Chair of Programming Methodology
Department of Computer Science
ETH Zürich

Towards Customizability of a Symbolic-Execution-Based Program Verifier

Bachelor's Thesis

Robin Sierra

supervised by

Vytautas Astrauskas, Malte Schwerhoff

7th September 2017

Contents

Chapter 1

Introduction

Viper [1] is a verification infrastructure that is being developed at ETH Zurich and enables automatic reasoning about properties of a wide variety of concurrent and heap-manipulating programs. A core concept of Viper is the notion of *permissions* to control access to resources such as fields (e.g. `x.f`), recursive predicates and magic wands, all of which can be used to control access to complex data structures and enforce an orderly modification thereof.

To achieve automated verification of the aforementioned properties, Viper includes Silicon [2], a verifier that is based on a technique called *symbolic execution*. Silicon internally manages the resources as sets of *chunks* that contain the permission to read/write the represented resource and a symbolic value, e.g. for the value of a heap location.

The problem with Silicon is, that the handling of chunks is closely coupled to the properties of the respective resource. The goal of this thesis is to change that. Additionally, chunk handling is made more independent of the specific chunk type used. Together, this should enable easier customization of any of the three parts (chunk handling, resource properties, chunk implementation), without interfering with the other two. This is achieved by

1. classifying resources and chunks, and describing their handling in a formal way (chapter 3),
2. developing an approach that enables separate customization of the aforementioned components (chapter 4), and
3. implementing some customizations to enrich Silicon (chapter 6).

Additionally, the approach is evaluated in chapter 8 and a conclusion is drawn in chapter 9.

Chapter 2

Background

We will introduce the core concepts of the Viper language and their handling in Silicon.

2.1 Viper

Viper supports three resource types: fields, predicates, and magic wands. The resources and the operations they allow are explained in the following sections. For a full description of the Viper syntax please refer to [2].

2.1.1 Permissions

A core concept of Viper is the notion of permissions to control access to heap locations. Permissions are modeled by a rational number, where 0 denotes no permission, 1 denotes write permission and everything in between are read permissions.

2.1.2 Fields

One type of heap locations to which one can have permissions, i.e. one resource type, are *fields*. Just as in programming languages, fields are accessed through a receiver and a field name, e.g. `x.f`, where `x` is a reference, and `f` is the name of the field. Let us look at an example showing how fields are used in Viper:

```
field f: Int

method writeToField(x: Ref)
{
  inhale acc(x.f, write)
  x.f := 1
  exhale acc(x.f, 1/2)
```

```
8   assert x.f == 1
9 }
```

Listing 2.1: Field example

In the first line a field `f` of type `Int` is declared. Then, in line 3, the method `writeToField` is defined. It takes a reference `x` as an argument. Since one can only write to a heap location to which one has write permissions, one first has to gain the permissions. This is done in line 5 by *inhaling* access to `x.f` with a permission amount of `write`, i.e. `1`. After gaining enough permissions, one can write a value to the heap location (line 6). The opposite of gaining permissions is losing permissions, called *exhaling*. In line 7, a permission amount of `1/2` to `x.f` is given up, i.e. exhaled. At that point, one still has read permissions, therefore (in line 8) one can assert that the value of `x.f` is indeed equal to `1`.

Write access to `x.f`, written as `acc(x.f, write)` can also be shortened to `acc(x.f)`.

2.1.3 Predicates

In addition to fields that describe single heap locations, there is also the possibility to specify potentially unbounded data structures through *recursive predicates*. Predicates can take arguments and provide access to heap locations in their body. A singly linked list, for example, is declared as follows:

```
1 field next: Ref
2 field value: Int
3
4 predicate list(head: Ref) {
5   acc(head.value) &&
6   (head.next != null ==> acc(list(head.next)))
7 }
```

Listing 2.2: Predicate declaration

First two fields are declared, one for the next node in the linked list, and one for the value of a node. With that, one can construct a list by defining a predicate. It takes a reference as an argument, and in its body, it provides write access to the nodes value `head.value` by using an accessibility predicate `acc(head.value)`. Again, if needed, a permission amount could be specified, e.g. `acc(head.value, 1/2)`, if only read permission is desired. Additionally, the predicate provides access to the rest of the list starting at

`head.next`, if it is non-null. Just as for fields, access to predicates can be inhaled and exhaled:

```
1 inhale acc(list(head))
2 exhale acc(list(head))
3 inhale list(head)
4 exhale acc(list(head), 1/2)
```

Listing 2.3: Predicate usage

Note that line 3 is a short notation for line 1. As for fields, a permission amount can be specified (line 4).

When inhaling a predicate, one does not automatically get access to all the permissions in the body. Instead, the user has to specify at which point a predicate should be replaced by its body. This operation is called *unfolding*. We again use the predicate shown in listing 2.2.

```
inhale list(head) // same as: inhale acc(list(x))
// exhale acc(head.value) would fail at this point
unfold list(head) // replace list(head) by its body
// exhale list(head) would fail at this point
exhale acc(head.value)
```

Listing 2.4: Unfolding a predicate

The reverse operation is called *folding* and uses the `fold` keyword.

2.1.4 Magic Wands

A *magic wand* is a binary operator that encodes the promise, that in a state where the left-hand side holds, the left-hand side can be replaced by the right-hand side. In Viper, this is written as

```
A --* B
```

Listing 2.5: Magic wand

where A and B are some expressions. Magic wands can, for example, be used to specify the traversal of partial data structures.

2.1.5 Working Example

A complete, working example looks as follows:

```
field next: Ref
field value: Int

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

method writeHeadValue(head: Ref)
  requires list(head)
  ensures head.value == 31415
  ensures list(head)
{
  unfold list(head) // get perm to write head.value
  head.value := 31415
  fold list(head) // get back list(head)
}
```

Listing 2.6: Viper example

The same field and predicate definitions as before are used; additionally, there is a *method* which takes a reference to the head of a list as a parameter and changes the value of the head of the list. A method can take preconditions (in this case that one must have access to the list) and postconditions (that ensures that the head's value is `31415` and that we return access to the list).

2.2 Silicon

Silicon uses a symbolic heap to represent an actual heap during program execution. Additionally, it uses path conditions to represent assumptions made during execution, which are then given to the underlying SMT solver (which we will call *prover* in this thesis).

2.2.1 Heap

The symbolic heap consists of *chunks*, which essentially are the heap locations in the program. Chunks also contain a permission amount. There are different types of chunks for different types of locations:

- Fields: Field chunks consist of the field name, the receiver, the field's value and a permission amount in the interval $[0, 1]$.

- Predicates: Predicate chunks consist of the predicate name, the arguments, a snapshot and a permission amount. The snapshot represents the heap values the predicate abstracts over. Unlike for fields, for predicates there is no upper bound for permissions, i.e. it is possible to have a permission amount of 2 to a predicate. Since Silicon treats predicates as opaque resources, meaning that locations inside a predicate are hidden, predicates can be seen as *abstract heap locations* similar to fields.

- Magic Wands: Magic wand chunks have the same shape as predicate chunks except they are identified by their structure, instead of their name. This is explained in more depth in section 2.2.3.

A heap can contain multiple chunks for the same heap location.

2.2.2 Chunk Handling

To show how chunks are handled in Silicon, we present an example:

```
1 inhale acc(x.f, 1/2)
2 inhale acc(y.f, 1/2)
3 inhale acc(z.f)
4
5 assume x == y
6
7 exhale acc(x.f)
```

Listing 2.7: Chunk handling example

In the first line, access to the field `x.f` is inhaled. In Silicon, a new field chunk with field name `f`, receiver `x`, some symbolic value and permission value 1/2 is added. At this point, some assumptions can be made about the chunks. For example, since the permission value is more than 0, the receiver cannot be null (one cannot have permission to null). This assumption gets added to the path conditions. Creating a new chunk, adding it to the heap and assuming the properties of the chunk, is called *producing*.

The second inhale on line two works exactly the same as the first one, and so does the third one (just with a permission value of 1 instead of 1/2).

In Line 5 we learn that `x == y`. This is a case of *aliasing*, i.e. multiple references pointing to the same heap location. Just like the assumptions about chunks, it gets added to the path conditions.

Finally, in line 7 we try to exhale permissions to `x.f`. In Silicon, permissions have to be removed from the heap, which is called *consuming*. In this case, consuming should succeed, since `x == y`, so the total amount of permissions to location `x.f` is 1. However, consuming fails, since the consumption algorithm looks for a (single) chunk that provides enough permissions to remove. Such a chunk does not exist on the heap at the moment. Therefore, a *state consolidation* is triggered.

In a state consolidation, Silicon "cleans up" the heap by merging chunks for the same location. In this case, the chunk that was added for `x.f` and the one added for `y.f` get merged to a single chunk with a permission amount of 1. Additionally, more assumptions about the chunks are added to the path conditions, e.g. that `x != z`. This follows from the fact, that one cannot have more than write permissions to a field and if `x == z`, the total permission amount to `x.f` would be 2. After the state consolidation, consuming is retried and this time it succeeds since there is a chunk with enough permissions on the heap.

The reason why there are two points at which assumptions about chunks are added to the path conditions (when producing and during state consolidation) is, that state consolidations are very expensive and should, therefore, be avoided as much as possible. Therefore, it is better to add properties when producing. However, if adding the properties itself is very expensive, it is postponed to the next state consolidation.

