Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions

Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers

ETH Zurich, Switzerland
stheule@ethz.ch,
{ioannis.kassios,peter.mueller,alexander.summers}@inf.ethz.ch

Abstract. Abstract predicates are the primary abstraction mechanism for program logics based on access permissions, such as separation logic and implicit dynamic frames. In addition to abstract predicates, it is useful to also support classical abstraction functions, for instance, to encode side-effect-free methods of the program and use them in specifications. However, combining abstract predicates and abstraction functions in a verification condition generator leads to subtle interactions, which complicate reasoning about heap modifications. Such complications may compromise soundness or cause divergence of the prover in the context of automated verification. In this paper, we present an encoding of abstract predicates and abstraction functions in the verification condition generator Boogie. Our encoding is sound and handles recursion in a way that is suitable for automatic verification using SMT solvers. It is implemented in the automatic verifier Chalice.

1 Introduction

Program logics based on access permissions, such as separation logic [22] and implicit dynamic frames [25] are the foundation of many program verifiers for heap-manipulating programs [3, 9, 13, 19, 24]. They associate an access permission with each heap location and enforce that a method accesses a location only if it has the permission to do so. To enable modular verification, each method specification states which permissions the method requires from its caller (in its precondition) and returns to its caller (in its postcondition). Upon a call, the caller relinquishes the required permissions (we say the caller *exhales* the precondition) and transfers them to the callee (the callee *inhales* them). Conversely, when a method terminates, the method exhales its postcondition, while its caller inhales it. This technique simplifies framing; as long as a method holds on to the permission for a location (that is, does not exhale it), no other method can access that location and, thus, its value remains unchanged.

Abstract Predicates. Enumerating all locations for which a method requires or returns permissions is not possible for recursive data structures. For instance, a

```
class List {
  var value: int;
  var next: List;

  predicate valid { acc(value) && acc(next) && (next ≠ null ⇒ next.valid) }

  function length(): int
    requires valid;
    ensures result > 0;
  { unfolding valid in next = null ? 1 : 1 + next.length() }

  function itemAt(i: int): int
    requires valid && 0 ≤ i && i<length();
  { unfolding valid in i = 0 || next = null ? value : next.itemAt(i-1) }

  method set(i: int, v: int)
    requires valid && 0 ≤ i && i < length();
    ensures valid && length() = old(length()) && itemAt(i) = v;
    ensures ∀ j in [0..length()-1] • i ≠ j ⇒ itemAt(j) = old(itemAt(j));
  {
    unfold valid;
    if (i = 0) { value := v; }
    else       { call next.set(i-1, v); }
    fold valid;
  }
}
```

Fig. 1. A Chalice [19] implementation of a singly-linked list. Methods have preconditions (keyword **requires**) and postconditions (keyword **ensures**). In addition to regular methods, Chalice supports side-effect free functions, which may be used in specifications. An access permission to a field $o.f$ is denoted by **acc**($o.f$), which corresponds to $o.f \mapsto \_$ in separation logic. The Chalice conjunction && treats permissions multiplicatively (i.e., requiring the sum of the permissions in each conjunct), similarly to the separating conjunction $*$ of separation logic. The recursive abstract predicate `valid` represents the memory locations of the list structure. The **unfold** and **fold** ghost statements replace a predicate by its body and vice versa. The ghost expression construct **unfolding**...**in** is intuitively analogous to an **unfold**-**fold** block and can be used in functions and in specifications, where statements cannot occur.

method that traverses a linked list, such as method `itemAt` in Fig. 1 would require permission to access `this.value`, `this.next`, `this.next.value`, `this.next.next`, and so on. To solve this problem, Parkinson and Bierman [21] introduced abstract predicates. The definition of an abstract predicate declares a predicate body that may contain permissions to concrete heap locations, constraints on their values, and possibly further predicate instances. Due to this recursion, abstract predicates potentially represent permission to an unbounded number of heap locations. For instance, the abstract predicate `valid` in Fig. 1 represents the permissions for `value`, `next` and, if `next` is non-null, the permissions in `next.valid`. Just as with permissions to field locations, a method may require predicate instances from its caller. It may access a location if it possesses the corresponding permission, either directly or as part of a predicate instance. For example, method `itemAt` requires the predicate instance `valid` to get access to all locations of the list.

In this paper, we employ ghost locations to describe the predicates held for a particular object (cf. Sec. 3). Holding an instance of a predicate is represented by holding permission to its ghost location. We use the term *location* for both concrete field locations and predicate locations. We use the term *permission* to describe permissions to both kinds of locations.

Abstraction Functions. Abstraction functions [12] map the concrete representation of a data structure (say, a linked list) to an abstract value (say, a sequence). Specifications can then be expressed in terms of the abstract values, which is important for information hiding. In the form of side-effect-free inspector methods (pure methods), abstraction functions are a key ingredient of contract languages such as Eiffel, JML, Spec#, and .NET CodeContracts, which support runtime assertion checking in addition to static verification.

Many permission logics express data abstraction via parameterised abstract predicates whose parameters represent abstracted values of a data structure and whose bodies relate the concrete representation of the data structure to these abstract values. However, there are several advantages to supporting classical abstraction functions *in addition to* abstract predicates: (1) Most data structure implementations include side-effect-free inspector methods (or *functions* in Chalice) such as `itemAt` and `length` in our example. It is convenient to re-use these methods in specifications; side-effect free methods can be encoded naturally as abstraction functions [6]. (2) Declaring abstraction functions does not affect the signatures and definitions of abstract predicates. This allows additional abstractions to be added to a library during maintenance, without changes to the abstract predicates and, thus, re-verification of existing client code. (3) Specifications written without abstraction functions typically use logical variables for the parameters of an abstract predicate, which can then be used in postconditions to describe how the abstract value has changed. Finding witnesses for these logical variables is not supported well by SMT solvers[1]. By contrast, abstraction functions can be used within **`old`** expressions to refer to their pre-state evaluation, avoiding logical variables.

For concreteness, we present our approach in the context of Chalice, which provides only non-parameterised predicates. In this setting, the primary role of predicates such as `valid` in Fig. 1 is to abstract over access permissions, whereas functions such as `length` and `itemAt` abstract over the contents of a data structure. Our results apply also to frameworks with parameterised predicates.

Contributions. The main contribution of this paper is an encoding of abstract predicates and abstraction functions for verification condition generators that is sound and amenable to automation via SMT solvers. The challenge that we

[1] Existing tools that support rich parameterised predicates use custom reasoning engines based on symbolic execution, rather than verification condition generation.

solve is how to avoid giving uncontrolled recursive axioms to the prover, which may cause the prover to diverge because it instantiates these axioms indefinitely in a so-called *matching loop*.

It is tempting to generate such recursive axioms in three situations: (1) to define which permissions are part of a recursive predicate instance (for example, to determine which permissions are transferred when it is exhaled), (2) to define the values of recursive abstraction functions, and (3) to define which heap locations the value of a recursive abstraction function depends on (for example, to determine whether a heap update affects the value of a function application).

The key insight motivating our solution is that, although each of the three questions above are in principle concerned with a statically unbounded number of unrollings of a recursive definition, the unrolling of these definitions is typically only relevant up to the depth at which the program to be verified (including its contracts) has explicitly inspected the corresponding data structure, at either the current or an earlier program point. That is, our solution focuses on ensuring that recursively-defined information is made available for predicate and function bodies which have been *syntactically observed* at some program point up to the current one. By applying this idea to the various aspects of the encoding problem, our solution expresses each proof obligation in a way that does not give recursive definitions to the prover that can be unfolded in an unbounded manner. Our solution employs some existing ideas, but combines and enhances them in an original way to achieve the first verification condition generator that avoids matching loops and is sound[2]. In particular, we present a novel encoding of those permissions inside a predicate that are needed to express proof obligations.

We present our solution for implicit dynamic frames [25], but it also applies to other permission logics, such as separation logic. We have implemented our solution in a new version of Chalice.

Outline. Secs. 2 and 3 present our solution informally. The details of our encoding are explained in Secs. 4 and 5, and we argue why our solution is sound in Sec. 6. We discuss related work in Sec. 7 and conclude in Sec. 8.

2 Abstract Predicates

In this section, we describe informally how our technique avoids the first form of uncontrolled recursive axioms described above. The other two forms are discussed in the next section.

