Witnessing the Elimination of Magic Wands

Stefan Blom and Marieke Huisman
University of Twente

November 8, 2013

Abstract. This paper discusses the use and verification of magic wands. Magic wands are used to specify incomplete resources in separation logic, i.e., if missing resources are provided, a magic wand allows one to exchange these for the completed resources. We show how the magic wand operator is suitable to describe loop invariants for algorithms that traverse a data structure, such as the imperative version of the tree delete problem (Challenge 3 from the VerifyThis@FM2012 Program Verification Competition).

Most separation-logic-based verification tools do not provide support for magic wands, possibly because validity of formulas containing the magic wand is, by itself, undecidable. To avoid this problem, in our approach the program annotator has to provide a witness for the magic wand, thus circumventing undecidability due to the use of magic wands. We show how this witness information is used to encode a specification with magic wands as a specification without magic wands. Concretely this approach is used in the VerCors tool set: annotated Java programs are encoded as Chalice programs. Chalice then further translates the program to BoogiePL, where appropriate proof obligations are generated. Besides our encoding of magic wands, we also discuss the encoding of other aspects of annotated Java programs into Chalice, and in particular, the encoding of abstract predicates with permission parameters.

We illustrate our approach on the tree delete algorithm, and on the verification of an iterator of a linked list.

1 Introduction

Verification of sequential programs with pointers has significantly profited from the advance of separation logic. Separation logic is an extension of classical Hoare logic that explicitly considers the heap \[39\], which makes it highly suitable to reason about pointer structures, the permission to access a heap location, and (absence of) aliases.

In classical Hoare logic \[18\], a program is extended with annotations that express properties about the program’s intermediate state. Separation logic is an extension of Hoare logic that explicitly separates the program state into the heap and the store. Characteristic for separation logic is that annotations can also express properties about resources, where the most fundamental kind of resource is access permission to a part of the heap, i.e., to read or write a location on the heap. Formulas about access permissions can be combined into larger formulas using the separating conjunction operator \(\&\). A key feature of separation logic is that in a formula \(\phi_1 \& \phi_2\), the formula is only valid for a heap, if \(\phi_1\) and \(\phi_2\) are valid for disjoint parts of the heap.

**Magic Wands** A less common feature of separation logic is the magic wand operator, also known as the separating implication, usually written \(\leftarrow\). A formula \(\phi_1 \leftarrow \phi_2\) holds for a heap \(h\) if whenever \(h\) is extended with a heap \(h_1\) that satisfies \(\phi_1\), then the combined heap satisfies \(\phi_2\). In the literature, this operator is often described as a trading operator: the resources associated to \(\phi_1\) are traded for the resources associated to \(\phi_2\). Another way to think about it is to see the magic wand operator as a promise: it denotes a special kind of resource that provides the ability to exchange one set of resources (the required resources) to a different (possibly larger) set of resources (the ensured resources).

This paper shows how the magic wand operator can be used to elegantly specify loop invariants for iterative algorithms that explore data structures: knowledge about the current location in the data structure can potentially be exchanged for knowledge about the complete
data structure explored so far. We use this approach to specify a loop invariant for Challenge 3 from the VerifyThis@FM2012 Program Verification Competition [22]. This challenge is to verify an iterative tree delete algorithm: the removal of the element with the least key in a binary search tree. In the literature, also several other examples that illustrate the usefulness of the magic wand operator can be found, e.g., to specify an iterator protocol (see [17, 22]) (discussed in Section 6.2), to reason about sharing in data structures [19], and to specify several common object-oriented design patterns [24].

In addition, we also discuss how tool support to reason about magic wands is developed. There are several tools that allow reasoning about programs annotated using separation logic, such as VeriFast, SmallFoot, and jStar [24, 5, 12]. However, none of these tools support the magic wand operator. Only the notion of lemma in VeriFast [26] resembles a magic wand partially, but it cannot be used to exchange knowledge about permissions in the same way as magic wands (see Section 7.1 for more information).

We believe that the main reason that most separation logic-based program verification tools do not support reasoning about magic wands, is that deciding the validity of a formula containing magic wands is almost always undecidable [8]. To overcome this, this paper provides the means to associate a proof term with an assertion introducing a magic wand formula, by letting the user describe how to construct proofs. The proof term is then used to help the verifier to establish the correctness of the implementation w.r.t. the specification. Concretely, a proof annotation syntax is defined that allows one to specify how to build a proof term for a magic wand. Support for this proof annotation syntax has been integrated into our VerCors tool set, a tool set for the verification of concurrent Java programs.

In order to actually construct the term, we introduce the notion of a witness object to encode the magic wand formula. The witness object stores the resources that are needed to exchange the required resources of the magic wand into the ensured resources. Moreover, it also contains a description how to perform the conversion. This witness object is generated based on the proof term provided by the user. It contains several methods that model the logic rules for this particular magic wand formula.

Our solution to the verification problem of the magic wand has been inspired by the Curry-Howard isomorphism [20], which turns a verification problem into a type checking problem by constructing a proof term. This intuition is further emphasised by the way that we write annotations: formulas are typically manipulated using logical rules, while witnesses are manipulated by methods defined on them. Thus, the encoding of magic wand formulas in this paper transforms the program verification problem into the programmatic manipulation of specification-only (or ghost) objects. We believe that this approach is attractive for software engineers that have a more imperative way of thinking about program behaviour, while they struggle with logical manipulation of complex formulas.

**Predicates** A commonly used extension of separation logic is the use of abstract predicates [34]. They provide abstraction and allow one to reason about larger structures, and recursive definitions. Typical use of an abstract predicate is to encapsulate the state of an object and to hide the internal implementation details from the specification. Abstract predicates can be parameterised by parameters, which can be program variables or permissions. The latter is useful, for example to specify different access permissions to different parts of a data structure. Abstract predicates behave as resources in their own right. In the tree delete challenge, we use abstract predicates to capture for example the state of the tree.

To be able to reason about abstract predicates with arbitrary parameters (including permissions), there is ready-available tool support, but using a different style of specifications. In particular, VeriFast supports predicates with arguments, but it does not support reasoning about resources and functional properties separately in order to reuse existing functional specifications. This disagrees with the philosophy of VeriFast. Jost and Summers present an approach to reason about predicates that can be split into a resource and a functional predicate [26]. However, there are many predicates for which this approach does not work, for example when the predicate specifies different access permissions to different parts of a data structure. Therefore, in this paper we also propose an encoding of abstract predicates, using the same idea of constructing a witness object. Concretely, a predicate with parameters is encoded as a witness object with fields to encode the parameters. This allows replacing the original predicate by a predicate over the newly defined object that refers to the object’s fields, where the original predicate referred to its parameters. Thus, again we trade complexity on the level of logical formulas for complexity on the level of specification-only code.

**Approach.** Concretely, in this paper, we consider specifications in permission-based separation logic using the magic wand operator. Permission-based separation logic is an extension of standard separation logic, where fractional permissions [7] are used to distinguish read and write access permissions to a location on the heap [6, 10], which makes it suitable to reason about concurrent programs.

Permission-based separation logic is used as a specification language for the VerCors tool set [1], which targets the verification of concurrent Java programs. It leverages existing verification tools, and in particular it encodes annotated Java programs into Chalice pro-
grams [29]. Chalice is another verification tool for a concurrent programming language, but working on a more idealised programming language (without inheritance), based on the theory of implicit dynamic frames [40]. This is an alternative approach to reason explicitly about pointers in the heap, which is equivalent to separation logic [37].

Notice that all examples in this paper are sequential. Therefore, in most examples all points-to predicates are decorated with write permissions only.

We use our encoding to verify the tree delete challenge. Concretely, we show that the data structure is treated properly, i.e., only locations for which appropriate permissions are held are being accessed. Moreover, we show that the list of elements represented by the result is the tail of the list represented by the input. In addition, we also illustrate our approach on a linked list with an iterator (with remove).

Overview The remainder of this paper is organised as follows. First, we provide an introduction to separation logic, and how this is supported in Chalice and the VerCors tool set. Section 3 presents the tree delete challenge of the VerifyThis competition and discusses an intuitive solution for the challenge that uses a magic wand. Section 4 focuses on the encoding of predicates with parameters. We continue in Section 5 with the elimination of the magic wand. Then in Section 6, we present machine-checked versions of the challenge and of an additional example, namely an iterator protocol. Finally, we conclude and discuss related and future work.

