Lightweight Support for Magic Wands in an Automatic Verifier

Malte Schwerhoff and Alexander J. Summers

ETH Zurich, Switzerland
malte.schwerhoff@inf.ethz.ch
alexander.summers@inf.ethz.ch

Abstract. Permission-based verification logics such as separation logic have led to the development of many practical verification tools over the last decade. Verifiers employ the *separating conjunction* $A * B$ to elegantly handle aliasing problems, framing, race conditions, etc.
Introduced along with the separating conjunction, the *magic wand* connective, written $A \mathrel{-\!\!*} B$, can describe hypothetical modifications of the current state, and provide guarantees about the results. Its formal semantics involves quantifying over states: as such, the connective is typically not supported in automatic verification tools. Nonetheless, the magic wand has been shown to be useful in by-hand and mechanised proofs, particularly for specifying loop invariants and partial data structures.
In this paper, we show how to integrate support for the magic wand into an automatic verifier, requiring low specification overhead from the tool user. We present additional features to make the magic wand interact elegantly with abstract predicates, abstraction functions and "old" expressions. Our solution is compatible with a variety of logics and underlying implementation techniques.
Our approach is implemented, and a prototype is available to download.

1 Introduction

Permission-based verification logics, most notably separation logic [15], have been widely developed in recent years, both to explore their theoretical properties and to serve as the bases for a variety of practical tools. The most well-known feature of separation logic is its *separating conjunction* connective, $*$. An assertion of the form $A * B$ intuitively expresses that the two conjuncts hold for *separate* portions of the program heap; support for this connective can be used to handle aliasing, framing, race conditions etc.

The *magic wand* connective $\mathrel{-\!\!*}$ was introduced along with the separating conjunction. In a semantics based on *partial heaps* h (i.e., heaps modelled by partial functions), its meaning can be defined as follows:

$$h \vDash A \mathrel{-\!\!*} B \;\Leftrightarrow\; \forall h' \perp h.\ (h' \vDash A \;\Rightarrow\; h \uplus h' \vDash B)$$

Here, $h' \perp h$ expresses that the two states are *compatible* (neither do both require access to the same heap location, nor do they disagree on any heap values), while

$h \uplus h'$ denotes the *composition* of two partial heaps. Informally, an assertion of the form $A \mathrel{-\!\!*} B$ can be understood as describing the effect of a hypothetical modification of the current state (h, above): "if we add on any partial heap satisfying A, then B will hold in the resulting state".

Despite being included in the earliest works on separation logic, the magic wand connective is generally not supported in automatic verifiers built upon separation logic (and related) theories [2, 6, 8, 13]. The quantification over states in the wand's semantics makes lightweight support for the wand difficult, and with *no* user direction, reasoning with magic wand assertions is undecidable [5].

The magic wand is, however, an important reasoning tool, particularly for specifying partial versions of data structures, or protocols regarding their modification [10, 7, 9]. When working directly with loop invariants, iterative traversals of data structures can be awkward to specify [21, 14] in separation logics; as we illustrate in this paper, the magic wand can provide a simple solution.

Contributions. In this paper, we provide support for the magic wand in an automated verifier, with lightweight annotation overhead for the user. The core of our approach is an algorithm to automatically compute *footprints* for magic wand instances, on which we build our encoding of wand-related operations. We present extensions of this approach to integrate *ghost operations* (such as *folding* predicates), and a novel expression construct ("`now`" expressions) which enables functional properties of wands to be simply expressed. The running example in our paper focuses on specifying a loop invariant, but our support for magic wands is general, and can be directly employed for other use-cases. Our approach is implemented, and our prototype is available to download [1].

2 Background and Presentation

In the PhD of Smans [19], the verification logic *implicit dynamic frames* was introduced, providing permission-based reasoning similar to separation logic (which can be encoded [17]), but with specifications closer to the (object-oriented) programming language. We present our work with respect to a variant of implicit dynamic frames, based on that used in the Chalice [13] tool.

Definition 1 (Assertions and Expressions). Assertions*, ranged over by A, and* expressions*, ranged over by e, are defined as follows (in which x ranges over variables, f over field names, g over functions, P over predicates):*

$$A ::= e \mid \texttt{acc}(e.f) \mid P(e) \mid A \mathbin{\&\&} A \mid e \Rightarrow A \mid A \mathrel{-\!\!*} A$$
$$e ::= x \mid e.f \mid g(e) \mid !e \mid e \mathbin{op} e \mid e \mathbin{?} e : e \mid \texttt{unfolding}\, P(e)\, \texttt{in}\, e \mid \texttt{old}(e)$$

Permissions are managed in the logic via *accessibility predicates* $\texttt{acc}(e.f)$, which denote the permission to access heap location `e.f`. The conjunction && behaves as the separating conjunction $*$ in separation logics; in particular, an assertion $\texttt{acc}(x.f) \mathbin{\&\&} \texttt{acc}(y.f)$ is analogous to the separation logic assertion $x.f \mapsto \_ *$

$y.f \mapsto \_$, and implies that $x \neq y$. $P(e)$ represents an instance of a predicate, while $g(e)$ is an invocation of a function; in this paper, we provide only unary predicates and functions, for simplicity of presentation.