2.1 Folding and Unfolding

Whenever a method attempts to access a location $o.f$, the verifier needs to check whether the method has the access permission for $o.f$, either directly or as part

[2] Some verifiers based on symbolic execution [24, 14] support both abstract predicates and abstraction functions, and many more support only the former (e.g., [9, 13, 3]). In the equally important domain of verification condition generation, there is currently no solution that supports both features, is sound, and avoids matching loops.

of a predicate. However, since the definitions of predicates may be recursive, a verifier cannot determine precisely which permissions are part of a predicate; since the recursion is (statically) unbounded, it is neither possible to inline the predicate's body fully, nor is it useful to let an SMT solver reason directly about recursive definitions because the solver might unfold a recursive axiom infinitely often in a matching-loop and therefore not terminate.

Many verifiers [13, 19, 24] work around this problem by distinguishing between a predicate and its body. Instead of letting the SMT solver expand predicate definitions automatically, the verifier expands only specific predicate definitions at specific points in the program execution. *Unfolding* replaces a predicate by its body, while *folding* has the inverse effect. Until a predicate has been unfolded, the permissions and information implied by the predicate's body are generally not made available to the prover.

We call a permission that has not been folded into a predicate instance *direct*; all other permissions are called *folded*. Accessing a field, unfolding a predicate, or exhaling a predicate all require appropriate direct permissions.

Folding and unfolding transforms the problem of deciding how deeply to unroll a recursive definition to the problem of how deeply to unfold a predicate instance. Some tools provide heuristics for inferring unfold and fold operations in certain common situations, while others require programmers to indicate these operations through ghost statements. For the approach presented in this paper, it is irrelevant whether the unfold and fold operations are indicated explicitly or inferred, as long as there are specific points in the program execution at which the transition between a predicate and its body takes place. In our examples, we use explicit **`unfold`** and **`fold`** ghost statements for clarity.

The method `set` in Fig. 1 illustrates these concepts. The method body first inhales its precondition; in particular, the predicate `valid`. After the inhale, it holds `valid` as a direct predicate, whereas the permissions to `value` and `next` (as well as the fields of the rest of the list) are folded. Consequently, before accessing these locations, the method must unfold `valid`. The **`fold`** statement at the end of the method is necessary to regain direct permission to `valid`, which gets exhaled as part of the postcondition.

Folding and unfolding allows one to phrase most proof obligations in terms of *direct* permissions. For instance, the prover does not have to unroll recursive predicate definitions to determine whether a method may access a field or transfer a permission to another method since the corresponding proof obligations require direct permissions, and appropriate fold or unfold operations may be necessary to show this. However, abstraction functions require the verifier to track also certain folded permissions, as we will explain shortly.

The key insight mentioned in the introduction can now be made more concrete: although some information about the contents of recursively-defined predicates and functions is essential, we need concern ourselves only with those predicate instances whose bodies were unfolded earlier in the program text. Therefore, we can avoid dependencies on arbitrary unrollings of recursive definitions, and instead focus on recording detailed information for precisely these instances.

2.2 Exhaling Predicates

When a method releases the permission to a location, the verifier must invalidate any information about the location's value. Since another method may obtain the permission and modify the location, it would be unsound to retain any information. Verification condition generators invalidate information about a location by assigning an arbitrary value to the location; we say that the verifier *havocs* the location.

It is not useful to havoc locations immediately when the *direct* permission to the location is released; in particular, when the location is being folded inside a predicate. The reason for this is that abstraction functions provide a way of inspecting memory locations whose permissions have been folded. For instance, the function `itemAt` yields the value of a `value` field whose permission is folded (possibly deeply) into the `valid` predicate. If we were to havoc locations upon **`fold`** then a function that inspects a location after is has been folded into a predicate would return an arbitrary value, which would defeat the purpose of abstraction functions. For instance, the call to `itemAt` in the postcondition of `set` would yield an unknown value, and the postcondition could not be verified. To avoid this problem, it is necessary to havoc locations not when their permissions get folded inside a predicate but only when the predicate is subsequently exhaled; that is, when the method releases the *folded* permissions to these locations.

As we explained above, a verification condition generator based on SMT solvers cannot precisely determine all the permissions that are (transitively) folded into a predicate and, thus, which locations to havoc when the predicate is exhaled. We solve this dilemma by havocing very aggressively. Each time a predicate is exhaled, we havoc *all* memory locations for which the method does not retain a direct permission. This solution is sound since it havocs all locations whose permissions are folded, not just the ones that are inside the predicate instance given away. We will discuss in the next subsection how to make this crude approximation more precise, by recovering location values to make the verification sufficiently complete.

2.3 Framing of Locations—Known-Folded Permissions

Aggressively havocing all locations to which the program does not hold direct permission is, by itself, too incomplete to be useful. When the verifier has observed the value of a heap location earlier in the program, it is important to preserve this information even though the permission to this location is now folded inside a predicate instance. This is a particular case of our earlier insight: it is not *all* folded permissions that we care about, but those (a) for which the program previously held the direct permission and (b) which are folded inside a predicate instance that has been retained up to the current program point. We call these permissions the *known-folded permissions* of a predicate instance. In our encoding, in addition to recording direct permissions, we record for each predicate instance those locations to which the predicate holds known-folded permission.

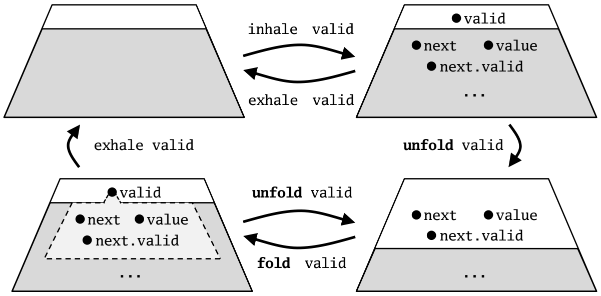

Fig. 2. Each trapezoid depicts all permissions held at a particular point in a method invocation, where the white regions contain direct permissions and the gray areas depict folded permissions. Known-folded permissions appear in the light gray region with dashed border (connected to the predicate they belong to).

Known-folded permissions provide an under-approximation of folded permissions, which (unlike the latter) can be precisely tracked in our encoding. A predicate instance contains known-folded permission to a location exactly when the direct permission to the location was folded (possibly deeply) inside the predicate instance at some earlier program point, either through a **fold** statement or through an **unfolding** expression. Because the known-folded permissions are always a subset of the folded permissions held by the current method, it is sound to havoc only those locations when exhaling a predicate for which the method holds neither direct nor known-folded permission.

Fig. 2 illustrates how **fold**, **unfold**, **exhale**, and **inhale** operate on some of the possible states of the `valid` predicate. The upper row depicts states in which the values of fields `value` and `next` have not been observed; in the upper right trapezoid, the permissions in the body of `valid` are folded, but not known-folded. In the lower row the values of these fields have been observed, and permission to them is either direct or known-folded, which protects the fields from being havoced by an **exhale**.

As an example, consider two distinct `List` objects `x` and `y`, for which the `valid` predicate is held at the beginning of the execution of the following code:

```
var i: int := unfolding x.valid in x.value;
var j: int := unfolding y.valid in y.value;
call y.set(0,10);
assert unfolding x.valid in (i = x.value); // succeeds
assert unfolding y.valid in (j = y.value); // correctly fails to verify
```

The **unfolding** expressions at the beginning of the code make the permissions to `x.value` and `y.value` known-folded. The call to `y.set` exhales `y.valid` and thereby removes the permission to `y.value` from the known-folded permissions, since `y.value` is folded inside `y.valid`. It havocs the state, but preserves locations for which the method holds known-folded permission, in particular, `x.value`. This makes the first assertion at the end of the code succeed, while the second assertion correctly fails to verify.

3 Abstraction Functions

The distinction between direct, known-folded, and other folded permissions avoids the need for recursive axioms in our encoding of recursive predicates. In this section, we explain how we handle the other two forms of problematic axioms mentioned in the introduction, namely the axioms for the definition and the framing of abstraction functions.

