhaled permissions and $\overline{e}$ are the arguments used for the resource to which we inhale permissions. The reason we use the snapshot map of the new chunk and not a summary snapshot map abstracting over the complete heap is that this would only add unnecessary instantiations of the axioms. The axiom where the trigger is used is inv_2 in Listing 5.7. This axiom is an implication whose left-hand-side is true if and only if this chunk provides permissions. Since the domain of the snapshot map belonging to a chunk is all values for which the chunk provides permissions, it is enough to only trigger using this snapshot map.

Listing 5.7 shows the symbolic execution rule for inhaling quantified permissions using a heap-dependent trigger with the changes highlighted. The rule is slightly simplified from the version found in [8], however nothing additional found in the original rule is relevant to the changes made here. In addition to the changes to the trigger of the quantified permission assertion respectively the inverse function defined by it, this rule also includes a new axiom in line 12. This axiom generates a trigger for all locations to which this quantified permission assertion provides permissions.

```
1  produce: Σ → A → Snap → (Σ → R) → R
2  produce(σ1, forall x: T :: {id(e1(x))} c(x) ==> acc(id(e(x)),
       p(x)), sm, Q) =
3    eval(σ1, c(x), (λ σ2, c(x)' ·
4      eval(σ2{π := σ2.π ∪ c(x)'}, e(x) (λ σ3, e(x)' ·
5        eval(σ3, p(x), (λ σ5, p(x)' ·
6          evalTrigger(σ5, id(e1(x)), (λ σ6, loc_id(sm1(e1(x)'), e1(x)') ·
7            Let e^-1 be a fresh function of type E → T
8            inv1 := ∀ r : E · {e^-1(r)} c(e^-1(r))' ∧ 0 < p(e^-1(r))' ⇒ ⋀
                 e_i(e^-1(r))' = r_i
9            inv2 := ∀x : T· {loc_id(sm(e1(x)'), e1(x)')} c(x)' ∧ 0 < p(x)'
                 ⇒ e^-1(e(x)') = x
10           ch := id(r; sm(r), ite(c(e^-1(r))', p(e^-1(r)), 0))
11           Let h_id ⊆ σ2.h be all chunks for identifier id
12           (sm', sm_def, _) = qp-summarise(h_id, id)
13           loc_ax := ∀r : E· {e^-1(r)} c(e^-1)' ⇒ loc_id(sm'(r), r)
14           h3 := σ2.h ∪ {ch}
15           π3 := pc-add(σ2.π, {inv1, inv2, sm_def, loc_ax})
16           Q(σ2{h := h3, π := π3}))))))))))
```

Listing 5.7: Inhaling with heap-dependent trigger

When exhaling permissions, we need to make sure to generate triggers for the permissions that we are exhaling. This is required to make sure we can trigger quantified permission assertions that use a heap-dependent trigger which were inhaled earlier. In the case of quantified permissions, the triggering function is used twice: first we need it for the injectivity check. The reason it is required is that the originally inhaled quantified permissions need to be triggered to be able to assert injectivity. Additionally, we also add the axiom $\forall \overline{r}$ {$e^{-1}(\overline{r})$} c($e^{-1}(\overline{r})$)' ⇒ loc_id(sm($\overline{r}$), $\overline{r}$). Recall that a quantified permission assertion is of the form $\forall \overline{x : T} \cdot c(x) \Rightarrow \texttt{acc}(id(\overline{e(x)}), p(x))$. If we replace x with $e^{-1}(\overline{r})$ in this assertion, we can clearly see the similarity between the two axioms. The difference between them is that the right hand side of the implication of one axiom is an accessibility predicate, whereas for the other axiom, it

