Lightweight Support for Magic Wands in an Automatic Verifier

Malte Schwerhoff and Alexander J. Summers

ETH Zurich, Switzerland
`{malte.schwerhoff,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, for example, 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, due to a novel approach for choosing *footprints* for magic wand formulas automatically. We show how to extend this technique to interact elegantly with common specification features such as recursive predicates. Our solution is designed to be compatible with a variety of logics and underlying implementation techniques.

We have implemented our approach, and a prototype verifier is available to download, along with a collection of examples.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Magic Wand, Software Verification, Automatic Verifiers, Separation Logic, Implicit Dynamic Frames

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.614

Supplementary Material ECOOP Artifact Evaluation approved artifact available at
`http://dx.doi.org/10.4230/DARTS.1.1.10`

1 Introduction

Permission-based verification logics, most notably separation logic [24], 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; such an assertion is true in a program state σ, if we can *split* the state into two parts, $\sigma = \sigma_1 \uplus \sigma_2$ such that A is true in σ_1 and B in σ_2. Here, $\uplus$ denotes the combination of two compatible partial program states; in particular, the two parts must describe disjoint heap locations. Support for this connective has been used to handle aliasing, framing, race conditions etc., both in by-hand proofs, and in a variety of tools.

The separating implication, or *magic wand* connective $\mathrel{-\!\!*}$ was originally introduced along with the separating conjunction, in the first papers on separation logic. The semantics of this connective is defined as follows:

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

Here, $\sigma' \perp \sigma$ expresses that the two states are compatible: neither do both require access to the same heap location, nor do they disagree on the values of any local variables. Informally, an assertion of the form $A \mathrel{-\!\!*} B$ can be understood as describing the effect of a hypothetical addition to the state σ, in the above: "if we add on any partial heap satisfying A, then B will hold in the resulting state". The ability to express guarantees about hypothetical (future) additions to the state, makes the magic wand well-suited for concisely specifying *partial* versions of data structures, e.g. for describing ongoing traversals of those structures [34, 22], or for allowing clients to reason about "recombining" a view on the whole data structure while hiding the internal definitions, which has been used for specifying protocols that enforce orderly modifications of data structures [17, 10, 15]. Yang [35] employs the magic wand for a by-hand proof of the Schorr-Waite graph marking algorithm, while Dodds et al. employ it for specifying synchronisation barriers for deterministic parallelism [9].

Despite its history and this variety of applications, the magic wand connective is generally not supported in automatic verifiers built upon separation logic (and related) theories [1, 8, 14, 21]. The quantification over states in the wand's semantics makes the connective challenging. Recent developments in *propositional* separation logics [19, 13] show its proof theory to be intricate. In the presence of variables and other logical features arising in program verification, reasoning without any user guidance is known to be undecidable [5].

We address the problem of magic wand support in the context of a general-purpose verifier, via lightweight user annotations and a novel approach for automatically choosing suitable *footprints* for magic wand assertions. We describe our solution in the context of imperative code annotated with generic user-defined predicate and function definitions (for describing program data structures). Our technique is defined in terms of core operations which most verification tools for separation logic (and similar permission-based logics) already support; it is designed to be easily implementable as an extension to existing tools. The specification language supported by our prototype implementation [29] is richer than the fragment used in this paper, and includes fractional permissions [3], quantifiers and custom domains such as mathematical sequences and sets.

Contributions

This paper shows how to support the magic wand connective in an automated verifier, including the following specific contributions:

- A design for the representation of wands in a verification state, and the provision of suitable *ghost operations* for directing their use (Section 3.2).
- An automatic strategy and algorithm for choosing suitable *footprints* for magic wand instances, without additional user direction (Section 3.4).
- A mechanism for integrating existing ghost operations (such as *folding* predicates) with our automatic footprint computation, and a soundness argument for the presented algorithms. (Section 4).
- A set of additional heuristics, which aim to infer the magic-wand-related annotations required by our approach (Section 5).
- An implementation of our techniques [29], built as an extension of an existing verification tool, along with examples demonstrating the conciseness and versatility of our approach (Section 6).

```
var val: Int
var next: Ref

predicate List(ys: Ref) {
  acc(ys.val) * acc(ys.next) * (ys.next ≠ null ⇒ List(ys.next))
}

function sum_rec(ys: Ref): Int
  requires List(ys)
{
  unfolding List(ys) in
    (ys.val + (ys.next = null ? 0 : sum_rec(ys.next))) }

method sum_it(ys: Ref) returns (sum: Int)
  requires ys ≠ null * List(ys)
  ensures List(ys) * sum = old(sum_rec(ys))
{
  var xs := ys
  sum := 0

  while (xs ≠ null)
    invariant ((xs ≠ null) ⇒ List(xs)) *
      sum = (old(sum_rec(ys)) - (xs = null ? 0 : sum_rec(xs)))
  {
    unfold List(xs)
    sum := sum + xs.val
    xs := xs.next
  }
  // postcondition error: no permission List(ys) available
}
```

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

Our work is agnostic as to the implementation of the underlying verifier, and is designed to be easily adaptable to related verificaton logics (Section 3.1).

2 Background and Motivation

We present our work using *implicit dynamic frames* [30] as the specification logic. It provides permission-based reasoning similar to separation logic (which can be encoded [26]), and is suitable for verification both by tools based on verification-condition-generation [31, 21] and verifiers built around symbolic execution [32, 16]. We present examples in the Silver intermediate verification language [16]; our implementation extends the existing verifier Silicon for this language.

2.1 Running Example

Figure 1 shows a simple example Silver program used as our running example: a straightforward iterative implementation to calculate the sum of the nodes in a linked list. In this subsection, we give a high-level overview of the concepts involved in the specification and attempted verification of this example.

Specifications (such as the precondition marked with `requires` or the declared loop invariant) express not only the intended functional properties of the code, but also the required *permissions*; it is a general requirement that heap (field) locations may only be dereferenced if corresponding permissions are currently held. Permission to access a single field location is denoted by assertions such as `acc(ys.val)`, while permission to access an unbounded number of field locations can be expressed using predicate instances such as

```
 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
```

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

`List(ys)`, which requires permission to all `val` and `next` fields of the linked-list beginning from `ys`. For example, the precondition of the method `sum_it` requires an instance of the `List` predicate, and its postcondition promises that such an instance will be returned to the caller, along with the guarantee that the returned value is the sum of the values stored in the list.

The verification of the while loop (line 20) relies on the provided loop invariant (line 21), which also specifies which permissions are carried along in the invariant. At the beginning of each iteration of the loop body, an `unfold` annotation (line 24) directs the verifier to unroll the (recursive) definition of the `List` predicate instance. According to the definition of the predicate (line 3), this makes available the permissions to access the fields of `xs`, which is necessary for the verifier to allow the subsequent assignments (lines 25 and 26).

After the loop (line 28) the verifier will have a copy of the state described by the loop invariant, as well as the assumption that the loop condition is false. Unfortunately, in this case, the provided loop invariant provides no permissions; essentially, the `List` predicate instance has been totally unfolded during traversal of the list, and the left-over permissions were not retained in the loop invariant. As a consequence, the postcondition (line 15) will fail to verify, since the required predicate instance cannot be found.

2.2 Overview of Magic Wand Support

Figure 2 shows the body of the `sum_it` method, specified using our magic wand support. The loop invariant has been strengthened (line 9) to include an additional *magic wand instance*[1] `(xs ≠ null ⇒ List(xs)) −∗ List(ys)`.

Informally, this magic wand instance represents the following promise: “if you give up permission to the remainder of the list (starting from `xs`), I will give you back permission to the entire list structure (starting from `ys`)”. This assertion plays the role of representing the

[1] 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.

permissions to the partial list inspected so far by the loop; we say these permissions make up the *footprint* of the magic wand.

The footprint of a magic wand must include enough permissions to make this informal promise justified. We can direct the verifier to create a new magic wand instance (and choose a suitable such footprint) using a `package` statement, such as that on line 6, which creates the wand instance necessary for showing that the loop invariant holds on entry. An empty footprint suffices on line 6, since `xs` and `ys` are equal at this point (line 0).

During verification of the loop body, we need to maintain the magic wand instance in the loop invariant; this is achieved by the `package` statement on line 19, which produces a new suitable magic wand instance to represent the current left-over permissions to the already-inspected portion of the list. Apart from the assertions A and B, the extra annotations on this line explain to the verifier *how*, given the left-hand-side assertion, the right-hand-side assertion can be obtained[2]. Given these annotations, the calculation of the extra permissions which must be associated with this new wand instance (i.e. its footprint) is performed automatically; our techniques for achieving this are an important contribution of this paper.

A wand instance can be combined with its left-hand-side (LHS) assertion, and the combination exchanged for the right-hand-side (RHS) assertion; this is called *applying* the magic wand instance. For example, after the loop body in our example, the magic wand instance from the loop invariant is *applied* (on line 21); its LHS must be given up, and its RHS `List(ys)` is added to the state, providing the method's postcondition.

In the rest of the paper, we use this simple example to help explain the details of our general magic wand support: the representation of magic wands, related annotations, and our automatic footprint computation algorithm (Section 3); the integration of other ghost operations such as `folding` on line 19 (Section 4); and a set of heuristics used to infer magic-wand-related annotations (Section 5). In the remainder of this section, we provide more-detailed background and foundational definitions.

