Towards better Function Axiomatization in a Symbolic-execution-based Verifier

Bachelor Thesis
Mauro Bringolf
September 6, 2019

Advisors: Prof. Dr. Peter Müller, Dr. Malte Schwerhoff, Dr. Alex J. Summers
Department of Computer Science, ETH Zürich
Abstract

One of the challenges in software verification is reasoning about the heap. Silicon is a symbolic-execution-based verifier and part of the Viper verification infrastructure developed at ETH Zürich. In its symbolic execution, Viper functions are axiomatized on the SMT level. Since Viper functions are potentially heap-dependent, this requires some encoding of the heap on the SMT layer. We present a new design for such an encoding with the goal of improving function axiomatization in Silicon.
Contents

1 Introduction ........................................... 1
   1.1 Background ........................................... 1
   1.1.1 Function axiomatization ............................ 1
   1.1.2 Snapshots for function applications ............... 2
   1.1.3 Snapshots for predicate instances ................ 2
   1.1.4 Generalizing: Snapshots for arbitrary assertions .. 2
   1.1.5 Quantified versus non-quantified permissions ...... 2
   1.1.6 Current snapshot representation and its limitations 3
   1.2 Notation ............................................. 6
   1.3 Outline .............................................. 7

2 Heap snapshot representation in SMT .................. 9
   2.1 Requirements ......................................... 9
   2.1.1 Capturing field values ............................. 9
   2.1.2 Monomorphic and polymorphic snapshots .......... 10
   2.1.3 Capturing predicate instances .................... 11
   2.1.4 Support for framing ................................ 11
   2.1.5 On injectivity of the $loc_P$ function ............ 14
   2.2 Building snapshots from assertions .................. 14
   2.2.1 Atomic assertions .................................. 15
   2.2.2 Compound assertions ............................... 16

3 Symbolic Execution Rules ............................. 19
   3.1 Evaluating function applications .................... 19
   3.2 Symbolic Execution Rules ............................ 19
      3.2.1 Symbolic execution notation ...................... 19
      3.2.2 Consuming assertions ............................ 20
      3.2.3 Producing assertions ............................ 21
Chapter 1

Introduction

One approach to program verification is to provide means for specifying program behavior precisely. If program and specification are both mathematical objects then the verification task is well-defined: Does the code allow any behavior violating its specification? Complete formalization of this question enables tool support for answering it. Among the existing tools is Viper, a verification infrastructure developed at ETH Zürich [7]. It provides an intermediate verification language for imperative, heap-manipulating programs whose behavior can be specified in a permission based model. Additionally, it has two backends for automatically verifying programs against their specifications. Carbon is based on verification condition generation while Silicon implements symbolic-execution-based verification. Our work is solely concerned with Silicon.

1.1 Background

In Viper, permissions are used to specify what parts of the heap can be accessed by a certain piece of code. To this end so-called accessibility predicates are provided: \( \text{acc}(x.f, p) \) denotes \( p \) permissions to the field \( f \) of object \( x \). Additionally, a program can define parametrized assertions as predicates. More details on the language, including everything necessary for what follows, can be found in the official Viper tutorial [5].

1.1.1 Function axiomatization

Functions in Viper programs can be understood as abstractions over expressions. Conceptually, a function application is evaluated by substituting the arguments for the formal parameters in its body. While it is possible treat functions this way in symbolic execution (ref?), essentially performing the substitution in the symbolic execution engine, Silicon takes another
1. Introduction

Instead of doing the substitution during evaluation, it is axiomatized as an axiom per function. This definitional axiom equates a function application with its body and thus, in a sense allows the underlying SMT solver to perform the substitution instead.

Formulating this axiom for the SMT solver requires a translation of the function body, in particular of any heap-dependent expressions contained in it. And this is exactly where heap snapshots come into play.

1.1.2 Snapshots for function applications

For a function application, its heap snapshot stores the values of all heap locations it might depend on. This set of locations is called the heap footprint of the function and is defined by its precondition. This snapshot is added as an additional argument to the application, giving the SMT solver a window to read out the necessary heap locations from the state when the application was made.

1.1.3 Snapshots for predicate instances

Similarly to a function application, a predicate instance covers some part of the heap which is stored in its heap snapshot. This part of the heap is hidden behind the predicate instance and cannot be modified unless it is explicitly taken out again. Thus, the heap described by its snapshot remains in its original state and serves exactly the same purpose as the one for a function application. So indeed, Silicon uses only one type of heap snapshot to cover these two use cases.

1.1.4 Generalizing: Snapshots for arbitrary assertions

More careful inspection of the last two sections reveals that we can formulate the general idea of a heap snapshot as follows:

A snapshot captures all current values of heap locations needed to evaluate an assertion

In the case of a function application, the assertion is the functions precondition and for a predicate instance it is the predicate body. Now, the word current in the above characterization indicates that a snapshot is always tied to a particular symbolic state and indeed the arguments for computing a snapshot are an assertion and a symbolic state.

1.1.5 Quantified versus non-quantified permissions

From a semantic perspective, non-quantified permissions can be considered to be a special case of quantified permissions (see [9], section 4.2.1). Still,
for snapshots Silicon currently makes a distinction between the two type of assertions. A lightweight snapshot representation is used in the absence of quantified permissions, while a second, more powerful one for quantified permissions is used only when needed. The price to pay for this optimization is some added complexity from having to deal with two different kinds of snapshots as well as their integration.

Until chapter 6, we will limit our discussion to the simpler case of non-quantified permissions and any reference to snapshots will implicitly be for this case. That is, we develop an alternative design for the lightweight snapshot representation which is heavily inspired by the one for quantified permission and part of the motivation for a new design is to do away with this separation completely.

1.1.6 Current snapshot representation and its limitations

From the above it follows that a snapshot should be some data structure combining symbolic values of heap locations. The choice of this data structure directly affects the shape of function axioms, since this is where values are read out from snapshots. Silicon’s original choice of binary trees as data structure to represent snapshots turns out to be suboptimal in this respect. Despite its apparent simplicity, reading out values from binary trees can translate to large, unintelligible terms.

Conceptually, the tree structure for an assertion is derived from its syntax tree by replacing each conjunction \( \&\& \) with an inner node with two sub-trees, accessibility predicates with symbolic values. For further details of the design and its axiomatization, see [9].

An example of a function axiom

In (ref malte), a small example involving conditional aliasing is used to describe the function axiomatization algorithm. This example is reproduced in listing 1.1.6 and we will use it to illustrate some of the difficulties of the structural design. The key point of this example is that the function body contains a field access \( z.val \) which is syntactically absent from the precondition. But using a semantic argument one can see that \( z \) either aliases \( x \) or \( y \), so in any case the permissions in the precondition guarantee that its value can be found in the functions snapshot.

```plaintext
1  field val: Int
2
3  function double (b: Bool, x: Ref, y: Ref, z: Ref): Int
4    requires acc(x.val) & acc(y.val)
5    requires b ? z == x : z == y
6  {  
7    b ? x.val + z.val : y.val + z.val
```
1. Introduction

Silicon currently computes the function axiom of \texttt{double} as:

\[
\forall s : \text{Snap}, b : \text{Bool}, x : \text{Ref}, y : \text{Ref}, z : \text{Ref}. \text{ite}(b, z = x, z = y) \\
\Rightarrow \text{double}(s, b, x, y, z) = \text{ite}(b, \\
\text{SnapToInt}(\text{fst}(s)) + \text{ite}(\neg b, \text{SnapToInt}(\text{fst}(\text{snd}(s))), \text{SnapToInt}(\text{fst}(s))), \\
\text{SnapToInt}(\text{fst}(\text{snd}(s))) + \text{ite}(\neg b, \text{SnapToInt}(\text{fst}(\text{snd}(s))), \text{SnapToInt}(\text{fst}(s)))
\]

This function axiom is computed in two phases. Silicon first checks that the function is well defined, i.e. it only accesses heap locations which its precondition allows for. Our informal argument from before about the access to \texttt{z.val} in listing 1.1.6 belongs to this phase. For any heap access in the function body, such an argument reveals where to find the corresponding access predicate in the precondition. And since the snapshot structure directly corresponds to the one of the precondition, this mapping effectively describes how to find values resulting from heap accesses in the function’s snapshot parameter. Thus it is stored and used to translate the function body in a second phase. As a consequence, the variables \texttt{x,y} and \texttt{z} do not appear in the axiom’s resulting term at all. Instead the translations are as follows:

<table>
<thead>
<tr>
<th>x</th>
<th>\text{SnapToInt}(\text{fst}(s))</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>\text{SnapToInt}(\text{fst}(\text{snd}(s)))</td>
</tr>
<tr>
<td>z</td>
<td>\text{ite}(\neg b, \text{SnapToInt}(\text{fst}(\text{snd}(s))), \text{SnapToInt}(\text{fst}(s)))</td>
</tr>
</tbody>
</table>

Not only does this indirect referencing of heap locations hinder readability, but can also carry over branching from the permission check into the definitional axiom. Generally speaking, an author of a Viper function (or a tool generating a Viper function) would need to learn a lot of implementation details to make sense of the resulting SMT function. More seriously, a considerable fraction of Silicon issues are caused by or at least involve function axioms and their computation making the SMT function axiom an important tool in the debugging process.

Independent of readability concerns, this representation has a known soundness issue. Since values from a snapshot are mapped to heap locations from the corresponding assertion structurally, it is required that a snapshot is only used with a single syntactic assertion. But with \textit{inhale-exhale assertions} this premise is not fulfilled, since this feature essentially allows a snapshot to be used with an arbitrary assertion. Presumably, an inhale-exhale assertion
[a1,a2] generated by a sensible Viper frontend allows using one snapshot covering a single heap footprint for both a1 and a2. Thus by empowering our snapshot data structure to map values to heap locations semantically instead of structurally, we can overcome this current limitation.

**Pair swap example**

A minimal example exposing this limitation is given in listing 1.1. The body of the predicate pair is an inhale-exhale assertion, which contains two field permissions in swapped order. It manifests as \( \text{acc}(x.f) \land \text{acc}(x.g) \) when inhaling the predicate body but as \( \text{acc}(x.g) \land \text{acc}(x.f) \) during an exhale operation. These two assertions are semantically equivalent \(^1\) and we expect the assertion in line 18 to hold.

```plaintext
1 field f : Int
2 field g: Int
3
4 predicate pair(x: Ref)
5 {
6   \[ \text{acc}(x.f) \land \text{acc}(x.g), \text{acc}(x.g) \land \text{acc}(x.f) \]\n7 }
8
9 method fold_unfold_pair(x: Ref)
10 {
11   inhale acc(x.f) \land acc(x.g)
12   x.f := 0
13   x.g := 1
14   fold pair(x)
15   unfold pair(x)
16   assert x.f == 0
17 }
```

Listing 1.1: Inhale-exhale assertions in predicate bodies can cause the current structural snapshot design to be unsound (Silicon issue #271 [8]).

But instead, the current representation of the snapshot as follows:

1. Line 15: consumes the predicate body yielding the snapshot pair(0,1).
   This snapshot is stored with the obtained predicate instance pair(x).

2. Line 16: The predicate instance is removed and its snapshot recovered.
   The snapshot pair(0,1) is used to produce the predicate body.

\(^1\)at least in the case all permissions are available which we always assume for our work; see 3.2.1.
1. Introduction

The mismatch comes from the fact that in step 1, the predicate body is \( \text{acc}(x.f) \land \text{acc}(x.g) \) while in step 2 it is \( \text{acc}(x.g) \land \text{acc}(x.f) \) but the same snapshot relying on the assertions structure is used for both. As a consequence, line 16 restores the value of \( x.f \) as 1 and of \( x.g \) as 0. This means that not only does the expected assertion fail, but the false assertion \( x.f == 1 \) can be proven. To avoid this unsoundness, Silicon disallows inhale-exhale assertions in predicate bodies and function preconditions.