2.2.3 Handling Magic Wands

To illustrate the handling of magic wands, consider the following example (taken from [2]), where `m` and `n` are integers and `x` and `y` are references:

```
1 n := m - 1
2 inhale acc(x.f) && x.f > n + 1 --* acc(x.f) && x.f > m
3 y := x
4 n := n + 1
5 exhale acc(y.f) && y.f > n --* acc(y.f) && y.f > m
```

Listing 2.8: Magic wand handling

When inhaling a magic wand (line 2), the wand gets separated into its *structure* and its *arguments*:

$$(\texttt{acc(\_.f) \&\& \_.f > \_ --* acc(\_.f) \&\& \_.f > \_})(x, x, n+1, x, x, m)$$

The structure has holes where the arguments fit from left to right. When exhaling the magic wand again (line 5), a magic wand has to be found, that

has the same structure and where the prover can prove the arguments to be equal. In this case, the magic wand to be exhaled is

$$(\texttt{acc(\_.f) \&\& \_.f > \_ --* acc(\_.f) \&\& \_.f > \_})(y, y, n, y, y, m)$$

They both have the same structure, and the prover is able to prove that `x == y`. Additionally, since the value of `n` changed, the old `n + 1` can be proven to be equal to the new `n`, which makes exhaling succeed.

Magic wand chunks consist of the structure of the wand, the arguments and also a snapshot [2] [3], which, among other things, stores information about the values of fields when the magic wand chunk was created. This makes magic wands very similar to predicates, where the structure corresponds to the predicate name.

Chapter 3

Separation Algebra and Silicon

In order to get a better understanding of chunk handling in Silicon, we will describe how it relates to a more formal approach using *separation algebras*. In the first section, the definitions are given, and in the second section, they are compared to how chunk handling is done in Silicon.

3.1 Separation Algebra

Separation logic [4] is an extension to Hoare logic that has found many applications in verifying a wide variety of programs. A way to build a separation logic is by defining *separation algebras* [5]. A separation algebra is a tuple $\langle A, \oplus, u \rangle$ where A is some set, $\oplus$ is a partial relation over that set which is

- associative: $x_1 \oplus (x_2 \oplus x_3) = (x_1 \oplus x_2) \oplus x_3$
- commutative: $x_1 \oplus x_2 = x_2 \oplus x_1$
- cancellative: $x_1 \oplus y = x_2 \oplus y \rightarrow x_1 = x_2$

and u is a unit. Additionally, there are also multi-unit separation algebras, where every element in A has a unit, but not necessarily the same. The separation algebras described in this section are based on [6].

3.1.1 Heaps

A heap is a function $h : L \rightarrow V$ that maps heap locations $l \in L$ to an element of some set C. Let H be the set of all such heaps. A lookup in heap h for a location l is, therefore, done by applying $h(l)$.

For elements $v_1, v_2 \in V$, we define the partial operation $v_1 \oplus v_2$ to be the *combine* operation which produces a combined element $v' \in V$, if it is defined for the chosen elements.

Two heaps $h_1, h_2 \in H$ are *compatible*, iff $\forall l \in L : h_1(l) \oplus h_2(l)$ is defined.

For two compatible heaps $h_1, h_2 \in H_F$ the composition operation $\bullet$ produces a new heap $h_1 \bullet h_2$ for which for all $l \in L$ a lookup is defined as

$$(h_1 \bullet h_2)(l) = h_1(l) \oplus h_2(l).$$

The $\bullet$ operation is, therefore, a location-wise combine operation.

We can also define the reverse operation: For two heaps $h_1, h_2 \in H$ we define $h_1 \backslash h_2 = h$ iff $\exists h$ such that $h_2 \bullet h = h_1$.

3.1.2 Fields

We can instantiate the sets L and V from the previous section to define a separation algebra for fields. A field location is a tuple (f, r) where f is a field name and r is a receiver. Let L_F be the set of field locations. V can be instantiated as the set $V_F := (S \times P_F)$ where S is the set of snapshots and P_F is the set of field permissions. This leads to the following definition for field heaps: A field heap is a function from L_F to $(S \times P_F)$. Let H_F be the set of field heaps.

We can now define the $\oplus$ operator for fields. For that, we first define it for snapshots and permissions. Let $s_1, s_2 \in S$. Then

$$s_1 \oplus s_2 = \begin{cases} s_1 & \text{if } s_1 = s_2 \\ \text{undef.} & \text{otherwise.} \end{cases}$$

Let P_F be the interval $[0, 1] \subset \mathbb{Q}$ and let $p_1, p_2 \in P_F$. Then

$$p_1 \oplus p_2 = \begin{cases} p_1 + p_2 & \text{if } p_1 + p_2 \in [0, 1] \\ \text{undef.} & \text{otherwise.} \end{cases}$$

Let $(s_1, p_1), (s_2, p_2) \in (S \times P_F)$. Then

$$(s_1, p_1) \oplus (s_2, p_2) = \begin{cases} (s_1 \oplus s_2, p_1 \oplus p_2) & \text{if } s_1 \oplus s_2 \text{ and } p_1 \oplus p_2 \text{ are defined,} \\ \text{undef.} & \text{otherwise.} \end{cases}$$

Additionally, we will replace every unit element, i.e. every element with permission value 0, by $\perp$ in order to make sure that every value is only paired with a non-empty permission. Therefore the new set is

$$V_F := (S \times P_F) \cup \{\perp\} - \{(s, 0) \in (S \times P_F)\}.$$

3.1.3 Predicates

To define a separation algebra for predicates, we first define L_P. A predicate location $l \in L_P$ is a tuple $(n, [args])$, where n is a predicate name and $[args]$ is a list of arguments. V can be instantiated as the set $V_P := (S \times P_P)$ where S is the set of snapshots and P_P is the set of predicate permissions. This leads to the following definition for predicate heaps: A predicate heap is a function from L_P to $(S \times P_P)$. Let H_P be the set of predicate heaps.

We can now define the $\oplus$ operator for predicates. Let S be the separation algebra over snapshots as before. Let P_P be the interval $[0, \infty) \subset \mathbb{Q}$ and let $p_1, p_2 \in P_P$. Then $p_1 \oplus p_2 = p_1 + p_2$. Combining $(s1, p1) \oplus (s2, p2)$ is analogous to fields.

Again we replace all unit elements by $\perp$ analogous to fields.

3.1.4 Generalized Heaps

In general, one might have both fields and predicates. Therefore we define *general heaps* to account for such situations.

A general heap h is an n-tuple of heaps, $h \in H_1 \times H_2 \times ...$ where $H_1, H_2, ...$ are sets of heaps. For example, the set of heaps that contains both fields and predicates would then be given as $H = H_F \times H_P$. Compatibility and composition $\bullet$ of two heaps are defined component-wise.

3.2 Silicon

The separation algebras defined in the last section can now be used to understand the resource handling of Silicon.

3.2.1 Chunks

A field chunk is a tuple $((x, f), s, p)$ where `x.f` is a field with snapshot s and permission p. In separation algebra, it corresponds to a single mapping of the heap function for location (x, f). If we interpret a function to be a subset of domain-codomain tuples, we get exactly the same representation.

Analogously we define a predicate chunk to be a tuple $((pred, args), s, p)$ where $pred$ is a predicate name with arguments $args$, snapshot s and permission p.

Let C be the set of chunks and L be the set of Silicon heap locations for both field and predicate chunks. Additionally, for a chunk $c = (l, s, p)$ let

$$\begin{aligned} \texttt{loc}(c) &= l \\ \texttt{snap}(c) &= s \\ \texttt{perm}(c) &= p \end{aligned}$$

be the accessor functions for the chunk components. We will also use $\texttt{perm}(h(l))$ and $\texttt{snap}(h(l))$ for the permission and value of a separation algebra heap location.

3.2.2 Symbolic State

A symbolic state in Silicon is a tuple $(\xi, \pi) \in (\Xi \times \Pi)$, where $\xi \in \Xi$ is a heap and $\pi \in \Pi$ is a set of path conditions.

As in separation algebra, a heap in Silicon is a collection of chunks. Although in contrast to separation algebra, in Silicon it can happen that one does not know every aliasing relation at a specific point in time, therefore there can be multiple chunks for the same heap location which means that a heap is a *multiset* of chunks.

A set of path conditions π in Silicon represents the current assumptions that have been made. It is a set of boolean formulae. For path conditions π and heap locations l_1, l_2, we write $l_1 =_\pi l_2$ iff it follows from π that the heap locations l_1 and l_2 are the same, and $l_1 \neq_\pi l_2$ iff it follows that they are not equal.

Hence, we can define a function `chunks` that takes a set of path conditions, a heap, and a location and returns a multiset of all chunks in the heap, for which it follows from the path conditions that the location is the same:

$$\texttt{chunks}(\pi, \xi, l) = \{c \in \xi \mid \texttt{loc}(c) =_\pi l\}$$

Let $\xi_\pi(l)$ be short for $\texttt{chunks}(\pi, \xi, l)$.