In contrast to separation logic, implicit dynamic frames allows *heap-dependent expressions* such as $x.f_1.f_2 > 0$ to be used in assertions. This allows for a different style of specification; in particular, heap-dependent *functions* can be defined and used in expressions, similarly to the use of pure methods in assertions in first-order-logic-based approaches such as e.g. JML [11]. Heap-dependent expressions are only guaranteed a meaningful semantics when they are *framed* by the permissions held in the state in which they are evaluated, meaning that for any heap location dereferenced by the expression, a corresponding permission must currently be held. Accessing heap locations in program statements is similarly restricted; e.g. a field read `x := y.f` is allowed only in states in which a permission to the location `y.f` is held. An assertion is said to be *self-framing* if it requires at least permissions to those locations on which the expressions it mentions depend. For example, the assertion `x.f > 0` is not self-framing, but the assertion `acc(x.f) * x.f > 0` is. All assertions expressible in separation logic are self-framing by construction [17]. To simplify this check, we require permissions to locations to be syntactically mentioned before the location is dereferenced in an assertion (this is analogous to the restriction in some separation-logic-based tools that logical variables be bound to heap locations before their use).

2.1 Programming Language and Running Example

We present our work in the context of a small intermediate verification language called *SIL*, which includes variables and object fields, methods with pre/post-conditions, loops with invariants and if-conditionals. From a verification perspective, proof obligations can be expressed via *exhaling* and *inhaling* assertions [13], which are permission-aware analogues of traditional *assume*/*assert* statements used to express verification conditions. An operation `exhale` A (where A is an assertion) can be understood to *assert* all of the logical properties described by A, and to *give away* all of the permissions described by the assertion. `inhale` A is the dual operation: it *assumes* the logical properties, and *adds* the permissions to the current state. Thus, for example, a method call can be modelled by *exhaling* the method's precondition, and *inhaling* its postcondition.

Figure 1 shows a simple SIL program used as our running example. Unbounded data structures can be specified in SIL via *recursive predicates* [16], such as `List`[1], here. An instance of this predicate (written `List(x)`) represents permissions to all directly and transitively (via `next`) reachable fields. Permissions folded inside a predicate instance are not directly accessible by the verifier until the predicate instance is *unfolded*; the `unfold` statement on line 26 directs the verifier to exchange the predicate instance for its body, making the permissions to `xs.val` and `xs.next` available (and removing the predicate instance `List(xs)`). We will sometimes refer to having a predicate instance $P(e)$ by saying

[1] For brevity, our `List` predicate precludes empty lists, but they can be defined as well.

```
0  var val: Int
1  var next: Ref
2
3  predicate List(ys: Ref) {
4    acc(ys.val) && acc(ys.next) &&
5      (ys.next ≠ null ⇒ List(ys.next))
6  }
7
8  function sum_rec(ys: Ref): Int
9    requires List(ys)
10 {
11   unfolding List(ys) in
12     (ys.val + (ys.next == null ? 0 : sum_rec(ys.next))) }
13
14 method sum_it(ys: Ref) returns (sum: Int)
15   requires ys ≠ null && List(ys)
16   ensures List(ys) && sum == old(sum_rec(ys))
17 {
18   var xs := ys;
19   sum := 0
20
21   while (xs ≠ null)
22     invariant ((xs ≠ null) ⇒ List(xs)) &&
23       sum == (old(sum_rec(ys)) - (xs == null ? 0 :
   sum_rec(xs)));
24   {
25     unfold List(xs)
26     sum := sum + xs.val
27     xs := xs.next
28   }
29   // postcondition error: no permission List(ys) available
30 }
```

Fig. 1. Running sum example (with insufficient loop invariant)

that we have permissions to $P(e)$; a view that is motivated by fact that predicate instances and their bodies need to be exchanged explicitly. The use of fold and unfold statements is a common approach to taming recursive definitions in automatic tools [20].

Following other implicit dynamic frames approaches [19, 13], SIL also supports recursively-defined *functions*. In our example, the function sum_rec returns the sum of the integer values stored in the linked-list. A SIL function's body is an *expression*, not a statement; evaluating a function invocation is therefore side-effect free. A function's *precondition* must require enough permissions to guarantee that the function's body is *framed*; in this case, it requires an instance of the List predicate. The body of the sum_rec function computes the sum in the natural recursive manner; the only non-standard feature is the `unfolding`. This construct does not affect the value returned by the function; its only purpose is to make the folded permissions available to the verifier while the actual function body (after the "`in`") is evaluated. We call an operation which modifies only available permissions (but doesn't change heap/expression values) a *ghost operation*; the unfold statement in our example is also a ghost operation.

```
0   var xs := ys;
1   sum := 0
2
3   define A (xs ≠ null ⇒ List(xs))
4   define B List(ys)
5
6   package A --* B
7
8   while (xs ≠ null)
9     invariant (xs ≠ null ⇒ List(xs)) && (A --* B) &&
10      sum == old(sum_rec(ys)) - (xs == null ? 0 : sum_rec(xs))
11  {
12    wand w := A --* B // give magic wand instance the name w
13
14    var zs := xs // value of xs at start of iteration
15    unfold List(xs)
16    sum := sum + xs.val
17    xs := xs.next;
18
19    package A --* (folding List(zs) in (applying w in B))
20  }
21  apply A --* B
```

Fig. 2. Our verified version of the body of sum_rec, from Figure 1.

The method sum_it from the figure shows a straightforward iterative implementation to calculate the sum of the linked-list's nodes. The method's precondition requires permission to an instance of the List predicate, and the postcondition states that this predicate will be returned to the caller, along with the knowledge that this iterative approach computes the same value as the function sum_rec. The old construct specifies that the nested expression should be evaluated in *heap* of the method's prestate; note that old expressions do not affect the evaluation of local variables.