2.3 Assertion Language

The assertion language used in this paper consists of the following constructs:

$$A ::= e \mid \texttt{acc}(e.f) \mid P(e) \mid A * A \mid e \Rightarrow A \mid A \mathrel{-\!\!*} A$$

This is a core fragment of the assertions employed in the Silver language, extended (in the last case) with *magic wand assertions*; A denotes assertions, e denotes side-effect-free expressions. Permissions are managed in the logic via *accessibility predicates* $\texttt{acc}(e.f)$, which denote the *exclusive* permission to access a heap location $e.f$. For example, see line 4 of Figure 1. The conjunction $*$ behaves as the separating conjunction in separation logics; in particular, an assertion $\texttt{acc}(x.f) * \texttt{acc}(y.f)$ requires the *disjoint union* of the permissions required by the two conjuncts; this implicitly requires that $x \neq y$, otherwise the same (exclusive) permission would be required twice (this is analogous to the meaning of the assertion $x.f \mapsto \_ \; * \; y.f \mapsto \_$ in separation logic). The grammar above imposes the standard restriction [1] that accessibility predicates (as well as predicate and magic wand instances) may not occur on the left of conditional assertions: the value of the condition e is therefore independent of the current permissions held.

[2] Variable `zs` records the node that `xs` pointed to at the beginning of the current loop iteration, while `w` gives a name to the magic wand instance belonging to the loop invariant at the start of the iteration. Both are not strictly necessary, but make the annotations on line 19 succinct.

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. In particular, heap-dependent *functions* can be defined and used in expressions, such as the `sum_rec` function in Figure 1 (line 7). 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 such as `xs.val` on line 25 of Figure 1 is allowed only in states in which a permission to the location `xs.val` 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 `xs.val > 0` is not self-framing, whereas `acc(xs.val) * xs.val > 0` is self-framing. Invocations of functions such as `sum_rec` must analogously occur in states in which their preconditions (e.g. line 8) hold. Only self-framing assertions can be used in specifications. As a technical simplification of this check, we assume that the permission to access a heap location comes syntactically *before* any expressions depending on the value at that location (this is analogous to the restriction in some separation-logic-based tools that logical variables must be bound to heap locations before their use). In [26], Parkinson and Summers have shown that separation logic assertions can be encoded into implicit dynamic frames, and that the resulting assertions are self-framing by construction.

To simplify the presentation of our algorithms, we restrict ourselves in this paper to unary predicates $P(e)$ and functions $g(e)$ (the details of which will be discussed in Section 2.5), but this arity restriction is not relevant for the techniques presented in this paper. Our implementation [29] supports unrestricted predicate and function definitions, as well as fractional permissions [3].

2.4 Verification via Exhale and Inhale Operations

From a verification perspective, proof obligations can be expressed in Silver via *exhaling* and *inhaling* assertions [21], which are permission-aware analogues of traditional *assume*/*assert* statements used to express verification conditions. Analogous operations are used internally in other verification tools (e.g. for separation logic); in tools based on symbolic execution, these operations are typically called *consume* and *produce*.

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. Once permissions have been given away, the verifier may no longer retain (or *frame*) facts about the values of these heap locations, even if permission is regained later. For example, before the while loop in our running example, the loop invariant is exhaled; the giving up of permissions reflects the fact that the loop may modify the locations to which the loop invariant requires access. The loop invariant must also be exhaled at the end of the loop body, reflecting the usual requirement that the invariant is preserved.

In terms of the state maintained by a verifier, an `exhale` operation can be understood as requiring the current verification state σ to be *split* into a part σ_1 satisfying the assertion A, and a remainder state σ_2, which is the result of the operation. From a soundness perspective, it is fine for a verifier to overapproximate σ_1, effectively giving more permissions away than is necessary; the precision of these operations depends on the completeness of the underlying tool. If such a split *cannot* be found (the assertion A could not be shown to hold in the original state), then this operation causes a verification error, similar to an assertion failure in first-order tools.

`inhale` A is the dual operation: it *assumes* the logical properties described by A, and *adds* the permissions to the current state. As an operation on states, this can be regarded as *combining* an arbitrary state satisfying A with the current state. The verification of high-level programming features can typically be modelled using combinations of these operations: for example, a method call can be modelled by *exhaling* the method's precondition, and *inhaling* its postcondition. In terms of Figure 2, the loop invariant (line 9) is inhaled at the beginning of checking the loop body (line 11) as well as after the loop, for verifying the subsequent code (line 21).

2.5 Recursive Definitions and Ghost Operations

In Silver, unbounded data structures can be specified via *recursive predicates* [25], such as `List`. An instance of this predicate (written e.g. `List(xs)`) represents permissions to all directly and transitively (via `next`) reachable fields[3]. A predicate definition can have any number of parameters, and its body may be any self-framing assertion; in particular, it may include instances of the same or other predicates, and express conditions over arbitrary combinations of the parameter and heap values accessed.

Complete reasoning in the presence of such predicates is undecidable; consequently, many automatic verifiers do not treat a predicate instance as simply a direct short-hand for its body (the *equi-recursive* interpretation [33]). Instead, tools typically differentiate between holding an instance of a predicate and holding the assertion defined by its body, while allowing the two to be exchanged via `unfold` and `fold` operations (the *iso-recursive* interpretation). An `unfold` operation directs that a currently-held predicate instance should be exchanged for its body, while a `fold` operation exchanges the body for a predicate instance. Until an `unfold` is specified, predicate instances are treated as *opaque*, in the sense that the permissions and logical facts entailed by their definitions are not directly available to the verifier. We call such operations (which rewrite the verification state to guide the tool, but do not involve changes to the program state) *ghost operations.*

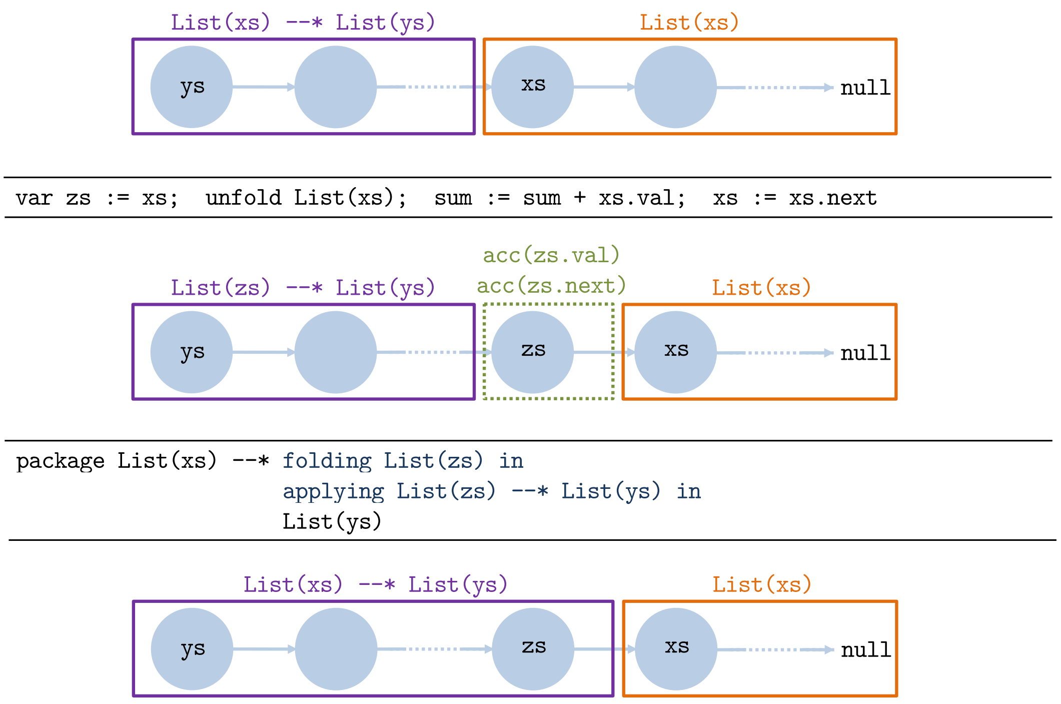
In some tools, ghost operations must be explicitly specified within the program code, while other tools may attempt to infer these via heuristics/static analyses. For the purposes of this paper, we will include ghost operations as explicit statements in the program text[4].

As alluded to above, Silver also supports recursive side-effect-free *functions* in specifications. In our example, the function `sum_rec` returns the sum of the integer values stored in the linked-list. A Silver function's body is an *expression*; the function's precondition must require enough permissions to guarantee that the function's body is framed (in the case of `sum_rec`, 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 role is to tell the verifier to *temporarily* apply an `unfold` ghost operation *before* evaluating the nested expression (after the "`in`"), explaining how to find the necessary permissions.

2.6 Revisiting the Running Example

Armed with the above background, we can explain the usage of magic wands in Figure 2 more clearly. In particular, the magic wand in the loop invariant retains the appropriate

[3] Technically, recursive predicate definitions should be understood via their least fixpoint interpretations. We do not concern ourselves with well-definedness details regarding recursive definitions, which are not the main focus of our paper.
[4] However, in Section 5 we will show how we can indeed infer these in many cases.



```
var zs := xs;  unfold List(xs);  sum := sum + xs.val;  xs := xs.next
```

```
package List(xs) --* folding List(zs) in
                     applying List(zs) --* List(ys) in
                     List(ys)
```

Figure 3 Illustration of the organisation of permissions in the loop invariant from Figure 2, via magic wand and predicate instances. The magic wand instance covers permissions to the prefix of the list (starting at `ys`) that has already been traversed by the loop. For simplicity, the cases of `xs`/`xs.next` being `null` have been ignored.

permissions to the already-inspected nodes in the original list, such that rather than these being lost after the loop terminates, they can be recovered by simply applying the wand (line 21). This enables the verification of the postcondition of `sum_it` (Figure 1, line 15), which expresses that a `List` predicate will be returned to the caller, along with the knowledge that this iterative code computes the same value as the function `sum_rec`[5].