A heap h in separation algebra corresponds to a Silicon heap/path condition pair (ξ, π) iff the following properties hold $\forall l \in L$:

$$\sum_{c \in \xi_\pi(l)} \texttt{perm}(c) \leq \texttt{perm}(h(l)) \tag{H1}$$

$$\forall c \in \xi_\pi(l) : \neg(\texttt{snap}(c) \neq_\pi \texttt{snap}(h(l))) \tag{H2}$$

i.e. the permission amount in Silicon is an underapproximation of the permission amount in separation algebra, it cannot be proven that a single location has multiple different values and it cannot be proven that the value

is different from the value at the heap location in separation algebra.

A situation where the path conditions are contradictory in Silicon leads to a contradictory state where one can assert `false`. This corresponds to a non-existent heap in separation algebra (e.g. due to an undefined operation).

3.2.3 Lookup

Consider looking up in a heap h whether a location l has at least p_{req} permission. In separation algebra, this is done by applying h to l, i.e. check whether $p_{req} \leq \texttt{perm}(h(l))$.

Doing the same in Silicon is more complicated. With path conditions π, looking up in a heap ξ whether a location l has at least p_{req} permission is defined as follows:

$$\exists c_0 \in \xi_\pi(l) : p_{req} \leq \texttt{perm}(c_0)$$

or in words, one has to look for a chunk whose location is l and for which one has enough permission. If one finds such a chunk c_0, soundness follows from property (H1):

$$p_{req} \leq \texttt{perm}(c_0) \leq \sum_{c \in \xi_\pi(l)} \texttt{perm}(c) \leq \texttt{perm}(h(l))$$

Therefore, if a lookup succeeds in Silicon, it also succeeds in separation algebra.

If a lookup does not succeed in separation algebra, neither will it in Silicon, again due to property (H1)

$$p_{req} > \texttt{perm}(h(l)) \geq \sum_{c \in \xi_\pi(l)} \texttt{perm}(c) \geq \texttt{perm}(c_0)$$

which again is sound. In all other cases, a lookup in Silicon will not succeed because of Silicon's incompletenesses, even though in separation algebra a lookup would succeed.

3.2.4 State Consolidation

State consolidation in Silicon means finding definite aliasing relations in the heap and merging the respective chunks. The goal is to reduce the number of chunks in the multiset for a given location. In separation algebra, such a step is never needed, since aliasing relations are always known. For Silicon, therefore, state consolidation is a form of 'catching up' with separation algebra.

To describe this more formally, we first define what it means for two chunks to be merged. Let $c_1, c_2 \in \xi$ be chunks for which $\texttt{loc}(c_1) =_\pi \texttt{loc}(c_2)$, let π be the path conditions. Then merging c_1, c_2 will produce a new chunk

$$c = (\texttt{loc}(c_1), \texttt{perm}(c_1) + \texttt{perm}(c_2), s)$$

where s is a new snapshot. This leads to a new heap ξ' where which is the same as ξ except for location $\texttt{loc}(c_1)$ which now is

$$\xi'_\pi(\texttt{loc}(c_1)) = \xi_\pi(\texttt{loc}(c_1)) \cup \{c\} - \{c_1, c_2\}$$

i.e. c_1, c_2 have been replaced by c. Additionally, multiple conditions will be added to the path conditions:

- $\texttt{perm}(c_1) > 0 \rightarrow \texttt{snap}(c_1) = s$
- $\texttt{perm}(c_2) > 0 \rightarrow \texttt{snap}(c_2) = s$

Merging preserves property (H1) since $\sum_{c \in \xi_\pi} \texttt{perm}(c) = \sum_{c \in \xi'_{\pi'}} \texttt{perm}(c)$. Note that merging two chunks is essentially the same thing as applying the $\oplus$ operator two $h_1(l)$ and $h_2(l)$ for some location l and heaps h_1, h_2 in separation algebra.

State consolidation means for all $l \in L$ merging all chunks in $\xi_\pi(l)$ as described above one after each other. Let π' be the path conditions after merging. Then we know that for all $l \in L$ the multiset $\xi'_{\pi'}(l)$ contains either no chunk or a single chunk c' for which

$$\texttt{perm}(c') = \sum_{c \in \xi'_{\pi'}(l)} \texttt{perm}(c)$$

In addition to the path conditions added in the merging steps, the following path conditions will be added to produce new path conditions π'':

- For all non-equal field chunks $c_1, c_2 \in \xi'$ Silicon adds

  $$\texttt{perm}(c_1) + \texttt{perm}(c_2) > 1 \rightarrow \texttt{loc}(c_1) \neq \texttt{loc}(c_2)$$

  to the path conditions in order to make use of the fact that field locations cannot have a permission value of more than 1. This is obviously also true in separation algebra.

- For the same reason, for all field chunks c the path condition

  $$\texttt{perm}(c) \leq 1$$

  will be added. If before one could have proven that the permission value is greater than 1, now Silicon is in a contradictory state and in separation algebra there would be no heap.

Hence, state consolidation leads to a new symbolic state (ξ', π'').

3.2.5 Inhaling

In separation algebra, inhaling permission p for a heap location l means composing the heap h corresponding to a Silicon ξ and the singleton heap h_c corresponding to a chunk c, i.e. $h' = h \bullet h_c$. If such a heap h' exists, it has the property $\texttt{perm}(h'(l)) = \texttt{perm}(h(l)) + \texttt{perm}(h_c(l))$.

In Silicon, inhaling permission p to a location l adds a new chunk $c = (l, p, s)$ to the heap ξ and tries to merge it with a chunk for the same location already on the heap if there is one. Let π again be the path conditions, ξ' the new heap and π' the new path conditions. This leads to the following two possibilities:

- $\xi_\pi(l) = \varnothing$. Then, $\forall l' \neq l : \xi'_{\pi'}(l') = \xi_\pi(l')$ and $\xi'_{\pi'}(l) = \{c\}$.
- $\xi_\pi(l) = C$ for some multiset $C \neq \varnothing$. Let $c_0 \in C$ be a chunk. Then c_0 and c will be merged as in a state consolidation which also adds to the path conditions.

In both cases, the assumption will be added that $\texttt{perm}(c) \geq 0$. Also, property (H1) is preserved in both cases, since for all $l \in L$

$$\sum_{c \in \xi'_{\pi'}(l)} \texttt{perm}(c) = \sum_{c \in \xi_\pi(l)} \texttt{perm}(c) + p \leq h(l) + p = h_c(l)$$

holds.

3.2.6 Exhaling

Next, we consider exhaling permission p to a location l from a heap h corresponding to a Silicon heap ξ. In separation algebra, this means finding a heap $h' = h \backslash h_c$ where h_c is the singleton heap for location l. If such a heap h' exists, it has the property $\texttt{perm}(h'(l)) = \texttt{perm}(h(l)) - \texttt{perm}(h_c(l))$.

Silicon, on the other hand, will look for a chunk $c \in \xi_\pi(l)$ for which $\texttt{perm}(c) \geq p$. If there is such a chunk, from which the permission amount p will be subtracted. This preserves property (H1), since

$$\sum_{c' \in \xi'_\pi(l)} \texttt{perm}(c') = \sum_{c' \in \xi_\pi(l)} \texttt{perm}(c') - p \leq \texttt{perm}(h(l)) - p = \texttt{perm}(h'(l))$$

where again ξ' is the new heap.

If there is no chunk $c \in \xi_\pi(l)$ for which $\texttt{perm}(c) \geq p$, a state consolidation will be performed and then the operation is retried. If there still is not enough permission, verification fails.

If no heap h' can be found in separation algebra due to $\texttt{perm}(h(l)) < p$, Silicon will fail as well (because of property (H1)):

$$\forall c \in \xi_\pi(l) : \texttt{perm}(c) \leq \texttt{perm}(h(l)) < p$$

therefore Silicon is sound in this case as well.

The path conditions do not get changed by exhaling permissions.

3.3 Notable Properties of Chunk Handling

We defined separation algebras for the different resources. From this we can identify some notable properties of chunk handling:

- From the separation algebras, we can derive that the only operation on a specific chunk type that has to be defined, is the $\oplus$ operator, i.e. how to merge chunks.
- In Silicon, on the other hand, chunks have even more similarities, since all operations only use the three accessor functions $\texttt{loc}(c), \texttt{snap}(c)$ and $\texttt{perm}(c)$.
- Heap handling is the same for all types of chunks, only the path conditions require a separate treatment.

These properties will be important for designing the new resource handling approach in chapter 4.

Chapter 4

Approach for Resource and Chunk Handling

A core goal of this thesis is to make resource handling in Silicon more easily customizable and adaptable to new resources. This chapter presents an approach to the handling of non-quantified resources in Silicon based on the separation algebras discussed in the previous chapter which is almost independent of the chunk type used. Furthermore, it enables more convenient customization of the existing resources: fields, predicates, and magic wands.

4.1 General Philosophy

The fundamental component of the resource API is the differentiation between *resources* and *chunks* and their role in Silicon. Resources are the general concept, for example fields or predicates. They have properties, which the user should be able to specify. Chunks, on the other hand, are the heap representation of a resource in Silicon. The approach has three main components:

- API for specifying properties of resources: Properties of resources should be specified independently of the chunk implementation and handling by using a *resource description*.
- Chunk implementation: Chunks implement a chunk interface, that works both with the API and chunk handling. It should be possible to add chunks without reimplementing chunk handling algorithms.
- Chunk handling: Producing, consuming and state consolidation should be as independent as possible from the chunk types used. They should work the same regardless of what type a chunk has.

If each of the three components is independent of the others, customizing each component is easier because changing it does not affect the other two.