is the location function corresponding to the same resource. The axiom that we are adding essentially generates a trigger for every location that is included in this quantified permission assertion. The `qp-remove` function used in line 10 is an auxiliary function that removes permissions from quantified chunks if they refer to the right resource.

```
1  consume: Σ → A → (Σ → Snap → R) → R
2  consume(σ1, h, forall x: T :: c(x) ==> acc(id(e(x)), p(x)), Q) =
3    Proceed as above to obtain c(x)',e(x)' and p(x)', let σ2 be
       the post-state
4    (sm,sm_def, _) := qp-summarise(h,id)
5    Let y1,y2 be fresh symbolic constants of type T
6    assert(σ2.π, c(y1)' ∧ c(y2)' ∧ loc_id(sm(e(y1)), e(y1)) ∧
       loc_id(sm(e(y2)), e(y2)) ∧ 0 < p(y1) ∧ 0 < p(y2) ∧
       e(y1)' = e(y2)') ⇒ y1 = y2)
7    Introduce a fresh function e^-1 and axioms inv1 and inv2 as
       above
8    locAx := ∀r̄ {e^-1(r̄)} c(e^-1(r̄))' ⇒ loc_id(sm(r̄), r̄)
9    π2 := pc-add(σ2.π, locAx)
10   h3 := qp-remove(σ2.π, h, id, (λ r̄ · ite(c(e^-1(r̄)), p(e^-1(r̄)),
       0)))
11   π3 := pc-add(σ2.π, {inv1, inv2, sm_def})
12   Q(σ2{π := π3}, h3, sm)
```

Listing 5.8: Exhaling with heap-dependent triggers

5.3 Generating Triggers from Statements and Expressions

To correctly trigger quantified axioms, we need to generate the triggering function whenever a particular resource is mentioned. To do this, we need to summarise the heap in order to make sure that we can trigger quantifiers no matter what snapshot map was used to create the trigger function.

5.3.1 Field Read

When reading a field, we need to generate a trigger instance for the field we are reading, i.e. summarize the heap to get a snapshot map for all relevant chunks, evaluate the receiver and then add the generated trigger instance to the path conditions. The corresponding evaluation rule can be seen in Listing 5.9.

```
1  eval: Σ → E → (Σ → V → R) → R
2  eval(σ1, e.f, Q) =
3    eval(σ1, e, (λ σ2 e′ ·
4      Let hf ⊆ σ2.h be all chunks for identifier f
5      (sm, smdef, _) = qp-summarise(hf, f)
6      locax := locf(sm(e′), e′)
7      π2 := pc-add(σ2.π, {smdef, locax})
8      perm := ∑ch(_,_,p)∈hf p
9      assert(π2, 0 < perm(e′))
10     Q(σ2{π := π2}, sm(e′)))))
```

Listing 5.9: Evaluation rule for field reads

5.3.2 Field Write

In contrast to field reads, for field writes we need to generate the trigger twice: once in the state before the assignment and once in the state after the assignment. This makes sure that we can trigger both quantifiers added to the state before the assignment and quantifiers that were added to the state after the assignment. Listing 5.10 shows the evaluation rule to achieve this.

```
1  exec: Σ → S → (Σ → R) → R
2  exec(σ1, e.f := v, Q) =
3    eval(σ1, e, (λ σ2 e′ ·
4      eval(σ2, v, (λ σ3 v′ ·
5        Let hf1 ⊆ σ3.h be all chunks for identifier f
6        (sm1, smdef1, _) = qp-summarise(hf1, f)
7        locax1 := locf(sm1(e′), e′)
8        π3 := pc-add(σ3.π, {sm1def, locax1})
9        consume(σ3{π := π3}, acc(e′.f, 1), (λ σ4, _ ·
10         produce(σ4, acc(e′.f, 1) && e′.f == v′, (λ σ5 ·
11           Let hf2 ⊆ σ5.h be all chunks for identifier f
12           (sm2, sm2def, _) = qp-summarise(hf2, f)
13           locax2 := locf(sm2(e′), e′)
14           π5 := pc-add(σ5.π, {sm2def, locax2})
15           Q(σ5{π := π5}))))))))))
```