Figure 3 conceptually illustrates the permissions that the magic wand instance in our loop invariant represents, by stepping through the important stages of verifying the loop body (for simplicity, the cases of `xs`/`xs.next` being `null` have been ignored). At the beginning of the loop body, the magic wand's footprint includes the permissions (to fields `val` and `next`) from the head of the linked list `ys` all the way down to – but excluding – the current node `xs`. The remaining permissions, i.e. those to the current node and the tail of the list, are contained in the predicate instance `List(xs)`. The latter is then unfolded, providing permissions to the fields of `xs`, i.e. to `acc(xs.val)` and `acc(xs.next)`, and `xs` is then advanced such that it points to the next node (i.e. to `zs.next`). In order to reestablish the loop invariant, in particular, to reestablish that the wand instance includes the permissions to the already visited prefix of the list, we therefore need to add the permissions to `zs.val` and `zs.next` to the wand instance. This is (conceptually) achieved by the final `package`-statement: the ghost operations on the RHS of the wand force the wand's footprint to include the footprint of the wand held so far, plus the permissions to `zs.val` and `zs.next` (necessary for the `folding` ghost operation specified). Ghost operations will be explained in Section 4.

[5] The `old` construct specifies that the nested expression should be evaluated in the *heap* of the method's prestate; the evaluation of program variables is not affected by `old`.

As we show in this paper, our support for the magic wand connective allows a natural specification of these "left-over" parts of data structures, in a way which requires few annotations, and applies equally well to other data structures and predicates. This is an important use-case for our work, but (as discussed in the introduction) the magic wand has various other known applications (e.g. [17, 10, 15, 35, 9]); the possibility of practical tool support via the contributions of this paper will likely also lead to further applications being explored.

3 Magic Wand Support with Automatic Footprints

We present our solution for 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 to other permission-based verification logics.

3.1 Basic Operations

We use σ to range 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 generation, they could consist of maps representing the heap and permissions held. States must be able to record assumptions, permissions and magic wand instances (see the next subsection). Figure 4 defines the interface we expect to be implemented by the states. We represent these interface operations as functions on (and producing) immutable states. In practice, the operations could be implemented by generating a corresponding program in an intermediate language, or by directly updating internal (potentially mutable) state in a verification tool.

The state operations `hasAcc`, `addAcc` and `removeAcc` are used respectively to check that a state holds permissions to a field, to add and to remove such permission from a state. Analogous operations are included for predicate and magic wand instances. With respect to our running example (Figure 2), `addPred` will be, for example, used to add an instance of `List(xs)` to the state when inhaling the loop invariant at the beginning of the loop body, and `removePred` will be used at the end of the loop body when exhaling the loop invariant. The `unfold` operation on line 15 would typically use (amongst other operations) `removePred` to remove the predicate instance `List(xs)` that is to be unfolded, and `addAcc` to add permissions to `xs.val` to the state. In line 16, `hasAcc` would be used to assert that reading the field `xs.val` is permitted, i.e. that the state holds permissions to the field.

Depending on the implementation approach taken for a particular verifier, the implementation of these operations will vary. In a verifier which translates to an intermediate language such as Boogie [20], operations such as `addAcc` and `removeAcc` would typically be implemented by generating modifications of the Boogie program state (used to model e.g. permissions held), while in a symbolic execution tool these operations would typically involve mutation of a collection maintained by the verification tool to represent the make up of the current state.

The idea behind `equate` is to be able to communicate information (i.e. logical constraints) from one state to another. In particular, we use this operation to model adding the information

σ.`eval`(e)	$\approx$ evaluates expression e in state σ, yielding a value v
σ.`assume`(e)	$\approx$ assume that e holds in σ
σ.`assert`(e)	$\approx$ assert that e holds in σ
σ.`hasAcc`(v, f)	$\approx$ true iff σ contains access to $v.f$
σ.`addAcc`(v, f)	$\approx$ add access to $v.f$ to σ
σ.`removeAcc`(v, f)	$\approx$ remove access to $v.f$ from σ
σ.`hasPred`(P, v)	$\approx$ true iff σ contains $P(v)$
σ.`addPred`(P, v)	$\approx$ add $P(v)$ to σ
σ.`removePred`(P, v)	$\approx$ remove $P(v)$ from σ
σ.`hasWand`$(A \mathbin{-\!*} B)$	$\approx$ true iff σ contains $A \mathbin{-\!*} B$
σ.`addWand`$(A \mathbin{-\!*} B)$	$\approx$ add $A \mathbin{-\!*} B$ to σ
σ.`removeWand`$(A \mathbin{-\!*} B)$	$\approx$ remove $A \mathbin{-\!*} B$ from σ
σ.`onlyvars`$()$	$\approx$ returns a state σ' that is empty, except for local variables
	$\approx$ declared in σ, and all assumptions σ has about them
σ_1.`equate`(σ_2, v, f)	$\approx$ update σ_1 s.t. it contains all assumptions from σ_2 about $v.f$
`if` $(\ldots)$ $\ldots$ `else` $\ldots$	$\approx$ conditional operation

Figure 4 Basic state operations. e denotes an expression, σ a state, and v denotes a value (of appropriate type), i.e. the result of evaluating an e in a σ. All operations except `eval`, `hasAcc`/`hasPred`/`hasWand`, `onlyvars` and `if` return an updated state.

that the value of an expression in σ_1 is the same as in σ_2. We expect σ_1.`equate`(σ_2, v, f) to produce a modified version of σ_1, in which information known about the value of $v.f$ 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 $v.f$ 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 for verification (such as Boogie) 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 overload `if` such that it can be used with (boolean) state values v, e.g. `if` σ.`eval`(e), and with (boolean) return values of operations on state, e.g. `if` σ.`hasAcc`(v, f).

We can now define the `inhale` and `exhale` operations as functions of states (Figure 5), 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, σ.`inhale`$(a_1 * a_2) \rightsquigarrow$ σ.`inhale`(a_1).`inhale`(a_2) represents that inhaling a conjunction is defined as inhaling the second conjunct in the state resulting from inhaling the first conjunct.

```
σ.exhale(a)                 ⇝   σ.exhale(σ, a)
σ.exhale(σ̃, e)              ⇝   σ̃.assert(e)
σ.exhale(σ̃, a₁ * a₂)        ⇝   σ.exhale(σ̃, a₁).exhale(σ̃, a₂)
σ.exhale(σ̃, e ⇒ a)          ⇝   if σ̃.eval(e) σ.exhale(σ̃, a) else σ
σ.exhale(σ̃, acc(e.f))       ⇝
    v := σ̃.eval(e)
    if σ.hasAcc(v, f) σ.removeAcc(v, f) else fail

σ.inhale(e)                 ⇝   σ.assume(e)
σ.inhale(a₁ * a₂)           ⇝   σ.inhale(a₁).inhale(a₂)
σ.inhale(e ⇒ a)             ⇝   if σ.eval(e) σ.inhale(a) else σ
σ.inhale(acc(e.f))          ⇝   v := σ.eval(e); σ.addAcc(v, f)
```

Figure 5 The interesting cases for the definitions of exhaling and inhaling assertions (see also [21]). 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). The cases for inhaling/exhaling predicate and wand instances are analogous to the case of inhaling/exhaling `acc`($e.f$). `fail` is a short-hand for σ.`assert(false)`.

3.2 Representing, Applying and Packaging Wands

Our approach to supporting magic wands can be related to the handling of recursive definitions via ghost operations (cf. Section 2.5) as follows: Just as for predicates, we wish to be able to derive new magic wand assertions, and to apply their meanings while verifying code, but tackling this problem automatically without any direction from the user is known to be undecidable, even for much more restricted assertion logics than those we wish to support [5].

Analogous to predicate instances, we treat instances of magic wands as *opaque*; when one is available in a verification state, the verifier need not attempt to deduce anything that follows from the wand's meaning, without direction to do so. The choice to use such a magic wand instance must be directed by a ghost statement `apply` $A \mathrel{-\!\!*} B$ (see, for example, line 21 of Figure 2). Recall the formal semantics of a magic wand assertion from Section 1:

$$\sigma_{foot} \vDash A \mathrel{-\!\!*} B \;\Leftrightarrow\; \forall \sigma_A \perp \sigma_{foot} \cdot\; (\sigma_A \vDash A \;\Rightarrow \sigma_{foot} \uplus \sigma_A \vDash B)$$

This semantics intuitively says that $A \mathrel{-\!\!*} B$ is true in a state σ_{foot} if it is guaranteed that the state created by combining this state with some additional state σ_A satisfying A, satisfies B. One can see this as a definition in terms of what can be *deduced* from a magic wand, according to the following *Modus-Ponens*-like inference rule from separation logic: $A * (A \mathrel{-\!\!*} B) \vDash B$, and we analogously define the operation of applying a wand instance in a state σ as follows:

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

Just as for predicate instances, the opaque treatment of magic wand instances requires for soundness that the state σ_{foot} in the semantics above must notionally *belong* to the magic wand instance, in the sense that the program is not allowed to modify that part of the state up until the wand instance is applied. We call such a state the *footprint* of the magic wand instance. Whenever a new magic wand instance is to be added to the state, we need to compute some suitable part σ_{foot} of the current state σ, that will suffice to guarantee the wand's semantics, and then remove σ_{foot} from the current state, and add the new magic wand instance. We call this operation (of choosing a suitable footprint for a magic wand instance, and exchanging the footprint for the wand instance) *packaging* the magic wand instance, and use a ghost command `package` $A \mathrel{-\!\!*} B$ to indicate that this operation should be performed.