4.2 API for Specifying Resource Properties

When producing a chunk and during state consolidation, various assumptions about the chunk get added to the path conditions. These, essentially, are properties of the resource the chunk represents and it should be possible to specify them separately from chunk handling.

4.2.1 Motivation

We will motivate the use of properties by showing fields as an example. For fields, there are four properties that need to be specified:

$$\forall x, f :: perm(x.f) \geq 0 \tag{4.1}$$

$$\forall x, f :: perm(x.f) \leq 1 \tag{4.2}$$

$$\forall x, f :: perm(x.f) > 0 \implies x \neq null \tag{4.3}$$

$$\forall x, y, f :: perm(x.f) + perm(y.f) > 1 \implies x \neq y \tag{4.4}$$

The first two properties bound the permission value of a field and the third property makes sure that one cannot have permissions to `null`. The last property describes the fact, that two receivers, if their permission values add up to more than one, cannot be equal.

In contrast to the properties given above, in Silicon the assumptions are not made about heap locations, but about chunks. The reason for that is, that there could be multiple chunks for the same heap location, which might not even be known when assuming the properties. It, therefore, makes sense to assume properties for each chunk separately. As described at the end of section 2.2.1, properties about chunks are assumed at two different times: on produce and during state consolidation. We will keep this distinction by introducing two types of properties: *instance properties*, which are assumed when a chunk is added, and *delayed properties*, which are assumed during state consolidation. To understand the difference, we consider again the properties for fields given above.

The property

$$\forall x, f :: perm(x.f) \geq 0$$

quantifies over all field locations. Chunks are an instantiation of the quantification, since they represent a single field location. Therefore, the body of the quantifier has to hold for each chunk separately, which makes it easy to add the property when a chunk is created, i.e. the property is an instance property. On the chunk level, the property can be described by

```
this.perm >= 0.
```

`this` is a reference to the chunk that is newly added, its permission value is accessed through dot notation just as in object-oriented programming and comparison works as in usual arithmetic.

The same can be done with

$$\forall x, f :: perm(x.f) > 0 \implies x \neq null$$

which becomes

```
this.perm > 0 ==> !(this.args == null).
```

Note the use of `this.args` to access the arguments of the chunk, which is the receiver of the field the chunk represents.

In the current version of Silicon, the property

$$\forall x, f :: perm(x.f) \leq 1$$

is assumed during state consolidation. If we want to keep that, we have to make it a delayed property. Instead of describing a single chunk that is added, a delayed property describes all chunks of a resource (fields in our case). The property therefore becomes

```
forEach c ::  c.perm <= 1.
```

The `forEach` expression iterates over all chunks in the heap and adds the body as an assumption. In this case, for every chunk `c` in the heap the assumption `c.perm <= 1` is added. While for iterations over 1-tuples of chunks a delayed property does not make much sense (and in fact, this property gets changed in section 6.1), it makes a lot of sense if we iterate over all *pairs* of chunks in the heap, which is exactly what needs to be done for the property

$$\forall x, y, f :: perm(x.f) + perm(y.f) > 1 \implies x \neq y.$$

A possible way of describing this property is

```
forEach c1, c2 :: c1.perm + c2.perm > 1
                ==> !(c1.args == c2.args).
```

Here, `(c1, c2)` are all possible pairs of *distinct* chunks in the heap. This restriction is important since the property does not hold if `c1 == c2`. Not allowing `c1 == c2` is not a problem though since in that case, one can just use an instance property. Additionally, it is important, that iteration happens for each field separately, i.e. that we do not mix e.g. fields `f` and `g`.

Therefore, the `forEach` expression treats each field separately, as if the property existed for each field on its own.

Since there are $O(n^2)$ pairs of chunks if n is the number of chunks in the heap, the number of assumptions becomes huge, which can slow the prover down. In that case, it is useful to only add the the assumption that `!(c1.args == c2.args)` if `c1.perm + c2.perm > 1` can be proven to be true. For this case, a `check then else` expression exists:

```
forEach c1, c2 :: check c1.perm + c2.perm > 1
                  then !(c1.args == c2.args)
                  else true
```

This is the version of the property that is actually used.

4.2.2 Syntax

We will now give an overview over the expressions that can be used in properties. A full grammar is given in appendix A. Properties support five types:

- `Boolean`: Usual Boolean values.
- `Argument`: Arguments of a chunk, e.g. the receiver for a field, or the arguments to a predicate.
- `Permission`: The permission amount of a chunk which is a rational value.
- `Value`: The value of a chunk, which represents a part of a heap.
- `Chunk`: A chunk itself.

These types can be used in properties. Some expressions enforce type rules, like e.g. the condition of an `if then else` expression has to be a `Boolean`. The possible expressions are:

- Boolean Algebra with literals `true` and `false`, and the usual Boolean operations `!`, `&&`, `||`, `==>`.
- Permission literals (fractions) and the usual arithmetic operations `+`, `-`, `*`, and `/`. Also, comparisons `>`, `<`, `>=`, `<=`.
- An equality operator (`==`). Both operands have to be of the same type, namely one of the `Boolean`, `Argument`, `Permission`, and `Value` types.
- A `this` keyword to refer to the chunk that is being added. Only allowed in instance properties.

- A `forEach` clause with an arbitrary number n of chunk variables to iterate over all n-tuples of distinct chunks in the heap. Only allowed in delayed properties.
- Accessors `.args`, `.perm`, `.val` of type `Argument`, `Permission`, and `Value` to access chunk components, namely the arguments, the permissions, and the values.
- A `null` literal of kind `Argument`.
- A ternary `if then else` clause, where the condition has to be of type `Boolean` and both branches have the same type, namely one of the `Boolean`, `Permission`, and `Value` types.
- A ternary `check then else` clause similar to `if then else`, but that checks the condition with the prover and only adds the resulting branch as an assumption.

Properties are Boolean expressions that can be built using any allowed combination of the above expressions.

4.2.3 Resource Descriptions

Properties built out of the aforementioned expressions can be used to then specify a *resource description* for a resource. The resource description for fields looks as follows:

```
Resource: Field

Instance:

this.perm >= 0
this.perm > 0 ==> !(this.args == null)

Delayed:

forEach c :: c.perm <= 1
forEach c1, c2 :: check c1.perm + c2.perm > 1
                  then !(c1.args == c2.args)
                  else true
```

Listing 4.1: Field resource description

The resource description contains the name of the resource in the first line, and then has two sections of properties, instance properties and delayed

properties, as indicated by the labels.

For predicates the resource description looks much simpler:

```
Resource: Predicate

Instance:

this.perm >= 0
```

Listing 4.2: Predicate resource description

Since there are no delayed properties, the label has been left out. The resource description for magic wands looks exactly the same and is therefore omitted here.

4.2.4 Usage in Silicon

During verification, the properties described in resource descriptions have to be translated to actual assumptions that can be given to the prover. This is done by using a *property interpreter*. For instance properties, a property interpreter takes a new chunk (the `this`-chunk) and a property and creates a symbolic expression. For delayed properties, it takes a resource identifier, a heap, and a property to create one. In both cases, the symbolic expression, later on, gets passed to the prover.

The implementation of property interpretation and the interpretation rules are explained in more depth in chapter 5.

4.3 Chunk Interface and Implementation

The second component of the approach is developing a chunk interface and implementing it for all chunks. Since it should work well with resource descriptions, it is apparent that it needs arguments, a value, and a permission amount corresponding to the accessors in the description grammar. Additionally, it needs to take into account e.g. the field name for fields, since the `forEach` expression works on each field name separately. And last, there has to be a possibility to tell which resource a chunk belongs to since again the `forEach` expression only iterates over chunks of a certain resource type. All these requirements lead to the following interface:

```
trait NonQuantifiedChunk {
  val resourceID: ResourceID
  val id: ChunkIdentifier
  val args: Seq[Term]
  val snap: Term
  val perm: Term
}
```

Listing 4.3: Non-quantified chunk interface

The arguments are given by `args`, the value by `snap`, and the permission amount by `perm`. They are all terms, which are symbolic expressions in Silicon. The `id` is the part of the abstract location, that only Silicon knows about (and not the prover), e.g. the field name for fields. It is of type `ChunkIdentifier` which is a trait that can be implemented by any class. Finally, the `resourceID` serves to identify, which resource a chunk belongs to.

We will now show how the chunks in Silicon implement the interface.

4.3.1 Fields and Predicates

Implementing a predicate chunk based on the interface for non-quantified chunks shown in listing 4.3 is straightforward. The `id` of a predicate chunk should be based on the name of the predicate. We, therefore, implement a class of chunk identifiers that use a string to identify the chunk:

```
case class BasicChunkIdentifier(name: String)
      extends ChunkIdentifier
```

Listing 4.4: Identifier based on a name

The rest is evident: The chunk arguments are the arguments of predicate instances, the snapshot is the predicate instance's snapshot and permissions are the permissions held to that predicate instance.

Field chunks have a very similar structure. They use the same chunk identifier as predicate chunks by using the field name. Furthermore, the location depends on the receiver of the field which can be seen as an argument. Fields, therefore, always have a single argument; snapshots and permissions are as before.