3.1 Definition of Abstraction Functions

An abstraction function is typically encoded as an uninterpreted function symbol together with a *definitional axiom* that relates the function symbol to the definition of the abstraction function. For recursive functions, this axiom is recursive and might lead to a matching loop if the instantiation of the axiom is not controlled. We avoid this matching loop by controlling when a recursive function definition may be expanded by the SMT solver; this is achieved via appropriate matching patterns (*triggers*) that control how the SMT solver instantiates universal quantifiers [7]. Our approach lets the prover unroll the function definition only one level deep, and it triggers additional unrollings as we explain next.

A typical recursive abstraction function such as `length` in Fig. 1 requires permission to the locations of the data structure in its precondition (often via a predicate) and recurses on the data structure to compute its result. In particular, the body of such a function typically unfolds a predicate from its precondition before recursing on the next node in the structure via the same function (cf. `length`). The definitional axiom of such a function relates a function application that depends on a predicate to an expression that depends on the locations whose permissions are folded inside the predicate's body. Therefore, an instantiation of the axiom will typically only provide useful information if the verifier has observed some information about these contents. For instance, if a method does not have direct or known-folded permission to `o.next` and `o.next.valid` then it cannot observe the values of `o.next` and `o.next.length()`. Without any knowledge about these values, the definitional axiom for `o.length()` is not useful.

This observation leads us to tie the instantiation of function definitions to the occurrence of predicate instances which have been unfolded at some earlier point in the program code. Essentially, we again follow our key insight; rather than concerning ourselves with arbitrary unrollings of a function's definition, it is enough to ensure that they are unrolled to at least the depth that the corresponding predicate instances have been unfolded (at some program point up to the current one). Thus, the instantiation of axioms proceeds in lockstep with the method's traversal of the corresponding data structure. To implement this, our approach generates an appropriate trigger whenever **`fold`** and **`unfold`** operations are encountered (see Sec. 5.1 for details).

The following example illustrates our approach. Assume that `unroll` is an additional method of class `List`. The method precondition mentions `length`, which allows the prover to expand the recursive definition only one level deep, which avoids a matching loop. Therefore, the prover cannot conclude from the precondition that `next.next` is **`null`**. However, since the method cannot observe the value of `next.next` at this point (the permission is folded inside `valid`), the information would be useless anyway. The second **`unfold`** statement unfolds the predicate instance required by the function application `next.length()` and, therefore, generates the trigger to instantiate the definitional axiom for this application, which provides the information to verify the assertion.

```
method unroll()
  requires valid && length() = 2
{
  unfold valid;
  unfold next.valid;
  assert next.next = null;
}
```

The approach outlined above avoids matching loops by controlling the instantiations of recursive definitional axioms. It supports the typical uses of abstraction functions, but does not handle recursion that is not tied to the traversal of a data structure (e.g., a factorial function). The automatic handling of such functions is beyond the scope of this paper.

3.2 Termination of Abstraction Functions

A common problem in allowing arbitrary recursive definitions in specifications, is the potential for introducing unsoundness to the logic, via non-well-founded recursion. For example, the definition of a function $f() = 1 + f()$ is inconsistent and therefore must be forbidden.

The usual approach to handling this problem is to insist on the existence of a well-founded *termination measure* for each recursive function definition provided. One particular termination measure that is tied to predicates is the number of (direct or folded) predicate instances held in the state. That is, among other measures, a function passes the termination-check if the verifier can show that every recursive function call is made within the body of an **`unfolding`** expression (that is, after unfolding a predicate instance). The number of predicate instances held defines a well-founded measure because every predicate instance can be unfolded only finitely many times. Only finitely many **`fold`** statements can ever be encountered during a program run and, thus, any point in the program at which an infinite predicate instance is assumed to be held is actually unreachable. For a more detailed and formal discussion of the treatment of termination issues here, we refer the reader to [26].

Note that this treatment of predicates does not rule out cyclic *heap structures*. For instance, doubly-linked structures can be easily handled with fractional permissions, while cyclic lists can be handled (as in separation logic) using list *segment* predicates[3].

Concerning our running example (Fig. 1), the termination measure described above permits the definitions of the functions `length` and `itemAt`; the recursive calls are inside **`unfolding`** expressions.

[3] See the online tool [1] for further examples, including a cyclic list.

3.3 Framing of Abstraction Functions

Since an SMT solver cannot unroll a recursive function definition arbitrarily deeply, it cannot use the function definitions to *frame* function applications, that is, to determine whether a heap update potentially affects the value of a function application or not. Therefore, an encoding of abstraction functions requires a *framing axiom* in addition to the definitional axiom, to express the circumstances under which the value of a function application can be framed.

Intuitively, the value of a function can be framed if none of the heap locations on which the function depends are modified. These locations are a subset of the locations for which the function requires permission in its precondition. However, if the function requires a recursive predicate then this set cannot be determined precisely by the verifier or the SMT solver.

Similarly to existing tools such as Spec# [2] and VeriCool [24], we handle this problem by abstracting over the locations folded inside a predicate instance, via *versioning*. The idea is as follows: if we can be sure that a predicate instance has been neither unfolded nor exhaled since an earlier program point, we know that all locations nested inside the predicate are unmodified. We label the predicate with the same version to identify this case.

Predicate versions are recorded as the values of predicate locations in the heap, which are treated like field locations: in particular, we retain knowledge of a predicate version so long as we hold either direct or known-folded permission to the predicate location. When we hold neither direct nor known-folded permission to such a location, it will naturally be havoced during an exhale. In addition, the version is havoced when the predicate is unfolded. Thus, our solution for function framing is closely tied to the framing of locations.

The details of our handling of functions are given in Sec. 5, but the framing axiom is, informally, as follows: two applications of the same function in two states evaluate to the same value if the receiver and all arguments of the function applications are the same and the two states agree on the values of all locations to which the function precondition requires direct permission; in particular, this includes the versions of the required predicates.

To illustrate our approach, consider again two distinct `List` objects `x` and `y` for which the `valid` predicate is held at the beginning in the following code:

```
var i: int := x.itemAt(0);
var j: int := y.itemAt(0);
call y.set(0,10);
assert x.itemAt(0) = i; // succeeds
assert y.itemAt(0) = j; // correctly fails to verify
```

The exhale operation of the method call gives away `y.valid`, thereby havocing the version of that predicate. The predicate `x.valid` is not affected, and keeps the same version, which allows the prover to correctly verify the first assertion, using the framing axiom. The second assertion is not necessarily true, and indeed fails to verify because the version of `y.valid` has changed.

4 Encoding of Abstract Predicates

In this and the next section, we present an encoding of our solution in the verification condition generator Boogie [16]. Verification with Boogie consists of three steps: (1) a *translator* translates the source program and its specification into the Boogie language, (2) Boogie computes verification conditions, and (3) an SMT solver attempts to prove the verification conditions. Here, we focus on how the translator encodes abstract predicates without giving recursive definitions or axioms to the prover. The complementary encoding of abstraction functions will be presented in the next section.

Heaps and Permission Masks. Our encoding represents the current heap with a variable `Heap`, which is a map from locations to values. The value of a location $e.f$ is denoted by $\texttt{Heap}[e, f]$. Permissions are tracked using *permission masks*, which map locations to booleans. The variable `Mask` stores the current mask, which represents the direct permissions held by the current method.

We also store information about predicate instances in the heap. For an abstract predicate p, we use (ghost) predicate locations $e.p$ to store an integer-mask pair. Such a pair represents the predicate version (as an integer) along with a mask representing the known-folded permissions under the predicate instance $e.p$; we call this the *predicate mask*. We write $\texttt{Heap}[e, p]\texttt{.version}$ to denote the version of predicate instance $e.p$, and $\texttt{Heap}[e, p]\texttt{.mask}$ to denote the corresponding predicate mask for known-folded permissions. At the beginning of verifying a method body, we assume all predicate masks to be empty masks (denoted by $\emptyset$), that is, to map all locations to false.

We say that a heap H' is a *framed heap for a state* with heap H and mask M, written $H' \overset{M}{\equiv} H$, if H and H' agree on the location values for which the method has direct or known-folded permission in that state, and for all other predicate locations, the predicate mask in H' is empty. More precisely:

1. For all locations $o.l$ for which $M[o, l]$ is true, $H'[o, l] = H[o, l]$.
2. For all predicate locations $o.p$ for which $M[o, p]$ is true, and for all locations $o'.l'$ for which $H[o, p]\texttt{.mask}[o', l']$ is true, $H'[o, l] = H[o, l]$.
3. For all predicate locations $o.p$, if $H'[o, p]$ has not been constrained by either of the two cases above, then $H'[o, p]\texttt{.mask} = \emptyset$.