Unlike the folding of a predicate instance, a suitable choice of footprint for a magic wand instance is not directly determined by its definition. As discussed, such a footprint must be *some* state guaranteeing the semantics of the wand, but this doesn't indicate *how* this state should be chosen. The key contribution which keeps our approach lightweight is that we have defined a useful strategy (and corresponding algorithm) for automatically choosing footprint states.

3.3 Strategy for Choosing Footprints

Automatically choosing a suitable footprint for a magic wand instance is challenging. According to the approach outlined in the previous subsection, a package operation `package` $A \mathrel{-\!\!*} B$ must attempt to choose a footprint σ_{foot}, which can be any portion of the current state so long as it satisfies the wand's semantics. In checking this criterion, it would be unsound to use any facts from the current state which are *not* framed by permissions that we choose to put into the footprint, since these might no longer be true by the time the wand instance is applied[6]. For example, when proving `acc(x.f) -* acc(x.f) * x.f = 3` in a state where `x.f = 3`, we may only use this fact if we store permission to `x.f` in the footprint; otherwise, the value of `x.f` could have been changed by the time the wand instance is applied.

In deciding on a strategy for choosing footprints, we could soundly restrict the choice of a wand instance's footprint state to be *any* portion of the current state, so long as we then check that this choice indeed guarantees the wand's semantics. Certain strategies for choosing a footprint are, however, more useful than others. For example, we *could* always choose the empty state as a footprint, and therefore not use up any permissions at a `package` statement; the check of the wand's semantics (with $\sigma_{foot} = \varnothing$) would typically then fail: effectively we would only support wands where A logically entails B, which, for example, would not be sufficient to specify our running example. Alternatively, we could always 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 would be almost certain not to verify, since all permissions from the current state would have been lost to the wand instance. Although 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 guaranteeing enough information for the wand's semantics. However, the notion of "as small as possible" is not straightforward to define precisely. For example, we can satisfy the semantics of a wand $A \mathrel{-\!\!*} B$ by choosing a footprint state which includes enough permissions such that a state satisfying A can never again be obtained; this would yield a true wand instance (the inability to find a suitable σ_A state makes the semantics of the wand vacuously true), but one which could never be applied, which is not useful as a verification construct.

Recall the previous example, in which `acc(x.f) -* acc(x.f) * x.f = 3` is to be packaged in a state in which `x.f = 3`. As discussed, this fact may only be soundly used when proving the RHS of the wand if permission to `x.f` goes into the wand's footprint. Although this extra logical fact is useful in proving the right-hand-side, such a decision would again yield a wand instance which cannot be applied, since the LHS of this wand requires this

[6] We handle only magic wand assertions $A \mathrel{-\!\!*} B$ in which the assertions A and B are self-framing. Thus, we disallow awkward assertions such as $true \mathrel{-\!\!*} x.f = 3$. This is not a strong restriction in practice; indeed, in standard separation logics, all assertions are self-framing [26].

```
transfer($\overline{\sigma_i} \cdot \sigma, \sigma_{used}$, acc($e.f$)) $\rightsquigarrow$
    $v$ := $\sigma_{used}$.eval($e$)
    if $\sigma$.hasAcc($v, f$) {
        $\sigma'_{used}$ := $\sigma_{used}$.addAcc($v, f$)
        $\sigma''_{used}$ := $\sigma'_{used}$.equate($\sigma, v, f$)
        $\sigma'$ := $\sigma$.removeAcc($v, f$)
        return ($\overline{\sigma_i} \cdot \sigma', \sigma''_{used}$)
    } else {
        ($\overline{\sigma'_i}, \sigma'_{used}$) := transfer($\overline{\sigma_i}, \sigma_{used}$, acc($e.f$))
        return ($\overline{\sigma'_i} \cdot \sigma, \sigma'_{used}$)
    }

transfer($\epsilon, \sigma_{used}$, acc($e.f$)) $\rightsquigarrow$ fail
```

Figure 6 $\overline{\sigma_i}$ denotes a (potentially empty) stack of states, and $\overline{\sigma_i} \cdot \sigma$ denotes a stack created by pushing a single state σ onto a stack of states $\overline{\sigma_i}$. Function `transfer` descends a stack of states and tries to find permissions to location $e.f$. If successful, the permissions are transferred into σ_{used}. The cases for $P(e)$ and $A \mathrel{-\!\!*} B$ are defined analogously.

permission to be provided when applying the wand instance. Essentially, any permissions which we choose to take from the current state when they are already provided by the LHS of the wand, are *leaked* at the point of packaging an instance of that wand, which is not typically useful for verifying the rest of the program.

Motivated by these observations, our strategy for choosing wand footprints is: *include all permissions required by the wand's RHS, which we cannot prove to be provided by the wand's LHS.* We observe that restricting the choice of footprint to *only* these permissions is not really a restriction in practice. If the tool user *intends* to include extra permissions from the current state to in a wand's footprint, they can achieve it by writing a RHS which requires more permission than the LHS provides. In the next subsection, we explain how this high-level strategy can be realised in practice.

3.4 Footprint Computation Algorithm

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

To break this circularity, we devised an algorithm called `exhale_ext` to *simultaneously* evaluate the wand's RHS to determine the permissions it requires, and construct a new state σ_{used}, containing these required permissions. These permissions are taken from the current state if they cannot be proved to be provided by the wand's LHS; thus, we implicitly carve out a suitable footprint for the wand from the current state.

Our algorithm is shown in Figure 7, and works as follows: Let σ be the current state in which a `package` $A \mathrel{-\!\!*} B$ operation takes place. We first construct a hypothetical extra state σ_A representing the information provided by the wand's LHS (by inhaling the LHS into

```
$\sigma$.package($A \mathrel{-\!\!*} B$) $\leadsto$
    $\sigma_{emp}$ := $\sigma$.onlyvars()
    $\sigma_A$ := $\sigma_{emp}$.inhale($A$)
    ($\sigma' \cdot \sigma'_A, \sigma'_{used}$) := exhale_ext($\sigma \cdot \sigma_A, \sigma_{emp}, B$)
    return $\sigma'$.addWand($A \mathrel{-\!\!*} B$)

exhale_ext($\overline{\sigma_i}, \sigma_{used}$, acc($e.f$)) $\leadsto$
    return transfer($\overline{\sigma_i}, \sigma_{used}$, acc($e.f$))

exhale_ext($\overline{\sigma_i}, \sigma_{used}, A_1 * A_2$) $\leadsto$
    ($\overline{\sigma'_i}, \sigma'_{used}$) := exhale_ext($\overline{\sigma_i}, \sigma_{used}, A_1$)
    ($\overline{\sigma''_i}, \sigma''_{used}$) := exhale_ext($\overline{\sigma'_i}, \sigma'_{used}, A_2$)
    return ($\overline{\sigma''_i}, \sigma''_{used}$)

exhale_ext($\overline{\sigma_i}, \sigma_{used}, e$) $\leadsto$
    $\sigma_{used}$.assert($e$)
    return ($\overline{\sigma_i}, \sigma_{used}$)
```

Figure 7 Packaging a wand instance. σ_{emp} is empty except for local variables and assumptions about those from σ. Permissions transferred into σ_{used} contribute to the footprint. `exhale_ext` of $P(e)$ and $A \mathrel{-\!\!*} B$ is defined analogous to the case of `acc`($e.f$). Other cases of `exhale_ext` are analogous to `exhale`, but expressions are evaluated in σ_{used}, as in the e case, above.

an empty state σ_{emp}). We then use `exhale_ext` to attempt to compute a state σ'_{used} that satisfies the wand's RHS, i.e. such that $\sigma'_{used} \vDash B$. State σ'_{used} is constructed by successively transferring permissions from the *stack* of states $\sigma \cdot \sigma_A$ into σ_{emp}. As motivated in Section 3.4, the algorithm tries to minimise the computed footprint by taking permissions preferentially from σ_A and only from σ when needed[7].

Our `exhale_ext` algorithm computes a footprint only implicitly; the state σ_{used} is not the footprint itself, but (if the algorithm terminates successfully) corresponds to (a part of) σ_A combined with a part of the input state σ which has been removed from the resulting state σ'; the removed part corresponds to the chosen footprint. At at any point *during* the execution of `exhale_ext`, the current σ_{used} satisfies the prefix of the wand's RHS that has been processed so far. Recall that both sides of a wand have to be self-framing; in particular, on a wand's RHS the permission to access a heap location occurs before any expressions that depend on the location's value. This is why expressions can be evaluated in σ_{used}; any necessary permissions (and assumptions about the location's value) will already have been transferred into this state by our algorithm.

The separation of the two initial states σ and σ_A in our algorithm is essential for a correct footprint computation. A naïve algorithm which simply *combined* the hypothetical extra state with the current state before trying to exhale the wand's RHS would be unsound: this combination might be inconsistent (due to holding too many permissions, or to conflicting value facts), which would trivialise the check of the wand's RHS assertion.