Listing 5.10: Execution rule for field writes

5.3.3 Perm

Perm is very similar to a field read, we just need to generate the trigger for whatever resource is the argument to the perm expression. Listing 5.11 shows the updated rule that includes trigger generation.

```
1 eval: Σ → E → (Σ → V → R) → R
2 eval(σ1, perm(id(ē)), Q) =
3   eval(σ1, ē, (λ σ2, ē' ·
4     Let h_id ⊆ σ2.h contain all heap chunks for identifier id
5     (sm, sm_def, _) = qp-summarise(h_id, id)
6     loc_ax := loc_id(sm(ē), ē)
7     π2 := pc-add(σ2.π, {sm_def, loc_ax})
8     sum := foldl(h_id, 0, (λ id(v̄;_,p), q · q + ite(⋀ ē' = v̄, p,
          0)))
9     Q(σ2{π := π2}, sum))
```

Listing 5.11: Evaluation rule for **perm**

5.3.4 Forperm

Since forperm already iterates over all chunks that provide access to the resource we are interested in, we do not need to create a summary, but can instead create a trigger instance using the snapshot map of each chunk that we are iterating over. Listing 5.12 shows the corresponding updated evaluation rule.

```
1 eval: Σ → E → (Σ → V → R) → R
2 eval(σ1, forperm x̄:T [id(x̄,ē)] :: b(x̄), Q) =
3   eval(σ1, ē, (λ σ2, ē' ·
4     Let h_id ⊆ σ2.h contain all heap chunks with identifier id
5     Let z̄:T be fresh program variables s.t. |z̄| = |x̄|,
6       and let z̄' be corresponding fresh symbolic values
7     loc_ax := true
8     cnj(z̄) := foldl(h_id, true, (λ id(v̄;_,p(v̄)),c ·
9       loc_ax := loc_ax ∧ loc_id(sm(z̄,y'), z̄,y')
10      Let (v̄1 :: v̄2) = v̄ be the arguments corresponding
11        to x̄ and ē respectively
12      c ∧ (p(z̄',ē') > none && ⋀ z̄' = v̄1 && ⋀ ē' = v̄2 ==> b(z̄)))
13    π2 := pc-add(σ2.π, loc_ax)
14    eval(σ2 {γ := σ2.γ[z̄ ↦ z̄'], π := π2}, cnj(z̄), (λ σ3, cnj(z̄)' ·
15      Q(σ3{γ := σ2.γ}, ∀ x̄:T · cnj(x̄)'))
```

Listing 5.12: Evaluation rule for **forperm**

5.3.5 Fold

When folding a new predicate, we create a trigger instance using the arguments of the predicate and a heap summary for this predicate that includes the newly folded predicate. In the symbolic execution rule in Listing 5.13, $pred_{body}$ is used to represent the body of the predicate.

```
1  exec: Σ → S → (Σ → R) → R
2  exec(σ1, fold acc(pred(ē), p), Q) =
3    eval(σ1, ē, (λ σ2 ē′ ·
4      eval(σ2, p, (λ σ3 p′ ·
5        assert(σ3.π, 0 ≤ p′)
6        consume(σ3, pred_body, (λ σ4 ·
7          produce(σ4, acc(pred(ē′), p′), (λ σ5 ·
8            Let h_pred ⊆ σ5.h be all chunks for identifier pred
9            (sm, sm_def, _) = qp-summarise(h_pred, pred)
10           loc_ax := loc_pred(sm(ē′), ē′)
11           π5 := pc-add(σ5.π, {sm_def, loc_ax})
12           Q(σ5{π := π5}))))))))))
```