Loop invariants play the usual role in SIL, but via `inhale`/`exhale` operations in place of where they would be assumed/asserted in a first-order setting. In particular, after the loop in method sum_it, the loop invariant will be inhaled before the postcondition is exhaled. However, since xs == null after the loop, the invariant provides no permissions, and so exhaling the postcondition fails. A simple fix for this problem is not obvious: there is no convenient way to specify where to keep permissions to the "partial list" inspected so far by the loop [21, 14], nor is there an obvious position to insert appropriate fold statements to reconstruct the desired predicate for the original list structure.

2.2 Specification using the Magic Wand

Figure 2 shows the body of the sum_it method, verified using our magic wand support. In particular, the loop invariant has been strengthened (line 9) to include the additional magic wand assertion[2] (xs != `null` ==> List(xs)) −∗ List(ys).

[2] We have used syntactic abbreviations (lines 3 and 4) to make the code more readable, and to save repetition of these assertions; they are also supported in our tool.

Informally, this magic wand states "give me the remainder of the list (from xs), and I will give you the entire list structure". Thus, this assertion plays the role of representing the permissions to the partial list inspected so far by the loop; we say these permissions make up the *footprint* of the magic wand. In particular, after the loop body, the magic wand from the loop invariant is *applied* (on line 21); its left-hand-side (LHS) assertion must be given up, and its right-hand-side (RHS) List(ys) is added to the state, providing the method's postcondition. Variable zs is used to refer to the linked-list node that xs points to at the beginning of the current loop iteration, its purpose will become clear in Section 4.1.

In the rest of the paper, we use this example to explain the details of our general magic wand support. Section 3 describes the core of our solution: in particular, the automatic handling of *footprints* for magic wand instances. In Section 4.1 we show how to integrate ghost operations such as folding and unfolding with our magic wand support (cf. line 19), and in Section 4.2 we introduce a magic-wand-specific generalisation of old expressions, which allows additional functional properties to be added to our specifications easily.

3 Magic Wand Instances and Footprints

We present our approach to supporting the magic wand without relying on any particular implementation strategy for the underlying verification tool. For example, we are agnostic as to whether the verifier is based on symbolic execution, verification condition generation, or some other technique, so long as the modelled program state admits a number of basic operations presented in the next subsection. Moreover, although we present our approach in the context of implicit dynamic frames, it is straightforward to adapt it to a separation-logic-based tool or other permission-based verification logics.

3.1 Basic Operations

We use σ as a meta-variable ranging over program states as modelled in the verifier. We do not prescribe a particular representation for these states; in a tool based on symbolic execution, they could be sets of heap chunks along with path conditions, while in a tool based on verification condition, they could consist of total maps representing the heap and permissions held. States must be able to record assumptions, permissions and magic wand instances. Figure 3 defines the interface we expect to be provided by the underlying tool, in terms of a handful of basic operations on states. We represent these basic operations as functions on (and producing) states; in practice they could be implemented by generating a corresponding program in an intermediate language, or by directly executing corresponding operations to update internal state in a verification tool. These definitions treat states as immutable data; modifications are reflected by returning a new state (this could also differ in a real implementation).

$\texttt{assume}(\sigma, e)$	$\approx$ assume that e holds in σ
$\texttt{assert}(\sigma, e)$	$\approx$ assert that e holds in σ
$\texttt{has}(\sigma, c)$	$\approx$ queries whether or not σ contains c (see caption)
$\texttt{add}(\sigma, c)$	$\approx$ add access to c to σ
$\texttt{remove}(\sigma, c)$	$\approx$ remove access to c from σ
$\texttt{equate}(\sigma_1, \sigma_2, v)$	$\approx$ update σ_1 s.t. it contains all assumptions from σ_2 about v
$\texttt{if}\ (e, \sigma)\ \ldots\ \texttt{else}\ \ldots$	$\approx$ Conditional operation, depending on meaning of e in σ

Fig. 3. Basic state operations. Some operations are overloaded: c can be an accessibility predicate $\texttt{acc}(x.f)$, a predicate instance $P(x)$ or a magic wand assertion $A \mathrel{-\!\!*} B$. v denotes a variable x or a field $x.f$, or a sequence of these (for which the operation is applied per element). All operations except `has` return an updated state σ'.

Most of the basic operations are self-explanatory, but `equate` is quite subtle. The idea is to be able to communicate information (e.g., logical constraints) from one state to another. In particular, we use this operation to model adding the information that the value of an expression in σ_1 is actually the same value as it had in σ_2. $\texttt{equate}(\sigma_1, \sigma_2, x.f)$ should produce a modified version of σ_1, in which information known about the value of $x.f$ in σ_2 has been copied/made available. In practice, this operation often has a simple implementation; it could amount simply to equating the symbolic values of $x.f$ in the two states, or (in a tool based on verification condition generation) simply adding the assumption that the value of the expression is the same in the two states. In implementations in which logical constraints (path conditions) are not stored globally, the operation might require selectively copying such constraints.

The `if` conditional can be implemented differently in different tools: those which translate to another language may represent this as an actual conditional, whereas a symbolic-execution-based verifier would typically *branch* (that is, split the proof of the program) at this point.

We can now define `inhale` and `exhale` operations as functions of states (Figure 4), in terms of the basic operations above. We use the $\rightsquigarrow$ symbol to denote the desugaring/compilation of an operation into simpler ones. For example, $\texttt{inhale}(\sigma, \texttt{acc}(e.f)) \rightsquigarrow \texttt{add}(\sigma, \texttt{acc}(e.f))$ represents that inhaling an accessibility predicate is defined as adding the appropriate permissions to the state.