2 Background

This section first introduces permission-based separation logic. Then it introduce Chalice as a tool that can check a subset of the logic and finally it explains the basics of the VerCors tool set, and the syntax for permission-based separation logic in the tool set.

2.1 Permission-based Separation Logic

As mentioned above, separation logic [39] was originally developed as an extension of Hoare logic [18] to reason about programs with pointers, as it allows to reason explicitly about the heap. In classical Hoare logic, assertions are properties over the state, while in separation logic, the state is explicitly divided in the heap and a store, related to the current method call. Parkinson adapted separation logic for Java [34], and this is the variation of the logic that we will use.

Separation logic is also suited to reason modularly about concurrent programs [39]: two threads that operate on disjoint parts of the heap, do not interfere, and thus can be verified in isolation. However, classical concurrent separation logic requires use of mutual exclusion mechanisms for all shared locations, and it forbids simultaneous reads to shared locations. To overcome this, Bornat et al. [6] extended separation logic with fractional permissions. Permissions, originally introduced by Boyland [7], denote access rights to a shared location. A full permission 1 denotes a write permission, whereas any fraction in the interval (0, 1) denotes a read permission. Permissions can be split and combined, thus a write permission can be split into multiple read permissions, and sufficient read permissions can be merged into a write permission. The use of permissions makes permission-based separation logic suitable to prove data race freedom of multithreaded programs using different synchronisation mechanisms.

Most variants of separation logic strictly distinguish logical values in specifications from the contents of locations on the program heap. This provides a more elegant theory, but allows less freedom when writing specifications. Therefore, we choose to use the more liberal style exemplified by Total heap Permission Logic [35].

Formally, we define the set of formulas in this resource logic as follows:

\[ R := b | \kappa(\overline{e}) | \text{Perm}(e,f,S) | R \ast R \]
\[ | R \rightarrow R | b \Rightarrow R | * R(\alpha) \]
\[ S \in (0,1], \kappa \in \text{Pred} \]

where \( b \) is a first-order logic formula, \( \kappa \) is a predicate in set \( \text{Pred} \), \( e \) denotes an expression (with \( \overline{e} \) for a vector of expressions), \( \alpha \) is a logical variable with type \( T \), and \( \pi \) is a fractional permission, i.e., a value in the interval \( (0,1] \). Expressions can be qualified expressions of the form \( e.f \), where \( f \) is a field name. Below we will also give the grammar of the permission-based separation logic, as supported by the VerCors tool set. All examples will be given following the VerCors grammar. However, in the text, we will sometimes also use the more classical, mathematical notation.

Validity of this logic is defined over resources \( \mathcal{R} \), which consist of a heap and a partial permission table for the locations in this heap. The permission predicate \text{Perm}(x,f,\pi)\) expresses that the field \( x.f \) points to a location on the heap and the current thread holds a permission \( \pi \) on this location. If a thread holds permission 1, this denotes a write permission; any permission less than 1 denotes a read permission. Permissions can be split and combined (by division and addition). When reasoning about concurrent programs, permissions can be transferred between threads, and they can be stored as resource invariants into locks. A global correctness property of the logic is that the total number of permissions to access a certain location never exceeds 1, which ensures that any program that can be verified is free of data races.

The separating conjunction operator \( \ast \) expresses that a formula \( \phi_1 \ast \phi_2 \) holds for a resource \( \mathcal{R} \) if the resource can be split into two disjoint resources \( \mathcal{R}_1 \) and \( \mathcal{R}_2 \), such
that $\phi_1$ holds for $R_1$, and $\phi_2$ holds for $R_2$. Separating conjunction can be lifted over a set of formulas, ranged over by the logical variable $\alpha : T$, using the universal separating conjunction, written $\forall R(\alpha)$.

The magic wand operator $\rightarrow$ is formally defined as follows. A resource $R$ satisfies a formula $\phi_1 \rightarrow \phi_2$ if for every disjoint resource $R'$ satisfying the property $\phi_1$, the combination of $R$ and $R'$ satisfies $\phi_2$. In other words: any information or permissions described by the formula $\phi_1$ can be exchanged for the formula $\phi_2$. It is important to note that by obtaining $\phi_2$, one has to give up $\phi_1$, thus linear reasoning is used here. We mimic this in our encoding of magic wands. When creating a witness object for a magic wand formula, we store permissions inside the witness object, which can be retrieved in exchange for the permissions or the formula required by the magic wand.

Finally, a predicate $\kappa$ is a, possibly recursive, definition of a separation logic formula:

$$\forall e' : \kappa(\overline{e'}) = \phi(\overline{e'})$$

Every predicate must be well-defined, so termination of the definition must be guaranteed. An abstract predicate is a predicate whose definition is not available to the client of a class. This is useful to encapsulate details about the state that should not be used by the client. It is also conveniently used for axiomatizing the behaviour of low-level data structures and operations, such as volatile variables, locks and thread operations. By declaring a predicate abstract, the client is prevented from having access to internal details, these are only visible to the instances of the class itself. This feature is used in the iterator example (see Section 6.2) to enforce the calling convention for iterators: only by following the protocol can the client get access to the predicates that are required for the next call. For the verification of a method, abstract predicates that are within the scope of the receiver object can be opened and closed.

An important restriction on formulas is that they must be self-framed, which means that the set of locations that must be read to evaluate the formula is a subset of the set of locations for which the formula specifies at least read access.

Finally we introduce a derived notation: when we want to express both permission and value at the same time, we use the PointsTo predicate that is used in classical separation logic:

$$\text{PointsTo}(x, f, \pi, v) \stackrel{\text{def}}{=} (\text{Perm}(x, f, \pi) \land x.f = v)$$

Notice that this relation between Perms and PointsTo is formally established by Parkinson and Summers [35].

2.2 Chalice

Chalice is a tool for the verification of concurrent programs [29], written in a simple object-oriented language with built-in specification features. The VerCors tool set uses Chalice as an intermediate language to encode annotated Java programs. Before providing more details about the VerCors tool set, we first briefly introduce Chalice.

Supported features of the Chalice language are basic classes (no static members or inheritance), fields and three kinds of ‘methods’:

- standard methods, which can be used in execution;
- functions, i.e., to evaluate a property about the state, which do not use any access permissions and can therefore be used both during execution and in specifications;
- predicates, which can contain access permissions and can only be used in specifications.

The specification language of Chalice supports field permissions in the same way as the logic in the previous section, albeit with a different syntax (acc, instead of Perm, and $\&\&$ is used as a connective for permissions, instead of $\ast$). Standard boolean expressions and functions can be used in specifications. Additionally, Chalice has support for (recursive) predicates, however these predicates cannot have explicit parameters, i.e., they are limited to the implicit parameter $\text{this}$. Both functions and predicate definition should terminate.

The Chalice tool verifies annotated code by generating an annotated Boogie program [3], for which the Boogie verifier subsequently will generate first-order logic proof obligations. Standard methods and functions map into Boogie naturally and can make full use of all of the automatic verification features of Boogie. However, as Boogie is not natively aware of permissions, these are encoded into Boogie’s first-order logic annotations, and therefore reasoning about them is less automatic. Similarly, for predicates reasoning is less automatic, and the user has to tell the verifier explicitly to replace a predicate by its definition (unfold) or introduce a predicate in place of its definition (fold).

As a consequence, in this paper we write explicitly when to fold and unfold predicates, while we do not have to do this when reasoning about functions, as these definitions are opened and closed automatically by the Boogie verifier. Instead, to ensure that the verification process succeeds, we only sometimes have to add a (provable) property about a function definition.

2.3 The VerCors Tool Set

The target of the VerCors tool set is the verification of multi-threaded object-oriented programs, in particular of annotated Java code. Rather than developing yet
The tool considers two classes of expressions: resource expressions \( (R, \text{typical element } r_i) \) and logical expressions \( (E, \text{typical elements } e_i) \). An important subset of those are the logical expressions of type boolean \( (B, \text{typical elements } b_i) \). Using those classes, the syntax for the formulas that is supported by the tool is defined as follows:

\[
R ::= b \mid \text{Perm}(c.f, \text{frac}) \mid (\forall \text{forall} \ T \ v; b; r) \\
| r_1 \ast r_2 \mid r_1 \ast r_2 \mid b_1 => r_2 \\
| e.P(c_1, \ldots, c_n) \mid S(e_1, \ldots, e_m)
\]

\[
E ::= \text{any pure expression} \\
B ::= \text{any pure expression of type boolean} \\
\mid (\forall \text{forall} \ T \ v; b_1; b_2) \\
\mid (\exists \text{exists} \ T \ v; b_1; b_2)
\]

where \( T \) is an arbitrary type, \( f \) a field name, \( \text{frac} \) a fraction of a permission, \( v \) a variable name, \( P \) a (dynamic) predicate and \( S \) a static predicate. Notice the correspondence with the mathematical formulation of resource logic in Section [27].