[7] In the context of Figure 7 the stack of states making up the first argument to `exhale_ext` will always consist of *two* states, corresponding to the input $\sigma \cdot \sigma_A$, but our algorithm is defined to take a stack of states of any length. This generalisation will be shown to be necessary in Section 4 for supporting nested magic wand assertions.

```
$\sigma$.package($A \mathbin{-\!\!*} G$) $\rightsquigarrow$
    $\sigma_{emp}$ := $\sigma$.onlyvars()
    $\sigma_A$ := $\sigma_{emp}$.inhale($A$)
    ($\sigma' \cdot \sigma'_A, \sigma'_{used}$) := exec($\sigma \cdot \sigma_A, \sigma_{emp}, G$)
    return $\sigma'$.addWand($A \mathbin{-\!\!*}$ nested($G$))

exec($\overline{\sigma_i}, \sigma_{ops}$, folding $P(e)$ in $G$) $\rightsquigarrow$
    $\sigma_{emp}$ := $\sigma_{ops}$.onlyvars()
    $v$ := $\sigma_{ops}$.eval($e$)
    ($\overline{\sigma'_i} \cdot \sigma'_{ops}, \sigma'_{used}$) := exhale_ext($\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{emp}$, Body($P$)[param $\mapsto v$])
    $\sigma''_{used}$ := $\sigma'_{used}$.fold($P, v$)
    return exec($\overline{\sigma'_i}, \sigma'_{ops} \uplus \sigma''_{used}, G$)

exec($\overline{\sigma_i}, \sigma_{ops}$, unfolding $P(e)$ in $G$) $\rightsquigarrow$
    $\sigma_{emp}$ := $\sigma_{ops}$.onlyvars()
    $v$ := $\sigma_{ops}$.eval($e$)
    ($\overline{\sigma'_i} \cdot \sigma'_{ops}, \sigma'_{used}$) := exhale_ext($\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{emp}, P(e)$)
    $\sigma''_{used}$ := $\sigma'_{used}$.unfold($P, v$)
    return exec($\overline{\sigma'_i}, \sigma'_{ops} \uplus \sigma''_{used}, G$)

exec($\overline{\sigma_i}, \sigma_{ops}$, applying $A \mathbin{-\!\!*} B$ in $G$) $\rightsquigarrow$
    $\sigma_{emp}$ := $\sigma_{ops}$.onlyvars()
    ($\overline{\sigma'_i} \cdot \sigma'_{ops}, \sigma'_{used}$) := exhale_ext($\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{emp}, A * (A \mathbin{-\!\!*} B)$)
    $\sigma''_{used}$ := $\sigma'_{used}$.apply($A \mathbin{-\!\!*} B$)
    return exec($\overline{\sigma'_i}, \sigma'_{ops} \uplus \sigma''_{used}, G$)

exec($\overline{\sigma_i}, \sigma_{ops}$, packaging $A \mathbin{-\!\!*} G_1$ in $G_2$) $\rightsquigarrow$
    $\sigma_{emp}$ := $\sigma_{ops}$.onlyvars()
    $\sigma_A$ := $\sigma_{emp}$.inhale($A$)
    ($\overline{\sigma'_i} \cdot \sigma'_{ops} \cdot \sigma'_A, \sigma'_{used}$) := exec($\overline{\sigma_i} \cdot \sigma_{ops} \cdot \sigma_A, \sigma_{emp}, G_1$)
    $\sigma''_{ops}$ := $\sigma'_{ops}$.addWand($A \mathbin{-\!\!*}$ nested($G_1$))
    return exec($\overline{\sigma'_i}, \sigma''_{ops}, G_2$)

exec($\overline{\sigma_i}, \sigma_{ops}, A$) $\rightsquigarrow$
    $\sigma_{emp}$ := $\sigma_{ops}$.onlyvars()
    return exhale_ext($\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{emp}, A$)
```

Figure 8 Executing ghost operations. The first three exhibit the same structure: 1. `exhale_ext` determines the footprint of the operation and transfers it from $\overline{\sigma_i} \cdot \sigma_{ops}$ into σ_{emp}, yielding $\overline{\sigma'_i} \cdot \sigma'_{ops}$ and σ'_{used}; 2. the actual operation is performed, rewriting σ'_{used} into σ''_{used}; 3. the execution continues in updated states. The packaging ghost operation proceeds analogous to the package statement. The last case, `exec`($\_, A$), handles assertions with no further ghost operations.

For example, suppose that σ holds permission to fields `x.f` and `y.f`, with values 1, respectively, 2. The operation `package (acc(x.f) * x.f = 2)` $\mathbin{-\!\!*}$ `(acc(x.f) * acc(y.f) * x.f = y.f)` will succeed, and the footprint `acc(y.f)` will be removed from the current state. The fact `y.f = 2` from the original state will be used for checking the wand's RHS (which is justified, since the permission `acc`($y.f$) is taken from the current state for the wand's footprint). But importantly, the fact `x.f = 1` (which contradicts the wand's LHS) will *not* be used when checking the wand's RHS, since permission to this location is not taken from the current state. For this reason, packaging the alternative wand `package (acc(x.f) * x.f = 2) -* (acc(x.f) * acc(y.f) *` *`false`*`)` will fail, as it should.

For simplicity, our presentation in Figure 7 does not include cases for handling permissions under conditionals, i.e. when wands such as `true -* (b ? acc(x.f) : acc(x.g))` are to be packaged. Effectively, the footprint calculation must *branch* on the condition b, finding an appropriate footprint for when b is true, and another for when b is false. The tool can then either overapproximate, by removing at least as much as is taken in each branch, or (as in our implementation), can record the removal of the footprint information as being conditional on the value of b.

For example, assume that the current state satisfies `acc(x.f) * acc(x.g)`. Packaging an instance of the wand above succeeds, resulting in a state satisfying `(!b ? acc(x.f) : acc(x.g)) * (true -* (b ? acc(x.f) : acc(x.g)))`.
Consequently, trying to `assert acc(x.f)` will fail in that state (and likewise for `acc(x.g)`), but `assert (b ⇒ acc(x.g))` will succeed. Moreover, after applying the wand instance, `assert acc(x.f) * acc(x.g)` will succeed. We illustrate this handling of conditionals in one of our online examples (cf. Section 6).

4 Integrating Ghost Operations

The magic wand support described in the previous section forms the core of our solution, but it is not yet expressive enough to integrate well with features of the full logic such as predicates. In particular, in the proof (packaging) of a new magic wand instance, it is often necessary to be able to specify ghost operations *between* the hypothetical addition of the wand's LHS, and the proof of the RHS, for example, because the wand's RHS is a predicate instance that can only be folded once the state described by the wand's LHS is provided.

Our running example (Figure 2) exhibits an instance of this situation on line 19, when re-establishing the magic wand in the loop invariant. Recall that the wand `(xs ≠ null ⇒ List(xs)) -* List(ys)` expresses that we can obtain a predicate instance describing the complete list if we give up the "remainder list" starting from `xs`. Consider how we can re-establish this invariant at the end of the loop body: in particular, the state at line 18. In this state, we have permissions to the fields `zs.val` and `zs.next` of the current node (obtained from the `unfold` at line 15). We also have 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 instance 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 instance `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, which requires in the process the additional permissions `acc(zs.val)` and `acc(zs.next)` and the wand instance `w` (these constitute the footprint of the new wand instance).

In order to allow such ghost operations to be expressed when packaging wand instances, we generalise the `package` statement to the form `package` $A \mathbin{-\!\!*} G$, where G is an assertion A possibly nested inside ghost operations, as defined by:

$$\begin{array}{rcllll} G & ::= & A & \mid \ \texttt{folding}\,P(e)\,\texttt{in}\,G & \mid & \texttt{unfolding}\,P(e)\,\texttt{in}\,G \\ & & & \mid \ \texttt{packaging}\,A \mathbin{-\!\!*} G\,\texttt{in}\,G & \mid & \texttt{applying}\,A \mathbin{-\!\!*} A\,\texttt{in}\,G \end{array}$$

Note that we include syntax for nesting an assertion inside a ghost operation for each ghost operation that we support in statement position (`(un)fold`, `package`, `apply`; in other verifiers, more could be added). The difference is that the syntax here indicates that the ghost operation should be applied *during* the footprint computation for the new wand instance, rather than in the current state.

A successful `package` $A \mathrel{-\!\!*} G$ operation does *not* add a wand instance of the form $A \mathrel{-\!\!*} G$ to the state (which is not an assertion according to Section 2.3), but rather $A \mathrel{-\!\!*} A'$ where A' 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 they do not affect *what* the resulting wand instance represents. We write `nested`(G) to denote the assertion nested inside the ghost operations. For example, `nested(folding List(zs) in applying w in List(xs)) = List(xs)`.

Our automatic footprint computation is extended to support these ghost operations, as shown in the rules in Figure 8. These rules specify a modified definition of `package`, as well as rules for executing the ghost operations. Such execution requires finding and transferring suitable permissions from the (stack of) input states, such that the specified ghost operation can be executed; for example, in order to execute a `folding`, we must find the permissions required by the body of the corresponding predicate instance. The state *resulting* from the successfully-executed ghost operations is maintained as the separate parameter σ_{ops} (which is initially empty apart from assumptions about local variables). In effect, each ghost operation rewrites already existing permissions into a different representation, which is then available to subsequent ghost operations or the eventual call to `exhale_ext`, once all ghost operations have been handled.

As an example, consider the `package` statement on line 19 of Figure 2, and assume that σ denotes the state before the `package` statement. Hence, line 19 corresponds to performing an operation σ.package($A \mathrel{-\!\!*}$ `folding List(zs) in` ...). Following Figure 7, this will result in `exec`($\sigma \cdot \sigma_A, \sigma_{emp},$ `folding List(zs) in` ...), which means that all permissions necessary for executing the ghost operations must come from either the current state σ or the hypothetical LHS state σ_A.