Listing 5.13: Execution rule for **`fold`**

5.3.6 Unfold

When unfolding a predicate, we need to create the trigger in the previous state, i.e. the unfolded predicate is still included in the summary that is used to generate the trigger instance. Listing 5.14 shows the corresponding symbolic execution rule.

```
1  exec: Σ → S → (Σ → R) → R
2  exec(σ1, unfold acc(pred(ē), p), Q) =
3    eval(σ1, ē, (λ σ2 ē′ ·
4      eval(σ2, p, (λ σ3 p′ ·
5        Let h_pred ⊆ σ3.h be all chunks for identifier pred
6        (sm, sm_def, _) = qp-summarise(h_pred, pred)
7        loc_ax := loc_pred(sm(ē′), ē′)
8        π3 := pc-add(σ3.π, {sm_def, loc_ax})
9        assert(π3, 0 ≤ p′)
10       consume(σ3{π := π3}, acc(pred(ē′), p′), (λ σ4 ·
11         produce(σ4, pred_body, (λ σ5 ·
12           Q(σ5))))))))))
```

Listing 5.14: Execution rule for **`unfold`**

5.3.7 Package

Packaging a magic wand works pretty much the same as folding a predicate, we generate a trigger instance using a heap summary that includes the newly packaged magic wand. $(args(A --* B)$ is used here to refer to the arguments of the magic wand. `consume-ext` is a special rule for packaging a magic wand. Basically, it takes permissions from the current heap only if they are not provided by the a state satisfying the left-hand side of the wand. For the heap-dependent triggers, the rule is not very important, but we need to make sure to create triggers afterwards.

```
1  exec: Σ → S → (Σ → R) → R
2  exec(σ1, package A --* B, Q) =
3    produce(σ1{h := ∅}, A, (λ σlhs ·
4      consume-ext([σlhs.h, σ1.h], σlhs{h := ∅}, B, (λ [_, h2], _, _, ·
5        Let ch be a magic wand chunk corresponding to A --* B
6        Let wand be the corresponding identifier for A --* B
7        Let hwand ⊆ (h2 ∪ ch) be all chunks for identifier wand
8        (sm, smdef, _) = qp-summarise(hwand, wand)
9        locax := locwand(sm(args(A --* B)), (args(A --* B))
10       π1 := pc-add(σ1.π, {smdef, locax})
11       Q(σ1{h := heap-add(h2,ch), π := π1}))))))
```

Listing 5.15: Execution rule for **`package`**

5.3.8 Apply

Apply is closely related to unfold and creating triggers for it works similarly, the trigger is generated before actually applying the wand, i.e. when the corresponding wand chunk still exists. Similarly to above, $(args(A --* B)$ refers to the arguments of the magic wand.

```
1  exec: Σ → S → (Σ → R) → R
2  exec(σ1, apply A --* B, Q) =
3    Let idwand(e') be a magic wand chunk identifier corresponding
         to A --* B
4    consume(σ1, A, (λ σ2, _ ·
5      Let hid ⊆ σ2.h be all chunks for identifier id
6      (sm, smdef, _) = qp-summarise(hid, id)
7      locax := locid(sm((args(A --* B)), (args(A --* B))
8      π2 := pc-add(σ2.π, {smdef, locax})
9      consume(σ2{π := π2}, acc(idwand(e')), (λ σ3, _ ·
10       produce(σ3, B, fresh, Q)))))
```

Listing 5.16: Execution rule for **`apply`**

5.3.9 Unfolding, Applying

Both **`unfolding`** and **`applying`** do not need additional cases since they **`unfold`**/**`apply`** temporarily, so the rules presented above cover these cases as well.

5.3.10 Assert

assert also does not need any changes: **assert** essentially temporarily exhales permissions, but does not update the heap (i.e. no chunks are removed). Since we already have triggering for exhales, this works for asserts as well.

5.3.11 Assume

assume does not need any changes either: in the case of a pure **assume**, the evaluation rules already cover any required triggers (field reads, **perm**, **forperm**). Since any non-pure assume statement is translated to a pure version, these also do not require any changes. If a different approach for the assume statements were taken, non-pure assumes would need to generate triggers as well.