Note that we include both static predicates, which are defined independently of object instances, and dynamic predicates, whose definition is tied in with an instance of an object (passed as an implicit argument).

The notation for implication \( => \) is borrowed from JML, as are the notations for universal and existential quantification (using keywords \( \forall \text{forall} \) and \( \exists \text{exists} \), respectively). VerCors uses \( (\forall \text{forall} \ T \ v; b; r) \) to define the universal separating conjunction.

In addition to Java’s standard primitive types, the specification language has two additional primitive types: resource and frac, to type permission and fraction expressions, respectively. As in Chalice, the domain of frac is a value between 1 and 100, where 100 means a full write permission and any value less than 100 denotes a read-only permission. This restriction is made because we use Chalice as a back-end, not because the encoding requires it. In principle the techniques described in this paper work over any separation algebra [10].

Note that the syntax for resource formulas does not allow disjunction or negation, except for the special case of a boolean expression acting as a guard for a resource formula. This restriction does not apply to the boolean expressions, which can be used as part of a resource formula.

It may seem that this contradicts our earlier claim of extending JML, where disjunction is used heavily in order to choose between possible contracts. However, this does not introduce a real contradiction, as any method with multiple contracts can be respecified with a single contract, using an additional ghost parameter.

Like in JML, all comments that start with a \( \# \) are part of the specification. This holds for both single line (\( \# @ \ldots \)) and multiple-line (\( \# @ \ldots \)) comments.

Predicates are written in a simple functional syntax, preceded by the keyword resource. For example, a pred-
icate p providing write access to the field f is written as follows:

```java
/*@ resource p()=Perm(f,100); */
```

To indicate that a method is pure, as in JML, we add a pure modifier to the method declaration.

```java
/*@ pure */ boolean m()
```

In method contracts, we will often employ ghost parameters and ghost return values. These are declared by given and yields clauses that precede the method declaration. For example, an integer ghost parameter x and a boolean ghost return value b are specified as:

```java
given int x;
yields boolean b;
```

Implicitly, a method contract is universally quantified by the variables in the given clause, and existentially quantified by the variables in the yields clause.

For specification convenience, the tool also provides a polymorphic list or sequence type seq<T>, where T can be any type (not necessarily a class). This type translates directly to the Chalice type of the same name. The syntax for a constant list borrows from the syntax for a polymorphic list or sequence type.

Several standard operations on sequences are available. Given sequences s,t, we have:

- concatenation: s + t;
- first element: head(s);
- other elements: tail(s); and
- length: s.length.

Further, we provide syntactic sugar for the following common specification pattern:

```java
(e != null) ==> e.pred(e1, · · · ,en)
```

which we abbreviate as:

```java
e->pred(e1, · · · ,en)
```

Finally, the tool implements inheritance using the theory of abstract predicates and inheritance by Bierman and Parkinson [36], but cannot use the full power of that theory because specifications are restricted to monotone predicate families, as introduced by Haack and Hurlin [15,23]. We will not discuss the details of inheritance, because those are not needed for this paper.

In the examples below, whenever necessary we will explain more details of the specification syntax.

### 3 The Tree Delete Challenge

As a motivating example for our work, we use the iterative removal of the element with the least key from a binary search tree, i.e., Challenge 3 from the VerifyThis@FM2012 Program Verification Competition [22]:

Given: a pointer t to the root of a non-empty binary search tree (not necessarily balanced). Verify that the following procedure [in Fig. 3] removes the node with the minimal key from the tree. After removal, the data structure should again be a binary search tree.

Input for the tree delete algorithm is a binary search tree, in our case defined by the following (recursive) class definition:

```java
public class Tree {
  public int data;
  public Tree left;
  public Tree right;
  //...
}
```

The goal of the algorithm is to delete the element with the smallest key, i.e., the left-most node from the tree, and the challenge is to prove that the resulting tree remains a binary search tree.

To provide a specification for the algorithm, we first define:

- the predicate state, representing permissions to the field locations making up the tree;
- the pure function contents, capturing the list of integers stored in the tree; and
- the predicate state.contents, expressing the permissions and the stored values simultaneously.

```java
resource state() =
  Perm(data,100) **
  Perm(left,100) ** left ->state() **
  Perm(right,100) ** right ->state();
```

```java
requires t!=null ==> t.state();

pure seq<int> contents(Tree t){
  if(t == null){
    return seq<int>();
  } else {
    unfold t.state();
    return contents(t.left) + seq<int>(t.data) + contents(t.right);
  }
}
```

```java
resource state.contents(seq<int> L) =
  state() ** contents(this) == L;
```

The state predicate defines the permissions on the tree. If one holds the state predicate, one has write permission on the fields data, left and right, and recursively also on the subtrees pointed to by left and right, provided they are not null. The pure method contains defines the
The actual challenge description, is much more involved. One needs to specify an appropriate loop invariant that retains all permissions on the entire tree data structure during the iterations that compute the left-most node in the tree. The invariant must be written in such a way that the deletion of the left-most node afterwards is allowed and that the permissions on the whole tree can be recovered.

The core of the problem is the treatment of permissions, which are given in the form of a tree. In each iteration the focus on the tree (i.e. the variable current) is shifted by one step. However, once you have reached the left-most node, you want to move back the focus to the root of the tree, i.e., after the loop has finished, the method should continue with access to the root of the tree. The magic wand is highly suited to handle this: all the permissions on the traversed path are stored “inside” the magic wand, and by giving up the focus on the current node, focus on the root can be retrieved. Notice that our solution with the magic wand is general: it cannot only be used to specify the shift of focus to the root of the tree, but also to other nodes in the tree.

Fig. 3 contains the iterative implementation of the tree delete algorithm, with the key loop invariants necessary to verify this method.\footnote{For clarity of presentation, we have left out one bookkeeping invariant and a lot of proof script.} Note that rather than using the competition version verbatim, we have eliminated a superfluous variable and renamed all variables in order to make the code more understandable. Also note that while our ultimate goal is to further develop the tool set in such a way that this code can be verified essentially as it is written in Fig. 3, the current version of our tool set needs a lot more annotations. The fully annotated version, which can actually be verified with the VerCors tool set, will be given later in Section 6.1.

The variables in the algorithm denote the following:

- top is the pointer to the root of the complete tree;
- cur is the currently explored node; and
- left is the left subtree of cur.

The loop invariant expresses the following:

- the thread holds permissions for the currently explored node (state predicate);
- the currently explored node is the root of a tree (state predicate);
- the relationship between cur and left is as described above; and
- a promise (by means of a magic wand) that if cur is modified to represent a tree with the left-most element removed, and if the thread holds access permissions to this tree, then this can be exchanged to access permissions on a larger tree, which also has the left-most element removed compared to the tree at the start of the procedure.

Using these specifications, the method del_min, which implements the deletion of the element with the minimal key is specified as follows.

```java
VerCors

public Tree del_min(Tree t) {
    // unfold t.state();
    if (t.left == null) {
        // assert contents(t.left)
        return t.right;
    } else {
        t.left = del_min(t.left);
        // fold t.state();
        return t;
    }
}

Fig. 2. The recursive implementation of tree delete
```

In contrast to the recursive version, the verification of the iterative version of this algorithm, as requested by contents of a tree to be the contents of the nodes’ data fields, read from left to right.