The rules for executing the first three ghost operations shown in Figure 8 (`(un)folding` and `applying`) exhibit the same structure: First, `exhale_ext` is used to find the necessary state for executing the ghost operation at hand (including checking that all necessary boolean assertions are true), and to transfer that state from $\overline{\sigma_i} \cdot \sigma_{ops}$ into σ_{emp}, which yields $\overline{\sigma_i'} \cdot \sigma_{ops}'$ (the remainders of the input states) and σ_{used}'. Next, the actual ghost operation is performed on σ_{used}', which is thereby rewritten into σ_{used}''. Note that this operation is guaranteed to succeed because of the preceding `exhale_ext`. This rewriting of the state does not change which assertions are satisfied by the state in terms of the ideal (equirecursive) semantics of assertions, but for a verifier which differentiates between predicate instances and their bodies (and between wand instances and their footprints), the ghost operation can affect what the tool can show about the resulting state. Finally, the execution continues in the updated states.

In the context of the operations in line 19 of our running example, the execution of the ghost operation `folding List(zs) in` ... proceeds by invoking `exhale_ext`($\sigma \cdot \sigma_A \cdot \sigma_{emp}, \sigma_{emp},$ `Body(List(zs))`), which transfers the footprint of the body of `List(zs)` (see Figure 1) to σ_{emp} (the second argument). The footprint comprises the permissions corresponding to `acc(zs.val)` and `acc(zs.next)`, and, assuming `zs` $\neq$ *null*, the predicate instance `List(zs)`. The algorithm tries to take as many permissions as possible from the state on top of the stack (σ_A in our example), but since σ_A only provides `List(zs.next)`, the other permissions are taken from σ. The resulting stack of states is $\sigma' \cdot \sigma'_A \cdot \sigma_{emp}$ (where σ'_A is also essentially σ_{emp}), along with the single state σ'_{used} which is a state sufficient to satisfy `Body(List(zs))`, i.e. $\sigma'_{used} \vDash$ `Body(List(zs))`.

In the next step, σ'_{used} is rewritten into σ''_{used} by folding `List(zs)`, which replaces the footprint of the predicate body by an instance of the predicate[8].

Finally, the execution of the package statement from line 19 continues by invoking `exec(`$\sigma' \cdot \sigma'_A, \sigma_{emp} \uplus \sigma''_{used},$ `applying w in List(ys))`: executing the `applying` ghost operation proceeds by transferring `List(zs)` (the LHS of `w`) and the wand instance `w` itself from $\sigma' \cdot \sigma'_A \cdot \sigma''_{used}$to another fresh σ_{emp}. In particular, `List(zs)` is taken from σ''_{used}, and `w` is taken from σ'. Afterwards, `apply` is used to rewrite the state that now contains `List(zs)` and `w` such that it contains the RHS of `w`, i.e. `List(ys)`. Finally, `List(ys)` itself is transferred into a fresh σ_{emp} (see the last rule of Figure 8). The execution returns to the `package` rule from the start of the figure, in which the σ'_{used} is essentially a state satisfying exactly `List(ys)` – and as such, satisfies the RHS of the wand instance we set off to package. The remainder of the LHS state, σ'_A, is discarded (it is essentially empty in our case anyway), and the remainder of the current state, σ', is extended with an instance of $A \mathrel{-\!\!*}$ `List(ys)` before it is used for the verification of the rest of the program. The permissions taken from σ' compared with σ (i.e. `acc(zs.val)` and `acc(zs.next)`, as well as the wand instance `w`) conceptually belong to the newly-added wand's footprint.

Two cases from Figure 8 remain to explain: The ghost operation `packaging` $A \mathrel{-\!\!*} G$ (representing a recursive packaging of a wand instance, necessary for example for packaging nested magic wands of the form $A \mathrel{-\!\!*} (B \mathrel{-\!\!*} C)$) works similarly to the `package` statement, i.e. it creates an extra LHS state σ_A satisfying A, pushes it onto the stack of already existing states, and executes ghost operations potentially occurring in G (to which σ_A is available). Finally, it adds the magic wand $A \mathrel{-\!\!*}$ `nested`(G) to the state, and continues the execution.

The last case, executing a ghost-operation-free assertion A, only applies to the inner-most assertion nested inside a chain of ghost operations (i.e. A is `nested`(G) for some G). At this point, the footprint computation falls back to the `exhale_ext` algorithm of Figure 7.

Note that automatic footprint computation in the presence of ghost operations is especially challenging in the case of nested package operations; one has to track the various hypothetical states carefully, and ideally remove permissions from these states preferentially, since any removal from the original state σ can affect the further verification of the rest of the program. The precise details of the algorithms are quite complex, but specifying them at the level of abstraction shown here helped to guide our original implementation and avoid soundness issues. We give a soundness argument in the following subsection.

4.1 Soundness of the Footprint Computation

In this subsection we sketch a soundness argument for the core of our magic wand support: the soundness of our approach depends essentially on the correctness of our footprint computation. We focus on showing that the state removed by a package operation satisfies the properties that: it was a part of the original state, and it satisfies the semantics of the newly-packaged wand (thus, any future `apply` of the wand instance will be justified). We formulate similar results for each of our `transfer`, `exhale_ext` and `exec` definitions, which we then instantiate to show the desired property for `package`. By convention, we use unprimed σ variables

[8] One way of implementing σ'_{used}`.fold(`$\cdot$`)` is to exhale the predicate body and inhale the predicate instance. However, some verifiers might choose to implement folding predicate instances in a custom way which preserves extra information [11]; we leave the precise implementation up to the particular verifier.

to denote input states, primed versions σ' to indicate corresponding output states, and $\widehat{\sigma}$ versions to indicate the "removed parts" of the input state. We also use an "inclusion" relation $\sqsubseteq$ on states, which has the meaning: $\sigma \sqsubseteq \sigma'$ iff $\forall A.(\sigma \models A \Rightarrow \sigma' \models A)$.

▶ **Theorem 1.** *If* $\mathtt{transfer}(\overline{\sigma_i}, \sigma_{used}, \mathtt{acc}(e.f))$ *succeeds with result* $(\overline{\sigma_i'}, \sigma_{used}')$*, then there exist* $\overline{\widehat{\sigma_i}}$*, such that:*

(P_1) $\overline{\sigma_i' \uplus \widehat{\sigma_i} \sqsubseteq \sigma_i}$

(P_2) $\sigma_{used}' \sqsubseteq (\overline{\uplus \widehat{\sigma_i}}) \uplus \sigma_{used}$

(P_3) $\forall A.(\sigma_{used} \models A \Rightarrow \sigma_{used}' \models A * \mathtt{acc}(e.f))$

The first two properties of Theorem 1 state that `transfer` does not add permissions (and assumptions) when it rewrites $(\overline{\sigma_i}, \sigma_{used})$ into $(\overline{\sigma_i'}, \sigma_{used}')$. Essentially, the original states σ_i are each split into a $\widehat{\sigma_i}$ part (which is removed from the state and makes up part of σ_{used}') and a remaining σ_i'. The output σ_{used}' consists of these removed parts, plus anything that was originally in σ_{used}. The third property of Theorem 1 essentially expresses that it is the *extra* parts of σ_{used}' that satisfy $\mathtt{acc}(e.f)$). In particular, repeated (successful) calls to `transfer` build up a state which satisfies the conjunction of all transferred permissions.

Proof of Theorem 1. The proof proceeds by induction on the input stack. In case of the empty stack ϵ, `transfer` fails, which contradicts the assumption made in Theorem 1. In case of a non-empty input stack $\overline{\sigma_i} \cdot \sigma$, let us consider the if-branch first: $\overline{\sigma_i}$ remains unchanged, hence, we choose $\overline{\widehat{\sigma_i}}$ to be a stack of empty states $\varnothing$. In addition, let $\widehat{\sigma}$ be $\varnothing.\mathtt{addAcc}(v, f)$. Given these choices, $\overline{\widehat{\sigma_i}} \cdot \widehat{\sigma}$ satisfies (P_1) and (P_2). The output state σ_{used}'' is the input state σ_{used} with exactly one extra permission added (along with any assumptions about that location's value), corresponding to $\mathtt{acc}(e.f)$. Thus, (P_3) also holds.