1  field f : Int
2  field g: Int
3
4  function fun(x: Ref): Int
5    requires [acc(x.f) \land acc(x.g), acc(x.g) \land acc(x.f)]
6  { x.f }
7
8  method m(x: Ref)
9  {
10     inhale acc(x.f) \land x.f == 1
11     inhale acc(x.g) \land x.g == 0
12     assert fun(x) == 0
13  }

Listing 1.2: Based on the same problem as illustrated by listing 1.1, inhale-exhale assertions are not supported in function precondition position (Silicon issue #271 [8]).

The issue with function preconditions exploits the same problem, as illustrated in listing 1.2. The method \( m \) only exhales the function precondition when evaluating the term \( \text{fun}(x) \) in the assertion on line 12, so at first sight the inhale seems to be missing. But it occurs during the well-definedness check of the function \( \text{fun} \) itself. Essentially, this check inhales the function precondition into a fresh state and checks if this state contains enough permissions for all heap accesses in the function body. As explained above, this phase yields the mapping from heap accesses to expressions in the precondition which is necessary to formulate the function’s definitional axiom. It follows that two different assertions are used to compute the function axiom and compute a concrete snapshot for a function application. This mismatch results in the unsound verification of the assertion in line 12.

1.2 Notation

Any undertaking involving the translation of programs from one language into another necessarily introduces various syntactic entities in different languages denoting the same semantic object. In order to avoid cluttering the text with definitions or remarks concerning these relations which communicate very little to the reader, it is convenient to make them implicit using notational conventions. The following conventions are used hereafter:
First, we have three layers:

1. **Viper**: Entities of this layer are written in monospace font, e.g. \( x := f(y, 18) - 4 \) and represent the program as a syntactic object.

2. **SMT**: Its terms are written in as regular math formulas, e.g. \( x = f(y, 18) - 4 \). Any extended syntax which is not in direct correspondence with SMT-LIB will be explained upon first use. In the unfortunate case of confusion or ambiguity, please refer directly to the SMT-LIB version from our implementation in the appendix (ref).

3. **Symbolic execution in Silicon**: We use a custom math syntax described upon first use.

Second, we use the following mappings and shorthands:

- For each function `fun` in the Viper program, Silicon introduces a corresponding SMT level function which we denote by the same name `fun`.

- For each Viper type `T`, Silicon introduces a corresponding SMT sort which we denote by the same name `T`.

- Furthermore, for each Viper field `f` we denote the SMT sort corresponding to its type as `S_f`.

### 1.3 Outline

The objective of this project is a re-design of the snapshot representation used in Silicon which should alleviate the presented issues. In chapter 2 we will refine the requirements and introduce the basic design on the logical level. Chapter 3 presents the relevant symbolic execution rules of Silicon modified to the new representation. With these two chapters in place, we are ready to discuss triggering in chapter 4, making sure that our design not only works logically, but also practically.
Chapter 2

Heap snapshot representation in SMT

2.1 Requirements

The task at hand is to encode a partial heap described by some assertion into a snapshot term in SMT. Although Silicons background theory already contains a sort Snap for snapshots which implements the binary trees described earlier, we declare a new sort PHeap (short for partial heap). This is mainly for presentational purposes to avoid any confusion between the two designs. A complete implementation would most likely stick to the original name Snap since the term snapshot is already established as Viper terminology. Indeed, we will be using the term snapshot to describe our new design as well. Starting from the fresh sort PHeap, the following sections each take one requirement or design constraint and introduce the corresponding SMT entities and axioms.

In the following we will use the phrase all fields to mean all fields in the program under verification and similarly for predicates. These sets are always finite and can be determined statically (c.f. section 2.2.1).

2.1.1 Capturing field values

A snapshot should be able to store symbolic values resulting from field accesses. Thus, for each field $f$ we introduce a function to store the values of $x.f$ for any reference $x$:

$$\text{lookup}_f : \text{PHeap} \times \text{Ref} \rightarrow S_f$$

Note that by our notational conventions, $S_f$ denotes the sort of the field $f$. This function signature is a bit more subtle than it might appear at first sight since it already contains the main idea of using semantic instead of syntactic snapshots: For a fixed snapshot $h$ and field $f$, the domain of
the corresponding lookup function $\text{lookup}_f(h, \cdot)^1$ is $\text{Ref}$. As a consequence, looking up $x.f$ in a snapshot depends on the semantic $\text{Ref}$ object that $x$ evaluates to, but not on the syntactic variable $x$ itself (more generally the syntactic shape of the receiver expression).

Example

For an almost minimal example, consider the following function which accesses a field.

```plaintext
1  field f : Int
2
3  function getfneg(x: Ref): Int
4      requires acc(x.f)
5      { -x.f }

Listing 2.1: The value of a function application getfneg(x) depends on the value stored in the symbolic heap for location x.f
```

The function axiom of $\text{getfneg}$ should translate this access to a lookup in its snapshot parameter $h$, i.e. $\text{lookup}_f(h, x)$ (c.f. 3.3). How the symbolic value resulting from such a lookup is stored in a snapshot will be covered in section 2.2.

2.1.2 Monomorphic and polymorphic snapshots

Our design makes snapshots implicitly polymorphic, meaning that they can contain values of different sorts. That is, we declared a single sort $\text{PHeap}$ with different lookup functions for each field yielding values of different sorts. In fact, for each reference $x$ and field $f$, each snapshot $h$ contains a value, namely the term $\text{lookup}_f(h, x)$. The reason for this is that SMT functions are always total. Fortunately, this aligns with the fact that in Viper all objects have all fields, so we are not creating additional values.

In contrast, the current snapshot representation is monomorphic: Binary trees are defined over a single sort, and since there is no way of introducing a subtyping relation between sorts in SMT, this sort is chosen to be $\text{Snap}$. So to put a symbolic value of sort $T$ into such a tree, it needs to be converted into a term of sort $\text{Snap}$. The current design achieves this by providing a wrapper function for each sort $T$ that does exactly this embedding from $T$ to $\text{Snap}$. Thus, switching to the new design allows us to drop all of the sort wrappers, simplifying Silicon’s background theory and interaction with snapshots in the implementation.

---

$^1$This denotes the function $x \mapsto \text{lookup}_f(h, x)$ from $\text{Ref}$ to $S_f$.
2.1. Requirements

2.1.3 Capturing predicate instances

Predicate instances themselves do not represent symbolic values, but abstract over assertions. Translating to snapshots, this means each predicate instance represents a snapshot. This idea can be represented in the language of $PHeap$ terms by introducing a lookup function for predicate instances. An instance of a predicate $P$ is defined by a list of expressions $e_1, \ldots, e_n$ serving as arguments. A natural choice for our lookup function would be:

$$\text{lookup}'_P : PHeap \times \text{Pargsorts} \rightarrow PHeap$$

However, we will use a slightly modified version as will be explained in the next section and hence the prime in the name.

2.1.4 Support for framing

One of the main features of snapshots is that they enable framing (ref ref ref). That is, not only must the SMT solver be able to lookup values in a snapshot, but also infer what is and what is not in a snapshot. Considering that all our lookup functions are total (as SMT functions), any snapshot $h : PHeap$ yields all terms of the form $\text{lookup}_f(h, x)$ and thus actually represents a total heap. This mismatch between our concept of a snapshot as a partial heap and its axiomatization so far is revealed by Viper programs which rely on framing:

```
1 field f : Int
2 field g : Int
3
4 function fun(x: Ref): Bool
5   requires acc(x.f)
6
7 method m1(x: Ref)
8   requires acc(x.f) && acc(x.g)
9   {
10  var v : Bool
11  v := fun(x)
12  m2(x)
13  assert v == fun(x)
14   }
15
16 method m2(x: Ref)
17   requires acc(x.g)
18   ensures acc(x.g)
```

Listing 2.2: The value of $\text{fun}(x)$ is framed across the method call $m2(x)$.

Listing 2.2 illustrates a simple framing argument: From its precondition we know that the function $\text{fun}$ depends only on the location $x.f$. Addition-
2. Heap snapshot representation in SMT

ally, the precondition of \texttt{m2} restricts its view of the heap to just the location \texttt{x.g}.

From the perspective of the caller (method \texttt{m1}), the method call \texttt{m2(x)} can arbitrarily modify the value stored at \texttt{x.g}. So intuitively, executing this call makes \texttt{m1} lose all information about the value of \texttt{x.g}. Additionally, the call \texttt{m2(x)} implicitly leaves any part of the heap not included in its precondition unchanged. And since the caller knows the precondition of the method it calls, \texttt{m1} should be able to conclude that evaluating \texttt{fun(x)} again still yields the same value and thus prove the assertion in line 13. The crucial point here is that only knowing the value of \texttt{x.f} in the snapshot for \texttt{fun(x)} would not be enough for this proof to work, because we also need to know that \texttt{fun(x)} is independent of \texttt{x.g}.

One way to formalize this intuition is to make the functions \texttt{lookup\_f} and \texttt{lookup\_p} partial. We will then only consider the values mapped by the lookup functions to be in the snapshot\(^2\). But in SMT all functions are total, so we choose to model partiality using a manual encoding of function domains instead. For \texttt{lookup\_f}, a natural choice is to model the domain as a set of \texttt{Ref} terms. Sets are not available per default in SMT, but fortunately a parametric set formalization is already available in Silicon’s background theory (ref?) since Viper natively supports the type \texttt{Set[T]} for any Viper type \texttt{T}. For the purposes of this project we assume Silicon’s set axiomatization to be complete in the sense that any constants and functions like \texttt{∅}, \texttt{∪} and \texttt{∩} exist and behave as expected. This assumption is sensible since we have not encountered any missing constants, axioms or triggers during implementation. Based on this we will use standard set notation, for what are in fact constants and functions about sets from Silicon’s background theory. The sort of sets with elements of sort \texttt{T} is written as \texttt{Set[T]}. With this, we can define a snapshot’s domain per field as follows:

\[
dom_f : \texttt{PHeap} \rightarrow \texttt{Set[Ref]}
\]

Going back to our framing example from listing 2.2, a snapshot \texttt{h} for a function application of \texttt{fun(e)} should satisfy \texttt{dom\_f(h) = \{x\}} and \texttt{dom\_g(h) = ∅} where \texttt{x} is the reference that the function argument \texttt{e} evaluates to. Now, let us make the previous framing argument more precise using snapshot domains:

1. On line 9, a snapshot \texttt{h1} is computed for the function call. As explained above, it should satisfy \texttt{dom\_f(h1) = \{x\}} and \texttt{dom\_g(h1) = ∅}.

2. The method call on line 10 effectively removes any previous information about \texttt{x.g} by replacing it with a fresh symbolic value. However

\(^2\)We also refer to this as covered by the snapshot, contained in the snapshot or stored in the snapshot.
2.1. Requirements

since $x.f$ does not appear in its precondition, its symbolic value is preserved.

3. To check the assertion on line 11, a new snapshot $h_2$ is computed, again with $\text{dom}_f(h_2) = \{x\}$ and $\text{dom}_g(h_2) = \emptyset$. And since the symbolic value of $x.f$ has not been changed, the two snapshots also contain the same value, i.e. $\text{lookup}_f(h_1, x) = \text{lookup}_f(h_2, x)$.

4. Thus, comparing the two function applications $\text{fun}(h_1, x)$ and $\text{fun}(h_2, x)$ amounts to comparing $h_1, h_2$ which succeeds based on our understanding of $h_1$ and $h_2$ as partial functions. Of course, we will have to make sure that our axiomatization is strong enough to capture this intuition.