Using these specifications, the method del_min, which implements the deletion of the element with the minimal key is specified as follows.

```java
/**
 * requires t!=null && t.state();
 * ensures result->state();
 * ensures contents(result)==
 * tail(old(contents(t)));
 * @*
 * public Tree del_min(Tree t);
 *
 * This contract states that the algorithm removes the first element of the list that is represented by a tree. As the element with the minimal key in a binary search tree is the first element of the list representation of the tree, this contract is sufficient to meet the challenge. The permissions used are full write permissions, which implies that the underlying linked data structure is tree-shaped, and it cannot be a DAG or contain cycles. The contents of the result are the contents of the input, minus the first element, which implies that the result tree is again a binary search tree. Note that the result is even slightly stronger than required. According to the challenge it would have been acceptable to require that the contents are ordered, but due to the way in which the contract is written, this information is not needed.

Figure 2 contains a recursive implementation of this algorithm. It is easy to see, and to verify, that this implementation respects the specification of del_min. To illustrate this, we have decorated this recursive implementation with the annotations that are needed for the VerCors tool to verify that this implementation respects the specification above: essentially all that is needed are opening of the state predicate at the beginning of the method, closing of the state predicate at the end of the method body, and an explicit assertion that if t.left is null then the contents of t.left are the empty list.

In contrast to the recursive version, the verification of the iterative version of this algorithm, as requested by
```
4 The Encoding of Predicates

This section shows how predicates with explicit parameters, different from the implicit this parameter, are encoded in Chalice. Below, we first discuss how some predicates can be split in a data part and a resource part. However, there are many predicates for which this approach does not work. Therefore, we propose an alternative approach, encoding a predicate with parameters as a witness object. We describe this approach first for simple predicates, and then for recursive predicates. Finally, we describe the encoding algorithm.

4.1 Splittable Predicates

To provide the intuition behind our approach, we first discuss the most simple case, where it is easy to separate a predicate into a permission and a data part.

For example, suppose we have a predicate on a linked list expressing that one has write access to the list and that the list contains a given sequence of elements. A common way to define this in separation logic is as follows:

```cpp
class List {
    int val;
    List next;
}

resource list(List l, seq<int> L) =
    (l==null)?(L==seq<int>):
    (Perm(l.val,100) ** Perm(l.next,100))
    ** L.length>0 ** l.val==head(L)
    ** list(l.next,tail(L));
}
```

Obviously, since Chalice does not support predicates with parameters, this cannot be directly encoded in Chalice. However, it is straightforward to see that this recursive predicate can be split into a permission part, a value part and a non-recursive predicate, each with the same control flow. Notice that non-recursive predicates can always be inlined, so they do not provide any difficulties for verification, however they are very useful for clarity of specifications.

```cpp
resource state() =
    Perm(val,100) ** Perm(next,100) ** next->state();
requires l->state();

pure boolean contains(List l, seq<int> L) =
    (l==null)?(L==seq<int>):(L.length>0 &&
    l.val==head(L) && contains(l.next,tail(L));
}

resource list(List l, seq<int> L) =
    l->state() ** contains(l,L);
```

Note that after splitting, the resource part of the original list predicate is written as a recursive predicate state that no longer has a parameter, and thus it can be encoded directly in Chalice. The data part of the original list predicate is captured by the recursive pure method contains, which still has a parameter. However, this is no problem, as this can be encoded directly as a Chalice function. Finally, there is a non-recursive predicate list, which has parameters, but those are easy to eliminate in the encoding: simply inline the definition throughout the specification.

4.2 Predicate Witnesses

It is not always possible to split a predicate into a permission part without parameters and a data part. For
example, suppose you wish to define a predicate that captures that you have fraction \( p \) to access the elements stored in the list (via the \( \text{val} \) pointer), and fraction \( q \) to traverse the \( \text{next} \) pointer to the next element in the list, as is done in Figure 4. For this predicate, the split as described above is impossible. To get around this problem, we introduce the notion of a witness object, containing a valid predicate, which holds for the witness object if and only if the original predicate holds.

4.2.1 Witnesses for Non-Recursive Predicates

To describe how this witness object is constructed and reasoned about, we first consider the encoding of a very simple predicate, whose body is just true. Class \( \text{Twice} \) in Figure 5 defines such a predicate, called \( \text{state} \).

Fig. 5 shows the definition of the witness object in Chalice that encodes the \( \text{state} \) predicate declared in class \( \text{Twice} \). The witness object is an instance of class \( \text{Twice} \_\text{state} \). This class definition is generated by the VerCors tool set. The class has two fields: \( \text{ref} \) refers to the object where the original predicate is defined, and \( p \) holds the value of the parameter \( p \). Further, it defines a predicate \( \text{valid} \) that encodes the original \( \text{state} \) predicate, but using the fields of the witness object, instead of the original predicate’s parameters. In addition, the class contains a function \( \text{check} \) that relates the predicate parameters in the original specification to the fields of the witness object, and requires them to be the same. If \( o \) is the witness object for a predicate \( \text{state} \), then an assertion \( o.\text{valid}(p) \) becomes essentially \( o.\text{valid}() \times o.\text{check}(this,p) \) in the encoding (see for example Lines 4, 5 and 10, 11 in Fig. 5).

To complete the description of the class \( \text{Twice} \_\text{state} \), it also defines getter methods for all fields in the class.

Using the witness object of the predicate, we wish to show that class \( \text{Twice} \) in Figure 5 is correct. For a human, is easy to see that the body of \( \text{twice} \) satisfies its contract: the pre- and postconditions of \( m \) and \( \text{twice} \) are all the same, and the calls to \( m \) thus can be put in sequence.

To ensure that the tool can establish the correctness of \( \text{twice} \), we need to decorate it with some additional proof annotations, as shown in Fig. 7. First of all, every usage of predicate \( \text{state} \) has been prefixed with a
4.2.2 Witnesses for Recursive Predicates

When a predicate is recursively defined, such as the state predicate on linked lists in Fig. 7, the witness encoding does not result in a single object, but in a tree of witness objects. For every recursive invocation of the predicate in the predicate definition, the witness contains a field that refers to the witness that provides the evidence for the recursive call. Thus, the witness for a predicate being valid on an object is actually a tree of witness objects, whose structure matches the calling structure of the evaluation of the original predicate. For example, in Fig. 7, the linked list of witnesses at the top reflects that the definition of \( \text{rec.state}(p,q) \) applied to the list of three elements at the bottom makes two recursive calls. The definition of the witness object is given in Fig. 8, where the field \( \text{rec} \) refers to the witness object for the recursive call of the predicate. Note further how the conditional part of the valid predicate in lines 17-19 matches the conditional invocation \( \text{rec.next} \rightarrow \text{state}(p,q) \) in the original predicate definition.

4.3 Recipe for the Encoding

As presented above, the encoding into Chalice generates a witness class for every predicate definition, replaces predicate invocations in logical statements by validity checks on witnesses, and adds getter methods for use by witness classes as well as variables that are needed to store witness objects.

The complete recipe for the encoding is as follows:

1. Every predicate definition

   \[
   \text{resource pred(type1 arg1,...,typeN argN) = body ;}
   \]

   declared inside class \( \text{Class} \), gives rise to the declaration of a sibling of \( \text{Class} \), called \( \text{Class.pred} \), containing:

   - a field \( \text{Class.ref} \), to refer to the object for which the validity of the original predicate is encoded;
The general idea behind our encoding is as follows: each magic wand formula is encoded as an instance of class Wand. Assuming that we have a type that can represent formulas, the formulas describing the required and ensured resources of the magic wand are fields of this class. In addition, the class contains a description of how the required resources can be exchanged for the ensured resources (encoded in the method \texttt{applying} and of the extra resources needed to do so. These extra resources are inhaled by the constructor, and not returned. Instead they are stored inside the magic wand object (in the extra field), and folded into the valid predicate. The executing thread cannot directly access these extra resources anymore, the only way to retrieve them is by using the \texttt{apply} method. The specification of the \texttt{apply} method requires the required formula of the magic wand and ensures the ensured formula, while its body is the exchange description. In addition, the \texttt{apply} method also requires the valid predicate, for two reasons: first, this prevents the \texttt{apply} method from being called more than once; and second, this allows the \texttt{apply} method to retrieve the extra resources needed for the conversion. Correctness of the \texttt{apply} method w.r.t. its specification ensures that the resources stored in the wand, together with the required

\footnote{We use \texttt{applying} as the method name because we are going to use \texttt{apply} as a keyword.}

5 The Encoding of Magic Wands

Now that we have seen how predicates with parameters are encoded in Chalice using witness objects, we discuss how magic wands are encoded. The syntax that was used above, labeling calls and uses of predicates, and providing explicit instantiations for the ghost arguments, will also be used in this section. Additionally, we introduce new syntax to create a magic wand and to apply it. The creation of a magic wand requires the user to provide a proof script. The proof script language will be discussed below.