An important property of our encoding is that the known-folded permissions of a predicate instance $e.p$ include the known-folded permissions of predicates instances in the body of $e.p$ (informally, the predicate masks record information transitively). This property is maintained as part of our encoding of a **`fold`** statement, and when evaluating an **`unfolding`** expression. Storing transitive information in this "flattened" form means we never need to recursively traverse predicate masks in our encoding. The operation of "copying" information from one predicate mask to another is encoded via assumptions and quantifiers, but we represent it here using a disjunction operator on masks. We denote the disjunction of two masks by $(M_1 \bigvee M_2)$ and define it pointwise:

$$(M_1 \bigvee M_2)[e,l] \Leftrightarrow (M_1[e,l] \vee M_2[e,l])$$

Encoding of Exhale. As explained in Sec. 2, the exhale operation aggressively havocs the heap and preserves information only for those locations to which the method holds direct or known-folded permission after the exhale. In the encoding of `exhale` (see top of Fig. 3), this is reflected by introducing a fresh heap H', assuming that H' is a framed heap for the state after the exhale and then making H' the new current heap. The actual exhaling is encoded via an auxiliary operation **exhale**′, which is explained next.

$$\llbracket \textbf{exhale } A \rrbracket = \textbf{var } \mathtt{H'} \; ; \; \textbf{havoc } \mathtt{H'} \; ; \; \llbracket \textbf{exhale}' \ A \rrbracket \; ; \; \textbf{assume } \mathtt{H'} \overset{\mathtt{Mask}}{\equiv} \mathtt{Heap} \; ; \; \mathtt{Heap} := \mathtt{H'}$$

$$\begin{aligned}
\llbracket \textbf{exhale}' \ e \rrbracket &= \textbf{assert } \lfloor\!\lfloor e \rfloor\!\rfloor \\
\llbracket \textbf{exhale}' \ A_1 \ \&\& \ A_2 \rrbracket &= \llbracket \textbf{exhale}' \ A_1 \rrbracket \; ; \; \llbracket \textbf{exhale}' \ A_2 \rrbracket \\
\llbracket \textbf{exhale}' \ e \Rightarrow A \rrbracket &= \textbf{if } (\lfloor\!\lfloor e \rfloor\!\rfloor) \ \{ \ \llbracket \textbf{exhale}' \ A \rrbracket \ \} \\
\llbracket \textbf{exhale}' \ \textbf{acc}(e.f) \rrbracket &= \textbf{assert } \mathtt{Mask}[\lfloor\!\lfloor e \rfloor\!\rfloor, f] \; ; \; \mathtt{Mask}[\lfloor\!\lfloor e \rfloor\!\rfloor, f] := \mathtt{false} \\
\llbracket \textbf{exhale}' \ \textbf{acc}(e.p) \rrbracket &= \textbf{assert } \mathtt{Mask}[\lfloor\!\lfloor e \rfloor\!\rfloor, p] \; ; \; \mathtt{Mask}[\lfloor\!\lfloor e \rfloor\!\rfloor, p] := \mathtt{false} \\
\llbracket \textbf{exhale}' \ \textbf{unfolding } e.p \ \textbf{in } e' \rrbracket &= \llbracket \textbf{inhale } \mathtt{body}(\lfloor\!\lfloor e \rfloor\!\rfloor, p) \rrbracket^{\mathtt{false}}_{\mathtt{Heap}[\lfloor\!\lfloor e \rfloor\!\rfloor, p].\mathtt{mask}} \; ; \\
& \ \textbf{assert } \lfloor\!\lfloor e' \rfloor\!\rfloor
\end{aligned}$$

Fig. 3. Encoding of exhale. We use A to denote general assertions, which may include permissions, and e to denote expressions without any permissions. To emphasize the uniform treatment of field permissions and predicates, we write **acc**$(e.p)$ to denote the predicate instance $e.p$, but simply write $e.p$ in our examples for brevity. $\llbracket \_ \rrbracket$ denotes the translation of source statements to Boogie instructions. In the encoding, we treat **exhale** and **exhale**′ like statements even though they cannot occur in source programs. $\lfloor\!\lfloor \_ \rfloor\!\rfloor$ denotes the translation of source expressions to Boogie expressions; it is straightforward and therefore omitted. Both translation functions refer to the global variables `Heap` and `Mask`. `body`() yields the declared body of a predicate instance.

The **exhale**′ operation recursively traverses the assertion to be exhaled, asserting all logical properties, and removing the required permissions from the current mask (see Fig. 3). Exhaling a boolean expression amounts to asserting that the expression holds. Exhaling a conjunction results in exhaling the two conjuncts sequentially; the side-effects of these exhales are accumulated. Implications are handled via if-statements in the Boogie output. Exhaling permission to a field or predicate location amounts to checking that the permission is currently held, and then removing it. Exhaling an **`unfolding`** expression is the most complicated case. First, the body of the unfolded predicate is inhaled, but using the predicate mask for the predicate instance instead of the `Mask` variable (since the permissions encountered are known-folded and not direct permissions). The translation of **inhale** used here is described below. Finally, the body of the **`unfolding`** expression is asserted to be true.

Encoding of Inhale. The translation of **inhale** is given in Fig. 4. In contrast to **exhale**, the translation of **inhale** is parameterised by a mask because it sometimes operates on primary and sometimes on known-folded permissions stored in a particular predicate mask. The boolean parameter b of the translation function indicates whether the inhale is of direct permissions or not. **inhale** traverses the assertion to be inhaled, assuming all logical properties, and adding the required permissions to the current mask. The assumption of $\neg\mathtt{M}[\llbracket e \rrbracket, f]$ in the case of inhaling field permissions encodes the fact that we cannot hold permission to the same location twice. This assumption is not made for permissions to predicate locations, since it is possible to hold the same predicate more than once[4].

$$
\begin{aligned}
\llbracket \textbf{inhale } e \rrbracket^{b}_{\mathtt{M}} &= \textbf{assume } \llbracket e \rrbracket \\
\llbracket \textbf{inhale } A_1 \text{ \&\& } A_2 \rrbracket^{b}_{\mathtt{M}} &= \llbracket \textbf{inhale } A_1 \rrbracket^{b}_{\mathtt{M}} \; ; \; \llbracket \textbf{inhale } A_2 \rrbracket^{b}_{\mathtt{M}} \\
\llbracket \textbf{inhale } e \Rightarrow A \rrbracket^{b}_{\mathtt{M}} &= \textbf{if } (\llbracket e \rrbracket) \; \{ \; \llbracket \textbf{inhale } A \rrbracket^{b}_{\mathtt{M}} \; \} \\
\llbracket \textbf{inhale } \textsf{\textbf{acc}}(e.f) \rrbracket^{b}_{\mathtt{M}} &= \textbf{assume } \neg\mathtt{M}[\llbracket e \rrbracket, f] \; ; \; \mathtt{M}[\llbracket e \rrbracket, f] := \mathtt{true} \\
\llbracket \textbf{inhale } \textsf{\textbf{acc}}(e.p) \rrbracket^{b}_{\mathtt{M}} &= \mathtt{M}[\llbracket e \rrbracket, p] := \mathtt{true} \; ; \\
& \; \textbf{\#if } (\neg b) \; \{ \; \mathtt{M} := \mathtt{M} \bigvee \mathtt{Heap}[\llbracket e \rrbracket, p].\mathtt{mask} \; \} \\
\llbracket \textbf{inhale } \mathtt{\textbf{unfolding}} \; e.p \; \mathtt{\textbf{in}} \; e' \rrbracket^{b}_{\mathtt{M}} &= \textbf{\#if } (b) \; \{ \; \llbracket \textbf{inhale } \mathtt{body}(\llbracket e \rrbracket, p) \rrbracket^{\mathtt{false}}_{\mathtt{Heap}[\llbracket e \rrbracket, p].\mathtt{mask}} \; ; \; \} \\
& \; \textbf{assume } \llbracket e' \rrbracket
\end{aligned}
$$

Fig. 4. Encoding of inhale. The **#if** -conditionals are resolved by the translator.