Framing for predicates

One can construct analogous examples to show that domains for predicates are also needed. From our first version of $\text{lookup}'_P$ from above we can see that the construction for fields does not apply directly, since a predicate domain consists of tuples while $\text{Set}[\cdot]$ is parametrized by a single sort. One option is to encode tuples into SMT which could be done conveniently enough using SMT-LIB datatypes (ref). However, borrowing an idea from [10] we can get away with a simple function instead:

$$
\text{loc}_P : P_{\text{args}} \rightarrow \text{Loc}
$$

Here $\text{Loc}$ is a new sort modeling abstract locations and instead of a tuple of arguments $(x_1, \ldots, x_n)$ the term $\text{loc}_P(x_1, \ldots, x_n)$ is put into a predicate domain. That is to say, predicate domains are sets of $\text{Loc}$ terms:

$$
\text{dom}_P : \text{PHeap} \rightarrow \text{Set}[\text{Loc}]
$$

To see why this is already sufficient, consider what information we need $\text{dom}_P$ to convey. For any snapshot $h$ and tuple of argument terms $t := (x_1, \ldots, x_n)$ for a predicate $P$, it should tell us whether $t \in \text{dom}_P(h)$ or not. A bit of reflection shows that the only property of tuples needed for this operation is the following implication over two tuples of arguments $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$:

$$
x_1 = y_1 \land \ldots \land x_n = y_n \Rightarrow (x_1, \ldots, x_n) = (y_1, \ldots, y_n)
$$

If, for the sake of argument, we consider the tuple constructor $(\cdot \cdot \cdot)$ to be a function (which it would be in an SMT formalization), then this implication

---

This is implied by the theory of uninterpreted functions.
is given by the theory of uninterpreted functions (ref). This is the reason why it is enough to use an uninterpreted function without any additional properties instead of tuples to obtain the desired property of $\text{dom}_p$. This solution is a bit more indirect since it is not formulated in terms of tuples but instead introduces a new sort and function without any properties. But compared to a solution with tuples, it does not introduce any (currently) unused properties and functions like projections.

### 2.1.5 On injectivity of the $\text{loc}_p$ function

So far we have introduced the following functions for a predicate $P$:

\[
\begin{align*}
\text{loc}_p &: \text{Argsorts} \rightarrow \text{Loc} \\
\text{dom}_p &: \text{PHeap} \rightarrow \text{Set}[\text{Loc}] \\
\text{lookup}_p &: \text{PHeap} \times \text{Argsorts} \rightarrow \text{PHeap}
\end{align*}
\]  

(2.1)

Now consider a scenario with a predicate $P$ with one argument and some snapshot $h : \text{PHeap}$ which satisfies $\text{dom}_p(h) = \{\text{loc}_p(x)\}$, i.e. contains exactly the instance $P(x)$. As explained above, framing usually relies on facts about what is and what is not in a snapshot. For example, for another argument $y$ it should follow from $x \neq y$ that $P(y)$ is not in the snapshot. But with the setup above, this inference is not possible:

\[
\text{dom}_p(h) = \{\text{loc}_p(x)\} \land x \neq y \not\Rightarrow \text{loc}_p(y) \notin \text{dom}_p(h)
\]  

(2.2)

The reason is simply that our constraints still allow $\text{loc}_p(x) = \text{loc}_p(y)$, although this is conceptually incorrect. This can be avoided by axiomatizing the injectivity of $\text{loc}_p$. However, we were unable to construct an example which relies on this snapshot based framing reasoning, because this is usually done based on permissions. Thus, we note the conceptual necessity of injectivity of $\text{loc}_p$, but omit it from our design.

### 2.2 Building snapshots from assertions

One way that snapshots are built is by taking the relevant parts of the symbolic heap and combining them into a $\text{PHeap}$ term. We now explain the correspondence between assertions and snapshots from section 1.1.4 by using this idea and following the structure of assertions from smaller into larger ones. Since assertions have a recursive structure, we start with atomic assertions and then consider operators that build compound ones.

---

4Of course, by $x$ and $y$ we mean the symbolic values stored in the program variables $x$ and $y$. 
2.2. Building snapshots from assertions

2.2.1 Atomic assertions

Field access predicate

Field locations are accessed from assertions via accessibility predicates. For a field \( f \) this specifies some receiver expression \( e \) of type \( \text{Ref} \) and a permission amount \( p \). To describe such a field access predicate \( \text{acc}(e.f, p) \), for each field \( f \) we declare a function which acts as a constructor for a snapshot containing mapping a single location to a symbolic value.

\[
\text{singleton}_f : \text{Ref} \times S_f \rightarrow P\text{Heap}
\]

Since we treat the domain and lookups for each field and predicate separately, a complete axiomatization of such a singleton heap needs to define all of them. In the following we will do this by writing forall-quantifiers over fields and predicates, although Silicon’s background theory does not allow this quantification. However the set of fields and predicates is always finite and statically known, so we can understand a quantifier over fields or predicates as a shorthand on the presentation level for the corresponding finite conjunction (ref appendix?).

Axiom 1 (Field domain of field singleton)

\[
\forall f : \text{Field}, x : \text{Ref}, v : S_f. \quad \text{dom}_f(\text{singleton}_f(x,v)) = \{x\}
\]

Axiom 2 (Other field domains of field singleton)

\[
\forall f, g : \text{Field}, x : \text{Ref}, v : S_f. \quad g \neq f \Rightarrow \text{dom}_g(\text{singleton}_f(x,v)) = \emptyset
\]

Axiom 3 (Field lookup in field singleton)

\[
\forall f : \text{Field}, x : \text{Ref}, v : S_f. \quad \text{lookup}_f(\text{singleton}_f(x,v), x) = v
\]

Axiom 4 (Predicate domains of field singleton)

\[
\forall f : \text{Field}, P : \text{Predicate}, x : \text{Ref}, v : S_f. \quad \text{dom}_P(\text{singleton}_f(x,v)) = \emptyset
\]

In axiom 2 we used another bit of presentation level syntax over fields to express that the domain of every other field is empty and the logical equivalent can be found in the appendix (ref).
Predicate access predicate

Similarly, access to a predicate instance is written as acc(P(\(\overline{\ell}\)), p). Here \(\overline{\ell}\) stands in for a sequence of expressions of \(P\)'s argument types, whose corresponding sorts we denote as \(\overline{P_{argsorts}}\). As motivated earlier, a predicate instance represents some part of the heap which our formalization captures as a snapshot. So the snapshot corresponding to a predicate instance is constructed using a function \(\text{singleton}_P : \overline{P_{argsorts}} \times PHeap \rightarrow PHeap\) which is axiomatized much in the same way:

**Axiom 5 (Predicate domain of predicate singleton)**
\[
\forall P : \text{Predicate}, \overline{P_{args}} : \overline{P_{argsorts}}, h : PHeap. \quad \text{dom}_P(\text{singleton}_P(\overline{P_{args}}, h)) = \{\text{loc}_P(\overline{P_{args}})\}
\]

**Axiom 6 (Other predicate domain of predicate singleton)**
\[
\forall P, Q : \text{Predicate}, \overline{P_{args}} : \overline{P_{argsorts}}, h : PHeap. \quad Q \neq P \Rightarrow \text{dom}_Q(\text{singleton}_P(\overline{P_{args}}, h)) = \emptyset
\]

**Axiom 7 (Predicate lookup in predicate singleton)**
\[
\forall P : \text{Predicate}, \overline{P_{args}} : \overline{P_{argsorts}}, h : PHeap. \quad \text{lookup}_P(\text{singleton}_P(\overline{P_{args}}, h), \overline{P_{args}}) = h
\]

**Axiom 8 (Field domains of predicate singleton)**
\[
\forall P : \text{Predicate}, f : \text{Field}, \overline{P_{args}} : \overline{P_{argsorts}}, h : PHeap. \quad \text{dom}_f(\text{singleton}_P(\overline{P_{args}}, h)) = \emptyset
\]

**Pure assertion**

A pure assertion does not contain any heap references and is thus captured by a constant \(\text{emp} : PHeap\) representing the empty heap:

**Axiom 9 (Field domains of emp)**
\[
\forall f : \text{Field}. \quad \text{dom}_f(\text{emp}) = \emptyset
\]

**Axiom 10 (Predicate domains of emp)**
\[
\forall P : \text{Predicate}. \quad \text{dom}_P(\text{emp}) = \emptyset
\]

### 2.2.2 Compound assertions

Larger assertions are built from these atomic ones using Viper's boolean operators:
2.2. Building snapshots from assertions

Conjunction
The conjunction of two assertions in Viper is written as $a_1 \&\& a_2$ and we inductively define its snapshot to be a combination of the snapshots of the $a_1$ and $a_2$. The function $\text{combine}: \text{PHeap} \times \text{PHeap} \rightarrow \text{PHeap}$ is used to build this combination and its properties are axiomatized as follows:

Axiom 11 (Field domain of combination)
\[
\forall f : \text{Field}, h_1, h_2 : \text{PHeap}. \\
\text{dom}_f(\text{combine}(h_1, h_2)) = \text{dom}_f(h_1) \cup \text{dom}_f(h_2)
\]

Axiom 12 (Field lookup of combination)
\[
\forall f : \text{Field}, h_1, h_2 : \text{PHeap}, x : \text{Ref}. \\
x \in \text{dom}_f(h_1) \Rightarrow \text{lookup}_f(\text{combine}(h_1, h_2), x) = \text{lookup}_f(h_1, x)
\]

Axiom 13 (Predicate domain of combination)
\[
\forall P : \text{Predicate}, h_1, h_2 : \text{PHeap}. \\
\text{dom}_P(\text{combine}(h_1, h_2)) = \text{dom}_P(h_1) \cup \text{dom}_P(h_2)
\]

Axiom 14 (Predicate lookup of combination)
\[
\forall f : \text{Field}, h_1, h_2 : \text{PHeap}, P_{\text{args}} : P_{\text{argsorts}}. \\
\text{loc}_P P_{\text{args}} \in \text{dom}_P(h_1) \Rightarrow \text{lookup}_P(\text{combine}(h_1, h_2), \text{loc}_P P_{\text{args}}) = \text{lookup}_P(h_1, \text{loc}_P P_{\text{args}})
\]

The axioms above are asymmetric in the sense that they only relate a combination with its left argument, but not its right one. Instead of spelling out the symmetric axioms for the right argument, we chose to add a general commutativity axiom for $\text{combine}$ such that the axioms for the left argument suffice.

Axiom 15 (Commutativity of combine)
\[
\forall h_1, h_2 : \text{PHeap}. \quad \text{combine}(h_1, h_2) = \text{combine}(h_2, h_1)
\]

Other boolean connectives and branching
Besides conjunction, the Viper language contains more standard boolean connectives such as conditionals and disjunction. For each one, we will now briefly argue why there is no need to cover it in our work.

- A disjunction $a_1 \mid\mid a_2$ is only allowed between pure assertions $a_1$ and $a_2$. Consequentially, all disjunctions fall into the pure assertion category and are described by the snapshot $\text{emp}$. 
2. Heap snapshot representation in SMT

- A conditional expression \( b \ ? \ a_1 : a_2 \) case splits over a pure assertion \( b \). Silicon applies the following strategy: If \( a_1 \) or \( a_2 \) contain recourse assertions, the symbolic execution branches over \( b \) yielding two executions where the conditional has been replaced with \( a_1 \) and \( a_2 \) respectively. Otherwise it is pure and captured by the snapshot \( emp \).

- Implications are of the form \( b \Rightarrow a \), where \( b \) is a pure assertion. Since this is semantically equivalent (and in fact implemented as) to \( b \ ? \ a : true \), it is already covered.