5.1 General Idea

The general idea behind our encoding is as follows: each magic wand formula is encoded as an instance of class Wand. Assuming that we have a type that can represent formulas, the formulas describing the required and ensured resources of the magic wand are fields of this class. In addition, the class contains a description of how the required resources can be exchanged for the ensured resources (encoded in the method \texttt{applying} and of the extra resources needed to do so. These extra resources are inhaled by the constructor, and not returned. Instead they are stored inside the magic wand object (in the extra field), and folded into the valid predicate. The executing thread cannot directly access these extra resources anymore, the only way to retrieve them is by using the \texttt{apply} method. The specification of the \texttt{apply} method requires the required formula of the magic wand and ensures the ensured formula, while its body is the exchange description. In addition, the \texttt{apply} method also requires the valid predicate, for two reasons: first, this prevents the \texttt{apply} method from being called more than once; and second, this allows the \texttt{apply} method to retrieve the extra resources needed for the conversion. Correctness of the \texttt{apply} method w.r.t. its specification ensures that the resources stored in the wand, together with the required

\footnote{We use \texttt{applying} as the method name because we are going to use \texttt{apply} as a keyword.}

5 The Encoding of Magic Wands

Now that we have seen how predicates with parameters are encoded in Chalice using witness objects, we discuss how magic wands are encoded. The syntax that was used above, labeling calls and uses of predicates, and providing explicit instantiations for the ghost arguments, will also be used in this section. Additionally, we introduce new syntax to create a magic wand and to apply it. The creation of a magic wand requires the user to provide a proof script. The proof script language will be discussed below.

5.1 General Idea

The general idea behind our encoding is as follows: each magic wand formula is encoded as an instance of class Wand. Assuming that we have a type that can represent formulas, the formulas describing the required and ensured resources of the magic wand are fields of this class. In addition, the class contains a description of how the required resources can be exchanged for the ensured resources (encoded in the method \texttt{applying} and of the extra resources needed to do so. These extra resources are inhaled by the constructor, and not returned. Instead they are stored inside the magic wand object (in the extra field), and folded into the valid predicate. The executing thread cannot directly access these extra resources anymore, the only way to retrieve them is by using the \texttt{apply} method. The specification of the \texttt{apply} method requires the required formula of the magic wand and ensures the ensured formula, while its body is the exchange description. In addition, the \texttt{apply} method also requires the valid predicate, for two reasons: first, this prevents the \texttt{apply} method from being called more than once; and second, this allows the \texttt{apply} method to retrieve the extra resources needed for the conversion. Correctness of the \texttt{apply} method w.r.t. its specification ensures that the resources stored in the wand, together with the required

\footnote{We use \texttt{applying} as the method name because we are going to use \texttt{apply} as a keyword.}
resources are sufficient to establish the ensured resources of the magic wand. Figure 11 shows the definition of this wand class.

Notice that this is not a valid encoding, as it uses types such as Formula and Proof, which are not supported. Therefore, below we discuss how the ideas of this idealised encoding can be realised by a correct Chalice class.

5.2 Encoding of Magic Wands in Chalice

This section discusses how the idea described above is used to generate a specific class for each type of wand formula that is used. Moreover, the proof script cannot be passed as a parameter, instead we encode it by an identifier, and generate a body of apply that selects the appropriate proof script, depending on the value of the identifier.

Consider for example the WandDemo class in Fig. 12.

Every WandDemo object can be in two states: read mode and write mode. When an instance is created, it is created in write mode. In write mode, the only method that can be called is set. After setting, the object is in read mode. In read mode, the only method that can be called is get, and the object stays in read mode. However, it is always possible to reset the object to a different value.
To do so, write mode must be reestablished. Hence, the 
set method ensures not just the predicate readonly(), but 
in addition also the magic wand readonly() -- writeonly(), 
which can be applied to enable writing. Figure 12 also 
defines a method demo to illustrate how the magic wand is 
used.

The code is mostly self-explanatory, but three elements are worth noting:

- Every magic wand is given a label, which is used to be able to identify which magic wand is addressed. In this example, labels recover and wand are used.
- The syntax for creating a magic wand is

```java
create {
    proof script
    qed wand formula;
}
```

To write the proof script, the usual proof hints (fold/unfold/etc.) are allowed, and additionally two new statements are introduced: use R, and qed R: The use statement asserts that formula R holds at the time the magic wand is created and stores the resources represented by this formula inside the magic wand. For example, the proof in lines 27-32 stores 75% of the permission on the field x by means of the statement use Perm(this.x,75). The qed statement ends the proof script for the magic wand.

- In the demo method, the witness keyword in line 37 is used to indicate that the label wand refers to a witness for the corresponding magic wand formula.

Our encoding declares a class for the witness object, called Wandreadonly_for_writeonly, as shown in Fig. 13 to represent the magic wand. Moreover, it rewrites the WandDemo class to replace magic wand formulas by manipulations of the witness object (see Fig. 14).

Instances of Wandreadonly_for_writeonly are used as witnesses to the validity of all magic wands that match e1.readonly() -- e2.writeonly() for arbitrary expressions e1 and e2. Therefore, it is necessary to indicate which proof is to be used by the apply method, by setting the lemma field to the correct value. The results of evaluating the expressions at the time when the wand was created is stored in the fields in_1 and out_1, which have matching getter methods. Variables that are used in the proof are also stored in the object. In this case the value of this for lemma 1 is the only value needed and it is stored in the field this_1.

The contract of the apply method essentially requires the wand to be valid, and the the readonly predicate on in_1 to hold. It ensures the writeonly predicate on out_1. The actual code uses getters and has to deal with nullness and temporal issues as well.

The body of the apply method consists of an unfold statement, followed by a case distinction over the proofs that have to be verified. For each proof, the proof hints are copied into the corresponding branch in the apply method. In order to get error messages to point to the

```java
  class Wandreadonly_for_writeonly {
    int lemma;
    WandDemo in_1;
    */@
    requires this.valid();
    ensures true;
    @*/
    WandDemo get_in_1()=in_1;
    WandDemo out_1;
    */@
    requires this.valid();
    ensures true;
    @*/
    WandDemo get_out_1()=out_1;
    */@
    requires this.valid()**get_in_1()=null
    **get_in_1().readonly()**get_out_1()=null;
    ensures \old(get_out_1()).writeonly();
    @*/
    void _apply() {
      //@ unfold this.valid();
      if (lemma==1) {
        //@ unfold this_1.readonly();
        //@ fold this_1.writeonly();
        assert true==out_1.writeonly();
      }
    }
    WandDemo this_1;
    */@
    predicate boolean valid()=Perm(lemma,100)
    **lemma>0**Perm(in_1,100)**Perm(out_1,100)
    **Perm(this_1,100)
    **(lemma==1=>
      this_1!=null**Perm(this_1.x,75)**
      in_1==this_1*out_1==this_1)
    **lemma<=1;
    */
    requires true;
    ensures PointsTo(lemma,100,0)**PointsTo(in_1,100,null)
    **PointsTo(out_1,100,null)**PointsTo(this_1,100,null),
    @*/
    Wandreadonly_for_writeonly() {
      lemma=0;
      in_1=null;
      out_1=null;
      this_1=null;
    }
  }
```

Fig. 13. Encoding of the wand formula in class WandDemo
correct place, an assert statement is added, which in case of a bad proof will trigger error messages for the specific proof, rather than for the _apply method in general.

The valid predicate specifies (write) access to all fields in the object and for every proof it has a conditional requirement that all of the required permissions and properties are valid. For our example this means that for example if lemma is 1 then we have 75% permission on this._x and this._l == _in_1 == _out_1. Finally, the predicate valid also states that lemma must contain a valid proof number.

In Java, the logical way of creating new witness objects is to have an overloaded constructor for every proof script. That is not possible in Chalice, so instead we generate a factory method with a unique name for every proof script. These methods are placed in the class that calls them. Thus, in our case, the factory method Wand.readonly_for.writeonly Lemma_1, which requires the permissions and properties used in the proof and ensures a magic wand witness, is put into the WandDemo class.

We do not include a full listing of the generated Chalice code; however it can be generated using the online version of the VerCors tool set.

### 5.3 Correctness of the Encoding

This sections aims to provide an intuition why our encoding is correct. We will do so based on the view that a static verifier is a tool that, given a program with specifications establishes the existence of a correctness proof of those specifications. For example, for a sequential program it would establish the existence of a Hoare logic proof. Therefore, we will show that the program before the witness transformation can be proven correct if and only if the program after the witness transformation can be proven correct.

In order to avoid unnecessary clutter, we will ignore scoping and visibility rules for variables, as it is well-known how to fix these issues. Moreover, we will focus on the places where magic wands are introduced and eliminated. In proofs, magic wands can also occur in many other places, as they are carried along in proofs and specifications. However, as long as a magic wand is not introduced or eliminated, it is no different from any other formula and thus irrelevant for showing the correctness of the encoding.