When exhaling or inhaling an **`unfolding`** expression for a predicate instance $e.p$, the permissions in the body of $e.p$ become known-folded permissions. Therefore, they get inhaled into the predicate mask of $e.p$. In this case (which is indicated by b being false), inhaling a predicate permission $e.p$ "copies" any known-folded permissions from the inner predicate instance to the outer; that is, we flatten the known-folded permissions under $e.p$. Due to this flattening, an **`unfolding`** expression affects permissions only when we are inhaling direct permissions (that is, b is true). If we are already inhaling known-folded permissions, any permissions folded under an inner predicate instance are already part of the known-folded masks of the outer predicates.

Encoding of Source Statements. Chalice statements are generally desugared into appropriate combinations of **inhale** and **exhale** statements. For example, a (void) method call is simply encoded with an **exhale** of the precondition followed by an **inhale** of the postcondition (see Fig. 5). The havocing of heap locations that takes place as part of the exhale takes care of invalidating any information that the called method may have changed.

`fold` and **`unfold`** statements do not require a general havoc of the heap, since no permissions are actually released to another method; they are simply reorgan-

[4] In this paper, this can only happen for trivial predicates without permissions, but the assumption is important once fractional permissions are employed.

isations of permissions amongst predicate instances. Therefore, their translations use **exhale′**. In addition to swapping a predicate instance with its body, **`fold`** statements record known-folded permissions, and **`unfold`** statements havoc the version of the predicate instance.

We verify loops as usual by using a loop invariant and verifying the loop body independently of the surrounding code. To access locations in the loop body, the verifier requires the appropriate permissions in the loop invariant. To communicate knowledge about known-folded locations between the loop body and the surrounding context, one can use an **`unfolding`** expression in the invariant. Our encoding of **`unfolding`** then ensures that the necessary known-folded permissions are added when the loop invariant is evaluated.

$$
\begin{aligned}
\llbracket \textbf{call}\ e.m() \rrbracket &= \llbracket \textbf{exhale}\ \texttt{pre}(\llbracket e \rrbracket, m) \rrbracket\ ;\ \llbracket \textbf{inhale}\ \texttt{post}(\llbracket e \rrbracket, m) \rrbracket^{\texttt{true}}_{\texttt{Mask}} \\[1ex]
\llbracket \textbf{fold}\ e.p \rrbracket &= \llbracket \textbf{exhale}'\ \texttt{body}(\llbracket e \rrbracket, p) \rrbracket\ ;\ \llbracket \textbf{inhale}\ \texttt{body}(\llbracket e \rrbracket, p) \rrbracket^{\texttt{false}}_{\texttt{Heap}[\llbracket e \rrbracket, p].\texttt{mask}}\ ; \\
&\quad\ \llbracket \textbf{inhale}\ \textbf{acc}(e.p) \rrbracket^{\texttt{true}}_{\texttt{Mask}} \\[1ex]
\llbracket \textbf{unfold}\ e.p \rrbracket &= \llbracket \textbf{exhale}'\ \textbf{acc}(e.p) \rrbracket\ ;\ \textbf{havoc}\ \texttt{Heap}[\llbracket e \rrbracket, p].\texttt{version}\ ; \\
&\quad\ \texttt{Heap}[\llbracket e \rrbracket, p].\texttt{mask} := \emptyset\ ;\ \llbracket \textbf{inhale}\ \texttt{body}(\llbracket e \rrbracket, p) \rrbracket^{\texttt{true}}_{\texttt{Mask}}
\end{aligned}
$$

Fig. 5. Encoding calls, **`fold`**, and **`unfold`** statements. `pre`() and `post`() yield the precondition and postcondition of a method with the appropriate substitutions.

5 Encoding of Abstraction Functions

In this section, we present our encoding of abstraction functions, building upon the handling of predicates and known-folded permissions in the previous section. The presented approach is based on uninterpreted functions and axioms, but again avoids presenting the prover with recursive axioms that can be applied in an unbounded way. This is achieved by a combination of versioning predicate instances, and careful selection of axiom triggering strategies [7].

5.1 Function Definitional Axiom

Each Chalice function is represented by a corresponding uninterpreted function in the generated Boogie program, in which the heap, the receiver as well as parameters to the Chalice function are turned into explicit parameters. For example, function `length()` from Fig. 1 gives rise to the following uninterpreted Boogie function declaration (by convention, we prepend a # symbol to the function name, to differentiate it from its Chalice counterpart):

```
function #length(heap: HeapType, this: ref) returns (int);
```

In Boogie, it is standard to specify properties of an uninterpreted function via a *definitional axiom*. In the common case that this axiom introduces universal quantifiers, the prover needs a strategy to decide when to instantiate the axiom. For example, consider the “direct” translation of the Chalice definition of function length(), as a Boogie axiom (we assume here that H, M, and this range over heaps, masks, and references, respectively):

```
axiom ∀ H, M, this • M[this,valid] ⟹
  #length(H,this) = (H[this,next] ≠ null ? (1 + #length(H,H[this,next])) : 1)
```

This axiom states that, provided that the function’s precondition holds, a function application is always equal to its body. Allowing the prover to instantiate this axiom in arbitrary ways, would lead to two problems. First, the prover might instantiate it with a heap and a mask that belong to different execution states. Second, since the function is recursive, the prover might instantiate its definitional axiom indefinitely in a matching loop.

To solve the first problem, our encoding introduces a boolean uninterpreted function state which takes a heap and a mask as arguments. The function application state(Heap, Mask) is assumed to yield true in our encoding every time a state is changed, for instance, after an exhale is translated (but not for the intermediate states during the translation). All axioms that quantify over a heap and mask use the state function as a premise. We write ∀ (H,M) • P(H, M) to abbreviate ∀ H,M • state(H, M) ⇒ P(H, M).

To solve the second problem, we make use of Z3’s and Boogie’s facility to associate universal quantifiers with sets of syntactic *triggers*. Triggers are matching patterns denoted by the syntax {t1,t2,...}, where the terms t1, t2, etc. mention the variables bound by the universal quantifier. When the SMT solver has syntactically seen expressions e1, e2, etc. that match a trigger (modulo known equalities), it may instantiate the universal quantifier with the sub-expressions that correspond to the bound variables in the trigger.

In the following, we explain how we encode abstraction functions and use triggers in a way that allows the prover to obtain sufficient information from the recursive definitions of the functions, without having the possibility of entering a matching loop.

Limited and Unlimited Functions. To avoid matching loops in the definitional axioms for abstraction functions, we adopt a technique employed in other tools [17]. For each Chalice function we introduce *two* Boogie functions (called the *limited* and *unlimited functions*). Their logical meanings are intuitively the same, but their practical use in axioms and triggers is different. For example, for the length() function, we introduce the limited form (which we identify by adding a ’ to the name) along with the original definition:

```
function #length(heap: HeapType, this: ref) returns (int);
function #length’(heap: HeapType, this: ref) returns (int);
```

Now, we define the definitional axiom above as follows: every occurrence of a function application which comes from the body of the function definition, is replaced by its corresponding limited function. The unlimited form of the function is used in the trigger set for the axiom (and is still used in the translation of source-level Chalice expressions). For example, for `length` we generate the following definitional axiom (all our axioms that quantify over both the heap and mask include `state(H,M)` as part of each trigger, but we omit this for conciseness):

```
axiom ∀ (H,M), this • {#length(H,this)} M[this,valid] ⟹
  #length(H,this) = (H[this,next] ≠ null ? (1 + #length'(H,H[this,next])) : 1)
```

Because the body of this axiom does not introduce any new applications of the unlimited function `#length()`, an instantiation of the axiom does not give rise to any further instantiations; the potential matching loop is avoided. In order to give a meaning to the `#length'()` function, the following addition axiom is used, which also does not introduce any new applications of the unlimited function `#length()`.

```
axiom ∀ (H,M), this • {#length(H,this)} #length'(H, this) = #length(H, this)
```

Effectively, this allows to prover to unroll a function's definition exactly once for any given occurrence of that function in the source program.

Controlled Triggering. To implement the idea presented in Sec. 3.1 of allowing the prover to unroll a function definition when the corresponding predicate has been folded or unfolded at some point in the program, we proceed as follows. We introduce a boolean function for every predicate, to be used as a trigger for functions that depend on it. We illustrate this using the predicate `valid`:

```
  function #valid^trig(this: ref) returns (bool);
```

This function indicates that the corresponding predicate has been folded or unfolded in *some* state for the given receiver. This is introduced in our encoding by instrumenting the translation of **fold** `e.valid` and **unfold** `e.valid` with an extra assumption of `#valid`$^{\mathtt{trig}}$`(e)`.