3.2 Applying and Packaging Wand Instances

A magic wand instance held in a state represents a guarantee that the wand could be *applied* in that state: if the wand's LHS is given up, along with the wand instance itself, then the wand's RHS can be obtained. This amounts to the Modus-Ponens-like inference rule from separation logic: $A * (A \mathrel{-\!\!*} B) \vDash B$, and we analogously define the operation of applying a wand in a state as follows:

$$\texttt{apply}(\sigma, A \mathrel{-\!\!*} B) \rightsquigarrow \texttt{inhale}(\texttt{exhale}(\texttt{exhale}(\sigma, A \mathrel{-\!\!*} B), A), B)$$

$$
\begin{array}{lll}
\texttt{exhale}(\sigma, a) & \rightsquigarrow & \texttt{exhale}(\sigma, \sigma, a) \\
\texttt{exhale}(\sigma, \widetilde{\sigma}, e) & \rightsquigarrow & \texttt{assert}(\widetilde{\sigma}, e) \\
\texttt{exhale}(\sigma, \widetilde{\sigma}, c) & \rightsquigarrow & \texttt{if has}(\sigma, c)\ \texttt{remove}(\sigma, c)\ \texttt{else fail} \\
\texttt{exhale}(\sigma, \widetilde{\sigma}, a_1\ \&\&\ a_2) & \rightsquigarrow & \texttt{exhale}(\texttt{exhale}(\sigma, \widetilde{\sigma}, a_1), \widetilde{\sigma}, a_2) \\
\texttt{exhale}(\sigma, \widetilde{\sigma}, e \Rightarrow a) & \rightsquigarrow & \texttt{if}\ (e, \widetilde{\sigma})\ \texttt{exhale}(\sigma, \widetilde{\sigma}, a)\ \texttt{else}\ \sigma \\
 & & \\
\texttt{inhale}(\sigma, e) & \rightsquigarrow & \texttt{assume}(\sigma, e) \\
\texttt{inhale}(\sigma, c) & \rightsquigarrow & \texttt{add}(\sigma, c) \\
\texttt{inhale}(\sigma, a_1\ \&\&\ a_2) & \rightsquigarrow & \texttt{inhale}(\texttt{inhale}(\sigma, a_1), a_2)
\end{array}
$$

Fig. 4. The interesting cases for the definitions of exhaling and inhaling assertions (see also [13]). The second state parameter $\widetilde{\sigma}$ for `exhale` is used to carry a copy of the original state, used when checking boolean expressions (to avoid any loss of information due to removed permissions). Here, as in Figure 3, c can denote a field permission, predicate instance or wand assertion. `fail` is a short-hand for $\texttt{assert}(\sigma, \texttt{false})$.

We do not select when to apply wands automatically; instead, this is directed via an explicit $\texttt{apply}(A \mathrel{-\!\!*} B)$ command in the program source (see, for example, line 21 of Figure 2).

For applying wands to be sound, it is necessary that a wand instance represents sufficient state such that, when combined with some state satsifying its LHS, the RHS can be guaranteed to hold. In general, wand instances must (implicitly) carry around sufficient permissions to guarantee that they can be soundly applied; we call these permissions the *footprint* of the wand instance. For example, the (rather trivial) wand instance $\mathit{true} \mathrel{-\!\!*} \texttt{acc}(x.f)$ has a footprint consisting of the permission to location `x.f`; this permission must belong to the wand instance, otherwise, when applying the wand we would effectively forge it from nowhere.

When proving a new wand instance, we must not only guarantee that the wand's semantics holds in the current state (i.e., that it could be soundly immediately applied), but also that this guarantee will be preserved until the point where the wand instance is applied (perhaps in another method activation, since the wand instance could be `exhale`d). Thus, proving a new wand also requires determining a suitable footprint for the new wand instance that frames the information available in the current state on which the wand's truth depends. This is achieved by removing the footprint permissions from the current state, which ensures that the corresponding locations cannot be modified until the wand has been applied. Since the operation of proving a new wand instance also entails removing a suitable footprint from the current state, we call the operation *packaging* the wand instance. Once a wand instance is packaged, the permissions belonging to its footprint cannot be retrieved until the wand instance is applied.

Analogously to applying wands, we require a $\texttt{package}(A \mathrel{-\!\!*} B)$ statement to indicate where a new wand instance should be packaged. The key aspect of making this step lightweight for the tool user is that we provide an algorithm for computing a suitable footprint *automatically*. This is a challenging problem, and our solution is elaborated in the next subsection.

3.3 Determining Wand Footprints

Automatically deciding on a suitable footprint for a new magic wand is challenging. To constrain the problem slightly, we limit our wand support to only allow magic wand assertions $A \mathrel{-\!\!*} B$ in which the assertions A and B are self-framing. Thus, we disallow awkward assertions such as $\mathit{true} \mathrel{-\!\!*} x.f == 3$. This is not a strong restriction in practice; indeed, in separation logics, it is typically not even possible to represent assertions which are not self-framing [17].

A `package`$(A \mathrel{-\!\!*} B)$ command must not only determine a suitable footprint for the new wand instance, but must check that the semantics of the wand is guaranteed. That is, we must check that, given *only* the information from the current state which is framed by permissions in the chosen footprint, if the assertion is A is additionally added, we are guaranteed to reach a state in which the assertion B holds. So long as we check this property correctly, we observe that *any* choice of footprint for a new magic wand instance will be *sound*, in the sense that if we succeed in packaging the wand instance, its semantics will be guaranteed. For example, we *could* always choose the empty footprint, and not use up any permissions at a `package` statement. This would only succeed in the case where A and B are logically equivalent, which is not a particularly useful case in general. Alternatively, we could choose the *entire current state* to be the new wand instance's footprint. This would allow many wands to be proven, but would mean that the remaining program is almost certain not to verify. So, while either of these approaches would be sound, they would not be useful in practice.