The existence of proofs in a proof system is denoted with the symbol ⊢. That is,

\[ F_1, \ldots, F_n \vdash G \]

denotes that there is a proof that \( G \) logically follows from \( F_1 \star \cdots \star F_n \). The classical introduction and elimination rules for magic wands are the following [17]:

\[
F, G_1 \vdash G_2 \\
F \vdash G_1 \rightarrow G_2 \ (I \rightarrow)
\]

---

4 [http://fmt.ewi.utwente.nl/puptol/vercors-verifier/](http://fmt.ewi.utwente.nl/puptol/vercors-verifier/)
Using this class the fragments can be rewritten to
\[
\left\{ H \times F \right\} \\
\text{Wand witness=} \text{new Wand();} \\
\left\{ H \times \text{witness.valid()} \right\}
\]

and
\[
\left\{ H \times G_1 \times (G_1 \rightarrow G_2) \right\} \\
\text{witness._apply();} \\
\left\{ H \times G_2 \right\}
\]

Instead of using a magic wand in the fragments directly, we now have encapsulated the magic wand in the `valid` predicate of the ghost class `Wand`. Note how the resources needed to prove the magic wand `F_2` are explicitly required in the contract of the constructor.

The reverse of this first step is inlining, which is known to be correct, so this first step preserves correctness of the annotated program.

All occurrences of the magic wand are now located inside the ghost class `Wand`, and in particular in the definition of the predicate `valid`. As a second step, we eliminate the magic wand from the definition of `valid`, by moving the proof of the magic wand from the constructor to the `_apply` method:

```plaintext
Wand { // version 2 
  resource valid()=F; 
  requires F; 
  ensures valid(); 
  Wand() { 
    {F} 
    fold valid(); 
    {valid()} 
  } 
  requires G_1 \times valid(); 
  ensures G_2; 
  void _apply() { 
    {G_1 \times valid()} 
    unfold valid(); 
    {G_1 \times F} 
    // because F \vdash G_1 \rightarrow G_2 
    {G_1 \times (G_1 \rightarrow G_2)} 
    // by application of E \rightarrow 
    {G_2} 
  } 
}
```

Clearly, since the specifications of the constructor and method `_apply` did not change compared to the encoding in version 1 of class `Wand`, any program proven correct using the first `Wand` encoding, will also be proven correct using the `Wand` version 2 encoding and vice versa. Notice further that the correctness proofs of the constructor and the method `_apply` in version 2 of class `Wand` are correct if and only if they are correct in version 1 of class `Wand`.

Finally, as a last step to get to the encoding that we have actually presented above, we can use a cut elimination theorem from the underlying proof theory (similar to [32]), and simplify the annotated body of the `_apply` method as follows:

```plaintext
{G_1 \times valid()} 
unfold valid(); 
{G_1 \times F} 
// because F, G_1 \vdash G_2 
{G_2}
```

This removes the last occurrence of the magic wand from the annotated code.

We have sketched how a program before the witness encoding can be proven correct if and only if the program after the transformation can be proven correct. Thus the witness transformation for magic wands is both sound and complete.

For this correctness sketch, we have conveniently forgotten about visibility of fields and methods. However,
the actual implementation does take care of those. Similarly, the implementation takes care of multiple proofs by having multiple factory methods instead of a single constructor and by using a case distinction in the bodies of the valid predicate and apply method, to distinguish between the different proof resources and proofs.

5.4 Recipe for the Encoding

To conclude, we sketch the complete translation algorithm from a program that has been annotated using magic wands into a program that is annotated using witnesses. This requires to translate all occurrences of magic wand formulas and all occurrences of magic wand proof scripts.

The implementation supports magic wand formulas where both the required formula and the ensured formula are a conjunction of predicate invocations. For this presentation, we limit ourselves to a single required predicate and a single ensured predicate. We can do this without loss of generality because any formula can be turned into a single predicate formula, where the predicate body is the conjunction of the individual formulas and where the parameters are the free variables in the formulas.

Thus, we assume magic wands of the form:

\[
\text{name} : \mathcal{P}(\overline{e}) \rightarrow \mathcal{Q}(\overline{f})
\]

For every magic wand formula that uses the same combination of predicates, we use the same witness class, whose template is given in Fig. 15. First we generate a list of field declarations for the witness class, declaring all the parameters used in the magic wand formula:

- for \(i = 1 \cdots |\overline{e}|\): typeof \(e_i\) in \(i\);
- for \(i = 1 \cdots |\overline{f}|\): typeof \(f_i\) out \(i\);

where typeof is a meta operator that extracts the type of an expression.

Further, the witness class defines getters for all of its fields. This allows us to replace the magic wand formula

\[
\text{name} : \mathcal{P}(\overline{e}) \rightarrow \mathcal{Q}(\overline{f})
\]

with a formula that states that name is a valid witness of this formula:

\[
\text{name} = \text{null} \quad \text{null} \quad \text{name}.\text{valid}() \quad \text{**}
\]

\[
\forall i \in 1 \cdots |\overline{e}|, \quad \text{name}.\text{get}_i() = e_i \quad \text{**}
\]

\[
\forall j \in 1 \cdots |\overline{f}|, \quad \text{name}.\text{get}_j() = f_j
\]

Note that the quantifiers in this formula have to be expanded at code generation time because they use mathematical meta-notation that is not part of our syntax.

The annotated program will contain several proof scripts to create a witness, all matching the following template.

\[
\text{create }
\]

\[
\text{script;}
\]

\[
\text{qed name} : \mathcal{P}(\overline{e}) \rightarrow \mathcal{Q}(\overline{f});
\]

Fig. 15. Template for a witness class
This section presents two more-involved examples that use the magic wand in their annotations. First, we consider the tree delete challenge. Second, we consider an iterator protocol for iterators on a list of integers. For both examples, we show how to provide full annotations, so that it can be verified by the VerCors tool set.

### 6.1 Verification of the Tree Delete Challenge

Using the encoding, we can verify the tree delete challenge. Taking the annotated algorithm in Fig. 3 as a starting point, we have to provide proof scripts whenever we create a magic wand formula in order to make it verifiable. Since the magic wand formula is used in the loop invariant (and a new instance of it is needed with every iteration of the loop), we actually require witnesses for the creation of magic wand objects in two places in the annotated program: we need a witness to create a magic wand formula to show that the loop invariant holds upon loop initialisation, i.e., before the loop is actually executed, and in addition we need to provide a witness to create a magic wand formula inside the loop body, to show that every iteration of the loop preserves the loop invariant.

Fig. [17] shows the resulting fully annotated version of the tree delete algorithm.

The creation of the the magic wand formula in the different annotations uses different witness proofs (see Lines 25 and 38-48). However, the two different instances are used both in the loop invariant, which shows that it is essential to have a single \texttt{apply} method that can be used in different proof scripts.

The online version of the VerCors tool set can be used to inspect the full Chalice encoding of this example. Using the Chalice encoding, the VerCors tool can verify the iterative tree delete algorithm without any problems. The entire example verifies in 6 minutes on an Intel i7-2600 (3.40GHz).

### 6.2 The Iterator Protocol

As a second example, we present a variant of the iterator protocol from Haack and Hurlin \cite{haack2015}. To simplify our presentation, we have chosen to work with a list of integers rather than a list of objects.

The iterator protocol uses the following three states: \texttt{ready}, \texttt{readyForNext} and \texttt{readyForRemove}. The entire protocol is displayed in Fig. [13] When an iterator is created, permissions on the current list are folded in the \texttt{ready} state. In this state, one may apply a magic wand to recover the permissions on the current list, or one may call \texttt{hasNext} to test for the existence of a next element. If such an element exists, the next state is \texttt{readyForNext}, otherwise the next state is \texttt{ready}. If the state is \texttt{readyForNext}, the \texttt{next} method can be used to retrieve the current element and the next state will be \texttt{readyForRemove}. In the \texttt{readyForRemove} state, one can use either the \texttt{remove} method or the magic wand provided by next to go back to the \texttt{ready} state.

The specifications of the integer list and the list iterator can be found in Fig. [19] and [20] respectively. We have annotated implementations of these interfaces, which have been verified with the VerCors tool set. The fully annotated versions are available from the same website as the tree delete example.