Additionally, we are interested in unrolling the definitional axiom for a function application with a given list of arguments only if that application for the same arguments has been mentioned somewhere in the program. Otherwise, the prover cannot learn useful information by expanding the function's definition. To this end, we add another boolean function along with an axiom:

```
  function #length^trig(this: ref) returns (bool);
  axiom ∀ (H,M), this • {#length'(H, this)} #length^trig(this)
```

We use the function application `#length`$^{\mathtt{trig}}$`(e)` in triggers to indicate that `length` has been applied to the receiver `e` in *some* state. The axiom shown encodes this meaning.

We now add the following trigger set to our definitional axiom of `length`. It allows the prover to instantiate the definitional axiom in the cases described in Sec. 3.1, but still does not cause matching loops. Note that this trigger corresponds to our "key insight" in the introduction since it allows the prover to expand recursive definitions up to the depth at which the program has inspected the data structure.

```
{ #length$^{trig}$(this), #valid$^{trig}$(this) }
```

Such a trigger set is introduced for each predicate in the precondition of the function that also gets unfolded in the body of the function before any recursive function calls that occur. If the predicate is not unfolded, or the recursive calls are not in the body of the corresponding **unfolding** expression, the recursion of the function is not tied to traversing the predicate over the data structure, and the additional trigger is not added.

5.2 Function Framing Axiom

To frame functions, we employ an additional axiom which essentially states that, if no part of the state mentioned in a function's precondition differs between two heaps, then the function's value is also the same in the two heaps. As we discussed in Sec. 3.3, we use predicate versions to abstract over the locations folded into the predicate instance, thus avoiding giving recursive predicate definitions to the prover. For example, the framing axiom for `length()` is as follows:

```
axiom ∀ ($H_1$, $M_1$), ($H_2$, $M_2$), this • {#length'($H_1$,this),#length'($H_2$,this)}
  $H_1$[this,valid].version = $H_2$[this,valid].version ⟹
    #length'($H_1$, this) = #length'($H_2$, this)
```

This axiom is phrased in terms of the limited function `#length'`, but by the axiom relating unlimited and limited functions presented in the previous subsection, it can also be used to frame unlimited functions.

Note that our use of predicate versioning is asymmetrical; if a predicate version is the same in two heaps, then we assume the contents of the predicate instance to be the same, but not vice versa. For example, unfolding and folding a predicate (without modifying any contents) will yield a new version (see Fig. 5). However, our function definitional axiom will be triggered in this case, deducing the relationship to the locations immediately contained inside the predicate instance. If these are known to be preserved, then the prover can conclude that the function value has not changed.

6 Soundness

In this section, we give an informal justification for the soundness of our approach. As we have explained in Sec. 3.2, the definitional axioms for the Boogie functions that we use in our encoding are consistent. The soundness arguments in this section furthermore justify (1) our approach to function framing, and (2) our approach to heap location framing.

6.1 Soundness of Function Framing

To justify our approach to function framing, we use the following definition:

Definition 1 (Permission and Heap Footprints). *The* permission footprint of a predicate *in a heap is the set of heap locations defined by recursively evaluating the predicate definition in the given heap, and collecting the locations whose permissions are required by the definition.*

The permission footprint of a function *in a heap is the set of heap locations defined by evaluating the function's precondition in the given heap, and collecting the locations whose permissions the precondition requires, as well as the permission footprints of all predicates the precondition requires in that evaluation.*

The heap footprint of the predicate or function *in a heap is the set of location-value pairs such that the location is in the corresponding permission footprint, and the value is the value stored in the heap at that location.*

It is sufficient to observe the following:

(1) Evaluation of a function application (at runtime) reads only locations in its permission footprint. This property is enforced by a well-formedness check for each function definition, as is standard for logics supporting user-supplied definitions. In particular, a function application's value is a function of its receiver, arguments, and its heap footprint. Function bodies can also apply functions, but the checking of preconditions ensures that heap footprints for recursive applications are always subsets of the original one.

(2) Like functions, predicate definitions are checked to ensure that they read only heap locations that fall within their permission footprint. Thus, the permission and heap footprints of a predicate are fixed by the permissions folded inside of it. Since we havoc version numbers whenever a predicate is unfolded or exhaled, the version of a predicate location at two different program points can be known to be the same only if the heap footprint of the predicate is also identical at both points.

(3) Consider a situation in which our framing axiom allows the prover to equate a function value between two program points. By (1), we know that it is sufficient to know that the function's heap footprint remains the same between both points. The heap footprint is made up of locations to which explicit permission is required in the function's precondition (which the axiom requires to have the same values), and the heap footprints of those predicate locations that the function's precondition requires (which the axiom requires to have the same versions). By (2), the function value is the same in both states.

6.2 Soundness of Heap Location Framing

For the soundness of heap location framing, the argument depends on a precise definition of the notions of folded and known-folded locations for predicate instances. We address here the soundness of our use of predicate masks to record the known-folded permissions per predicate instance. Our encoding maintains an invariant regarding these masks, but its intuition is simple; we record and remember known-folded permissions until the corresponding predicate instances are lost, and make sure they are recorded in the masks of *all* predicate instances that enclose the locations. It is also important for our argument that we do not selectively record the known-folded permissions from a predicate's body, but always record everything that the body depends on at a time.

We need to define several notions to explain our argument. Since many of our definitions treat both types of location uniformly, we use the meta variable l to range over both predicate and field location names.

Definition 2 (Folded Locations).

- *In a heap* $\mathtt{H}$*, a location* $o'.l'$ *is* directly folded *inside a predicate location* $o.p$*, written* $\mathit{directFolded}(o,p,o',l',\mathtt{H})$*, if evaluating the body of the predicate instance* $o.p$ *in* $\mathtt{H}$ *results in directly requiring permission to the location* $o'.l'$*.*
- *In a heap* $\mathtt{H}$*, a location* $o'.l'$ *is* folded *inside a predicate location* $o.p$*, written* $\mathit{folded}(o,p,o',l',\mathtt{H})$*, as defined by (the least fixpoint of):*

$$\begin{aligned}&\mathit{folded}(o,p,o',l',\mathtt{H}) \;\Leftrightarrow\; (\mathit{directFolded}(o,p,o',l',\mathtt{H}) \vee \\ &\quad \exists o'',p''.(\mathit{directFolded}(o,p,o'',p'',\mathtt{H}) \;\wedge\; \mathit{folded}(o'',p'',o',l',\mathtt{H})))\end{aligned}$$

We now provide the definitions which characterise the auxiliary information that our encoding records about known-folded permissions.

Definition 3 (Recorded Predicate Bodies and Known-Folded Permissions).

- *In a heap* $\mathtt{H}$*, a predicate location* $o.p$ *has its* body recorded*, as defined by:*

$$\begin{aligned}&\mathit{bodyRecorded}(\mathtt{H},o,p) \;\Leftrightarrow\; (\forall o',l'.(\mathit{directFolded}(o,p,o',l',\mathtt{H}) \;\Rightarrow \\ &\quad \mathtt{H}[o,p].\mathtt{mask}[o',l'] \;\wedge \\ &\quad (l' \text{ is a predicate location } \;\Rightarrow\; \mathtt{H}[o',l'].\mathtt{mask} \subseteq \mathtt{H}[o,p].\mathtt{mask}))\end{aligned}$$

- *In a heap* $\mathtt{H}$*, a location* $o'.l'$ *is* known-folded *inside a predicate location* $o.p$*, written* $\mathit{knownFolded}(o,p,o',l',\mathtt{H})$*, as defined by (the least fixpoint of):*

$$\begin{aligned}&\mathit{knownFolded}(o,p,o',l',\mathtt{H}) \;\Leftrightarrow\; \mathit{bodyRecorded}(\mathtt{H},o,p) \;\wedge \\ &\quad (\mathit{directFolded}(o,p,o',l',\mathtt{H}) \vee \\ &\quad \exists o'',p''.(\mathit{directFolded}(o,p,o'',p'',\mathtt{H}) \;\wedge\; \mathit{knownFolded}(o'',p'',o',l',\mathtt{H})))\end{aligned}$$

Our definition of *bodyRecorded* requires not only that *every* location directly required by a predicate body is stored in the corresponding predicate mask, but also that the recorded information is transitive; any information in predicate masks for nested predicate instances must be included in the level above (in the definition, $M \subseteq M'$ denotes that mask M' is true at least for the locations for which M is true). Our definition of *knownFolded* insists on this organisation of the information in predicate masks "all the way down"—all of the predicate instances in between must also satisfy *bodyRecorded*. This definition of known-folded locations approximates the folded locations for a predicate instance (the definitions are similar, but *knownFolded* enforces extra constraints).