- An inhale-exhale assertion \([a_1,a_2]\) always presents itself as either \( a_1 \) or \( a_2 \) to the symbolic execution based on the context. Thus its snapshot is already defined by what is explained above.
3.1 Evaluating function applications

Building on the previous chapter, we will now describe how to adapt the symbolic execution rules of Silicon involving snapshots. From the perspective of Viper programs, this covers fold and unfold statements, unfolding expressions as well as functions including recursion. As before, we will take a purely logical perspective in this chapter, postponing the discussion of triggers to chapter 4.

3.2 Symbolic Execution Rules

We already explained the mapping between assertions and snapshots informally (c.f. 1.1.4 and 2.2). We will now make this idea more precise by adapting the symbolic execution primitives consume and produce to the new snapshot design. Then, using these two as building blocks, the rules for folding and unfolding predicates need very little modification themselves. And third, the translation from a function body to its definitional axiom is explained. Concluding, we use a few example functions to compare their new axioms with the ones based on the structural snapshot representation.

3.2.1 Symbolic execution notation

The symbolic execution rules are given in a notation similar to an imperative programming language. While this deviates from the original formalization (ref) which uses a continuation passing style, this syntax conveniently abstracts over some parts of symbolic execution which are not relevant for our work. We briefly explain the individual operations upon first use. In addition, we will always assume all permissions to be available in the symbolic heap. If this is not the case, verification will fail earlier and never apply the corresponding symbolic execution rule.
In the following we will have to refer to symbolic heap chunks storing field values or predicate instances. Our notation for a field chunk is $[x.f \mapsto v]$, meaning that $v$ is the value of the field $f$ of reference $x$. Similarly, a predicate chunk $[P(args) \mapsto h]$ describes a predicate instance with a snapshot $h$.

### 3.2.2 Consuming assertions

The symbolic execution rule `consume` takes a symbolic state, matches it against an assertion and computes the resulting snapshot. Intuitively, consuming an assertion corresponds to checking that the assertion holds. It also removes all matching heap chunks and potentially adds path constraints to the symbolic state. But these effects are independent of the snapshot design so we omit them from our description. Thus, the `consume` function below is actually just the part we change and can be specified as a function taking one assertion argument and returning a snapshot. Keeping the analogy with an imperative program, one can think of `consume` as a method of a class describing mutable, symbolic states such that it has access to any part of the symbolic state, in particular the symbolic heap.

**Field access**

Consuming a field access must check that the symbolic heap contains the referenced location and store its symbolic value. This is done by searching the symbolic heap for the location's symbolic value and returning it as a singleton snapshot. How exactly this heap chunk can be found is orthogonal to our concerns, and further specification of this procedure can be found in (malte ref).

```plaintext
consume(acc(e.f, p)):
  x := eval(e)
  find field chunk of shape [x.f \mapsto v]
  return singleton_f(x,v)
```

This snapshot communicates two pieces of information, which can both be obtained by applying the axioms about field singletons to it. First, it says that the assertion accesses only the location $e.f$ by specifying all involved domains (axioms 1, 2, 4). Second, via the $lookup_f$ function the symbolic value $v$ of $e.f$ from the symbolic heap used to consume this assertion is also stored (axiom 3) in the snapshot.
Predicate access

Access to a predicate instance is stored in the symbolic heap as a predicate chunk. Just like with a field access, translating this to a snapshot results in a singleton heap.

\[
\text{consume}\left(\text{acc}(P(\text{args}), p)\right):
\]
\[
t\text{Args} := \text{eval}(\text{args})
\]
\[
\text{find predicate chunk of shape } [P(t\text{Args}) \mapsto h]
\]
\[
\text{return } \text{singleton}_p(t\text{Args}, h)
\]

Combination

The semantics of a conjunction \( a_1 \land a_2 \) in Viper are such that both \( a_1 \) and \( a_2 \) must be satisfied by permission-compatible heap parts. Essentially this means that for any field location \( l \) contained in both assertions, the symbolic heap must contain at least the sum of all permissions to \( l \) contained in \( a_1 \) and \( a_2 \) and both heap parts must agree on its value. This is implemented in the symbolic execution rule by consuming the two operands one after another. But this aspect is not affected by the snapshot design at all and only the way the two snapshots for \( a_1 \) and \( a_2 \) are combined needs to be adapted to the new design:

\[
\text{consume}(a_1 \land a_2):
\]
\[
h_1 := \text{consume}(a_1)
\]
\[
h_2 := \text{consume}(a_2)
\]
\[
\text{return } \text{combine}(h_1, h_2)
\]

Since the permissions are checked separately, the combined snapshot is only computed in the case where both recursive consume calls succeed. By our explanation above this implies that the two snapshots \( h_1 \) and \( h_2 \) describe compatible partial heaps. Finally the combine function will give the union of the two as partial functions, as required by the semantics of \( a_1 \land a_2 \).

3.2.3 Producing assertions

Intuitively, producing an assertion from a snapshot means assuming that the assertion holds for the heap part described by the snapshot. To be the symmetric operation to consume, the state after produce\((a, h)\) should also have the information that \( h \) exactly describes \( a \), since consume always produces a snapshot matching the assertion exactly.
3. Symbolic Execution Rules

The fact that a snapshot exactly describes an assertion is encoded by its domains in our design. To show the necessity for this information to be learned via `produce`, consider listing 3.2.3. The `inhale` statement in line 11 translates to a `produce` call which yields the predicate instance \( P(x, y) \) and a function application \( f(x, y) \) based on it. To frame the value of this function application across the assignment \( z.g := 7 \) in line 12, information about the corresponding snapshots domains is required.

```plaintext
1    field g: Int
2
3    predicate P(x: Ref, y: Ref) {  
4      acc(x.g) && acc(y.g)
5    }
6
7    function f(x: Ref, y: Ref) : Int
8      requires P(x, y)
9
10   method m(x: Ref, y: Ref, z: Ref) {  
11      inhale P(x, y) && f(x, y) == 42 && acc(z.g)
12      unfold P(x, y)
13      z.g := 7
14      fold P(x, y)
15      assert f(x, y) == 42
16   }
```

### 3.2.4 Field access

Instantiating our general description of `produce` for a field access means that the stored symbolic value should be added to the symbolic heap and it should be learned that the snapshot covers exactly that field lookup.

\[
\text{produce}(\text{acc}(e.f, \ p), h) : \\
x := \text{eval}(e) \\
\text{add field chunk } [x.f \mapsto \text{lookup}_f(h, x)] \\
\text{assume } h = \text{singleton}_f(x, \text{lookup}_f(h, x))
\]  

A sidenote on the last equality: At first sight, it might look a bit suspicious to define \( h \) in terms of \( h \). However, applying axiom 1 and 3 yields:

\[
\text{dom}_f(h) = \{x\} \\
\text{lookup}_f(h, x) = \text{lookup}_f(h, x)
\]
3.2. Symbolic Execution Rules

This is just the right information, namely we did not learn anything new about \( \text{lookup}_f(h,x) \) but learned the domain of \( h \)\(^1\). The reason why chose this seemingly more complicated expression is that it matches our triggers better (see chapter 4).

3.2.5 Predicate access

Just like with \text{consume}, the mechanism for predicate instances is almost the same as for field accesses. The only difference is the \( \text{loc}_P \) function which is used to perform the lookup operation.

\[
\text{produce}(\text{acc}(P(\text{args}), p), h):
\]

\[
t\text{Args} := \text{eval}(\text{args})
\]

\[
\text{add predicate chunk } [P(t\text{Args}) \mapsto \text{lookup}_P(h, \text{loc}_P(t\text{Args}))]
\]

\[
\text{assume } h = \text{singleton}_P(t\text{Args}, \text{lookup}_P(h, \text{loc}_P(t\text{Args})))
\]

3.2.6 Combination

This case is a bit more subtle than the others, but fortunately the solution is simple. The assertions \( a_1 \) and \( a_2 \) both describe some part of the heap and we would like to say that \( a_1 \land a_2 \) describes the combination. This is exactly what we defined \text{combine} for, but what should be applied here? The idea is to say \textit{there exists a snapshot } \( h_1 \) \textit{describing } \( a_1 \) and learn its structure inductively by a recursive call to \text{produce}. Similarly we create \( h_2 \) for \( a_2 \) and can finally state that the shape of \( h \) is \text{combine}(\( h_1 \), \( h_2 \)).

\[
\text{produce}(a_1 \land a_2, h):
\]

\[
fresh h_1, h_2
\]

\[
\text{produce}(a_1, h_1)
\]

\[
\text{produce}(a_2, h_2)
\]

\[
\text{assume } h = \text{combine}(h_1, h_2)
\]

An inductive argument shows that, again this produces the same information: Assume that all relevant lookups from \( h \) are propagated to \( h_1 \) and \( h_2 \) and their domains match the assertions \( a_1 \) and \( a_2 \), that is to say the recursive calls to \text{produce} reveal all information about \( h_1 \) and \( h_2 \). Then, by axioms 11 and 13 the correct domain of the combined snapshot \( h \) can be inferred. Based on this, axioms 12 and 14 lift all lookups from \( h_1 \) and \( h_2 \) onto \( h \).

\(^1\)The domain could already be known before this assumption through explicit construction of \( h \) as singleton or combination. Logically, in such a case the assumption will have no effect (c.f. chapter 6).
3. Symbolic Execution Rules

3.2.7 Predicate folding and unfolding

Viper predicates provide abstraction over heap parts since, as mentioned before, a predicate instance stands for some partial heap. The operations used to introduce and look inside the abstraction are called fold and unfold respectively. More concretely, they exchange access to a predicate instance for access to the heap part it represents and thus modify the symbolic heap. The symbolic execution rules for both statements are formulated in terms of consume and produce. This modular design makes it easy enough to adapt them to the new snapshot design.

For fold, first the partial heap represented by the predicate body is consumed. This will remove it from the symbolic heap and return a snapshot representing the same information in turn. Here $P(args)_\text{body}$ stands in for the predicate’s body with the arguments $args$ substituted for its formal parameters. The second step adds a predicate instance storing this snapshot to the symbolic heap.

\[
\begin{align*}
\text{exec(fold } P(args)) : \\
h &:= \text{consume}(P(args)_\text{body}) \\
tArgs &:= \text{eval}(args) \\
\text{produce}(P(args), \text{singleton}(tArgs,h))
\end{align*}
\]

With fold being the abstraction operation, unfold works in the opposite direction. It first removes access to a predicate instance from the symbolic heap using consume. This yields the snapshot behind the abstraction. Then it restores the partial heap stored in the predicate instance from the snapshot by producing its body.

\[
\begin{align*}
\text{exec(unfold } P(args)) : \\
h &:= \text{consume}(P(args)) \\
tArgs &:= \text{eval}(args) \\
\text{produce}(P(args)_\text{body}, \text{lookup}(P,h,tArgs))
\end{align*}
\]

The symmetry between the two rules is now apparent: $P(args)$ and $P(args)_\text{body}$ simply swapped places and their snapshots are connected via one level of $\text{lookup}_P/\text{singleton}_P$ indirection.

\[2\]One can check that $\text{produce}(P(args), \text{singleton}_P(tArgs,h))$ is in fact equivalent to manually adding a predicate chunk $[P(tArgs) \rightarrow h]$. We still chose this wording to emphasize the symmetry between consume and produce.
3.2. Symbolic Execution Rules

3.2.8 Unfolding

Conceptually, evaluating an unfolding expression means executing the corresponding unfold statement, evaluating in the resulting state and using the resulting value with the old state. Thus, after adapting unfold there is nothing to add here.