We have also verified a small example that illustrates the usage of the list and the iterator, see Fig. [21] In this example, we create a list containing $[-1, 0, 1]$ and then use an iterator to remove the negative elements. The entire example verifies in 19 seconds on an Intel i7-2600 (3.40GHz).
VerCors

//@ requires top!=null ** top.state();
//@ ensures contains(result,\old(top.contents()));
public Tree del_min(Tree top){
//@ seq<int> orig_contents=top.contents();
//@ seq<int> target_contents=tail(top.contents());
//@ unfold top.state();
if (top.left == null) {
  return top.right;
} else {
  Tree cur, left;
  cur = top;
  left = cur.left;
//@ seq<int> cur_contents=orig_contents;
//@ assert cur_contents == left.contents() + seq<int>{top.data} + tolist(top.right);
//@ unfold left.state();
//@ loop invariant Perm(cur.left,100) ** Perm(cur.data,100) ** Perm(cur.right,100);
//@ loop invariant cur.left==left
//@ loop invariant cur.right->state() ;
//@ loop invariant Perm(left.left,100) ** Perm(left.data,100) ** Perm(left.right,100);
//@ loop invariant left.left->state() ** left.right->state();
//@ loop invariant cur_contents == (toList(left.left) + seq<int>{left.data} + tolist(left.right))
//@ loop invariant cur_contents=toList(left.left) + seq<int>{left.data} + tolist(left.right);
//@ unfold left.state();
//@ seq<int> prev_contents = cur_contents; */
while (left.left != null) /*@ with {
  create {} wand:(top.state contains(target_contents)) -- top.state contains(target_contents)); @*/
  { /*@ Tree prev = cur;
    seq<int> prev_contents = cur_contents; */
    cur = left;
    left = cur.left;
//@ unfold left.state();
    cur_contents = toList(left.left) + seq<int>{left.data} + tolist(left.right);
    cur_contents = cur_contents + seq<int>{cur.data} + tolist(cur.right);
    assert prev_contents.length > 0 ;
    assert cur_contents.length > 0 ;
    assert prev_contents == cur_contents + seq<int>{prev.data} + tolist(prev.right);
  }
//@ use prev_contents.length > 0 ;
//@ use cur_contents.length > 0 ;
//@ use Perm(prev.left,100)**Perm(prev.data,100);
//@ use Perm(prev.right,100)**prev.right->state();
//@ use prev.left==cur;
//@ use prev_contents == cur_contents + seq<int>{prev.data} + tolist(prev.right);
//@ fold prev.state();
//@ apply wand:(prev.state contains(tail(prev_contents)) -- top.state contains(target_contents));
//@ fold prev.state();
//@ apply wand:(cur.state contains(tail(cur_contents)) -- top.state contains(target_contents));
//@ apply wand:(top.state contains(target_contents)) -- top.state contains(target_contents));
//@ fold prev.state();
//@ create {}
//@ use prev_contents.length > 0 ;
//@ use cur_contents.length > 0 ;
//@ use Perm(cur.left,100)**cur->state();
//@ use Perm(cur.data,100) ** cur->state();
//@ use Perm(cur.right,100)**cur.right->state();
//@ use prev.left==cur;
//@ use prev_contents == cur_contents + seq<int>{prev.data} + tolist(prev.right);
//@ fold prev.state();
//@ apply wand:(prev.state contains(tail(prev_contents)) -- top.state contains(target_contents));
//@ fold prev.state();
//@ apply wand:(cur.state contains(tail(cur_contents)) -- top.state contains(target_contents));
//@ apply wand:(top.state contains(target_contents)) -- top.state contains(target_contents));
//@ fold prev.state();
//@ create {}
//@ use prev_contents.length > 0 ;
//@ use cur_contents.length > 0 ;
//@ use Perm(cur.left,100)**cur->state();
//@ use Perm(cur.data,100) ** cur->state();
//@ use Perm(cur.right,100)**cur.right->state();
//@ use prev.left==cur;
//@ use prev_contents == cur_contents + seq<int>{prev.data} + tolist(prev.right);
//@ fold prev.state();
//@ apply wand:(prev.state contains(tail(prev_contents)) -- top.state contains(target_contents));
//@ fold prev.state();
//@ apply wand:(cur.state contains(tail(cur_contents)) -- top.state contains(target_contents));
//@ apply wand:(top.state contains(target_contents)) -- top.state contains(target_contents));
//@ fold prev.state();
//@ create {}
//@ use prev_contents.length > 0 ;
//@ use cur_contents.length > 0 ;
//@ use Perm(cur.left,100)**cur->state();
//@ use Perm(cur.data,100) ** cur->state();
//@ use Perm(cur.right,100)**cur.right->state();
//@ use prev.left==cur;
//@ use prev_contents == cur_contents + seq<int>{prev.data} + tolist(prev.right);
//@ fold prev.state();
//@ apply wand:(prev.state contains(tail(prev_contents)) -- top.state contains(target_contents));
//@ fold prev.state();
//@ apply wand:(cur.state contains(tail(cur_contents)) -- top.state contains(target_contents));
//@ apply wand:(top.state contains(target_contents)) -- top.state contains(target_contents));
//@ fold prev.state();
//@ create {}
//@ use prev_contents.length > 0 ;
//@ use cur_contents.length > 0 ;
//@ use Perm(cur.left,100)**cur->state();
//@ use Perm(cur.data,100) ** cur->state();
//@ use Perm(cur.right,100)**cur.right->state();
//@ use prev.left==cur;
//@ use prev_contents == cur_contents + seq<int>{prev.data} + tolist(prev.right);
//@ fold prev.state();
//@ apply wand:(prev.state contains(tail(prev_contents)) -- top.state contains(target_contents));
//@ fold prev.state();
//@ apply wand:(cur.state contains(tail(cur_contents)) -- top.state contains(target_contents));
//@ apply wand:(top.state contains(target_contents)) -- top.state contains(target_contents));
//@ fold prev.state();
//@ create {}
7 Conclusions

In this paper, we have introduced two witness transformations from Java with separation logic annotations into a form that can be checked with Chalice. The first transformation replaces predicates with parameters by witness objects with parameter-free predicates (except for the implicit this parameter). The second transformation replaces magic wands by witness objects. To overcome undecidability when reasoning about magic wands, the user has to provide a proof script that can be used to check that resources are indeed correctly exchanged by the magic wand. Both transformations are not Chalice-specific, i.e., in principle they can be used as an encoding for any object-oriented language with separation logic annotations. This is reflected in the implementation, which will remove the witness extensions and replace them with objects and simpler predicates. In this process unknown expressions and annotation are simply copied, which means that the current implementation will work for other similar back-ends without modification. However, it might require additional or different proof hints.

In this paper, we have shown that the transformations that we define are sound and complete, i.e., if the
original specification is correct, if and only if the transformed specification is correct.

It should be emphasised that all examples in this paper have been machine-checked, however we feel it is too early to make any claim about the class of programs that can be validated with our approach: the fact that a proof exists does not always mean that it can be found automatically by a prover.

To illustrate our approach, we have presented two larger examples: the tree delete example, which demonstrates how magic wands can be used in loop invariants, and the iterator example, which shows how a magic wand is used to enforce that method calls happen in the prescribed order. Both examples with full annotations are automatically verified by the VerCors tool set. In this paper, we have not presented an example that uses both witness transformations at the same time, but such an example is available online.

7.1 Related Work

The Tree Delete Challenge Our use of the magic wand to verify the iterative implementation of the tree delete algorithm is just one of the ways of solving the tree delete challenge. Another way of solving the problem is using Türk’s rule for loops [11], which effectively offers the possibility of writing loop annotations as if the loop were a recursive function. In other words, the loop body can contain a block of proof hints, whose application is delayed, forming a stack of delayed proof steps. When the loop exits, all delayed steps are applied. As these steps can implicitly set aside resources too, each delayed step is equivalent to a magic wand. The advantage of using magic wand syntax rather than the Türk rule is that magic wands can be used in any location in the code, whereas the Türk rule is only applicable to loops. Yet another mechanism for specifying the iterative version of the tree delete problem is by using block contracts. The Krakatoa tool [14] implements this generalization of the Türk rule, which allows attaching pre- and post-conditions to arbitrary statement blocks, instead of just to loop bodies.

The tree delete challenge can also be addressed by using the Zipper data structure from functional programming [21]. This is an alternative way to treat the shift of focus: with little it allows to write the invariant in such a way that in each iteration the focus is shifted by one step. Once the left-most node is removed, the Zipper structure allows to move the focus back to the root. To use a Zipper structure, one would have to write the basic data structure, which requires quite a few lines of specification, but which can easily be automated. For those cases where the pre-defined functions work, the annotation work load would then be limited to a few instructions that move the focus. However, if a non-standard move is required, it would have to be spelled out completely at a considerable specification effort, because only pre-defined moves would be automatically generated.