Since both predicates and fields essentially have the same structure, it is even possible to use the same chunk type for both; the only difference being

the `resourceID`. We can, therefore, implement a `BasicChunk` as follows:

```
case class BasicChunk(resourceID: ResourceID,
                      id: BasicChunkIdentifier,
                      args: Seq[Term],
                      snap: Term,
                      perm: Term)
    extends NonQuantifiedChunk
```

Listing 4.5: Implementation of a chunk for fields and predicates

4.3.2 Magic Wands

Magic wands have a similar structure to predicates, which makes implementing them quite apparent. Still, they need special treatment which we will explain here.

In contrast to predicates, magic wands do not have a name. Instead, every occurring syntactic structure of magic wands is assigned an index which is then used as the `id`. As a consequence, magic wands have their own chunk type. Arguments, the snapshot, and the permission amount are exactly as for predicates. This leads to the following chunk:

```
case class MagicWandChunk(id: MagicWandIdentifier,
                          args: Seq[Term],
                          snap: Term,
                          perm: Term)
    extends NonQuantifiedChunk {
  override val resourceID = MagicWandID()
}
```

Listing 4.6: Implementation of a magic wand chunk

Since magic wand chunks are used exclusively for magic wands, the `resourceID` can be fixed to that of magic wands.

4.4 Chunk Handling

The combination of having resource descriptions with property interpretation and the new chunk interface enables us to make chunk handling more independent of the specific chunk types used. The only operation that has to be defined for each chunk type separately is *merging*.

4.4.1 Usage of Interpretation

A crucial part of decoupling chunk handling from the specific chunk types is the use of the property interpreter to add assumptions about the chunks every time a new chunk is added to the heap. This step is the same for all chunks since it only depends on the chunk interface and the resource identifier.

4.4.2 Unification

In addition to the usage of the property interpreter to assume properties in a unified way, the general chunk interface allows producing and consuming to be completely independent of the specific chunk type used.

The only part of resource handling that still requires the knowledge of which chunk type is used, is the state consolidator. State consolidation requires chunks to be merged. This step is unique to each chunk type:

- Basic chunks get merged by adding up their permission value and creating a fresh snapshot, which is then set equal to the original snapshots.
- Magic wand chunks do not get merged at all. Viper does not yet support fractional magic wands, one can therefore only either have write permission to the wand, or no permission at all.

Merging can be specified at a single location in the state consolidator for all chunk types.

4.5 Impact on Customization

Separating resource descriptions, chunk handling, and chunk implementation has an impact on how customization can happen.

- It should be possible for users of Silicon to customize resource descriptions. This includes adding new properties and changing existing properties, e.g. exchanging `check then else` clauses by `if then else` clauses to decrease the number of incompletenesses.

- Adding new chunk handling methods, like different ways of producing and consuming, requires changes to Silicon itself. Because of the unified chunk interface though, they do not require separate implementation for each chunk type.

- And finally, adding support for new resources in Silicon requires the most effort. It involves the following steps:

1. Changing the syntax of the Viper language to support the new resource.
2. Adding a resource description for the new resource.
3. Providing a translation from the Viper syntax of the resource to a Silicon chunk. If an existing chunk type can be used to represent the resource, nothing more has to be done.
4. If a new chunk type is needed, it has to be implemented. By using the new chunk interface, almost no changes to chunk handling have to be made, since the same produce, consume, and state consolidation methods as for the other chunks can be used. However, merging has to be specified in the state consolidator.

In chapter 6 the approach will be used to change resource descriptions and chunk handling.

Chapter 5

Property Interpretation

Until now we described how properties are defined, but what is still missing is how properties are used to create assumptions during verification. The main component that achieves this is the *property interpreter*.

A property interpreter gets created every time new assumptions have to be made about chunks on a heap. If multiple assumptions are made about the same heap, the same property interpreter can be used. The property interpreter takes properties and builds up terms that are then passed to the prover as assumptions. Recall that terms are symbolic expressions in Silicon. Its public interface is the following:

```
class PropertyInterpreter
            (heap: Seq[NonQuantifiedChunk],
             prover: Prover) {

  var relevantChunks: Seq[NonQuantifiedChunk]

  def buildPathConditionForChunk
            (chunk: NonQuantifiedChunk,
             property: Property)
            : Term

  def buildPathConditionForResource
            (resourceID: ResourceID,
             property: Property)
            : Term
}
```

Listing 5.1: Property interpreter interface

The constructor takes a collection of chunks for which assumptions will be

made (if `forEach` clauses are used) and a prover (which is used for `check then else` expressions). Additionally, two methods are provided. The first one builds a path condition for an instance property, where the chunk passed as argument is seen as the `this` chunk. The second method builds a path condition for a delayed property. The resource identifier passed specifies which chunks will be considered when iterating over chunks. For example, if the field identifier is passed, only field chunks will be considered in `forEach` expressions. Both methods return a term that is later given to the prover.

For the interpretation to work, we introduce a *placeholder map*: A placeholder map is a map from chunk placeholders (chunk variables in `forEach` expressions or the `this` literal) to actual chunks. Whenever a chunk placeholder occurs in an expression, the map is used to find the actual chunk it refers to. Additionally, a sequence of chunks that are considered in `forEach` expressions has to be specified. Interpretation works in three steps:

1. When interpreting an instance property, initialize the placeholder map such that `this` maps to the chunk for which the instance property should be built. When interpreting a delayed property, initialize the sequence of chunks considered to all chunks in the heap with the resource identifier passed as an argument to the build method.

2. Pattern match the outermost expression. Depending on the type of the expression, use one of the interpretation rule described in the following sections.

3. Rules can be recursively defined, so proceed until no subexpressions to be evaluated are left.

We will now present the rules used for interpreting. We will refer to terms by using the prefix `terms`, i.e. the term `True()` is referred to as `terms.True()`.

5.1 Pattern Matching

The main structure of an interpretation rule is the following:

```
def buildExpr(expr, pm, cs) = terms.Term(...)
```

Listing 5.2: General interpretation rule for expressions

The rule takes an expression as an argument, which should be interpreted. Additionally, it takes a placeholder map (`pm`) that provides the mappings between chunk placeholders and chunks, and a sequence of chunks (`cs`) used in `forEach` expressions. The result of interpretation is a term.

5.2 Literals

Literals are trivial to interpret since they directly correspond to a term. For example, the rule to interpret the property expression `true` is

```
def buildExpr(true, pm, cs) = terms.True()
```

Listing 5.3: Interpretation rule for `true`

which just returns `terms.True()`. The other literals `false`, `null`, and permission literals work the same way.

5.3 Operators

Most binary operators and also the unary not operator are easy as well. The plus operator for permissions, for example, works as follows:

```
1 def buildExpr(left + right, pm, cs) = {
2   terms.Plus(
3     buildExpr(left, pm, cs),
4     buildExpr(right, pm, cs)
5   )
6 }
```

Listing 5.4: Interpretation rule for binary operators

The other arithmetic operations (`-`, `*`, `/`) and the comparison operators work analogously. The `!` operator does the same, just for a single subexpression instead of having a left and a right subexpression.

Boolean operators are more interesting, since they implement the following optimization: If the left-hand side is syntactically equal to `terms.False()` (for `&&` and `==>`) or `terms.True()` (for `||`), the right-hand side will not be evaluated anymore, since the result of the operation is already determined. This can save a lot of time, if the right-hand side contains expensive `check then else` expressions. Therefore, the left hand side has to be interpreted first, then checked for early termination and only in the negative case the right hand side will be interpreted. For example, when interpreting the `&&` operator

```
1 def buildExpr(left && right, pm, cs) = {
2   val leftTerm = buildExpr(left, pm, cs)
3   if (leftTerm == terms.False()) {
4     terms.False()
```

```
5   } else {
6     val rightTerm = buildExpr(right, pm, cs)
7     terms.And(leftTerm, rightTerm)
8   }
9 }
```

Listing 5.5: Interpretation rule for the && operator

the left-hand side is checked for being `terms.False()` in line 3, and only if it is not the case, the right-hand side is interpreted in line 6.

A special case is the `==` operator. Equality is only defined for the `Boolean`, `Argument`, `Value`, and `Permission` type. The rule, therefore, has to make sure that both the left and the right expression's types agree. Else, it works just like the other binary operators.

5.4 Accessors

Accessors access the attributes of a chunk. There are three accessors: the argument accessor, the value accessor, and the permission accessor. They all work the same way: First, the chunk to be accessed is queried in the placeholder map. Then, the attribute is extracted from the chunk. The rule for permission accesses is

```
1 def buildExpr(chunkPlaceholder.perm, pm, cs) = {
2   pm(chunkPlaceholder).perm
3 }
```

Listing 5.6: Interpretation rule for the permission accessor

The chunk placeholder could either be the `this` keyword or a chunk variable defined in a `forEach` expression, e.g. as in `forEach c :: c.perm == 0`. Interpreting `c.perm` will return the permission value the chunk represented by `c` contains.

5.5 Conditionals and Loop

Interpreting the ternary `if then else` property expression is similar to binary operators, just with three arguments (the condition, the then branch and the else branch), since there is an if-then-else term in Silicon's term language.

The `check then else` expression is more interesting. It uses the prover to check the condition and returns only the term of the respective branch.

```
def buildExpr(check cond then e1 else e2, pm, cs) = {
  val condTerm = buildExpr(cond, pm, cs)
  if (prover.check(condTerm) {
    buildExpr(e1, pm, cs)
  } else {
    buildExpr(e2, pm, cs)
  }
}
```

Listing 5.7: Interpretation rule for the `check then else` expression

Note that `prover.check` makes use of the `prover` field in the property interpreter class shown in listing 5.1.