Fixed example: Pair swap

Recall the examples from chapter 1 where the structural snapshot representation causes unsoundness. Listing 3.1 reproduces the example from listing 1.1 with the symbolic heap chunks computed by the rules from this chapter inlined as code comments.

```plaintext
field f : Int
field g: Int
predicate pair(x: Ref)
{
    [acc(x.f) && acc(x.g), acc(x.g) && acc(x.f)]
}
method fold_unfold_pair(x: Ref)
{
    inhale acc(x.f) && acc(x.g)
x.f := 0
x.g := 1
// [x.f ↦→ 0], [x.g ↦→ 1]
fold pair(x)
// [P(x) ↦→ combine(sing_g(x, 0), sing_f(x, 1))] unfold pair(x)
// [x.f ↦→ lookup_f(combine(sing_g(x, 0), sing_f(x, 1)), x)]
// [x.g ↦→ lookup_g(combine(sing_g(x, 0), sing_f(x, 1)), x)]
// Fails
assert x.f == 0
// Succeeds
assert x.f == 1
}
```

Listing 3.1: Sound behavior of inhale-exhale assertions in predicate bodies as a result of our adapted snapshot design.

To see why the assertions in line 23 and 26 now behave in the sound, expected way it is enough to see what \( x.f \) evaluates to:
Symbolic Execution Rules

\[ \text{eval}(x.\xi) = \text{lookup}_f(\text{combine}(\text{sing}_g(x,0),\text{sing}_f(x,1)), x) \]
\[ = \text{lookup}_f(\text{combine}(\text{sing}_f(x,1),\text{sing}_g(x,0)), x) \]
\[ = \text{lookup}_f(\text{sing}_f(x,1), x) \]
\[ = 1 \]  

(3.4)

Here we used the axioms 12 and 15 to evaluate the term \( \text{lookup}_f(\text{sing}_f(x,1), x) \) under the condition that \( x \in \text{dom}_f(\text{sing}_f(x,1)) \) which follows from axiom 1. The value 1 is then obtained through axiom 3.

### 3.3 Function axiomatization

The function evaluation rule itself does not need adaption, but for the sake of consistency we present it in our syntax:

\[ \text{eval}(\xi(\text{args})) : \]
\[ 
\begin{align*}
  h & := \text{consume}(\xi(\text{args})_{\text{pre}}) \\
  t\text{Args} & := \text{eval}(\text{args}) \\
  \text{return} & \ f(h, t\text{Args})
\end{align*}
\]

Here \( \xi(\text{args})_{\text{pre}} \) stands in for the Viper assertion obtained by substituting the arguments \( \text{args} \) for the formal parameters in \( \xi \)'s precondition. According to the \textit{consume} rules given above, the domain of this snapshot covers all heap accesses in the function body. It is worth repeating that the function body is not evaluated by the symbolic execution engine, but instead the rule returns a symbolic function application (c.f. 1.1.1).

#### 3.3.1 Definitional axiom

The function body only plays a role for the definitional axiom which equates symbolic function applications with a corresponding term. We describe the construction of this term as a translation function \( tr \) mapping expressions to SMT terms, yielding the following axiom for a Viper function \( \xi \):

\[ \forall h : P\text{Heap}, f_{\text{args}} : f_{\text{argsorts}}. \quad f(h, x) = tr(\xi_{\text{body}}, h) \]

As usual, the overlined term stands in for quantification over variables of sorts determined by the type of \( \xi \). This simply means that the axiom can be instantiated for any snapshot and set of function arguments to rewrite
3.3. Function axiomatization

the symbolic function application to $tr(f_{\text{body}}, h)$. The translation of heap-independent expressions is unchanged and we will again only describe the cases that change.

**Field access**

A field access $e.f$ is translated to the term $\text{lookup}_f(h, x)$ where $x$ is the reference the receiver $e$ translates to:

$$tr(e.f, h):$$
$$x := tr(e, h)$$
$$\text{return } \text{lookup}_f(h, x)$$

As an example, consider how this rule composes with itself for nested field accesses.

```
1 field val : Int
2
3 field next : Ref
4
5 function nextVal(n: Ref) : Int
6     requires acc(n.next) && acc(n.next.val)
7     n.next.val
8 }
9
Listing 3.2: Nested field accesses inside a function body which need to be translated into the functions definitional axiom.
```

The translated function body for `nextVal` in listing 3.2 axiom can be obtained as:

$$tr(n.next.val, h) = \text{lookup}_{\text{val}}(h, tr(n.next, h))$$
$$= \text{lookup}_{\text{val}}(h, \text{lookup}_{\text{next}}(h, tr(n, h)))$$
$$= \text{lookup}_{\text{val}}(h, \text{lookup}_{\text{next}}(h, n))$$

The first two equalities are direct instantiations of our translation rule for field accesses, while the last equality translates the (heap-independent) formal Viper parameter $n$ to the formal parameter $n$ of the corresponding SMT function. Quantifying this over all function arguments and snapshots yields the function axiom for the symbolic function `nextVal`: 

27
∀ ℎ: PHeap, ℎ : Ref. nextVal(ℎ, ℎ) = lookupval(ℎ, lookupnext(h, ℎ))

**Unfolding**

The way predicate instances can occur in a function body is inside an expression of the form unfolding acc(ℙ(args), p) in e. This expression’s semantics are such that e is evaluated in a state where ℙ(args) has been unfolded temporarily without affecting the resulting state. Thus e can access any heap location captured by the predicate instance in addition to what is already accessible outside of the unfolding, except the instance ℙ(args).

Since the reasoning behind the final rule is based on triggers, in this chapter we only present a first approximation tr′ to convey the basic idea (c.f. 4.4). As a first approximation, consider the following seemingly direct translation of the semantics of unfolding:

\[
tr′(\text{unfolding acc(ℙ(args), p) in e, ℎ}) \colon \\
tArgs := \text{translate(args, ℎ)} \quad \text{return } tr′(e, \text{combine}(ℎ, \text{lookup}(ℙ(ℎ, \text{loc}(tArgs)))))
\]

This uses the combined snapshot of ℎ and whatever partial heap is stored behind the predicate instance ℙ(args), so it covers all heap accesses in e. But at first sight, this might seem incorrect since the snapshot used to translate e still has access to the instance ℙ(args) through ℎ. To put it more formally, let ℎ′ be the snapshot used to translate e. Then the ℎ′ constructed by the rule above satisfies \(\text{loc}(tArgs) \in \text{dom}(ℎ')\), which it should not according to the semantics of unfolding.

While this makes it seem unsound, in fact it is not. The reason is the aforementioned well-definedness check of the function body. Put succinctly, any Viper function exploiting our over-approximation of the snapshot to translate e would fail the permissions check during verification of the function body. The argument is a bit more delicate when considering fractional permissions to the predicate instance but still works out.

Interestingly enough, while this over-approximation is sound and to our knowledge no source of incompleteness we still be forced to render it more precisely for triggering reasons, hence the prime (c.f. 4.4).

**Function applications**

A similar question of over-approximation arises for function applications inside function bodies. Since Viper function applications translate to sym-
3.3. Function axiomatization

In function applications, the translation rule must specify its snapshot argument. During regular evaluation this snapshot is a result of consuming the precondition from the current symbolic heap (3.1). But in the current context, the snapshot needs to be computed from another snapshot.

```plaintext
function f(x: Ref) : Int {
  g(x)
}
```

Listing 3.3: Function application inside a function body which needs to be translated into the functions definitional axiom.

For example, translating the body of \( f \) in listing 3.3, requires the snapshot for \( g(x) \) to be computed from the snapshot parameter in \( f \)'s axiom. A similar argument with the well-definedness check as for unfolding can be made to argue that an over-approximating snapshot is sound. Unfortunately in this case over-approximation results in incompletenesses which can be avoided by computing a precise snapshot.

To illustrate this, a minimal example is given in listing 3.3.1. Essentially, a function value \( \text{one}(x) = 1 \) for a small snapshot is learned and needs to be reused inside a function with a bigger footprint. If the function \( \text{two} \) would pass its complete snapshot to \( \text{one}(x) \), then it would not obtain the same symbolic function application and be unable to prove the assertion in line 17. Using \( x_f \) and \( x_g \) for the symbolic values of \( x.f \) and \( x.g \) respectively, the two function applications of \( \text{one}(x) \) would be translated to:

\[
\begin{align*}
\text{one}(\text{singleton}_g(x, x_g), x) & \quad (\text{line } 15) \\
\text{one}(\text{combine}(\text{singleton}_f(x, x_f), \text{singleton}_g(x, x_g)), x) & \quad (\text{line } 17)
\end{align*}
\]