Tool Support The VeriFast tool implements lemmas [25], which are equivalent to a subset of magic wands. That is, lemmas can transform sets of resources, just like magic wand can. But magic wands can exchange resources as well, for example, a magic wand can express the capability to exchange a read-only permission for accessing a location on the heap to a full write permission for the same location. If one were to express this using a lemma, the lemma would have to be paired with a predicate that captures the extra permissions in order to match the functionality of the magic wand.

Jost and Summers have written a tool that translates VeriFast specification to Chalice [26]. They analyse VeriFast predicates to see if they can be split into a parameterless permission predicate and a boolean function. If such a split is impossible they resort to using ghost variables in the object to represent the predicate parameters. The advantage of such an approach is that it leads to a simpler encoding that requires less annotations to be verified. The disadvantage is that the encoding does not work if the same predicate must be held twice on the same object, whereas our encoding supports this.

Automated proving for special cases of the magic wand is being worked on. For example, in the setting of the logic of boolean bunched implications (a close relative of separation logic) Park et al. have given an algorithm for deciding the magic wand [32]. Further, Atkey has shown that for a restricted magic wand syntax it is possible to directly derive verification conditions [2].

Finally, the transformation that turns magic wands into objects resembles defunctionalization of closures in a functional programming language (see for example [38]).

The specification style of the generated object, defining a valid predicate is inspired by the standard methodology for Boogie of Barnett et al. [1].

7.2 Future Work

Annotation Generation Clearly, the major drawback of our approach is the large number of (long) annotations that the user has to provide at the moment. To address this issue, we will first of all investigate heuristics to come up with good default specifications. We will also investigate if methods for automatically deriving specifications can be adapted to our situation. For example, it might be possible to use the constraint solving algorithm developed by Ferrara and Müller [13] to infer a large number of the witness management annotations.

Additionally, as mentioned above, the approach taken by Jost and Summers in their translation of VeriFast to Chalice can deal with a number of cases with less annotation load. We will study if we can integrate their work, so that the simple cases can take advantage of their en-
coding, while the encoding of this paper can be used for more complicated cases.

The current implementation of the witness transformation for magic wands requires that all the resources that are stored inside the magic wand are explicitly mentioned using the `use` keyword. The problem of finding out which resources have to be stored is similar to the problem of finding out which resources to pass to a method during a call and which resources to keep. Therefore, we will study the techniques for solving this, such as the use of frame inference and bi-abduction [9], and see if they can be reused and/or adapted to our approach.

**Extensions** We believe the witness encoding approach can be extended to basic permissions on fields too. Such an extension replaces the permissions-as-logic-formulas view completely by a permissions-as-a-data-type view. This would make any check on permissions executable, enabling the re-use of existing run-time checkers for run-time checking of permissions. It might even be possible to reuse existing static verifiers that do not support permissions, however there is a risk that this would even further increase the annotation workload, which would make this approach practically unfeasible.

We are currently investigating how permission-based separation logic can be used to reason about Scala programs. This requires the possibility to specify the behaviour of closures. We are investigating if we can extend our approach also to encode closure specifications.

We also plan to further investigate if temporal aspects have an impact on the encoding. In the current encoding, everything in the footprint of the magic wand is evaluated in the current state, while everything else is evaluated against the state in which the magic wand is eventually applied. We have to investigate further if this causes any problems.

**Acknowledgement**

The work presented in this paper is supported by ERC grant 258405 for the VerCors project.

**References**

1. A. Amighi, S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. The VerCors project: Setting up basecamp. In *Programming Languages meets Program Verification (PLPV 2012)*, pages 71–82. ACM Press, 2012.
2. R. Atkey. Amortised resource analysis with separation logic. In A. D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 85–103. Springer, 2010.
3. M. Barnett, R.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *LNCS*, pages 364–387. Springer-Verlag, 2005.
4. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
5. J. Berdine, C. Calcagno, and P.W. O’Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F.S. de Boer, M.M. Bonsangue, S. Graf, and W.P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.
6. R. Bornat, C. Calcagno, P.W. O’Hearn, and M.J. Parkinson. Permission accounting in separation logic. In Palsberg and Abadi [31], pages 259–270.
7. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2004 of *LNCS*, pages 55–72. Springer-Verlag, 2003.
8. S. Brochonin, S. Demri, and É. Lozes. On the almighty wand. *Inf. Comput.*, 211:106–137, 2012.
9. C. Calcagno, D. Distefano, P.W. O’Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011.
10. C. Calcagno, P. W. O’Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE Computer Society, 2007.
11. L. Mendonça de Moura and N. Bjørner. Z3: An efficient SMT solver. In C.R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer-Verlag, 2008.
12. D. DiStefano and M. Parkinson. jStar: Towards practical verification for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 213–226. ACM Press, 2008.
13. P. Ferrara and P. Müller. Automatic inference of access permissions. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*, *LNCS*, pages 202–218. Springer-Verlag, 2012.
14. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.
15. Roberto Giacobazzi and Radhia Cousot, editors. *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’13, Rome, Italy - January 23 - 25, 2013*. ACM, 2013.
16. C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. In J. Meseguer and G. Rosu, editors, *Algebraic Methodology and Software Technology*, volume 5140 of *LNCS*, pages 199–215. Springer-Verlag, 2008.
17. C. Haack and C. Hurlin. Resource Usage Protocols for Iterators. *Journal of Object Technology*, 8(4):55–83, 2009.
18. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
19. A. Hobor and J. Villard. The ramifications of sharing in data structures. In Giacobazzi and Cousot [15], pages 523–536.
20. W. Howard. *The formulae-as-types notion of construction*, pages 479 – 490. ACM, 1980. Original paper manuscript from 1969.
21. G. P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

22. M. Huisman, V. Klebanov, and R. Monahan. VerifyThis verification competition 2012 – organizer’s report. Technical Report 2013-01, Department of Informatics, Karlsruhe Institute of Technology, 2013. Available at [http://digbib.ubka.uni-karlsruhe.de/volltexte/1000034373](http://digbib.ubka.uni-karlsruhe.de/volltexte/1000034373).

23. C. Hurlin. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic*. PhD thesis, Université Nice Sophia Antipolis, 2009.

24. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW520, Katholieke Universiteit Leuven, 2008.

25. B. Jacobs, J. Smans, and F. Piessens. VeriFast: Imperative programs as proofs. In *VSTTE workshop on Tools & Experiments*, August 2010.

26. D. Jost and A. J. Summers. An automatic encoding from VeriFast predicates into Implicit Dynamic Frames. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, 2013.

27. N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In Andrew Kennedy and Amal Ahmed, editors, *TLDI*, pages 105–116. ACM, 2009.

28. G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, February 2007. Dept. of Computer Science, Iowa State University. Available from [http://www.jmlspecs.org](http://www.jmlspecs.org).

29. K.R.M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *Lecture notes of FOSAD*, volume 5705 of *LNCS*, pages 195–222. Springer-Verlag, 2009.

30. P. W. O’Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

31. J. Palsberg and M. Abadi, editors. *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*, Long Beach, California, USA, January 12-14, 2005. ACM, 2005.

32. J. Park, J. Seo, and S. Park. A theorem prover for boolean BI. In Giacobazzi and Cousot [15], pages 219–232.

33. M. Parkinson. Local reasoning for Java. Technical Report UCAM-CL-TR-654, University of Cambridge, 2005.

34. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In Palsberg and Abadi [31], pages 247–258.

35. M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3:01):1–54, 2012.

36. M.J. Parkinson and G.M. Bierman. Separation logic, abstraction and inheritance. In G.C. Necula and P. Wadler, editors, *POPL*, pages 75–86. ACM, 2008.

37. M.J. Parkinson and A.J. Summers. The relationship between separation logic and implicit dynamic frames. In G. Barthe, editor, *ESOP*, volume 6602 of *LNCS*, pages 439–458. Springer-Verlag, 2011.

38. J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

39. J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

40. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2, 2012.

41. T. Türk. Local reasoning about while-loops. In R. Joshi, T. Margaria, P. Müller, D. Naumann, and H. Yang, editors, *VSTTE 2010. Workshop Proceedings*, pages 29–39. ETH Zürich, 2010.