The last expression is the `forEach` expression. It is used to create assumptions about the whole heap. Its semantics is somewhat complex: The chunks in the heap get grouped by their chunk identifier, so there is a sequence of chunks for each identifier. For each of these sequences, a conjunction gets created by iterating over the chunks and for each chunk creating a term out of the body of the `forEach` expression. We will show the rule in two parts: The first part splits the heap into the groups, and the second part creates a term from the chunks in a group.

The first part looks as follows:

```
1  def buildExpr(forEach c1, ... :: body, pm, cs) = {
2    val groups = ... // group cs by chunk identifier
3    val assumptions = Seq() // sequence of terms
4    for each group in groups {
5      val assumption =
6         buildForEach(forEach c1, ... :: body, pm, group)
7      assumptions.append(assumption)
8    }
9    terms.And(assumptions)
10 }
```

Listing 5.8: Grouping chunks by identifier

In the second line, the groups get created. Then, for each group of chunks, the second part of the rule gets used to build the term (lines 5 and 6). In line 9, a conjunction is created out of all terms.

The second part of the rule builds a term out of a group of chunks with the same identifier:

```
def buildForEach(forEach c1, ..., cn :: body, pm, cs) = {
  for each n-tuple of distinct chunks in cs {
    val pm2 = ... // add mappings
                  // c1 -> chunk1
                  // c2 -> chunk2
                  // ...
                  // cn -> chunkn
                  // to pm
    buildExpr(body, pm2, cs)
  }
}
```

Listing 5.9: Building a term out of a group

For each n-tuple of distinct chunks in `cs`, a mapping from the first chunk variable to the first chunk in the tuple, the second chunk variable to the second chunk in the tuple, and so on, is added and the body of the `forEach` expression is built using the updated mapping.

To illustrate how the `forEach` rule works, consider the following example: Let `forEach c1, c2 :: !(c1.val == c2.val)` be a property for fields. This property makes sure, that for every field name each chunk on the heap has a unique value. Also, let the heap consist of four chunks:

$$(x.f, 1), (y.f, 4), (x.g, 3), (y.g, 1)$$

The outermost expression of the property is a `forEach` expression. Therefore, pattern matching will result in calling the `forEach` rule in listing 5.8. There, the heap gets split into two groups: A group for field f $(x.f, 1), (y.f, 4)$ and a group for field g $(x.g, 3), (y.g, 1)$. Now, for each group, the body gets created.

For the first group, there is only one tuple of distinct chunks, namely the tuple $((x.f, 1), (y.f, 4))$. Therefore, the mappings $c1 \rightarrow (x.f, 1)$ and $c2 \rightarrow (y.f, 4)$ get added to the placeholder map. Now, the body of the `forEach` expression gets created, i.e. `!(c1.val == c2.val)`. In this case this becomes `!(1 == 4)`. The same thing is done for the second field name, which results in the assumption `!(3 == 1)`. In the end, the conjunction `!(1 == 4) && !(3 == 1)` is created.

Chapter 6

Customizations

In this chapter, we will use the approach developed in chapter 4 to customize resource descriptions and chunk handling. Because of the separation of resources from chunk handling, both changes can be done in isolation, without affecting one another.

6.1 Changing Resource Descriptions

In section 4.2.3 we presented the resource descriptions for the resources currently supported in Silicon. One element that does not seem to fit very well with the spirit of resource descriptions is that the assumption for the upper bound of 1 on field permissions is a delayed property, but the lower bound of 0 is an instance property. There is no need use a delayed property if the property only mentioned a single chunk. Changing this by using the resource API is simple: The delayed property

```
forEach c :: c.perm <= 1
```

gets replaced by the instance property

```
this.perm <= 1
```

in the field resource description. In addition to being more consistent, the change also affects the number of times Silicon has to perform a state consolidation. Since the property is assumed on produce, no state consolidation has to be performed when trying to use the fact, that the permission value to a field location is at most 1.

Another possibility is to add a new property. Consider the following example:

```
field f: Int

method foo(x: Ref, y: Ref)
  requires acc(x.f, 1/2) && acc(y.f, 1/2)
  requires x.f == 0 && y.f == 1
{
  assert x != y
}
```

Listing 6.1: Example of value inequality

The method requires read permission to `x.f` and `y.f` and additionally that their values are 0 and 1 respectively. From that, it should be possible to assert that `x != y`, since the same field location cannot have both the value 0 and 1 at the same time. However, this property is not implemented in Silicon. We can, therefore, use our approach to change the resource description and make this example verify.

The property that is required is similar to the property about the receivers not being equal if the permission amounts add up to more than 1. It has to be checked that the values of two chunks differ, and if so, add the assumptions that the receivers are not the same. Written as a property, this leads to

```
forEach c1, c2 :: check !(c1.val == c2.val)
                  then !(c1.args == c2.args)
                  else true
```

This encoding, however, would be unsound in the presence of chunks with no permissions. If one has no permission to a location, one is not allowed to read its value, since some other thread could have changed it. Therefore, a check for enough permissions has to be included which leads to the final version:

```
forEach c1, c2 :: check (c1.perm > 0 && c2.perm > 0)
                     && c1.val != c2.val
                  then c1.args != c2.args
                  else true
```

Adding the disequality constraint makes the Viper example verify.

The same can be done for predicates: If their snapshots differ, at least one of the arguments to the predicates has to be different. An example where this happens is the following:

```
field f: Int

predicate pred(x: Ref)
{
  acc(x.f)
}

method foo(x: Ref, y: Ref)
{
  inhale acc(x.f)
  x.f := 0
  fold pred(x)

  inhale acc(y.f)
  y.f := 1
  fold pred(y)

  assert x != y
}
```

Listing 6.2: Example for predicate snapshot inequality

First, access to `x.f` gets inhaled and a value gets assigned to the field. Subsequently, the permissions to the field get folded into the predicate, which makes Silicon temporarily forget the value of `x.f` (since one does not hold any permissions to it anymore). The same happens for `y.f`. In the end, one can still assert that `x != y`, even though the values of `x.f` and `y.f` are unknown, because one can see that the snapshots of `pred(x)` and `pred(y)` are different, which implies that `x != y`.

The final versions of the resource descriptions for all three resource types in Silicon can be found in appendix B.

6.2 Changing Chunk Handling

In addition to changing resource descriptions, this thesis also adds a new consumption method that can be enabled via a command line option. To motivate this, some understanding of how consuming is implemented in Silicon is needed.

Greedy Consumption Algorithm

The way Silicon consumes permissions to a heap location involves the following steps:

- First, Silicon tries to find a chunk for the required heap location on the heap. If the chunk has sufficient permissions, the permissions are taken from that chunk and consuming succeeds.
- If there is no matching chunk on the heap or the chunk found does not have enough permissions, a state consolidation is triggered. In state consolidation, all chunks with the same location are merged.
- After state consolidation, consuming is retried, i.e. Silicon again tries to find a chunk for the required heap location on the heap. If the chunk has sufficient permissions, the permissions are taken from that chunk and consuming succeeds.

Since the algorithm greedily tries to find a chunk and remove permissions from it, it is called the *greedy* consumption algorithm.

This approach has an incompleteness in the presence of *disjunctive aliasing*. Disjunctive aliasing describes the following situation:

```
1 field f: Int
2
3 method disjAliasing(x: Ref, y: Ref, z: Ref)
4 {
5   inhale acc(y.f) && acc(z.f)
6   assume x == y || x == z
7   exhale acc(x.f)
8 }
```

Listing 6.3: Disjunctive aliasing

In line 5, two chunks get added to the heap: one for `y.f` and one for `z.f`. In line 6 a disjunctive aliasing relation is learned, i.e. that `x` is either equal to `y` or to `z`. Since in both cases there is enough permission, exhaling permission to `x` (line 7) should be possible. The greedy algorithm is not able to do this though since it needs a specific chunk with enough permission to consume. In this case, it can neither prove that `y` is an alias of `x` nor that `z` is an alias of `x`. Therefore there is no chunk where permissions can be removed from and exhaling fails.

Complete Consumption Algorithm

The solution to this is a more complete consumption algorithm that can be enabled by using the command line option `-enableMoreCompleteExhale`. The complete consumption algorithm is based on the algorithm for consuming quantified permissions used by Silicon and described in [7]. The idea of the algorithm is, that instead of consuming permissions from a single chunk, one iterates over all chunks in the heap and removes as many permissions from each chunk as possible, provided that the arguments can be proved to be equal. Once enough permissions have been removed, the iteration can be stopped.