5.4 Pure Quantified Assertions

When Silicon evaluates quantified expressions, there might be new path conditions generated for the quantified variable. For example, consider the following:

```
forall x: Ref :: x in xs ==> x.f > 0
```

Listing 5.17: Pure quantifier

To symbolically evaluate the body, Silicon instantiates the quantified variable `x` as some arbitrary `x` and evaluates the body using this assignment. During the symbolic evaluation, we add path conditions according to the evaluation rules given above. In particular, when evaluating `x.f` we add $loc_f(sm(\mathtt{x}), \mathtt{x})$ as a new path condition, where sm is a new summarising snapshot map. However, what we actually need is a quantified path condition since this `x` does not really have meaning. We would want $\forall$ `x: Ref :: x` $\in$ `xs =>` $loc_f(sm(\mathtt{x}), \mathtt{x})$ as the new path condition. In order to solve this problem, Silicon creates a second, *auxiliary quantifier* after the evaluation of the body. This quantifier contains all new path condition that are gathered during the evaluation of the body and quantifies over the same variable as the main quantifier. Importantly, the auxiliary quantifier also uses the same trigger as the the main quantifier. Consider the example in Listing 5.18: here, we first **assume** and then **assert** a pure quantifier, both times using `x.f` as trigger.

```
inhale forall x: Ref :: {x.f} x in xs ==> acc(x.f)
assume forall x: Ref :: {x.f} x in xs ==> x.f > 0
assert forall x: Ref :: {x.f} x in xs ==> x.f > 0
```

Listing 5.18: Pure quantified assertions using heap-dependent triggers

The term that is generated in order to trigger quantifiers using `x.f` as their trigger ends up in the auxiliary trigger that is generated by Silicon. In particular, this means that the main trigger that is generated for the **assert** statement does not contain the trigger expression. This is a problem, because in the case of an **assert**, the main and auxiliary quantifier are actually treated differently: on the SMT level, the auxiliary quantifier is assumed, whereas the main quantifier is asserted. What this means is that the auxiliary quantifier is added as an additional condition and is only instantiated if its trigger is encountered. The

main quantifier, on the other hand, gets instantiated with some arbitrary symbolic value as the quantified variable and the SMT Solver then checks whether the assertion holds. However, because in the case of Listing 5.18, the auxiliary quantifier is uses `x.f` as its trigger, it will not be triggered by the main quantifier. This also means that the main quantifier cannot trigger any other earlier quantifiers that used `x.f` as their trigger. Listing 5.19 illustrates the problem that is occurring here. In Listing 5.19, `e(x)` represents the symbolic expression corresponding to a Viper expression that contains a resource access. In Listing 5.18, this would be the expression `x in xs ==> x.f > 0`. `aux` represents all the auxiliary terms that are gathered during the symbolic execution of the aforementioned expression. In particular, only `aux` contains the term that is used as trigger for all these quantifiers.

```
1 assume ∀ x :: {loc_f(sm(x), x)} aux
2 assume ∀ x :: {loc_f(sm(x), x)} e(x)
3 assume ∀ x :: {loc_f(sm(x), x)} aux
4 assert ∀ x :: {loc_f(sm(x), x)} e(x)
```

Listing 5.19: Regular evaluation of the pure quantifiers in 5.18

5.4.1 Rewriting Quantifiers

One possible solution to this problem is to make sure that the main quantifier contains the triggering expression. The idea is to add the trigger function as close to the field read as possible. This requires saving all the evaluated receivers of field reads and **perm** expressions that are encountered, because we need to generate the trigger function using these receivers. An example of this can be seen in Listing 5.20. Here, we inserted an additional implication to make sure the trigger is available when evaluating `x.f`. To keep the rewritten quantifier as close to the original quantifier as possible, we need to find the innermost expression that is both boolean and contains a resource access. This expression is then rewritten to be the right-hand side of a new implication, where the left-hand side is the trigger that we generate.
This solution does work as intended, however, it is not a very nice solution, because it goes against the general design philosophy used in Silicon. We are trying to keep symbolic evaluation local, for the evaluation of an expression it should not matter what expressions are nested under it. A better solution would be local to the evaluation of the field read or **perm** expression and would not need the rewriting of the whole quantifier.

```
inhale forall x: Ref :: {x.f} x in xs ==> acc(x.f)
assume forall x: Ref :: {x.f} x in xs ==> loc_f(sm1(x), x)
    ==> x.f > 0
assert forall x: Ref :: {x.f} x in xs ==> loc_f(sm1(x), x)
    ==> x.f > 0
```