One of the properties we assume about the underlying methodology, is that locations to which folded permission is held, never have their permissions also in the direct mask. This is a special case of the more-general property that permissions should never be forged/duplicated, but only transferred. Note that this is a soundness property of the underlying semantic model.

Lemma 1 (Folded Locations cannot be in the Direct Mask). *Before and after every Chalice program statement, it holds that*

$$\forall o, p, o', l'.\ (\mathit{Mask}[o,p] \ \wedge\ \mathit{folded}(o,p,o',l',\mathit{Heap})) \ \Rightarrow\ \neg\mathit{Mask}[o',l']$$

Finally, we can state the invariant that describes how the information in our predicate masks relates to the definition of *knownFolded* locations above:

Theorem 1 (Predicate Mask Permissions are Known-Folded Locations). *Our encoding preserves the following invariant:*

$$\begin{array}{l}\forall o, p, o', l'.(\mathit{Heap}[o,p].\mathtt{mask}[o',l'] \ \Rightarrow \\ \quad ((\mathit{Mask}[o,p] \ \vee\ \exists o'', p''.(\mathit{Mask}[o'',p''] \ \wedge \mathit{knownFolded}(o'',p'',o,p,\mathit{Heap}))) \ \wedge \\ \quad \mathit{knownFolded}(o,p,o',l',\mathit{Heap})))\end{array}$$

Proof Sketch. We show that the property is preserved across the four most relevant operations concerning known-folded permissions: folding, unfolding, inhaling, and exhaling predicate instances. For each operation, we assume the invariant holds beforehand, and show that it holds afterwards. To do this, we consider the cases in which $\mathtt{Heap}[o,p].\mathtt{mask}[o',l']$ can newly have been made true by the operation (in which case the consequent of the implication must also be checked), as well as the cases in which the consequent of the implication may have been falsified by the operation (in which case we must be sure that the antecedent is also now false).

Folding $o_1.p_1$: Consider the set of locations directly required in the body of $o_1.p_1$. Permissions to all of these locations are removed from `Mask` and added to $\mathtt{Heap}[o_1,p_1].\mathtt{mask}$. Furthermore, any of these locations which are predicate locations have their predicate masks copied into $\mathtt{Heap}[o_1,p_1].\mathtt{mask}$ (cf. Fig. 3). In particular, these operations result in $\mathit{bodyRecorded}(\mathtt{Heap}, o_1, p_1)$ holding. Finally, $\mathtt{Mask}[o_1,p_1]$ is set to `true`. Since these operations only add to predicate masks, they do not falsify any previous instances of *knownFolded*.

Considering the invariant, the antecedent $\mathtt{Heap}[o,p].\mathtt{mask}[o',l']$ can newly have been made true only in the case $o = o_1$, $p = p_1$ and for $o'.l'$ being one of the locations directly folded in the body of $o_1.p_1$. For all such cases, we have $\mathit{knownFolded}(o_1,p_1,o',l',\mathtt{Heap})$ as required. On the other hand, since a set of locations is removed from `Mask`, we must also take care that the antecedent is not falsified when $o.p$ is one of those locations. However, since all such locations are recorded in $\mathtt{Heap}[o_1,p_1].\mathtt{mask}$ by the operation, the second disjunct can be shown in these cases, taking $o'' = o_1$ and $p'' = p_1$.

Unfolding $o_1.p_1$: This operation sets $\mathtt{Mask}[o_1,p_1]$ to false, and removes any information associated with the predicate instance. In particular, $\mathtt{Heap}[o_1,p_1].\mathtt{mask}$

is set to false for all locations, which we write as $\emptyset$. Since we do not add to any predicate masks, we cannot make the antecedent of the invariant true in any new cases. Since we only remove information from the predicate mask $\texttt{Heap}[o_1, p_1].\texttt{mask}$, the only instances of *knownFolded* that can be falsified by this operation are those concerning locations known-folded in $o_1.p_1$; i.e., those $o_2.l_2$ for which $\mathit{knownFolded}(o_1, p_1, o_2, l_2, \texttt{Heap})$ holds (note that by Lemma 1, we know that $o_1.p_1$ was not itself folded inside any predicate instance). Thus, considering the consequent of the invariant, the only cases we need worry about are when either $o = o_1$ and $p = p_1$ (in which case, the antecedent of the invariant is necessarily false, since we reset the predicate mask to $\emptyset$), or $o'' = o_1$ and $p'' = p_1$. In this latter case, unrolling the definition of $\mathit{knownFolded}(o'', p'', o, p, \texttt{Heap})$ in the state before the operation, in combination with the fact that we record all direct-folded locations from the body of $o_1.p_1$ in the $\texttt{Mask}$, provides the necessary information to show that the invariant still holds.

Inhaling $o_1.p_1$: This operation simply sets $\texttt{Mask}[o_1, p_1]$ to true, which cannot falsify the consequent of the invariant, or make the antecedent newly true.

Exhaling $o_1.p_1$: This operation sets $\texttt{Mask}[o_1, p_1]$ to false, and then generates a "global havoc", constrained by the assumption $H' \overset{\texttt{Mask}}{\equiv} \texttt{Heap}$. Let's consider the point just after this havoc operation (see Fig. 3), but before the new heap H' is assigned to $\texttt{Heap}$ (so that we have names for both the new and old heaps). The havoc operation means that all predicate masks $H'[o.p].\texttt{mask}$ are set to $\emptyset$, except in the case that either $\texttt{Mask}[o, p]$ still holds after the operation, or, for some $o''.p''$, $\texttt{Mask}[o'', p''] \wedge \texttt{Heap}[o'', p''].\texttt{mask}[o, p]$ holds. In particular, consider the cases in which the antecedent of the invariant $H'[o, p].\texttt{mask}[o', l']$ could possibly hold. This would require that $H'[o, p].\texttt{mask} \neq \emptyset$, and, since nothing is added to the predicate masks in the operation, also that $\texttt{Heap}[o, p].\texttt{mask}[o', l']$ held. By the argument above, along with the assumption of the invariant in the state before the operation, we deduce that:

$$\begin{aligned} & H'[o, p].\texttt{mask}[o', l'] \;\Rightarrow \\ & \quad ((\texttt{Mask}[o, p] \;\vee\; \exists o'', p''.(\texttt{Mask}[o'', p''] \;\wedge \mathit{knownFolded}(o'', p'', o, p, \texttt{Heap})) \wedge \\ & \quad \mathit{knownFolded}(o, p, o', l', \texttt{Heap}))) \end{aligned}$$

Thus, all we need to know in order to deduce that the invariant holds in the new heap H' is that the occurrences of *knownFolded* mentioned here are still true for H'. This follows because, in both cases, the known-folded information is recorded under a predicate instance to which direct permission is held. By the soundness of the underlying permission logic, we know that the locations folded inside this outer predicate instance cannot be modified, and thus, that the meanings of all nested predicates are preserved. Furthermore, any predicate instances known to be folded inside this outer instance must have their entire bodies recorded in the corresponding predicate mask, and therefore, their meanings and directly folded locations remain unaffected by the global havoc. Therefore, a simple induction shows that *knownFolded* information is preserved in the new heap. □

Corollary 1 (Predicate Masks record only Folded Permissions). *Before and after the translation of every Chalice statement, the following property holds:*

$$\forall o, p, o', l'.\ (\texttt{Mask}[o, p] \ \wedge\ \texttt{Heap}[o, p].\texttt{mask}[o', l'] \ \Rightarrow\ \mathit{folded}(o, p, o', l', \texttt{Heap}))$$

By this corollary, which follows directly from Theorem 1, the locations framed across a global havoc are always either locations to which direct permission is held, or which are folded inside a predicate instance to which direct permission is held; in both cases, the soundness of the underlying permission handling implies that framing these location values is sound.