As a result, the SMT solver cannot reuse the previous knowledge of \( \text{one}(x) = 1 \) inside the body of \( \text{two} \) because an imprecise snapshot was passed.

```plaintext
field f : Int
field g : Int

function one(x: Ref) : Int
  requires acc(x.g)

function two(x: Ref) : Int
  requires acc(x.f) && acc(x.g)
  { one(x) }

method m(x: Ref) {
```

```
3. Symbolic Execution Rules

\[
\begin{align*}
&\text{inhale acc}(x,f) \land \text{acc}(x,g) \\
&\text{inhale} \ one(x) == 1 \\
&\text{assert} \ two(x) == 1
\end{align*}
\]

Our solution is to add a \( \text{restrict}_\text{fun} : P\text{Heap} \times \text{fun}abs \rightarrow P\text{Heap} \) function for every Viper function \( \text{fun} \). The idea is that it takes a snapshot and returns the part of it that describes the functions precondition. For our example, these two functions are axiomatized as follows:

\[
\begin{align*}
\forall h : P\text{Heap}, x : \text{Ref}. \quad & \text{restrict}_\text{one}(h, x) = \text{singleton}_g(x, \text{lookup}_g(h, x)) \quad (3.5) \\
\forall h : P\text{Heap}, x : \text{Ref}. \quad & \text{restrict}_\text{two}(h, x) = \text{combine} (\text{singleton}_g(x, \text{lookup}_g(h, x)), \text{singleton}_f(x, \text{lookup}_f(h, x))) \quad (3.6)
\end{align*}
\]

Now, assuming that the function axiom of \( \text{two} \) passes the restricted snapshot \( \text{restrict}_\text{one}(h, x) \) to \( \text{one} \), the proof succeeds:

\[
\begin{align*}
two(\text{combine}(\text{sing}_f(x, x_f), \text{sing}_g(x, x_g)), x) \\
= \text{one}(\text{restrict}_\text{one}(\text{combine}(\text{sing}_f(x, x_f), \text{sing}_g(x, x_g)), x)) \\
= \text{one}(\text{sing}_g(x, \text{lookup}_g(\text{combine}(\text{sing}_f(x, x_f), \text{sing}_g(x, x_g)), x)), x) \\
= \text{one}(\text{sing}_g(x, \text{lookup}_g(\text{sing}_g(x, x_g), x)), x) \\
= \text{one}(\text{sing}_g(x, x_g), x)
\end{align*}
\]

These equalities follow from the following axioms:

- (3.9): Definitional axiom of \( \text{two} \)
- (3.10): Axiom for \( \text{restrict}_\text{one} \) (3.5)
- (3.11): Symmetry of combination (Axiom 15)
- (3.13): Field lookup of singleton (Axiom 3)

The axioms for these restriction functions can be computed purely syntactically from the preconditions and we omit a detailed description since it is just another variant of the correspondence between assertions and snapshots.
we have seen before. Finally we can give the rule for translating function applications using this restriction function:

\[
\text{tr}(g(\text{args}), h) : \\
t\text{Args} := \text{translate}(\text{args}, h) \\
\text{return } g(\text{restrict}_g(h, t\text{Args}), t\text{Args})
\]
4.1 Triggers in SMT solvers

So far we have presented our design as a mathematical construction and focused on its properties and what it allows to prove. Preparing this theory for an SMT solver is mostly a syntactic translation into SMT-LIB code [1]. Additionally, it is necessary to choose trigger terms for universal quantifiers in order to guide the SMT solver to the right instantiations. Basically, a trigger says that whenever the solver obtains a term of a certain syntactic form, it should infer a corresponding instantiation of the quantified formula. The mechanism is detailed in [4]. The goal of this chapter is to give triggers for all our existing axioms and reason about their behavior and safety, focusing in particular on matching loops [2].

4.2 Triggering our existing axioms

This section restates all axioms introduced earlier, only this time with their triggers added. The triggers are written in curly braces after the quantified variables and before the quantified formula. Recall that our axioms contain quantifiers over fields and predicates which, as explained before, are removed during the translation to SMT. Thus, these quantifiers are not covered by our triggers.

4.2.1 Field singleton axioms

The triggers for the axioms about field singleton snapshots are quite simple to trigger. They allow the SMT solver to simplify the domains and lookup of field singletons yielding only new set terms. And thus, since none of our triggers involve any set terms, we argue that these triggers will not cause matching loops.
4. Additional Axioms and Triggers

Axiom 1′
\[ \forall f : \text{Field}, x : \text{Ref}, v : S_f \{ \text{dom}_f(\text{singleton}_f(x,v)) \}. \]
\[ \text{dom}_f(\text{singleton}_f(x,v)) = \{ x \} \]

Axiom 2′
\[ \forall f, g : \text{Field}, x : \text{Ref}, v : S_f \{ \text{dom}_g(\text{singleton}_f(x,v)) \}. \]
\[ g \neq f \Rightarrow \text{dom}_g(\text{singleton}_f(x,v)) = \emptyset \]

Axiom 3′
\[ \forall f : \text{Field}, x : \text{Ref}, v : S_f \{ \text{lookup}_f(\text{singleton}_f(x,v), x) \}. \]
\[ \text{lookup}_f(\text{singleton}_f(x,v), x) = v \]

Axiom 4′
\[ \forall P : \text{Predicate}, f : \text{Field}, x : \text{Ref}, v : S_f \{ \text{dom}_P(\text{singleton}_f(x,v)) \}. \]
\[ \text{dom}_P(\text{singleton}_f(x,v)) = \emptyset \]

4.2.2 Predicate singleton axioms

With the exception of the \( \text{loc}_P \) function, predicate singletons are axiomatized just like field singletons. This modification presents no problem for triggering so we use the same triggers as above.

Axiom 5′
\[ \forall P : \text{Predicate}, \overline{P} : \text{Pargsorts}, h : \text{PHeap} \{ \text{dom}_P(\text{singleton}_P(\overline{P}, h)) \}. \]
\[ \text{dom}_P(\text{singleton}_P(\overline{P}, h)) = \{ \text{loc}_P(\overline{P}) \} \]

Axiom 6′
\[ \forall P, Q : \text{Predicate}, \overline{P} : \text{Pargsorts}, h : \text{PHeap} \{ \text{dom}_Q(\text{singleton}_P(\overline{P}, h)) \}. \]
\[ Q \neq P \Rightarrow \text{dom}_Q(\text{singleton}_P(\overline{P}, h)) = \emptyset \]

Axiom 7′
\[ \forall P : \text{Predicate}, \overline{P} : \text{Pargsorts}, h : \text{PHeap} \{ \text{lookup}_P(\text{singleton}_P(\overline{P}, h), \overline{P}) \}. \]
\[ \text{lookup}_P(\text{singleton}_P(\overline{P}, h), \overline{P}) = h \]

Axiom 8′
\[ \forall P : \text{Predicate}, f : \text{Field}, \overline{P} : \text{Pargsorts}, h : \text{PHeap} \{ \text{dom}_f(\text{singleton}_P(\overline{P}, h)) \}. \]
\[ \text{dom}_f(\text{singleton}_P(\overline{P}, h)) = \emptyset \]
4.2. Triggering our existing axioms

4.2.3 Empty heap axioms

These do not contain any quantified variables except fields and predicates such that there is no need to choose any triggers.

Axiom 9’
\[ \forall f : Field \{ \text{dom}_f(\text{emp}) \}. \]
\[ \text{dom}_f(\text{emp}) = \emptyset \]

Axiom 10’
\[ \forall P : Predicate \{ \text{dom}_P(\text{emp}) \}. \]
\[ \text{dom}_P(\text{emp}) = \emptyset \]

4.2.4 Combine axioms

The triggers and arguments for domains and lookups are identical for fields and predicates, so we chose to only present the case of fields.

Field domain of combination

Axiom 11’
\[ \forall f : Field, h_1, h_2 : \text{PHeap} \{ \text{dom}_f(\text{combine}(h_1, h_2)) \}. \]
\[ \text{dom}_f(\text{combine}(h_1, h_2)) = \text{dom}_f(h_1) \cup \text{dom}_f(h_2) \]

The argument for the safety of this trigger is as follows: While the axiom potentially yields new terms matching the trigger, namely \( \text{dom}_f(h_1) \) and \( \text{dom}_f(h_2) \), these are strict subterms of \( \text{combine}(h_1, h_2) \) and thus a direct matching loop is impossible. There is also no possibility for longer loops from these terms, since this is the only axiom triggered by the \( \text{dom}_f \) symbol.

Field lookup of combination

For this axiom we found two triggers to be necessary:

Axiom 12’
\[ \forall f : Field, h_1, h_2 : \text{PHeap}, x : \text{Ref} \{ \text{lookup}_f(\text{combine}(h_1, h_2), x) \} \]
\[ \{\text{lookup}_f(h_1, x), \text{combine}(h_1, h_2)\} \]. \]
\[ x \in \text{dom}_f(h_1) \Rightarrow \text{lookup}_f(\text{combine}(h_1, h_2), x) = \text{lookup}_f(h_1, x) \]
4. **Additional Axioms and Triggers**

The first one is probably more intuitive: Whenever the solver tries to lookup a field in a combined heap $\text{combine}(h_1, h_2)$, this trigger yields an instantiation which equates this with a lookup in the strictly smaller heap $h_1$. Effectively, this trigger in combination with the symmetry axiom allows to solve to resolve field lookups inside combined heaps. Looking at our symbolic execution rules, these snapshots are created by the $\text{consume}$ function. Listing 4.1 contains a Viper program that requires this trigger to evaluate the function using its axiom. By the same argument as for the trigger of axiom 11', this trigger alone cannot directly create a matching loop.

```viper
field f : Int

function fun(x: Ref) : Int
    requires acc(x.f, 1/2) && acc(x.f, 1/2)
    { x.f }

method m(x: Ref)
    requires acc(x.f)
    {
        x.f := 1
        assert fun(x) == 1
    }
```

Listing 4.1: Checking the assertion will consume the precondition of $\text{fun}$, yielding a snapshot which combines two singletons mapping $x.f$ to 1. The heap access in its body translates to a $\text{lookup}_f$ in this snapshot and should trigger axiom 12' to resolve its value.

The second trigger is a bit more indirect. It says that whenever the solver obtains a field lookup in some snapshot $h_1$ and a combination of this snapshot with any other $h_2$, then the axiom can be instantiated. We explain why this is necessary using the example from listing 4.2. Clearly, $x.f$ in the assertion in line 18 evaluates to $\text{lookup}_f(h_1, x)$ and the semantics of Viper also tell us that this value should be 1. From a logical perspective, this is implied by our symbolic execution:

\[
\text{lookup}_f(h_1, x) = \text{lookup}_f(\text{combine}(h_1, h_2), x) = \text{lookup}_f(\text{combine}(\text{sing}_f(x, 1), \text{sing}_g(x, 2)), x) = \text{lookup}_f(\text{sing}_f(x, 1), x) = 1
\]

Here the equalities 4.1 and 4.3 are instances of axiom 12 and 4.4 of axiom 3. In step 4.2 we applied the path condition from line 17. The key point is that the first step of this proof expands the snapshot term in order to rewrite it before resolving the lookup. This is the rationale behind our second trigger.
4.2. Triggering our existing axioms

for axiom 12: The solver is allowed to add arbitrary combinations to snapshots under lookups, given that it already knows something about these combinations, i.e. has the corresponding \textit{combine} term.\footnote{Based on our example and more generally our symbolic execution rules for snapshots, such facts about combined snapshots will always be equalities so we would ideally like to trigger on "equality with the combination". But only uninterpreted functions are allowed in triggers, so we have to resort to this more general trigger.}

\begin{verbatim}
1 field f : Int
2 field g : Int
3
4 predicate P (x: Ref) { acc(x.f) && acc(x.g)
5 }
6
7 method m(x: Ref)
8 requires acc(x.f) && acc(x.g)
9 {
10  x.f := 1
11  x.g := 2
12  fold P(x)
13  // [P(x) \mapsto \text{combine}(\text{sing}_f(x,1),\text{sing}_g(x,2))]}
14 unfold P(x)
15 // [x.f \mapsto \text{lookup}_f(h_1,x)], [x.g \mapsto \text{lookup}_g(h_2,x)]
16 // combine(h_1,h_2) = \text{combine}(\text{sing}_f(x,1),\text{sing}_g(x,2))
17 assert x.f == 1
18 }
\end{verbatim}

Listing 4.2: The comments in line 14 and 16 describe the symbolic heap by listing its chunks in the same notation used earlier. Line 17 is a path condition learned through the \textit{produce} call inside \texttt{unfold}. These changes in the symbolic state result directly from application of our symbolic execution rules 3.2 and 3.3 from section 3.2.7.

Compared to the first trigger, this one allows resolving lookups inside snapshots created by \textit{produce}.

4.2.5 Commutativity of combination

In some sense, there is no choice for the trigger of the commutativity axiom. The only terms covering both quantified variables without containing interpreted functions are \textit{combine}(h_1,h_2) and \textit{combine}(h_2,h_1). But since the two universally quantified variables $h_1$ and $h_2$ can always be interchanged, these two triggers are equivalent.

\textbf{Axiom 15'}

$$\forall h_1, h_2 : \text{PHeap}\{\text{combine}(h_1,h_2)\}.$$ 

\textit{combine}(h_1,h_2) = \textit{combine}(h_2,h_1)$$
4. Additional Axioms and Triggers

It seems expensive to have an axiom trigger on any combination of heaps because these can potentially be nested heavily. But instead we claim that the nesting does not increase the number of instantiations and a `combine` term yields a linear number of instantiations in its size\(^2\). In fact, every `combine` symbol triggers the axiom at most twice, no matter how many nested combinations it contains.

The key to this insight is the fact that multiple triggers yielding the same instantiation of quantified variables result in only one instantiation of the quantifier. Put more formally, let a term \(t\) trigger a quantifier by instantiating its variables to \(s\). If the solver now encounters another term \(t'\) matching the same trigger with variable instantiations \(s'\), it will not trigger a second instantiation if it has already established that \(s = s'\). This is a direct consequence of the way the triggering mechanism works [4].

With this fact, let us consider the instantiations triggered by some term `combine(h, h')`:

1. It matches the trigger via the unifier \(h_1 = h, h_2 = h'\). Assuming that these have not been used to instantiate commutativity yet, this axiom is now triggered and yields the equality `combine(h, h') = combine(h', h)`.

2. The newly obtained term `combine(h', h)` matches the trigger via the unifier \(h_1 = h', h_2 = h\). Assuming that this instantiation of the commutativity axiom has not happened yet either, this will trigger a second time but not yield any new terms.

Not that in cases where our assumptions are not fulfilled, i.e. one of the two instantiations has already happened, then the process just stops. So much for the top level `combine` symbol, but what about the cases where \(h\) or \(h'\) are combinations themselves? Let \(h'\) be the term `combine(h'', h''')`\(^4\) and thus trigger (at most) two more instantiations of commutativity for this `combine` symbol which effectively add `combine(h'', h''') = combine(h''', h'''')` to the solvers knowledge. While the solver can propagate this equality about \(h'\) back up to the top level combination, e.g. infer the equality `combine(h, combine(h'', h''''')) = combine(h, combine(h''', h''''))`, the crucial point is that this will not trigger new axiom instantiations for the top level `combine` symbol because the resulting instantiations for \(h_1\) and \(h_2\) are known to be equal to the previous one:

---

\(^2\)i.e. number of nodes in its syntax tree representation

\(^3\)As usual, \(s\) and \(s'\) stand in for sequences of variables and similarly, \(s = s'\) stands in for the conjunction expressing that both sequences are element-wise equal.

\(^4\)Since our argument revolves around equivalence classes of SMT terms it might be considered too imprecise to say that \(h'\) is another term. What we mean is that we treat the symbol \(h'\) as a meta-variable for which we substitute the term `combine(h'', h''')`. 

38
4.3 Triggering function axioms

4.3.1 Termination of Viper functions

Since functions in Viper can be defined recursively they are a potential source of non-termination. As an intermediate verification language, the design of Viper is such that it does not check termination of functions but leaves this up to the user or frontend. While this is the current state of Viper, there is work on a project adding optional termination proofs via measure functions. This is relevant for our work since each function application corresponds to an instantiation of the definitional axiom, for which we need to decide on the triggers. Since there is no mechanism for ensuring progress towards termination, we cannot simply instantiate the function axiom for all applications i.e. use the application itself as trigger.

4.3.2 Basic approach

The basic approach taken by Silicon is to instantiate the definitional axiom of a function $f$ only once per application. This is implemented by introducing an additional symbolic function $f_{\text{limit}}$ with the same signature used for (transitively) recursive calls. The definitional axiom is triggered on applications of $f$, so the recursive calls to $f_{\text{limit}}$ effectively stop the recursion after one step. The only axioms about $f_{\text{limit}}$ introduce the postcondition for these applications and do not affect how the recursion is axiomatized.

4.3.3 Additional trigger heuristic

Since unrolling the function definition only once is of course highly incomplete, a second trigger for function axioms is added. This mechanism is based on a heuristic from Heule et al. [6] and allows the solver to make additional instantiations based on how deep predicates have been folded and unfolded in the rest of the program. Basically, for every predicate instance...
4. Additional Axioms and Triggers

mentioned in the function body the function axiom can be instantiated for instances which have been folded or unfolded in the program.

The implementation is as follows: it introduces a trigger function $P_{\text{trig}} : P_{\text{argsorts}} \times P_{\text{Heap}} \rightarrow \text{Bool}$ for each predicate $P$. The purpose of this function is to tag predicate instances which have been folded or unfolded explicitly in the program. This is done simply by adding the term $P_{\text{trig}}(P_{\text{args}}, h)$ to the path condition whenever such an operation occurs during symbolic execution. Roughly speaking, the function axiom can then be instantiated for all snapshots which contain such tagged predicate instances. Additionally, an instantiation must specify the function arguments. The heuristic implemented in Silicon is to allow any arguments which have been used in a recursive application, that is one of $f_{\text{lim}}$. These sets of arguments are tagged with the additional function symbol $f_{\text{stateless}}$ which has the same signature as $f$, except that it does not take a snapshot parameter $^5$.

The example from listing 4.3 successfully verifies only because the function axiom of $\text{length}$ can be instantiated multiple times with this mechanism. Starting from $\text{length}(n_5)$ in the assertion in line 36, the following terms and instantiations are obtained by the solver:

1. The snapshot $h$ for $\text{length}(n_5)$ is computed and call evaluates to $\text{length}(h, n_5)$.
2. This term triggers the definitional axiom of $\text{length}$ which contains a recursive call that simplifies to $\text{length}_{\text{lim}}(h', n_4)$ where $h'$ is the snapshot describing the predicate instance $\text{node}(n_4)$.
3. The limited function application triggers the tagging mechanism for function arguments and yields the term $\text{length}_{\text{stateless}}(n_4)$.
4. From the fold statement on line 30, the solver obtained the tagged snapshot $\text{node}_{\text{trig}}(n_4, h')$. Together with $\text{length}_{\text{stateless}}(n_4)$ this triggers the second instantiation of the function axiom.
5. This instantiation contains another recursive call $\text{length}(h'', n_3)$ for another snapshot $h''$ and the process repeats at step 3.