Listing 5.20: Updated pure quantifier containing the trigger

5.4.2 Localized Triggering

The approach described in the previous section works, however it is not very localized. Evaluated terms should not have to be manually rewritten later. An

alternative, more localized approach is to make sure that any pure mentioning of a resource is translated to a function that can be used to trigger other functions. For field reads, this was already the case: every field read is translated into a lookup using a snapshot map. To make sure that field reads contained in pure quantified assertions can be used to create heap-dependent triggers, the additional triggering axiom in the summary function needed to be changed. Instead of triggering on $loc_{id}(sm(\overline{e}), \overline{e})$ it is weakened to trigger on $sm(\overline{e})$, as can be seen in the updated Listing 5.21. This will still trigger for any terms that triggered it before, since every $loc_{id}(sm(\overline{e(r)}), \overline{r})$ of course contains the subterm $sm(\overline{r})$. Additionally, any lookup using a summarising snapshot map will now also trigger this axiom. Since field reads are evaluated using a summarising snapshot map they will trigger this axiom, allowing us to verify Listing 5.18.

```
1  qp-summarise(h, id) =
2    Let h_id ⊆ h be all chunks for identifier id
3    Let sm be a fresh snapshot map of type E⃗ → Snap
4    sm_def := ∅
5    perm := λ r⃗ · 0
6    foreach id(r⃗; sm_i(r⃗), q_i(r⃗)) ∈ h_id do
7      sm_def := sm_def ∪ {∀r⃗ : E⃗ · 0 < q_i(r⃗) ⇒ sm(r⃗) = sm_i(r⃗)}
8      perm := λr⃗ · perm(r⃗) + q_i(r⃗)
9    loc_ax := ⋀_{i∈h_id} loc_id(sm_i(r⃗), r⃗)
10   sm_def := sm_def ∪ {∀r⃗ : E⃗ {sm(r⃗)} loc_ax}
11   (sm, sm_def, perm)
12   //where E⃗ are the sorts corresponding to the arguments of id,
         for example Ref if id denotes a field
```

Listing 5.21: qp-summarise, with weakened trigger

perm expressions, on the other hand, were not translated to any function application, but instead to a sum. This can of course not be used as a trigger. To create a suitable function for **perm**, the symbolic evaluation of **perm** functions was changed to include a summary similar to the summary used for heap values. This summary essentially just wraps the sum that was previously used for **perm** expressions in a function application. The summary axiom inside the perm-summarise function allows us to create triggers from an application of the perm snapshot map.

```
1  perm-summarise(h,id) =
2    Let h_id ⊆ h contain all chunks for identifier id
3    Let pm be a fresh perm snapshot map of type E⃗ → Perm
4    perm(r⃗) := foldl(h_id, 0, (λ id(r⃗;sm(r⃗),q(r⃗), p ·
5      p + q(r⃗)))
6    locAx(r⃗) := foldl(h_id, 0, (λ id(r⃗;sm(r⃗),q(r⃗), c ·
7      c && loc_id(sm(r⃗),r⃗)))
8    pm_def := ∀r⃗ : E⃗ {pm(r⃗)} perm(r⃗)
9    summaryAx := ∀r⃗ : E⃗ {pm(r⃗)} locAx(r⃗)
10   (pm, {pm_def, summaryAx})
```