In the else-branch, the induction hypothesis yields appropriate $\overline{\widehat{\sigma_i}}$ for all states but the last; choosing $\widehat{\sigma}$ to be $\varnothing$ yields the desired properties. ◀

▶ **Theorem 2.** *If* $\mathtt{exhale\_ext}(\overline{\sigma_i}, \sigma_{used}, A)$ *succeeds with result* $(\overline{\sigma_i'}, \sigma_{used}')$*, then there exist* $\overline{\widehat{\sigma_i}}$*, such that* (P_1) *and* (P_2) *from Theorem 1 hold. Moreover:*
(P_3) $\forall A'.(\sigma_{used} \models A' \Rightarrow \sigma_{used}' \models A' * A)$ *holds.*

Proof of Theorem 2. The proof proceeds by induction on the structure of the assertion A. In case of $\mathtt{acc}(e.f)$, the desired results follow directly from `transfer`; in case of e, it suffices to choose $\overline{\widehat{\sigma_i}}$ to be a stack of empty states. The only interesting case is that of $A_1 * A_2$: Let $\overline{\widehat{\sigma_i}'}$ and $\overline{\widehat{\sigma_i}''}$ be the states whose existence follows from applying the induction hypothesis to the first and second recursive invocations of `exhale_ext`, respectively. Choosing $\overline{\widehat{\sigma_i}}$ to be $\overline{\widehat{\sigma_i}' \uplus \widehat{\sigma_i}''}$, it is straight-forward to combine the assumptions about $\overline{\widehat{\sigma_i}'}$ and $\overline{\widehat{\sigma_i}''}$ to show that (P_1), (P_2) and (P_3) hold for $\overline{\widehat{\sigma_i}}$. ◀

Theorem 2 is essentially a generalisation of Theorem 1: the first two properties of Theorem 2 are identical to those of Theorem 1; the third differs since the algorithm finds suitable substates to satisfy a general assertion A, instead of only a single required permission. We can use this result to reason about `exec`:

▶ **Theorem 3.** *If* $\mathtt{exec}(\overline{\sigma_i}, \sigma_{ops}, G)$ *succeeds with result* $(\overline{\sigma_i'}, \sigma_{ops}')$*, then there exist* $\overline{\widehat{\sigma_i}}$*, such that:*

(P_1) $\overline{\sigma_i' \uplus \widehat{\sigma_i} \sqsubseteq \sigma_i}$

(P_2) $\sigma_{ops}' \sqsubseteq (\overline{\uplus \widehat{\sigma_i}'}) \uplus \sigma_{ops}$

(P_3) $\sigma_{ops}' \models \mathtt{nested}(G)$

Proof of Theorem 3. The proof proceeds by induction on the structure of the (ghost operations in) the assertion G. The cases of `folding`, `unfolding` and `applying` are all similar to each other: From Theorem 2 and from the induction hypothesis applied to `exec`, the existence of $\overline{\widehat{\sigma_i}}'$ and $\overline{\widehat{\sigma_i}}''$ satisfying the appropriate conditions follows. Choosing $\overline{\widehat{\sigma_i}}$ to be $\overline{\widehat{\sigma_i}}' \uplus \overline{\widehat{\sigma_i}}''$, it suffices to combine the assumptions about $\overline{\widehat{\sigma_i}}'$ and $\overline{\widehat{\sigma_i}}''$, as well as to observe that the particular ghost operation applied does not (with respect to the semantics of the logic) increase the true assertions in the resulting state, to show that P_1 and P_2 hold for $\overline{\widehat{\sigma_i}}$. Since σ'_{ops} (the state returned) is the second component of the result of the recursive invocation of `exec`, the assumptions gained for this call from the induction hypothesis suffice to show that (P_3) holds. The case of `packaging` is similar to the above cases, but showing P_2 is more involved (similarly to our argument about `package` in Section 4.1): it is necessary to observe that: (i) the call to `exec`$(\ldots, G_1)$ removes a suitable footprint of $A \mathbin{-\!\!*} G_1$ from $\overline{\sigma_i} \cdot \sigma_{ops} \cdot \sigma_{emp}$ (by similar argument to that for `package` in Section 4.1), and (ii) that any footprint of $A \mathbin{-\!\!*} G_1$ is at least as expressive (in terms of which assertions can be deduced from it) as the wand $A \mathbin{-\!\!*}$ `nested`(G_1) which is added to the state in its place. In case of an assertion A without ghost operations, all properties follow directly from Theorem 2 applied to the call of `exhale_ext`. ◀

In Theorem 3, the first two properties are similar to those for our other results, essentially expressing that we remove parts of the input states to obtain σ'_{ops}. The third property differs; in general, executing the ghost operations involves rewriting the original state σ_{ops} (and pulling in extra parts of the stack of input states which are found to be missing). This is different from `exhale_ext`, which seeks to *add* everything in the required assertion to the pre-existing σ_{used}, which gives us the stronger third property in Theorem 1 and Theorem 2.

Nonetheless, if we consider the definition of `package` in Figure 8, we can see that these properties are sufficient to guarantee that the call to `exec` removes a correct footprint. In particular, instantiating Theorem 3 for this call, we obtain that there exist $\widehat{\sigma}, \widehat{\sigma_A}$ such that $(\sigma' \uplus \widehat{\sigma}) \sqsubseteq \sigma$, and, (combining the three properties from the theorem) $\widehat{\sigma} \uplus \widehat{\sigma_A} \models$ `nested`(G) (note that σ_{ops} is σ_{emp} for this call). From this, we obtain $\widehat{\sigma} \uplus \sigma_A \models$ `nested`(G). Since σ_A is an arbitrary state satisfying A, $\widehat{\sigma}$ is an appropriate footprint state for the wand. This state is removed from σ (resulting in the returned σ'), thus (assuming our algorithm reaches this point without failure) adding the wand instance in its place is justified, according to its semantics.

5 Inferring Annotations

In order to reduce the annotation overhead involved in specifying magic-wand-related ghost operations, we have extended the approach presented so far with a set of simple (and optional) heuristics, which attempt to insert additional `package` and `apply` operations into an input program. As described in Section 4, a `package` operation may also require nested ghost operations in order to succeed; our heuristics also attempt to infer these appropriately.

Our heuristics are *failure-directed*: if exhaling an assertion A, (e.g. a loop invariant), fails due to insufficient permissions (to a field, a predicate or a wand), then we apply the heuristics to search for ghost operations that avoid the failure. The heuristics search (in a depth-first manner) for a sequence of ghost operations that would rewrite the state such that the initially missing permissions can be found. The width of the search tree is bounded by the number of predicate and magic wand instances held in the current state (which is finite and typically small). The depth of the search is bounded by a configurable threshold. We also order the candidate ghost operations according to a number of syntactic criteria on the

```
var xs := ys;
sum := 0

define A
define B List(ys)

while (xs ≠ null)
  invariant (xs ≠ null ⇒ List(xs)) *
    ((xs ≠ null ⇒ List(xs)) −∗ List(ys)) *
    sum = old(sum_rec(ys)) - (xs = null ? 0 : sum_rec(xs))
{
  unfold List(xs)
  sum := sum + xs.val
  xs := xs.next;
}
```

Figure 9 Verified version of the body of `sum_rec` (Figure 1), with heuristics enabled.

symbolic state and program text, as a coarse estimate of which operations are "likely" to be successful, for example by preferentially unfolding predicate instances whose bodies appear to contain a suitable permission.

As an example, consider the loop body from our running example (Figure 2), and assume that the `package` statement in line 19 were removed, which would prevent the verifier from (immediately) showing that the invariant is preserved by the loop body. In particular, the verifier would fail to find the wand instance `A` −∗ `List(ys)` in the current state. This would trigger the heuristics, which would first detect that there is no predicate (or wand) instance in the current state that could be unfolded (or applied) to get a suitable wand instance. The heuristics would then try to package `A` −∗ `List(ys)`, which would fail because the desired predicate instance `List(ys)` would not be found either in the current state or the hypothetical state from the wand's left-hand-side. This would trigger the heuristics again, and result in an attempt to apply the wand instance `w` (which mentions `List(ys)`). Applying `w` would fail as well - because this wand's left-hand-side `List(zs)` is missing. The heuristics would be triggered once again, and try to fold `List(zs)`, which succeeds. The previously failing operations are then retried: that is, applying `w` and exhaling `List(ys)`, both of which succeed now. With these nested ghost operations, the initially triggered packaging of `A` −∗ `List(ys)` also succeeds, which enables the verifier to find the previously missing wand instance, and therefore, to show that the loop invariant is preserved.

Our heuristics allow us to remove all `package` and `apply` statements from the examples listed in Section 6. We can also remove `w` and `zs` (which were only used to facilitate writing a `package` statement) declared on line 12 and line 14, respectively, of Figure 2. The code shown in Figure 9 is verified by our implementation [29] when heuristics are enabled.

6 Implementation

Our implementation [29][9] includes the examples listed below (as well as a number of regression tests), all of which have been verified on a Intel Core i7-2600K 3.40GHz machine running Windows 7 x64 from an SSD. For each example we also include a version with the suffix `_heuristics.sil`, which is the example with heuristics activated and with all magic-wand-related annotations removed. The reported runtimes are averaged over ten runs per example

[9] In Silver, the separating conjunction is denoted by `&&`; for simplicity, we used `*` in this paper

(the standard deviations were always less than 0.1s). We state two (averaged) runtimes per example: the first figure is the overall runtime, which includes time for parsing and type-checking the example, starting-up the prover (i.e. Z3), and the verification time; the second records the verification time only.

- `list_sum.sil` is the running example from our paper. It verifies in 2.2s/0.7s, both with and without heuristics enabled (to infer the necessary magic-wand-related annotations).
- `list_insert.sil` is an encoding of an iterative algorithm for inserting a value into a sorted linked list. It verifies in 3.0s/1.5s (3.5s/2.0s with activated heuristics).
- `tree_delete_min.sil` is an encoding of challenge 3 from the VerifyThis verification competition at Formal Methods 2012, which was to verify an iterative implementation removing the minimal element from a binary search tree. The example verifies in 2.6s/1.1 (2.8s/1.4s with activated heuristics). VerCors [2] (the only comparable tool we are aware of with magic wand support) requires substantially more annotations to specify this example, and takes 6 minutes (on a comparable machine).
- `un_currying.sil` demonstrates how nested ghost operations can be used to prove the standard "currying" and "uncurrying" property of magic wands: $A * B \mathrel{-\!\!*} C \Leftrightarrow A \mathrel{-\!\!*} (B \mathrel{-\!\!*} C)$. The "$\Rightarrow$" case is especially interesting since it requires nested `packaging` operations. The example verifies in 1.6s/0.3s (both with and without heuristics).
- `conditionals.sil` illustrates and explains how our tool handles magic wands where the footprint is affected by conditionals whose guards depend on locations that are provided by the LHS of the wand. It verifies in 1.8s/0.4s. (We do not provide a version with activated heuristics because adding assertions that trigger packaging and applying the involved wands turned out to be more overhead than explicitly packaging/applying them.)