Intuitively, it makes sense to choose a footprint which is as small as possible, while still allowing the packaging to succeed. However, the notion of "small as possible" is not straightforward to define precisely. For example, consider a state in which permission to `x.f` is held, and `x.f` has the value 3. Suppose that we attempt to package the wand $\texttt{acc}(x.f) \mathrel{-\!\!*} (\texttt{acc}(x.f)\ \&\&\ x.f == 3)$ in this state. Thinking purely in terms of permissions, an empty footprint seems sensible, since the wand's LHS provides all of the permission that the RHS requires. However, the heap fact $x.f == 3$ is not guaranteed to hold when one adds on an arbitrary state satisfying $\texttt{acc}(x.f)$; it could indeed be that the value has changed, since no permission is to be kept with the wand instance. Thus, it might be tempting to think that the wand's footprint *should* include some permission to `x.f`, to preserve the value.

We do not take this approach for three main reasons. Firstly, attempting to determine precisely which logical properties entailed by the current state might allow the truth of the wand RHS to be guaranteed, is too challenging to be achieved automatically, and any attempt to do so is likely to produce results which are hard to understand for a tool user (who doesn't even explicitly see the choice of footprint made). In particular, the user might be surprised that more permissions have been lost in this step than they expected, because (for example) they enabled enough extra facts to be preserved to contradict the information from the wand's LHS, and thus make the truth of the wand vacuous. Secondly, any extra permissions added on to attempt to frame such extra facts, are essentially guaranteed to be *leaked*; if the wand does not explicitly include

$\texttt{package}(\sigma, A \mathrel{-\!\!*} B) \rightsquigarrow$
  σ_{emp} := $\texttt{equate}(\varnothing, \sigma, \texttt{vars}(A \mathrel{-\!\!*} B))$
  σ_A := $\texttt{inhale}(\sigma_{emp}, A)$
  $(\sigma'_A, \sigma', \sigma'_{foot})$:= $\texttt{exhale\_ext}(\sigma_A, \sigma_A, \sigma, \sigma, \sigma_{emp}, B)$
  return $\texttt{add}(\sigma', A \mathrel{-\!\!*} B)$

$\texttt{exhale\_ext}(\sigma_{lhs}, \widetilde{\sigma_{lhs}}, \sigma_{curr}, \widetilde{\sigma_{curr}}, \sigma_{foot}, \texttt{acc}(e.f)) \rightsquigarrow$
  if $\texttt{has}(\sigma_{lhs}, e.f)$ {
    σ'_{lhs} := $\texttt{remove}(\sigma_{lhs}, e.f)$
    return $(\sigma'_{lhs}, \sigma_{curr}, \sigma_{foot})$
  } else if $\texttt{has}(\sigma_{curr}, e.f)$ {
    σ'_{curr} := $\texttt{remove}(\sigma_{curr}, e.f)$
    σ'_{foot} := $\texttt{add}(\sigma_{foot}, e.f)$
    σ''_{foot} := $\texttt{equate}(\sigma'_{foot}, \widetilde{\sigma_{curr}}, e.f)$
    return $(\sigma_{lhs}, \sigma'_{curr}, \sigma''_{foot})$
  } else fail

$\texttt{exhale\_ext}(\sigma_{lhs}, \widetilde{\sigma_{lhs}}, \sigma_{curr}, \widetilde{\sigma_{curr}}, \sigma_{foot}, A_1 \mathbin{\&\&} A_2) \rightsquigarrow$
  $(\sigma'_{lhs}, \sigma'_{curr}, \sigma'_{foot})$:= $\texttt{exhale\_ext}(\sigma_{lhs}, \widetilde{\sigma_{lhs}}, \sigma_{curr}, \widetilde{\sigma_{curr}}, \sigma_{foot}, A_1)$
  $(\sigma''_{lhs}, \sigma''_{curr}, \sigma''_{foot})$:= $\texttt{exhale\_ext}(\sigma'_{lhs}, \widetilde{\sigma_{lhs}}, \sigma'_{curr}, \widetilde{\sigma_{curr}}, \sigma'_{foot}, A_2)$
  return $(\sigma''_{lhs}, \sigma''_{curr}, \sigma''_{foot})$

$\texttt{exhale\_ext}(\sigma_{lhs}, \widetilde{\sigma_{lhs}}, \sigma_{curr}, \widetilde{\sigma_{curr}}, \sigma_{foot}, e) \rightsquigarrow$
  $\texttt{assert}(\widetilde{\sigma_{lhs}} \uplus \sigma_{foot}, e)$
  return $(\sigma_{lhs}, \sigma_{curr}, \sigma_{foot})$

Fig. 5. Packaging a wand instance. $\texttt{vars}(A)$ returns the set of local variables in A. σ_{emp} is empty except for local variables from $A \mathrel{-\!\!*} B$ and assumptions about those from σ. Permissions taken from σ_{curr} contribute to the footprint, assumptions about the corresponding locations are preserved via `equate`. Boolean assertions are checked in the combination of the initial LHS state and the footprint. Other cases of `exhale_ext` are analogous to `exhale`, but expressions are evaluated in $\widetilde{\sigma_{lhs}} \uplus \sigma_{foot}$. $\uplus$ combines two states by adding permissions and conjoining boolean assumptions, similar to $\uplus$ used in separation logic.

the permissions in the right-hand-side, then they will not be recovered when the wand is applied. Thirdly, if the required permissions are potentially available from the wand's LHS, it could be that the wand is *intended* to describe a different state from that in which the wand is packaged. For example, in the same state as above, packaging a wand $(\texttt{acc}(x.f) \mathbin{\&\&} x.f == 4) \mathrel{-\!\!*} (\texttt{acc}(x.f) \mathbin{\&\&} x.f > 3)$ will only result in a wand which can be applied at some later program point if we *do not* put any permissions from the current state into the footprint. Furthermore, if the tool user *intends* extra permissions from the current state to be folded into a wand, this can be easily controlled: by writing a RHS which requires more permission than the LHS, this intention is made explicit.