Listing 5.22: perm-summarise function

In Listing 5.23, we see the new updated evaluation rule for **perm** using this new summary function.

```
1 eval(σ1, perm(id(ē)), Q) =
2   eval(σ1, ē, (λ σ2, ē′ ·
3     Let h_id ⊆ σ2.h contain all heap chunks for identifier id
4     (sm, sm_def, _) = qp-summarise(h_id, id)
5     assume sm_def
6     assume loc_id(sm(ē), ē)
7     (pm, pm_def) = perm-summarise(h_id, id)
8     π2 := pc-add(σ.2.π, pm_def)
9     Q(σ2{π := π2}, pm(ē′))
```

Listing 5.23: Evaluation rule for **perm**

5.5 Non-quantified permissions

Heap-dependent triggers as described in this chapter only work in Silicon if used in conjunction with quantified permissions. If we only use non-quantified permissions, Silicon does not create snapshot maps, which makes it difficult to use the trigger function. Essentially, many of the supporting functions used in this chapter would need to be recreated for the non-quantified algorithms of Silicon.
To allow heap-dependent triggers to be used even if we do not use quantified permissions, a simple remedy was used: we simply switch to using quantified algorithms when using heap-dependent triggers, the same way Silicon switches if there is a quantified permission assertion. To the user, this does not really make a difference, and in the future, Silicon might possibly completely switch to only using quantified algorithms.

5.5.1 Using non-quantified algorithms

If we want to use heap-dependent triggers while using the non-quantified algorithms, a different approach would have to be taken. A possibility is to essentially make a summary function for non-quantified algorithms. This summary would look something like the following:

$$sm(r) = \begin{cases} t_1 & \text{if } r = v_1 \\ t_2 & \text{if } r = v_2 \end{cases}$$

Here, the heap would consist of two relevant chunks, the receivers of these chunks are v_1 respectively v_2 and their symbolic values are t_1 respectively t_2. Such a summary would allow to use the same triggers also for non-quantified algorithms, because once again we have the same basic functionality.

Chapter 6

Evaluation

To test the performance impact of the new support for heap-dependent triggers in Silicon, multiple tests were performed. The tests were performed using the Viper runner [12] which in turn uses the Nailgun [5] software as an attempt to make the tests independent of warm-up time of the Java Virtual Machine.
To evaluate the performance impact of the support for heap-dependent triggers, the Viper test suite was used. This is a collection of mostly relatively simple tests that covers the whole variety of features the Viper language has to offer. In total the Viper test suite contains 1016 different tests that were run.
To lessen the performance impact of the heap-dependent trigger support, caching was additionally added to any summary snapshot maps. If the relevant part of the heap is still the same, we can reuse cached summaries and save re-doing the summary every time. The performance of Silicon before and after the addition heap-dependent trigger support, as well as both with and without caching is listed in Table 6.1.

	Silicon old, caching	**Silicon old, no caching**	**Silicon new, caching**	**Silicon new, no caching**
Average runtime [s]	0.623	0.702	0.745	0.870
Standard deviation [s]	2.226	2.787	2.656	3.156
Full runtime of the test suite [s]	633.223	713.736	756.978	884.245
Maximum runtime for a single test [s]	39.089	44.858	39.187	40.893

Table 6.1: Runtime comparison of Silicon with and without caching, as well as before and after the addition of the new features

The fact that Silicon becomes slower with the addition of the support for heap-dependent triggers was almost inevitable since for every statement and expression mentioning a resource, we need to summarise the heap. However, as can be seen, the addition of caching definitely had an impact. The caching significantly improved the performance, although there still is a performance loss compared to the previous version.

Chapter 7

Conclusion

The goal of this thesis was to make Viper's and Silicon's features more complete with regards to resources and permissions in particular. I achieved this by making **perm**, **forperm** and **assume** work with all resources and allowing resources to be used as triggers for quantifiers.
The permission introspection features of Viper now work with all resources that Viper currently supports and if there ever are additional resources added in the future, extending the permission introspection to support these resources should be very simple, because an additional case in Silicon's evaluation is all that is needed. Viper should not need any additional changes.
assume now works with any assertion, bringing it in line with **inhale**, **exhale** and **assert**. We expect this to make it easier to use these statements and that this will avoid user confusion in the future.
Any Viper resource can now also be used as a trigger in a quantified expression. This allows to verify some new quantified expressions that Silicon was unable to verify before. For example, we now can use `x.f` as a trigger for **forall** `x: Ref :: perm(x.f) > none ==> x.f != 0`. Silicon's performance is slightly worse because of this addition, however by adding caching we managed to reduce the performance impact by a significant margin. Additionally, seven Silicon issues related to heap-dependent triggers were also resolved as part of my thesis.