7 Conclusions and Related Work

We have presented a novel technique enabling the support of magic wands in automatic verification tools. Our approach requires moderate additional 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. We have implemented the described support as an extension of the verification tool Silicon [16], which supports implicit dynamic frames assertions. Our work makes few assumptions about the underlying verifier and specific logic, and should be easy to apply in other tools, such as verifiers for separation logics.

Lee and Park have recently developed a proof system for a separation logic supporting the magic wand connective [19], which also provides a decision procedure for propositional separation logic (i.e. without variables). In a richer logic such as ours, however, the magic wand is known to be undecidable [5]. Our work addresses this difficulty with the combination of `apply` and `package` annotations (which can often be inferred by our heuristics), along with novel algorithms for computing appropriate magic wand footprints automatically.

In parallel with our work, Blom and Huisman [2] have developed support for magic wands in their VerCors verifier. VerCors translates Java programs with separation-logic-style specifications into Chalice programs [21], and magic wands are eliminated during the translation by a clever encoding into additional Chalice classes whose instances ("witness objects") represent magic wand instances. This translation is automatic, but similar annotations to our approach are needed to direct the creation and use of magic wands. In contrast to our approach, the user must also manually specify annotations defining the permissions and logical facts to be used from the current state for each wand's footprint, which are then combined to show the wand's RHS via arbitrary user-defined ghost code. The ability to use arbitrary code is potentially more flexible than the ghost operations our tool supports (for example, ghost methods could be employed), but the resulting annotation overhead is significantly higher than with the automatic footprint computation presented in this paper (even comparing without the additional heuristics described in Section 5). Moreover, their translation does not support nested wands such as $A \mathrel{-\!\!*} (B \mathrel{-\!\!*} C)$ or wands inside predicate definitions (although we believe it could be extended to handle the latter).

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 is sufficient for many loop invariants, such as the one in our example. The PhD thesis of Retert [28] provides an abstract-interpretation-based approach supporting this connective.

The specific problem of rewriting and maintaining appropriate predicate definitions during data structure traversals has already received much attention. Without an alternative to simple `fold`/`unfold` annotations, one needs to define a new predicate type to represent "partial" versions of the data structure, and write ghost methods to "append" to this partial version, as well as to rewrite it into the original predicate once the traversal is completed. The problems of tracking suitable permissions in loop invariants are discussed in detail by Tuerk [34], 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 his idea (where "*rest*" refers to the remaining loop iteration, and "*all*" the entire loop). Making use of magic wand support is more general than Tuerk's proof rule, for example when further code after the loop is needed before restoring the overall predicate, as in the tree-min-delete challenge (Section 6).

A variety of existing work aims to reduce the annotation overhead associated with managing and rewriting predicate definitions with explicit `fold` and `unfold` operations. For example, Smallfoot [1] and Grasshopper [27] achieve concise specifications without user direction by building in specific support for list and tree predicates. Lee et al. [18] provide a static analysis capable of identifying when objects participate in many such data structures simultaneously. Nguyen and Chin [23] and Brotherston et al. [6] provide techniques for proving and applying user-supplied lemmas automatically. Chin et al. [7] provide support for a wider class of predicate definitions, including functional abstractions of data structures, provided that one reference parameter is traversed in the predicate's definition. Their entailment checker "carves out" a suitable portion of the input state, which (for one input state) is similar to the operation of our footprint computation algorithm.

These techniques improve the usability of recursive predicate reasoning, and can complement our work in a practical tool. Each comes with limitations: they cannot be applied equally to fully general predicates. One consequence of available magic wand support is that iterative code (such as our running example) can be specified without the need for extra predicate types to represent "partial" versions of data structures. These extra predicates do not describe structures which the program operates on, and are cumbersome to define for structures more complex than linked lists; loop invariants employing magic wands can be defined analogously for other data structures, and also support the specification of functional properties (e.g. the use of `sum_rec` in our example).

VeriFast [14] is a mature and expressive verifier for programs annotated with separation logic. We believe it is possible to partly work around the absence of magic wands using *lemma function pointers* and predicates. One can encode a wand $A \mathrel{-\!\!*} B$, using a predicate F (representing the wand's footprint), and a pointer to a lemma function with precondition

$F * A$ and postcondition B, whose body shows how to rewrite the state. The need to define the footprint manually, however, entails substantial additional overhead (to define a predicate for each footprint, and the appropriate lemma methods for manipulating them) for the user compared with our technique of automatic footprint computation.

As future work, we are interested to investigate other applications of our magic wand support, such as reasoning about *closures*, for which it is useful to be able to reason about connecting calls of closures together without knowing their specifications concretely. We are also interested in encoding existing by-hand proofs using our prototype implementation, e.g. parts of the proofs from [10, 9, 12]. The developers of the Viper verification tools also plan to incorporate our magic wand support into their tool infrastructure.

Acknowledgements. The authors were funded by the Hasler Foundation. We thank the anonymous referees for suggesting many improvements to the explanation of our work. We are also very grateful to Peter Müller, Ioannis T. Kassios, Uri Juhasz and John Boyland for extensive feedback on earlier versions of this paper.

References

1 Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.

2 Stefan Blom and Marieke Huisman. Witnessing the elimination of magic wands. Technical Report TR-CTIT-13-22, University of Twente, November 2013.

3 John Tang Boyland. Checking interference with fractional permissions. In *SAS*, 2003.

4 John Tang Boyland. Semantics of fractional permissions with nesting. *ACM Trans. Program. Lang. Syst.*, 32(6):22:1–22:33, August 2010.

5 Rémi Brochenin, Stéphane Demri, and Etienne Lozes. On the almighty wand. *Journal of Information and Computation*, 211:106–137, February 2012.

6 James Brotherston, Dino Distefano, and Rasmus Lerchedahl Petersen. Automated cyclic entailment proofs in separation logic. In *CADE'11*, pages 131–146, 2011.

7 Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1006–1036, 2012.

8 Dino Distefano and Matthew J. Parkinson. jStar: towards practical verification for java. In *OOPSLA*, pages 213–226, 2008.

9 Mike Dodds, Suresh Jagannathan, and Matthew J. Parkinson. Modular reasoning for deterministic parallelism. In *POPL*, pages 259–270, 2011.

10 Christian Haack and Clément Hurlin. Resource usage protocols for iterators. *Journal of Object Technology*, 8:55–83, 2009.

11 Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In *ECOOP*, pages 451–476, 2013.

12 Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. In *POPL'13*, pages 523–536. ACM, 2013.

13 Zhe Hou, Ranald Clouston, Rajeev Goré, and Alwen Tiu. Proof search for propositional abstract separation logics via labelled sequents. In *POPL*, pages 465–476, 2014.

14 Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical report, Katholieke Universiteit Leuven, August 2008.

15 Jonas Braband Jensen, Lars Birkedal, and Peter Sestoft. Modular verification of linked lists with views via separation logic. *Journal of Object Technology*, 10:2:1–20, 2011.

16 U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. Technical report, ETH Zurich, 2014.

17 Neelakantan R. Krishnaswami. Reasoning about iterators with separation logic. In *SVCBS*, pages 83–86, New York, NY, USA, 2006. ACM.

18 Oukseh Lee, Hongseok Yang, and Rasmus Petersen. Program analysis for overlaid data structures. In *CAV*, volume 6806 of *LNCS*, pages 592–608. Springer Berlin Heidelberg, 2011.

19 Wonyeol Lee and Sungwoo Park. A proof system for separation logic with magic wand. In *POPL*, pages 477–490, 2014.

20 K. Rustan M. Leino. This is Boogie 2. Available from `http://research.microsoft.com/en-us/um/people/leino/papers.html`.

21 K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In *ESOP*, pages 378–393, 2009.

22 Toshiyuki Maeda, Haruki Sato, and Akinori Yonezawa. Extended alias type system using separating implication. In *TLDI*, pages 29–42. ACM, 2011.

23 Huu Hai Nguyen and Wei-Ngan Chin. Enhancing program verification with lemmas. In *CAV*, volume 5123 of *LNCS*, pages 355–369. Springer Berlin Heidelberg, 2008.

24 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19, London, UK, 2001. Springer-Verlag.

25 M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM Press, 2005.

26 M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *In LMCS*, 8(3:01):1–54, 2012.

27 Ruzica Piskac, Thomas Wies, and Damien Zufferey. GRASShopper – complete heap verification with mixed specifications. In *TACAS*, pages 124–139. Springer, 2014.

28 William S. Retert. *Implementing Permission Analysis*. PhD thesis, University of Wisconsin at Milwaukee, Milwaukee, WI, USA, 2009.

29 Malte Schwerhoff and Alexander J. Summers. Implementation. Available from `www.pm.inf.ethz.ch/research/viper.html`.

30 Jan Smans. *Specification and Automatic Verification of Frame Properties for Java-like Programs*. PhD thesis, FWO-Vlaanderen, May 2009.

31 Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, pages 148–172, 2009.

32 Jan Smans, Bart Jacobs, and Frank Piessens. Heap-dependent expressions in separation logic. In *FMOODS/FORTE*, pages 170–185, 2010.

33 Alexander J. Summers and Sophia Drossopoulou. A formal semantics for isorecursive and equirecursive state abstractions. In *ECOOP*, pages 129–153, 2013.

34 Thomas Tuerk. Local reasoning about while-loops. In *VSTTE*, 2010.

35 Hongseok Yang. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In *SPACE*, 2001.