The pseudo code of the implementation looks as follows:

```
def consumeComplete(heap: Seq[NonQuantifiedChunk],
                    id: ChunkIdentifer,
                    args: Seq[Term],
                    perms: Term,
                    prover: Prover)
                   : Seq[NonQuantifiedChunk] = {
    val relevantChunks = ... // all non-quantified
                             // chunks with chunk
                             // identifier id
    val otherChunks = ... // remaining chunks

    var pNeeded = perms
    val newChunks = Seq[NonQuantifiedChunk]()

    var moreNeeded = true

    foreach ch in relevantChunks {
      if (moreNeeded) {
        val eq = ... // set arguments equal

        // the maximum amount that could be taken
        // is the minimum of what is available
        // and what is needed
        val pMaxTake = PermMin(ch.perm, pNeeded)

        // If the arguments are equal, take
        // pMaxTake amount of permissions
        // else take no permissions
        val pTaken = Ite(eq, pMaxTake, NoPerm())

```

```
31          // remove pTaken permissions from ch
32          val newPerm = PermMinus(ch.perm, pTaken)
33          // create a new chunk with the new
34          // permission amount
35          val newChunk = ch.withPerm(newPerm)
36
37          pNeeded = PermMinus(pNeeded, pTaken)
38
39          // add newChunk to the new heap
40          newChunks.append(newChunk)
41
42          // create the stopping condition to
43          // check if pNeeded is 0
44          val stopCond = Equals(pNeeded, NoPerm())
45          moreNeeded = !prover.check(stopCond)
46        } else {
47          // if no more permissions need to be
48          // removed, just add the chunk to the
49          // new heap
50          newChunks.append(ch)
51        }
52      }
53
54      // concatenate newChunks and otherChunks
55      // to create a new heap
56      val newHeap = newChunks ++ otherChunks
57
58      // use property interpreter to assume
59      // properties of new chunks
60      val pi = new PropertyInterpreter
61                  (newHeap, prover)
62
63      foreach ch in newChunks {
64        // find resource description for chunk ch
65        val resource = ... // resource description
66        val properties = resource.instanceProperties
67        // build assumptions and add them
68        // to the prover
69        prover.assume(
70          pi.buildPathConditionsForChunk
71                        (ch, properties)
72        )
73      }
74
```

```
75      if (!moreNeeded) {
76        return newHeap
77      } else {
78        ... // failure due to insufficient permissions
79      }
80    }
```

Listing 6.4: Implementation of the more complete consumption algorithm

To explain the algorithm in more depth, it is easiest to explain it with an example. Recall the disjunctive aliasing example from before:

```
1 field f: Int
2
3 method disjAliasing(x: Ref, y: Ref, z: Ref)
4 {
5   inhale acc(y.f) && acc(z.f)
6   assume x == y || x == z
7   exhale acc(x.f)
8 }
```

Listing 6.5: Disjunctive aliasing

The complete consume is used in the last step: `exhale acc(x.f)`. We will give the line number of the algorithm in parentheses. The arguments for `consumeComplete` are the following:

- A heap `heap`: In the example, this is the heap before the exhale, which contains the chunks `BasicChunk("f", Seq(y), vy, 1)` and `BasicChunk("f", Seq(z), vz, 1)`, where `vy` and `vz` denote the values of `y.f` and `z.f`, respectively.
- A chunk identifier `id`: In the example, this is the name of the field, i.e. `"f"`.
- A sequence of arguments `args`: In the example, this is the receiver of the field to be exhaled, i.e. `Seq(x)`.
- A permission value `perms`: The permission amount to be exhaled, in this case `1`.
- A prover `prover`: The prover used to prove assertions.

The method returns a new updated heap.

The first step of the algorithm is to partition the heap into relevant chunks (chunks one might need to exhale from) and all other chunks (lines 7 and

10). In the example, both chunks on the heap have the desired identifier and are thus relevant chunks.

In line 12, the initialization of the loop starts. `pNeeded` denotes the permission amount that still has to be exhaled, which is initialized to the amount passed as an argument. In the example, that would be `1`. `newChunks` are the chunks in the new heap (line 13), which is initially an empty sequence. `moreNeeded` (line 15) is a Boolean variable that denotes whether there are still permissions to be exhaled.

Next, the iteration over the relevant chunks starts (line 17). In the example, the first chunk that is considered is `BasicChunk("f", Seq(y), vy, 1)`. If no more permissions are needed, the chunk is just added to the heap (line 50). Else (which happens in our case), the following is performed: First the arguments are set pair-wise equal (line 19), which for the example leads to `x == y`. Then, the maximum permission amount that could be taken is calculated (line 24), which is the minimum of what is available and what is still needed, i.e. `min(1, 1)` which is `1`. In line 29, the amount that can actually be taken is calculated. If the arguments are equal, the amount to be taken is `pMaxTake`, else no permissions can be taken. In the example, this leads to the expression `if x == y then 1 else 0` (the minimum has been simplified). Afterwards, the permission amount gets subtracted from the permission that the chunk had before (line 32), i.e. `1 - (x == y then 1 else 0)`. A new chunk gets created (line 35) by copying the old chunk and replacing the permission with the update value. This leads to the new chunk `BasicChunk("f", Seq(y), vy, 1 - (if x == y then 1 else 0))`.
Next, `pNeeded` needs to be updated to reflect the permission amount that is still needed. For that, we subtract `pTaken` from its old value and get `1 - (if x == y then 1 else 0)`. In line 40 the new chunk gets added to the new heap.

Finally, it is checked whether we still need more permissions or whether we already removed enough from the heap. In the example, `pNeeded` is `1 - (if x == y then 1 else 0)` which we cannot prove to be 0 (since this would require knowing that `x == y`). Therefore, `moreNeeded = false`.

The second iteration is analogous to the first one, but with different values. In the beginning, `pNeeded` is `1 - (if x == y then 1 else 0)`. Therefore `pMaxTake` becomes `min(1, pNeeded)`. `pTaken` therefore is `if x == z then pMaxTake else 0` and `newPerm` is `1 - pTaken`. Next, `pNeeded` gets updated to be `pNeeded - pTaken` which is `(1 - (if x == y then 1 else 0)) - (if x == z then min(1, (1 - (if x == y then 1 else 0))) else 0)`.

This complicated expression now gets tested for being equal to `0` (line 45).

We know that either `x == y` or `x == z`, so we inspect these two cases separately. Assume `x == y`. In that case, `if x == y then 1 else 0` just becomes `1` and the expression simplifies to `(1 - 1) - (if x == z then min(1, (1 - 1)) else 0)` which can be further simplified to `0 - (if x == z then 0 else 0)` which is `0`.

In the second case, `x != y`, therefore `x == z`. Again we can simplify the expression to be `(1 - 0) - (if x == z then min(1, (1 - 0)) else 0)` which can be further simplified to `1 - (if x == z then 1 else 0)` which again is `0`. Hence, the algorithm managed to remove enough permissions, since `moreNeeded` is definitely `0`.

After the iteration, in line 56, the new heap gets created out of all new chunks and the ones not considered by the algorithm. Then, a property interpreter gets instantiated (line 60) to create the assumptions about the new chunks. It gets initialized with the new heap and the prover. For each new chunk, its instance properties get constructed and assumed (lines 69 to 71).

If no more permissions are needed, we return the new heap, else we fail due to insufficient permissions. In the example, `moreNeeded` is `true`, therefore exhale succeeds.

Chapter 7

Quantified Chunks

In chapter 4 an approach was presented that introduces resource descriptions and their use in the handling of non-quantified chunks. The Viper language supports quantifications over heap locations, which in Silicon are modeled with quantified chunks. It is, of course, desirable to extend the use of resource descriptions to quantified chunks, most favorably by reusing the same resource description. This chapter first introduces the concept of quantified permissions and chunks by using fields as an example and then discusses a partial extension that makes the use of resource descriptions possible for instance properties.

7.1 Overview

While recursive predicates enable one to specify unbounded heap structures, they are hard to use when the access pattern does not correspond to the structure of the predicate, e.g. when having random access to an array. To deal with this problem, the Viper language supports *quantified* permission assertions, which can be used to get access to the entire array at once. An example for such an expression is

```
inhale forall x: Ref :: x != null ==> acc(x.f, 1/2)
```

Listing 7.1: Example of a quantified permission assertion

which provides read access to the field `f` for every reference that is non-null. More generally, a quantified permission assertion has the form

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

Listing 7.2: General quantified permission assertion

7.2 Supporting Quantified Chunks

Supporting quantified chunks in the approach requires building quantified assumptions out of the properties described in the resource descriptions and assuming them when adding the chunk to the heap. Recall the instance properties for fields described in section 4.2.3 (with the changes made in section 6.1):

```
1 this.perm >= 0
2 this.perm <= 1
3 this.perm > 0 ==> this.args != null
```

Listing 7.3: Field instance properties

For quantified chunks, one has to build a quantification over all heap locations mentioned in the quantified permission assertion. Recall the general form of quantified permission assertions given in listing 7.2. We can rewrite the properties to be:

$$\forall x : T :: p(x) \geq 0$$
$$\forall x : T :: p(x) \leq 1$$
$$\forall x : T :: c(x) \implies (p(x) > 0 \implies e(x) \neq null)$$

The condition $c(x)$ is not needed, if the assumption only contains $p(x)$, since assumptions about permissions should always hold regardless of the condition.

7.3 Interpretation

Interpretation of properties for quantified chunks is almost the same as for non-quantified chunks. Since only instance properties are supported, `forEach` expressions are not allowed. Additionally, `check then else` expressions are not really sensible since conditions can be true for some instantiations of quantified variables, but not for others. Therefore, `check then else` expressions are simply interpreted as `if then else` expressions. This enables one to reuse the properties described in the resource description introduced in section 4.2.3 (which may contain `check then else` expressions) while still being sound.