Our chosen criterion for defining footprints is therefore: *permissions required by the wand's RHS, which we cannot prove to be provided by the wand's LHS.*

This definition is simple, but writing an algorithm to compute it is still challenging: there is a potential circularity to the problem. The footprint for the wand is determined in terms of the permissions required by the RHS assertion. Exactly which permissions are required by the RHS can (due to implication/conditionals) depend on properties of heap values. Properties known about heap values in the *current state* may be soundly used if and only if permission to those heap locations is included in the wand's footprint (which we are trying to compute).

To break this circularity, we *simultaneously* construct a state corresponding to the wand's footprint, while evaluating the wand's RHS. Our algorithm is shown in Figure 5, and works as follows. We first construct a hypothetical extra state representing the information provided by the wand's LHS (we achieve this by *inhaling* the LHS, starting from an empty state). We then define `exhale_ext`: a generalisation of the `exhale` operation (Figure 4), which takes *two* initial states σ_{lhs} and σ_{curr} (and corresponding copies of their original values), and attempts to perform the exhale by taking permissions preferentially from σ_{lhs} (corresponding to the wand's LHS) and only from σ_{curr} (the original state in which the `package` takes place) when needed. In the process, we construct a third state σ_{foot} corresponding to the footprint we have found so far; we use this to track which information from the original state can be used when checking boolean assertions. In particular, we use the `equate` operation to copy information from the state before the `package` operation ($\widetilde{\sigma_{curr}}$), whenever we determine that permission is to be included in the footprint state. Whenever we need to evaluate an expression, we use the combined information from both σ_{lhs} and σ_{foot}. If the operation `exhale_ext` fails, this is either because a permission needed in the RHS could not be found in either state, or because some logical property required is not known to hold, given the information from σ_{lhs} and σ_{foot} (as computed so far).

It is important that only properties framed by the wand's chosen footprint are used when checking the RHS. For example, suppose that a state σ holds permission to integer fields `x.f` and `y.f`, with values 1 and 2, respectively. The operation $\texttt{package}(\sigma, (\texttt{acc}(x.f)\ \&\&\ x.f == 2) \mathrel{-\!\!*} (\texttt{acc}(x.f)\ \&\&\ \texttt{acc}(y.f)\ \&\&\ x.f == y.f))$ will succeed, and the footprint $\texttt{acc}(y.f)$ will be removed from the state as a result. Note that $y.f == 2$ is preserved from the original state, but, importantly, the fact $x.f == 1$ is *not* used when checking the wand's RHS.

4 Extensions: Handling Predicates and Functions

In this section, we present two extensions of the basic support for wands outlined so far. In Section 4.1, we show how we incorporate ghost operations (such as folding and unfolding predicates) into the packaging of new magic wands, which is crucial for making our approach usable in a verifier which distinguished a predicate from its body. In Section 4.2, we introduce a new feature based on the idea of *old expressions*, which enables relationships between the states in which a wand is packaged and applied to be expressed and reasoned about simply. Both of these extensions are supported by our implementation.

4.1 Packaging Wands with Ghost Operations

The support for packaging magic wands presented in Section 3 can be used with wands that include predicate instances (and even nested magic wands), but is not always useful as it stands. For example, it is often convenient to be able to express "missing parts" of predicate instances using magic wands, but to show that these parts can be reassembled back into a complete predicate instance, requires the predicate instance to be folded. More generally, it is convenient to be able to express wands $A \mathrel{-\!\!*} B$ in which the *proof* that, when given A we can obtain B may make use of ghost operations, such as folding predicates.

This situation arises in our running example (Figure 2), when re-establishing the magic wand in our loop invariant: (xs != null ==> List(xs))−∗List(ys), expressing that we can obtain the "full list" if we give up the "remainder list" starting from xs. Consider how we can re-establish this invariant: in particular, the state at line 18. In this state, we have the permissions acc(zs.val) and acc(zs.next) (obtained from the unfold at line 15), and the magic wand instance w from the loop invariant at the beginning of the iteration (line 12), which has the same RHS, but requires List(zs) on its LHS. We don't directly hold enough permissions to package the wand needed in our new loop invariant; conceptually, those missing are in the footprint of the wand instance w. However, given the LHS assertion (xs != null ==> List(xs)), we *can* obtain the RHS if we first fold the predicate List(zs), and then apply the wand instance w. These ghost operations explain *how*, given the LHS, we can rearrange the permissions in our state to obtain the desired RHS, requiring in the process the additional permissions acc(zs.val) and acc(zs.next) and the wand instance w.