```plaintext
1 field next: Ref
2
3 predicate node(this: Ref) {
4    acc(this.next) && (this.next != null ==> acc(node(this.next)))
5 }
6
7 function length(this: Ref): Int
8    requires acc(node(this))
9    ensures result > 0
```

$^5$Hence the name stateless; it refers to the heap state.
4.4 Matching loop

There is one rule in our axiomatization which we have presented as a first draft, but not yet delivered the final version: the translation of unfolding expressions inside function bodies. As explained in 3.3.1, we chose a sound overapproximation to translate such expressions. It turns out that this overapproximation in combination with the second trigger for axiom 14 and our predicate-based trigger for function axioms creates a matching loop.

4.4.1 Where the loop occurs

The root of the problem are the following two behaviors:

Listing 4.3: Evaluating the length of a linked list with 5 elements succeeds because the described triggering heuristic allows just enough function axiom instantiations.

We will refer to this second trigger mechanism for function axioms as predicate-based trigger below.

```java
10 { 
11   1 + unfolding acc(node(this)) in 
12       this.next == null ? 0 : length(this.next)
13 }
14
15 method test01() {
16   var n1: Ref; n1 := new(next)
17   n1.next := null
18   fold acc(node(n1))
19
20   var n2: Ref; n2 := new(next)
21   n2.next := n1
22   fold acc(node(n2))
23
24   var n3: Ref; n3 := new(next)
25   n3.next := n2
26   fold acc(node(n3))
27
28   var n4: Ref; n4 := new(next)
29   n4.next := n3
30   fold acc(node(n4))
31
32   var n5: Ref; n5 := new(next)
33   n5.next := n4
34   fold acc(node(n5))
35
36   assert length(n5) == 5
37 }
```
4. Additional Axioms and Triggers

- A predicate-based trigger allows instantiating a function axiom with any snapshot that contains the same predicate instance.

- A function axiom which contains an unfolding $P(args)$ in $e$ yields the term $h' := \text{combine}(h, \text{lookup}_P(h, args))$\(^6\) where $h$ is the function snapshot argument.

Now, $h'$ still contains the same predicate instance which was unfolded (hence it is an over-approximation) and is in general not equal to $h$. The second trigger of axiom 14 can therefore be used to instantiate it and effectively learn $\text{lookup}_P(h, args) = \text{lookup}_P(h', args)$. Using the theory of uninterpreted functions, the solver can use this equality to obtain the term $P_{\text{trig}}(args, h')$. This concludes the first round of the loop since the predicate-based trigger can now restart the process on the strictly larger snapshot $h'$.

4.4.2 An approach to work around it

Above we identified three components of the loop, so avoiding it can be done by adapting one of these appropriately. The second trigger for 14 is a necessity as argued in section 4.2.4 and thus out of question for modifications (simple ones at least). The predicateTrigger mechanism is based on a heuristic external to our design which contains a level of imprecision that seems difficult to avoid. In contrast, the imprecision introduced by our own overapproximation for nested function snapshots can be eliminated easier. This is the approach we decided to take and describe below.

Conceptually, the loop is no longer possible if the unfolded predicate instance is removed from the resulting snapshot, because it can no longer be used to obtain a new predicateTrigger. But snapshots are just SMT terms which are immutable, so it is not directly clear how something can be removed from it.

Our solution introduces a function $\text{remove}_P : P\text{Heap} \times \text{Pargsorts} \rightarrow P\text{Heap}$ for each predicate $P$. The result of $\text{remove}_P(h, args)$ is axiomatized to be the same as $h$, except that the instance $P(args)$ is no longer part of it. Although the idea is simple, the axiomatization unfortunately turns out to be a bit wordy. To save at least some of it, we omitted the triggers which are simply the left handside of the equality in all cases. First, the snapshot’s behavior with respect to fields should not change at all:

**Axiom 16 (Field domain of $\text{remove}_P$)**

\[
\forall P : \text{Predicate}, f : \text{Field}, h : P\text{Heap}, \overline{\text{args}} : \text{Pargsorts}.
\quad \text{dom}_f(\text{remove}_P(h, \overline{\text{args}})) = \text{dom}_f(h)
\]

\(^6\)here $args$ are the symbolic values obtained from evaluating $\overline{\text{args}}$. 

42
4.4. Matching loop

**Axiom 17 (Field lookup of \( \text{remove}_P \))**

\[
\forall P : \text{Predicate}, f : \text{Field}, h : \text{PHeap}, x : \text{Ref}, P_{\text{args}} : P_{\text{argsorts}}. \\
\text{lookup}_f(\text{remove}_P(h, P_{\text{args}}), x) = \text{lookup}_f(h, x)
\]

Similarly, all predicates different from \( P \) are unaffected:

**Axiom 18 (Other predicate domain of \( \text{remove}_P \))**

\[
\forall P, Q : \text{Predicate}, f : \text{Field}, h : \text{PHeap}, x : \text{Ref}, P_{\text{args}} : P_{\text{argsorts}}, Q_{\text{args}}. \\
P \neq Q \Rightarrow \text{dom}_Q(\text{remove}_P(h, P_{\text{args}})) = \text{dom}_Q(h)
\]

**Axiom 19 (Other predicate lookup of \( \text{remove}_P \))**

\[
\forall P, Q : \text{Predicate}, h : \text{PHeap}, P_{\text{args}} : P_{\text{argsorts}}, Q_{\text{args}}. \\
P \neq Q \Rightarrow \text{lookup}_Q(\text{remove}_P(h, P_{\text{args}}), \text{loc}_Q(Q_{\text{args}})) = \text{lookup}_Q(h, \text{loc}_Q(Q_{\text{args}}))
\]

Finally, the only part that changes is the domain with respect to \( P \) itself:

**Axiom 20 (Predicate domain of \( \text{remove}_P \))**

\[
\forall P : \text{Predicate}, f : \text{Field}, h : \text{PHeap}, x : \text{Ref}, P_{\text{args}} : P_{\text{argsorts}}. \\
\text{dom}_P(\text{remove}_P(h, P_{\text{args}})) = \text{dom}_P(h) - \{\text{loc}_P(P_{\text{args}})\}
\]

It might look like \( \text{remove}_P \) is the tool we need to render our over-approximated snapshot more precise, even exact. Closer inspection of the semantics of predicates reveals that it is not: In the snapshot created by an unfolding expression, the predicate instance can still be present, depending on the amount of permission unfolded. Consider the function in listing 4.4. Our current over-approximation of the snapshot created by the unfolding is exact in this case.

```
1 field f : Int
2
3 predicate P(x: Ref) {
4 acc(x.f)
5 }
6
7 function fun(x: Ref) : Bool
8 requires P(x)
9 { unfolding P(x, 1/2) in true
10 }
```

**Listing 4.4:** The snapshot to evaluate true still contains the predicate instance P(x)
4. **Additional Axioms and Triggers**

The next attempt could be to remove the predicate instance only if write permissions are unfolded. However, this is not precise either for the following two reasons:

- First, it is possible to have more than write permissions (1.0, that is) to a predicate instance. Consequently, unfolding write permissions can still leave some permissions untouched making the removal of the instance incorrect.

- Second and opposingly, it is not necessary to unfold write permissions in order to remove the instance since the precondition might only give fractional permissions from the start.

In fact both of these hold for any positive permission amount $p$, as illustrated in listing 4.5.