7.4 Delayed Properties

Due to time constraints, delayed properties are currently not supported for quantified chunks. Implementing them involves one major challenge: Identifying proper *triggers* which are used by the prover to instantiate quantifications, especially since currently on state consolidation, user-provided triggers are not available anymore. Apart from that, only the `forEach` rule for interpretation has to be implemented, which (instead of iterating over the heap) should create a quantified assumption for each quantified chunk.

Chapter 8

Evaluation

In this chapter, both the customization effort and the performance of the new resource API will be evaluated.

8.1 Customization

In chapter 6 changes were made to two of the three components of the approach. Especially changing the resource description is very convenient, since only a few lines of code at a single place have to be changed to implement new properties or change existing ones.

Even though changing the chunk handling rules is more difficult, it only has to be done once and it works for all non-quantified chunks, provided the chunk interface introduced in section 4.3 has been used.

8.2 Performance

Evaluating the performance of the new Silicon implementation involves two different discussions: Comparing the old implementation to the new one, and comparing the greedy consumption algorithm to the complete consumption algorithm introduced in section 6.2.

Old versus New

It is expected, that the additional overhead of the property interpreter and the newly added properties will slow down the Silicon version of this thesis compared to the old one, and indeed, this is the case for a lot of short tests in the Silicon test suite. For longer tests though, additional optimizations implemented in this thesis sometimes make the new version run considerably faster than the old one. Most notably, the longest running example of the test suite had an average runtime decrease of about 9% averaged over 5 runs.

Greedy versus Complete

When measuring, both the greedy and the more complete consumption algorithm had basically the same performance. The largest absolute difference was measured for the longest running example, where the greedy algorithm needed about 78 seconds averaged over 5 runs, and the complete algorithm 74 seconds. In all other cases, the absolute difference was less than half a second.

Chapter 9

Conclusion and Future Work

In this thesis, we described resources using separation algebras, from which we then derived an approach to make resource handling in Silicon not only more unified but also more customizable.

The approach still has some limitations: The resource descriptions are part of the Silicon code base, therefore, Silicon itself has to be recompiled when changing them. A possibility would be to support the syntax used for resource descriptions throughout this thesis as a text file input for Silicon. In that case, users could directly change the resources. Doing that is straightforward, the only thing needed is a parser to create the abstract syntax tree.

Additionally, it is desirable to extend the approach for quantified chunks to include delayed properties as well.

Last, there is also the possibility to either use the more complete consumption algorithm as the default or to implement heuristics for when to use the greedy consumption algorithm and the more complete one. This would allow users to verify Viper programs with disjunctive aliasing without having to set a command line flag.

Acknowledgements

Finally, I would like to thank Vytautas Astrauskas and Malte Schwerhoff for all the time and effort spent supervising and supporting me. Additionally, I want to thank Nils Becker for giving me early access to the new magic wand syntax and changes, in particular to magic wand snapshots, which made my implementation much easier and cleaner.

Listings

Bibliography

[1] P. Müller, M. Schwerhoff, and A. J. Summers: Viper: A Verification Infrastructure for Permission-Based Reasoning (VMCAI), 2016.

[2] M. Schwerhoff: Advancing Automated, Permission-Based Program Verification Using Symbolic Execution (Ph.D. Thesis), 2016.

[3] N. Becker: Generalized Verification Support for Magic Wands (Bachelor's Thesis), 2017.

[4] M. J. Parkinson and G. M. Bierman: Separation Logic, Abstraction and Inheritance (POPL), 2008.

[5] C. Calcagno, P. W. O'Hearn, and H. Yang: Local Action and Abstract Separation Logic, 2007.

[6] R. Dockins, A. Hobor, and A. W. Appel: A Fresh Look at Separation Algebras and Share Accounting, 2009.

[7] M. Schwerhoff and A. J. Summers: Automatic Verification of Iterated Separating Conjunctions using Symbolic Execution (CAV), 2016.

Appendix A

Resource Description Grammar

The grammar is type safe, but some additional constraints are enforced (either during construction or during interpretation):

- Identifiers are strings of any characters except for the string `"this"`.
- The body of a `forEach` expression may only contain chunk variables that are in its chunk variable list.
- Expressions may not contain both the `this` literal and `forEach` expressions.

Also, note that there is no `!=` operator. Use a combination of `!` and `==` instead.

Parentheses work as usual in most programming languages. Precedence is the same as in Viper. The property type labels can be omitted if no properties of that type is defined.

⟨*BoolLiteral*⟩ ::= ‘`true`’ | ‘`false`’

⟨*ChunkPlaceholder*⟩ ::= ‘`this`’ | ⟨*ChunkVariable*⟩

⟨*ChunkVariable*⟩ ::= ⟨*Identifier*⟩

⟨*ChunkVariableList*⟩ ::= ⟨*ChunkVariable*⟩
| ⟨*ChunkVariable*⟩ ‘`,`’ ⟨*ChunkVariableList*⟩

⟨*ArgsExpr*⟩ ::= ⟨*ChunkPlaceholder*⟩ ‘`.args`’ | ‘`null`’

⟨*PermExpr*⟩ ::= ⟨*ChunkPlaceholder*⟩ ‘`.perm`’
| ⟨*PermExpr*⟩ ‘`+`’ ⟨*PermExpr*⟩
| ⟨*PermExpr*⟩ ‘`-`’ ⟨*PermExpr*⟩
| ⟨*PermExpr*⟩ ‘`*`’ ⟨*PermExpr*⟩
| ⟨*PermExpr*⟩ ‘`/`’ ⟨*PermExpr*⟩
| ‘`if`’ ⟨*BoolExpr*⟩ ‘`then`’ ⟨*PermExpr*⟩ ‘`else`’ ⟨*PermExpr*⟩
| ‘`check`’ ⟨*BoolExpr*⟩ ‘`then`’ ⟨*PermExpr*⟩ ‘`else`’ ⟨*PermExpr*⟩

⟨*ValExpr*⟩ ::= ⟨*ChunkPlaceholder*⟩ ‘`.val`’
| ‘`if`’ ⟨*BoolExpr*⟩ ‘`then`’ ⟨*ValExpr*⟩ ‘`else`’ ⟨*ValExpr*⟩
| ‘`check`’ ⟨*BoolExpr*⟩ ‘`then`’ ⟨*ValExpr*⟩ ‘`else`’ ⟨*ValExpr*⟩

⟨*BoolExpr*⟩ ::= ⟨*BoolLiteral*⟩
| ‘`!`’ ⟨*BoolExpr*⟩
| ⟨*BoolExpr*⟩ ‘`&&`’ ⟨*BoolExpr*⟩
| ⟨*BoolExpr*⟩ ‘`||`’ ⟨*BoolExpr*⟩
| ⟨*BoolExpr*⟩ ‘`==>`’ ⟨*BoolExpr*⟩
| ⟨*BoolExpr*⟩ ‘`==`’ ⟨*BoolExpr*⟩
| ⟨*ArgsExpr*⟩ ‘`==`’ ⟨*ArgsExpr*⟩
| ⟨*ValExpr*⟩ ‘`==`’ ⟨*ValExpr*⟩
| ⟨*PermExpr*⟩ ‘`==`’ ⟨*PermExpr*⟩
| ⟨*PermExpr*⟩ ‘`>`’ ⟨*PermExpr*⟩
| ⟨*PermExpr*⟩ ‘`>=`’ ⟨*PermExpr*⟩
| ⟨*PermExpr*⟩ ‘`<`’ ⟨*PermExpr*⟩
| ⟨*PermExpr*⟩ ‘`<=`’ ⟨*PermExpr*⟩
| ‘`if`’ ⟨*BoolExpr*⟩ ‘`then`’ ⟨*BoolExpr*⟩ ‘`else`’ ⟨*BoolExpr*⟩
| ‘`check`’ ⟨*BoolExpr*⟩ ‘`then`’ ⟨*BoolExpr*⟩ ‘`else`’ ⟨*BoolExpr*⟩
| ‘`forEach`’ ⟨*ChunkVariableList*⟩ ‘`::`’ ⟨*BoolExpr*⟩

⟨*Properties*⟩ ::= ⟨*BoolExpr*⟩
| ⟨*BoolExpr*⟩ ⟨*Properties*⟩

⟨*ResourceDescription*⟩ ::= ‘`Resource:`’ ⟨*Identifier*⟩ ‘`Instance:`’ ⟨*Properties*⟩
‘`Delayed:`’ ⟨*Properties*⟩

Appendix B

Resource Descriptions

```
Resource: Field

Instance:

this.perm >= 0
this.perm <= 1
this.perm > 0 ==> !(this.args == null)

Delayed:

forEach c1, c2 :: check  c1.perm + c2.perm > 1
                  then !(c1.args == c2.args)
                  else   true

forEach c1, c2 :: check (c1.perm > 0 && c2.perm > 0)
                    && !(c1.val == c2.val)
                  then !(c1.args == c2.args)
                  else   true
```

Listing B.1: Final resource description for fields

```
Resource: Predicate

Instance:

this.perm >= 0

Delayed:

forEach c1, c2 :: check (c1.perm > 0 && c2.perm > 0)
                     && !(c1.val == c2.val)
                  then !(c1.args == c2.args)
                  else   true
```

Listing B.2: Final resource description for predicates

```
Resource: MagicWand

Instance:

this.perm >= 0
```

Listing B.3: Final resource description for magic wands

Appendix C

Declaration of Originality