In order to allow such ghost operations to be expressed when packaging wands, and inspired by the unfolding expression already in our language, we extend our syntax for package commands, to the form $\texttt{package}(A \mathrel{-\!\!*} G)$, where G is an assertion B possibly wrapped by *ghost operations*, as defined by:

$$G ::= A \mid \texttt{folding}\, P(e)\, \texttt{in}\, G \mid \texttt{unfolding}\, P(e)\, \texttt{in}\, G \mid \texttt{applying}\, A_1 \mathrel{-\!\!*} A_2\, \texttt{in}\, G$$

A successful $\texttt{package}(A \mathrel{-\!\!*} G)$ statement does *not* add a wand instance of the form $(A \mathrel{-\!\!*} G)$ to the state, but rather $(A \mathrel{-\!\!*} B)$ where B is the assertion nested in the ghost operations. The role of the ghost operations is to indicate only *how* the wand's semantics can be achieved, but do not affect *what* the resulting wand instance represents. Concretely, the above extended syntax is used for the package statement on line 19 of Figure 2, to justify the wand instance required for the re-established loop invariant, according to the argument presented above.

For space reasons, we do not present the details of how to extend our footprint algorithm (Figure 5) to these extended package statements. However, the ghost operations are taken into account when computing the wand's footprint. Essentially, the idea is that any permissions which *result* from a ghost operation can be provided for the wand's RHS *for free*, but instead, any permissions *required* by the ghost operation must instead be found (from either the wand's LHS, or the current state). In particular, in our example, the footprint of the new wand instance added at line 19 consists of the permissions acc(zs.val) and

acc(zs.next) (those required for the `folding`, and not found on the wand's LHS), and the wand instance w (required for the `applying`, and not provided by the LHS or the `folding`). The full support is implemented in our prototype [1].

4.2 Handling Additional Functional Properties

The features presented so far are sufficient to verify our original running example (Figure 1), but the specification is somewhat weak. We consider strengthening the postcondition of sum_it to express that the list sum does not change[3]:

```
ensures List(ys) && sum == old(sum_rec(ys)) &&
        sum_rec(ys) == old(sum_rec(ys))
```

How should we extend our loop invariant in order to verify this postcondition? In particular, it seems we should strengthen the RHS of the magic wand used in our loop invariant, to be List(ys) && sum_rec(ys) == old(sum_rec(ys)), thus guaranteeing the new postcondition. However, packaging this wand (both at line 6 and line 19) now fails. The reason is, that the wand LHS is too unconstrained; it allows for the "remainder" of the list (from xs) to be provided with *any* sum; therefore, we are unable to prove that the resulting sum for the whole list is unchanged.

Intuitively, we need to specify that the list provided in the wand's LHS must also be unchanged. However, we cannot write old(sum_rec(xs)); for the package statement on line 19, the verifier will complain that we don't hold the precondition of this function (i.e., the predicate instance List(xs), for the *current* value of xs in the loop body, in the method's pre-state. Thus, we have no direct way to express in the wand that the list provided when the wand is applied should have the same sum as that where the wand is packaged.

While it is possible to work around this problem, either by expressing the current sum of the remaining list indirectly as old(sum_rec(ys)) − sum, or by instrumenting the program with extra state recording the value of sum_rec(xs) at the point where the magic wand is packaged, for more complex examples (particularly those using more intricate data structures such as trees), these solutions become cumbersome. Instead, we add a small extra feature to our assertion language, inspired by old expressions.

We extend our *expression* syntax to include a new construct `now`(e), which may *only* occur in the LHS or RHS of a magic wand assertion. Similar to the `old`(e) construct, `now`(e) changes the heap in which e is evaluated, in this case to be the *heap in which this wand instance was added to the state*. Thus, `now`(e) evaluates e in the heap immediately after the enclosing magic wand instance was packaged, or inhaled. In particular, the expression e must be framed by permissions in that state.

We can now write a magic wand assertion to express our intended restriction for our example:

[3] It is also possible to express that the sequence of values stored in the list doesn't change, we chose the weaker property for simplicity of presentation.

```
(xs != null ==> List(xs) && (sum_rec(xs) == now(sum_rec(xs))))
--* List(ys)  &&  sum_rec(ys) == old(sum_rec(ys))
```

Given the semantics for `now` described above, we have to take care when *exhaling* a magic wand instance which uses `now` expressions: when the assertion is correspondingly *inhaled*, the expressions will effectively be *re-interpreted* with respect to this state. To avoid introducing unsoundness this way, we add an additional proof obligation when exhaling such a wand, that all `now` expressions are framed by permissions exhaled along with the wand, and have the same values when exhaled as they did in the heap in which `now` is currently interpreted. The restriction to have unmodified values could be weakened in some cases, but so far this has not been needed in our examples.

Using our `now` feature, we can verify the version of our running example with the strengthened postcondition. With respect to the code of Figure 2, only lines 3 and 4 (defining the magic wand assertion used) need changing; the resulting code verifies. Note that the feature employed can also be used in examples which *modify* the underlying data structure; the magic wand used could then specify a constraint on e.g. sum_rec(xs) as some function of now(sum_rec(xs)).

5 Conclusions and Related Work

We have presented a way of supporting magic wands in automatic verifiers, that requires little specification overhead and is still expressive enough to encode general uses of the logical connective. Most important is our ability to compute suitable footprints for magic wands automatically, which greatly simplifies the annotation effort required. Our work makes few assumptions about the underlying verifier, and should be relatively easy to apply in other tools/logics. The extra expressiveness of our "`now`" expressions may be hard to adapt to a separation-logic-based tool (in which expressions are usually too limited), but this feature is an extension.