Chapter 8

Future Work

Implement the newly extended permission introspection features in Carbon: currently, Carbon still only support the old versions of `perm` and `forperm`. It would of course be desirable to have the same features supported by both verifiers.

Switch to generating inverse functions in Viper: there is a lot of duplicate functionality concerning inverse functions. Both Silicon and Carbon were already generating inverse functions, because they are necessary to support quantified permissions. To support the rewriting of `assume` statements, inverse functions can now also be generated in Silver. By relying on inverse functions from Viper directly, the duplicate functionality could be reduced.

Add injectivity checks to the Viper inverse functions: for `assume` statements, it was not necessary to check the injectivity of quantified permissions, however for these inverse functions to be usable in all cases, the injectivity should also be checked in Silver.

Implement heap-dependent triggers in Carbon: Carbon currently only has limited support for heap-dependent triggers. Both predicates and fields can be used as triggers, however, axioms using these only trigger on field reads, respectively `fold` and `unfold` assertions. `perm`, `forperm`, `inhale`, `exhale`, `assert` and field writes all do not trigger quantifiers using fields or predicates as quantifier. Magic wands cannot be used as triggers at all in Carbon. Adding support for all these heap-dependent triggers would be desirable to have these features available in both verifiers.

Investigate potential to bring Silicon's performance to previous levels: The addition of support for heap-dependent triggers cost Silicon some valuable performance. Bringing the performance back on par with previous levels would be very useful.

Bibliography

[1] Charles Antony Richard Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[2] James C King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[3] P. Müller, M. Schwerhoff, and A. J. Summers. "Automatic Verification of Iterated Separating Conjunctions using Symbolic Execution". In: *Computer Aided Verification (CAV)*. Ed. by S. Chaudhuri and A. Farzan. Vol. 9779. LNCS. Springer-Verlag, 2016, pp. 405–425.

[4] P. Müller, M. Schwerhoff, and A. J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by B. Jobstmann and K. R. M. Leino. Vol. 9583. LNCS. Springer-Verlag, 2016, pp. 41–62.

[5] Nailgun. URL: `http://www.martiansoftware.com/nailgun/`.

[6] M. J. Parkinson and A. J. Summers. "The Relationship Between Separation Logic and Implicit Dynamic Frames". In: *Logical Methods in Computer Science* 8.3:01 (2012), pp. 1–54.

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

[8] M. Schwerhoff. "Advancing Automated, Permission-Based Program Verification Using Symbolic Execution". PhD thesis. ETH Zurich, 2016.

[9] Jan Smans, Bart Jacobs, and Frank Piessens. "Implicit dynamic frames: Combining dynamic frames and separation logic". In: *European Conference on Object-Oriented Programming*. Springer. 2009, pp. 148–172.

[10] A. J. Summers and P. Müller. "Automating Deductive Verification for Weak-Memory Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. LNCS. To appear. Springer-Verlag, 2018.

[11] Viper Project. URL: `viper.ethz.ch`.

[12] Viper Runner. URL: `https://bitbucket.org/viperproject/viper-runner/`.

[13] Z3 SMT Solver. URL: `https://github.com/Z3Prover/z3`.

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):

Advancing Non-Standard Permission Utilisation in Program Verification

Authored by (in block letters):

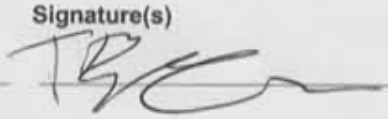
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Brodmann	Tobias

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)**
Zürich, 12.09.2018	

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