7 Related Work

Verification Condition Generation (VCG) is a popular technique for the construction of automated verifiers [2, 5, 10, 15, 16, 18]. The concept of abstract predicates was introduced by [21] and it is used in separation logic [22] and implicit dynamic frames [25]. In VCG verifiers, abstraction functions are used together with abstract predicates in Chalice [18] and VeriCool [25].

The VeriCool VCG implementation handles abstract predicates and abstraction functions soundly but, unlike our solution, introduces the possibility of matching loops in the SMT solver. To assess whether matching loops are a problem in practice, we took an example from VeriCool and translated it into Chalice. We experimented with partial specifications by leaving out parts of the main loop invariant. Such verification attempts are common when specifying a program and refining its contracts until the verification succeeds. We found that the verification time significantly increased for failed proof attempts (due to partial specifications) from 2.3 minutes to up to one hour (at which point we terminated the verifier). In some cases, removing logically redundant parts of the specification also increased the verification time (to 16 minutes). These are typical symptoms of matching loops. Our new version of Chalice verifies the translated program in less than 20 seconds (with or without the redundant specifications) and reports failed attempts for partially specified versions even quicker. Details of these examples can be found at [1].

The previous Chalice encoding of predicates and functions is unsound. The encoding considered only direct permissions and havoced the heap *lazily*, that is, when permissions are (re-)obtained. This ensures that values of fields are preserved, but also leaves invalid information in the heap, which caused the unsound behaviour. In particular, folded locations were never havoced.

Symbolic execution is an alternative technique to VCG and is used in tools such as [3, 9, 13, 14, 24]. Typically, symbolic execution engines use partial heaps and other more elaborate data structures for the representation of the program state. In the presence of such data structures, the problem treated in this paper is not as intricate. In particular, symbolic execution engines can iterate through their heap representation to determine folded permissions, whereas for VCG with SMT solvers, this is not possible in the presence of recursive predicates. Symbolic execution forgets heap information by chopping off the corresponding part from the partial heap. Framing of function values can be sufficiently handled as in VCG by predicate versioning.

The mechanism of predicate versioning typically seen in symbolic execution engines differs from ours. A version in these systems is a snapshot of the underlying heap. If a predicate is unfolded and folded back immediately, its version does not change and functions depending on that predicate are known to not have changed their value either. In contrast, our approach always changes the version number at **unfold** statements. However, the use of the definitional axiom allows the prover to unroll the definition one level and prove function equivalence nonetheless. In this way, we can achieve the same effect as in the predicate versioning of symbolic execution, without using the symbolic execution-specific data structures, which are unsuitable in a VCG setting. Versioning in Spec# [2] is also similar to our predicate versioning, but more incomplete: the user must explicitly mention the functions whose return values must be preserved.

Madhusudan et al. [20] present a logic to express complex properties of tree structures, as well as a procedure that decides such properties efficiently. Their core idea of expanding a recursive definition a statically known number of times that depends on the program under verification is similar to ours. Compared to our work, their logic is restricted in expressiveness: it tackles only tree structures and it considers only functions with a single tree argument and a specific definition pattern.

Shape Analysis has been successfully applied in the context of three-valued logic [23], and more recently adapted by many works to the setting of separation logic [8], with the aim of *inferring* the recursive structure of the current heap. The approach is typically based on a fixed set of recursive definitions, but some techniques (e.g., [11]) aim also to infer these recursive definitions. The idea that recursive definitions need only be handled up to a certain depth is central to shape analysis, but the problem tackled is different; we do not aim at such inference, while we do support arbitrary user-defined predicates and dependent abstraction functions for use with our encoding.

Suter et al. provide a generic approach to constructing decision procedures for recursive algebraic data types [27]. In particular, their work supports recursive abstraction functions, to allow a more abstract representation of the underlying data to be used in specifications. While their approach applies only to functional data types, they employ a notion of partial evaluation of the abstraction functions that is similar to our controlled instantiation of function definitions. It would be interesting to see if their work, as well as work on shape analyses could be adapted to our setting, perhaps to infer **unfold** and **fold** statements.

Triggering is a major challenge in the use of SMT solvers [7]. Our treatment of function definitional axioms involves intricate triggering, which extends the idea of limited functions [17].

8 Conclusion

In this paper, we have presented a VCG encoding technique for abstract predicates and abstraction functions. To prevent matching loops one must refrain from giving recursive definitions to the prover, even though in general the definitions of both predicates and functions can be recursive. We solve this challenge with the insight that proof obligations can typically be discharged by allowing the prover to unroll recursive definitions for the parts of the program data structures which have been observed by the program at some earlier point. Inspecting the program text allows us to encode this with the use of trigger strategies as well as the introduction of known-folded permissions.

Our encoding is, to the best of our knowledge, the only sound encoding of both features that prevents matching loops. Our comparison with the VeriCool VCG verifier shows that prevention of matching loops makes an important difference in the verification experience.

We have implemented the methodology for the more general setting of fractional permissions [4] in a new version of Chalice[5]. We ran our implementation on the Chalice test suite of 100 interesting examples and regression tests and observed no unsoundness or incompleteness. The timings are predictable, even for examples with faulty specifications. Our tool can be tried out online [1], where we also provide several challenging examples.

Acknowledgements. We would like to thank Jan Smans for explaining to us the operation of VeriCool and Malte Schwerhoff for various discussions and for making our implementation available online. We are especially grateful to the reviewers of FM 2012 for their insightful comments on an earlier version of this paper, which lead to a major redesign and improvement of our technique.

References

1. Chalice (online). `http://boogiebox2.inf.ethz.ch:1001/tuwin/tool/chalice`.
2. M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. *CACM*, 54(6):81–91, 2011.
3. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, volume 4111 of *LNCS*, pages 115–137. Springer, 2006.
4. J. Boyland. Checking interference with fractional permissions. In *SAS*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
5. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskał, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*. Springer, 2009.
6. Á. Darvas and P. Müller. Reasoning about method calls in interface specifications. *JOT*, 5(5):59–85, 2006.

[5] We provide an artifact containing a precompiled binary as well as the source code of our verifier at `http://www.pm.inf.ethz.ch/research/chalice/chalice.zip`

7. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
8. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.
9. D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In *OOPSLA*, pages 213–226. ACM, 2008.
10. J. C. Filliâtre. Why: a multi-language multi-prover verification tool. Technical Report 1366, LRI, Université Paris Sud, 2003.
11. B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *PLDI*, pages 256–265. ACM, 2007.
12. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.
13. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *APLAS*, volume 6461 of *LNCS*, pages 304–311. Springer, 2010.
14. I. T. Kassios, P. Müller, and M. Schwerhoff. Comparing verification condition generation with symbolic execution: an experience report. In *VSTTE*, volume 7152 of *LNCS*, pages 196–208. Springer, 2012.
15. K. R. M. Leino. Specification and verification of object-oriented software. In *Marktoberdorf International Summer School 2008, Lecture Notes*, 2008.
16. K. R. M. Leino. This is Boogie 2. Working Draft. Available at `http://research.microsoft.com/en-us/um/people/leino/papers.html`, 2008.
17. K. R. M. Leino and R. Monahan. Reasoning about comprehensions with first-order smt solvers. In *SAC*, pages 615–622. ACM, 2009.
18. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP*, volume 5502 of *LNCS*, pages 378–393. Springer, 2009.
19. K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *FOSAD V*, volume 5705 of *LNCS*, pages 195–222. Springer, 2009.
20. P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive proofs for inductive tree data-structures. In *POPL*, pages 123–136. ACM, 2012.
21. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM, 2005.
22. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*. IEEE Computer Society Press, 2002.
23. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, pages 105–118. ACM, 1999.
24. J. Smans, B. Jacobs, and F. Piessens. VeriCool: An automatic verifier for a concurrent object-oriented language. In *FMOODS*, volume 5051 of *LNCS*, pages 220–239. Springer, 2008.
25. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, volume 5653 of *LNCS*, pages 148–172. Springer, 2009.
26. A. J. Summers and S. Drossopoulou. A formal semantics for isorecursive and equirecursive state abstractions. Technical Report 773, ETH Zurich, 2012.
27. P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL*, pages 199–210. ACM, 2010.