The problems of tracking permissions in loop invariants are discussed in detail by Tuerk [21], who proposed alternative pre/postcondition specifications for loops. A magic wand of the form: pre_{rest} && ($post_{rest} \mathrel{-\!\!*} post_{all}$) gives an alternate expression of this idea (where "*rest*" refers to the remaining loop iteration, and "*all*" the entire loop). However, Tuerk's work does not provide a way to incorporate ghost operations: it is not suitable for a verifier using fold/unfold.

Recently, Blom and Huisman [3] added support for magic wands to their VerCors verifier, which translates Java programs with separation-logic-style specifications into Chalice programs. Chalice does not support magic wands natively, which are therefore eliminated during the translation by a clever encoding into additional classes and objects that represent magic wand instances. Although technically different from our approach, their approach is conceptually similar in that wands a treated as self-contained elements of the verification state that can be created and applied, and that "carry" enough permissions such that assuming the RHS of the wand when applying it is sound. However, their approach

requires the user to explicitly compute and specify the wand's footprint, which increases the annotation overhead, and can also be challenging if many path conditions need to be considered in the process. Moreover, their work does not address nested wands such as $A \mathrel{-\!\!*} (B \mathrel{-\!\!*} C)$ or $(A \mathrel{-\!\!*} B) \mathrel{-\!\!*} C$.

The VeriFast tool [8] can work around the absence of magic wands using *lemma function pointers* and predicates. One can model a wand $A \mathrel{-\!\!*} B$, using a predicate F (manually encoding the wand's footprint), and pointer to a lemma function with precondition $F * A$ and postcondition B, whose body shows how to rewrite the state. This requires significant insight and overhead for the user, and (we believe) cannot represent nested magic wands.

Lee and Park have recently developed a proof system for a separation logic supporting the magic wand connective [12], which also provides a decision procedure for propositional separation logic (without variables). In a richer assertion language such as ours, the magic wand is known to be undecidable [5]: our use of package and apply statements along with our deterministic approach for wand footprints makes our solution practical.

In the context of a permission-based type system, Boyland [4] has defined a "sceptre" operator to represent "borrowing" of permission. This connective is more restricted than the general magic wand, but sufficient for many loop invariants. The PhD thesis of Retert [18] provides an abstract-interpretation-based approach supporting this connective.

As future work, we are keen to try out our new tool support in other contexts. For example, we believe that magic wands can be used to support reasoning about *closures*, in which it is convenient to be able to express assertions of the form $post_1 \mathrel{-\!\!*} pre_2$, to express that, "after calling closure 1, we will be able to call closure 2", in a context in which the concrete pre/postconditions are not known. The relationship between old expressions in closure specifications and `now` expressions in such magic wands, seems particularly interesting.

We are also interested in exploring further automation of our approach. In particular, it seems that *applying* wands could be made an automatic heuristic when an assertion fails; since there are a (usually small) finite number of wand instances in the state, one could even try naïvely applying each (and backtracking if necessary). Automating package statements is harder, because of the need to integrate appropriate ghost operations in some cases. We are currently investigating whether static analysis can be employed to infer loop invariants which include these magic wands, along with appropriate package operations.

References

1. Our implementation. http://www.pm.inf.ethz.ch/research/semper/Silicon .
2. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.
3. S. Blom and M. Huisman. Witnessing the elimination of magic wands, November 2013.
4. J. T. Boyland. Semantics of fractional permissions with nesting. *ACM Trans. Program. Lang. Syst.*, 32(6):22:1–22:33, Aug. 2010.

5. R. Brochenin, S. Demri, and E. Lozes. On the almighty wand. *Inf. Comput.*, 211:106–137, Feb. 2012.
6. D. Distefano and M. J. Parkinson. jstar: towards practical verification for java. In G. E. Harris, editor, *OOPSLA*, pages 213–226. ACM, 2008.
7. C. Haack and C. Hurlin. Resource usage protocols for iterators. *Journal of Object Technology*, 8:55–83, 2009.
8. B. Jacobs and F. Piessens. The verifast program verifier. Technical report, Katholieke Universiteit Leuven, August 2008.
9. J. B. Jensen, L. Birkedal, and P. Sestoft. Modular verification of linked lists with views via separation logic. *Journal of Object Technology*, 10:2:1–20, 2011.
10. N. R. Krishnaswami. Reasoning about iterators with separation logic. In *Proceedings of SVCBS 2006*, pages 83–86, New York, NY, USA, 2006. ACM.
11. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT S.E. Notes*, 31(3):1–38, 2006.
12. W. Lee and S. Park. A proof system for separation logic with magic wand. In *Principles of Programming Languages (POPL) (to appear)*. ACM Press, 2014.
13. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP*, pages 378–393, 2009.
14. T. Maeda, H. Sato, and A. Yonezawa. Extended alias type system using separating implication. In *Proceedings of TLDI '11*, pages 29–42. ACM, 2011.
15. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19, London, UK, 2001. Springer-Verlag.
16. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM Press, 2005.
17. M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *In LMCS*, 8(3:01):1–54, 2012.
18. W. S. Retert. *Implementing Permission Analysis*. PhD thesis, Milwaukee, WI, USA, 2009.
19. J. Smans. *Specification and Automatic Verification of Frame Properties for Java-like Programs*. PhD thesis, FWO-Vlaanderen, May 2009.
20. A. J. Summers and S. Drossopoulou. A formal semantics for isorecursive and equirecursive state abstractions. In *Proc. of ECOOP '13*, pages 129–153, 2013.
21. T. Tuerk. Local reasoning about while-loops. In *VSTTE - Theory Workshop (VS-THEORY 2010)*, 2010.