```plaintext
1 define p() write
2 field g : Int
3
4 predicate P(x: Ref) {
5   acc(x.g)
6 }
7
8 function f1(x: Ref): Bool
9   requires acc(P(x), p / 2)
10 {  
11     unfolding acc(P(x), p / 2) in true
12 }  
13 
14 function f2(x: Ref): Bool
15   requires acc(P(x), 2*p)
16 {  
17     unfolding acc(P(x), p) in true
18 }  
19 
```

Listing 4.5: Line 1 is a macro definition which introduces the shorthand $p$ for write permissions. This illustrates how the example works for any permission amount, simply by changing the macro definition.

The function $f_1$ unfolds less than $p$ permissions but still removes the instance while $f_2$ unfolds $p$ permissions without removing the instance\(^7\). We conclude that without knowledge of the initial (symbolic) amount of permissions available when calling the function it is impossible to detect whether an unfolding expression removes a predicate instance or not. See chapter 6 for our proposed approach.

\(^7\)It could also remove more than $p$ without removing it, e.g. have $acc(P(x), 3*p)$ in the precondition and unfold $2*p$ in the body.
Our implementation

As a first approximation, we arrived at removing the predicate instance only when an unfolding expression unfolds write permissions. In general, this cannot be determined statically so the conditional goes into our snapshot term:

\[
\text{\texttt{tr}}(\text{unfolding acc}(P(\text{args}), p) \text{ in } e,h) : \\
\quad \text{\texttt{tArgs}} := \text{translate}(\text{args}, h) \\
\quad \text{\texttt{tp}} := \text{translate}(p, h) \\
\quad \text{\texttt{return}} \text{\texttt{tr}}(e, \text{\texttt{combine}}( \\
\quad \quad \text{\texttt{ite}}(\text{tp} = 1, \text{\texttt{remove}}_P(h, \text{\texttt{tArgs}}), h), \\
\quad \quad \text{\texttt{lookup}}_P(h, \text{\texttt{loc}}_P(\text{\texttt{tArgs}})) \\
\quad \))
\]

But as explained above, this is does not correctly depict the semantics of unfolding in general. The behavior of our implementation is described in chapter 5, but we suspect that some of its deficiencies are to be located in this rule.
Chapter 5

Evaluation

5.1 Supported subset of Viper

The presented modifications to Silicon are intended to support the Viper language without quantified permissions (QP) and without magic wands (MW). Our work also allows extension of the support for inhale-exhale assertions to positions where they are currently disallowed, namely function preconditions and predicate bodies. We have implemented the design from chapters 2 through 4 and now briefly discuss how it is tested and what the results are.

5.2 Testsuite

Our testsuite is an adapted version of the one for Silicon where all testcases which contain QP or MW are ignored or canceled. This leaves a bit more than 500 testcases of which some fifteen either fail or encounter a timeout at an SMT query. Because of the issues discussed in 4.4 we cannot expect to pass all tests.

5.3 Performance

Since the performance of an SMT solver might strongly depend on whether a single satisfiability query succeeds or fails, we ignored all of the failures and timeouts to get consistent behavior with the current version. For one testcase the behavior changed (correctly) from an expected failed to an expected successful assertion, so we excluded it from our testsuite for the same reason.

We end up with 503 remaining successful testcases (from a total of roughly 1000) on which we can evaluate performance. We ran the following versions of Silicon on this testsuite:
5. Evaluation

1. **Current**: The most recent version of Silicon which uses the structural snapshots described in chapter 1.

2. **Semantic heap snapshots**: Our implementation of Silicon with the snapshot design presented in this thesis substituted for the structural representation used by version 1.

3. **Semantic heap snapshots without commutativity**: A slight modification of version 2 where commutativity for `combine` (axiom 15) has been replaced with the symmetric alternatives for right subterm of all lookup axioms (c.f. 2.2.2).

We found that our new design causes a slow down of roughly 20 % over the current one. Additionally, similar performance of versions 2 and 3 supports our argument for the triggering behavior of commutativity:

<table>
<thead>
<tr>
<th>Version</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time for testsuite, averaged over 5 runs</td>
<td>284</td>
<td>349.6</td>
<td>361.8</td>
</tr>
</tbody>
</table>
Using a semantic heap snapshot representation for functions and predicates alleviates the unsoundness with inhale-exhale assertions and simplifies the function axiomization algorithm. However, the conceptually motivated predicate indirection through $\text{lookup}_p$ turned out to be problematic. We need a few helper functions and axioms such as $\text{remove}_p$ and $\text{restrict}_f$ to get the behavior of our snapshots closer to what is expected. Finally we discovered a matching loop that is unavoidable as long as Viper functions are permission-unaware\textsuperscript{1}. Below, we discuss two examples of function axioms as computed by our implementation.

### 6.1 Revisited example: Function axiom of double

```plaintext
1  field val: Int
2
3  function double (b: Bool, x: Ref, y: Ref, z: Ref): Int
4    requires acc(x.val) && acc(y.val)
5    requires b ? z == x : z == y
6    {  
7      b ? x.val + z.val : y.val + z.val
8    }
```

Listing 6.1: Reprint of listing 1.1.6

Using our new snapshot design, the function axiom is computed as:

\textsuperscript{1}By this we mean that currently, a function application does not have information about how much permissions are available.
∀h : PHeap, b : Bool, x : Ref, y : Ref, z : Ref.ite(b, z = x, z = y)
⇒ double(h, b, x, y, z) = ite(b,
lookup_val(h, x) + lookup_val(h, z)
lookup_val(h, y) + lookup_val(h, z)
)

Comparing this with the previous axiom (equation 1.1), the missing sort wrappers are immediately apparent. More importantly, the function parameters x, y and z actually appear in the function body and appear in the terms serving as x.val, y.val and z.val respectively. Although this is a very small example, we have not encountered any function axioms where our new design complicated the lookup of field values.

6.2 Example: Binary tree height

Listing 6.2: A standard binary tree predicate with a corresponding height function

This example combines most of the language features which are interesting from a snapshot perspective: unfolding expressions, recursion and conditionals. In the following discussion we ignore the macro for max and write an SMT function max instead of unfolding its definition as Viper does it. The function axiom as computed by our implementation is given in equation 6.2. Field accesses are explicit through the lookup functions and reference to function parameters, although lengthy because of the large snapshot terms.
Especially for the common pattern of recursion under an unfolding, the corresponding snapshot term is rather unintelligible. First, the unfolding results in a predicate lookup and predicate removal. The corresponding snapshot combination needs then be passed to the heap restriction function as to make it match the precondition again. And since any heap access under the unfolding will use this snapshot, the axiom grows in size quickly. A possibility we have not explored is to use SMT let-bindings to avoid repetition of terms. One should consider that it introduces another level of indirection which might again decrease understandability.
∀h : PHeap, r : Ref.  height(h, r) = 1 + max(
  ite(
    lookup_left(combine(lookupTree(h, locTree(r)), removeTree(h, r)), r) = null,
    0,
    heightLimit(restrictHeight(combine(
      lookupTree(h, locTree(r)),
      removeTree(h, r)
    ),
      lookup_left(combine(lookupTree(h, locTree(r)), removeTree(h, r)), r)
    ),
      lookup_left(combine(lookupTree(h, locTree(r)), removeTree(h, r)), r)
  )
  ),
  ite(
    lookup_right(combine(lookupTree(h, locTree(r)), removeTree(h, r)), r) = null,
    0,
    heightLimit(
      restrictHeight(combine(
        lookupTree(h, locTree(r)),
        removeTree(h, r)
      ),
      lookup_right(combine(lookupTree(h, locTree(r)), removeTree(h, r)), r)
    ),
      lookup_right(combine(lookupTree(h, locTree(r)), removeTree(h, r)), r)
  )
  )
)
)

(6.2)

6.3 Future work

6.3.1 Syntactic optimizations

Inspecting the SMT log of some examples reveals terms which seem a lot larger than necessary. The reason for this is that, as described so far, snapshots are not simplified as they are passed through our symbolic execution
rules but only internally by the SMT solver when processing a query. Of course, in general the symbolic execution engine should not strive to do this because this exactly what the SMT solver is used for. But we argue that some syntax based simplification of terms in their representation in the symbolic execution might not only improve performance but would definitely increase readability of the SMT log files. Without having validated this claim, we think the following rewrite rules would be worth lifting into Silicon itself:

- \( \text{combine}(\text{emp}, h) = h \) and \( \text{combine}(h, \text{emp}) = h \)
- \( \text{lookup}_{f}(\text{sing}_{f}(x, v), x) = v \)
- \( \text{lookup}_{P}(\text{sing}_{P}(x, h), x) = h \)

Another simple optimization is to implement the rules for \( \text{produce}(\text{acc}(x.f), p, h) \) and \( \text{produce}(\text{acc}(P(x), p), h) \) such that they inspect the form of \( h \) and do not add the singleton equality if it already is one syntactically (see rule 3.1). The spectrum of possible optimizations at this level is open-ended, but there is always the tradeoff in shifting work between the SMT solver and the symbolic execution engine itself. Therefore we propose to postpone this kind of work to a stage where a larger fraction of the testsuite is supported such that the performance implications can be better understood.

### 6.3.2 Resource specific heaps

In [3] different heap encodings for SMT solvers are evaluated. The authors suggest that field specific (and thus monomorphic) heaps are more efficient than polymorphic ones. Given the negative performance result from chapter 5, it could prove worthwhile to experiment with monomorphic snapshots. Logically, there is very little difference: a polymorphic snapshot \( h \) is replaced with e.g. a tuple of resource-specific snapshots \( (h_{f}, h_{g}, h_{p}, h_{Q}, \ldots) \) and functions and axioms are lifted where necessary.

### 6.3.3 Extension to quantified permissions

Our design is inspired by the field value functions (also called snapshot maps) used by Silicon for quantified permissions [9]. There are various ways in which our design can be used to support quantified permissions. Instead of exploring ways for achieving this, we just show the logical connection between the two designs in form of an embedding of field value functions into our \( PHeap \) sort. This should give the necessary understanding in order to evaluate how to proceed towards quantified permission support.

To present the embedding, we briefly review the mentioned field value functions. Silicon checks the program for quantified permissions on a per-field basis. If present, it introduces the sort \( FVF[S_f] \) which stands for field value
function and is monomorphic, i.e. only stores values of the field $f$ which are all of the same sort. Conceptually, a field value function allows the same operations as our snapshots:

$$\begin{align*}
FVF\text{.domain}_f & : FVF[S_f] \rightarrow \text{Set}[\text{Ref}] \\
FVF\text{.lookup}_f & : FVF[S_f] \times \text{Ref} \rightarrow S_f
\end{align*}$$

Thus, a snapshot should in principle be able to do whatever a field value function can do. Put more formally, a field value function for some field $f$ can be embedded in a snapshot by a function $\text{embed}_f : FVF[S_f] \rightarrow \text{PHeap}$. Its properties are:

$$\begin{align*}
\forall fvf : FVF[S_f]. & \quad \text{dom}(\text{embed}_f(fvf)) = FVF\text{.domain}_f(fvf) \\
\forall fvf : FVF[S_f], x : \text{Ref}. & \quad \text{lookup}_f(\text{embed}_f(fvf), x) = FVF\text{.lookup}_f(fvf, x) \\
\forall fvf : FVF[S_f], P : \text{Predicate}. & \quad \text{dom}_P(\text{embed}_f(fvf)) = \emptyset \\
\forall fvf : FVF[S_f], g : \text{Field}. & \quad f \neq g \Rightarrow \text{dom}_P(\text{embed}_f(fvf)) = \emptyset
\end{align*}$$

(6.4)
Bibliography


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):
Towards better function Axiomatization in a Symbolic-execution-based Verifier

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

Name(s):
Bringolf

First name(s):
Mauro

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
Zürich, 6.9.19

Signature(s)

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