Proof Pearl: Magic Wand as Frame

Qinxiang Cao[1], Shengyi Wang[2], Aquinas Hobor[2], and Andrew W. Appel[1]

[1] Princeton University
[2] National University of Singapore

February 2, 2018

Abstract. Separation logic is widely used to verify programs that manipulate pointers. It adds two connectives: separating conjunction $*$ ("star") and its adjoint, separating implication $-\!\!*$ ("magic wand"). Comparatively, separating conjunction is much more widely used. Many separation logic tools do not even support separating implication. Especially in interactive program verification or pen-paper proofs, people often find magic wand expressions not useful in expressing preconditions and postconditions.

We demonstrate that by using magic wand to express *frame*s that relate local portions of data structures to global portions, we can exploit its power while proofs are still easily understandable. This magic-wand-as-frame technique is especially useful when verifying imperative programs that walk through a data structure from the top down. We use binary search tree insert as an example to demonstrate this proof technique.

Keywords: Separation logic, Separating implication, Program verification

1 Thesis

When proving in separation logic a program that traverses (and possibly modifies) a list or tree data structure, use these PROOF RULES OF WAND-FRAME:

WANDQ-FRAME-INTRO: $Q \vdash \forall x.\, (P(x) -\!\!* P(x) * Q)$

WANDQ-FRAME-ELIM: $P(x) * \forall x.\, (P(x) -\!\!* Q(x)) \vdash Q(x)$

WANDQ-FRAME-HOR: $\forall x.\, (P_1(x) -\!\!* Q_1(x)) * \forall x.\, (P_2(x) -\!\!* Q_2(x)) \vdash$
$\forall x.\, (P_1(x) * P_2(x) -\!\!* Q_1(x) * Q_2(x))$

WANDQ-FRAME-VER: $\forall x.\, (P(x) -\!\!* Q(x)) * \forall x.\, (Q(x) -\!\!* R(x)) \vdash \forall x.\, (P(x) -\!\!* R(x))$

WANDQ-FRAME-REFINE: $\forall x.\, (P(x) -\!\!* Q(x)) \vdash \forall y.\, (P(f(y)) -\!\!* Q(f(y)))$

2 Introduction

Separation logic [14] is an extension of Hoare logic that has been widely used in program verification. The *separating conjunction* $P * Q$ ("star") in assertions represents the existence of two disjoint states, one that satisfies P and one that satisfies Q. Formally,

$m \vDash P * Q \;\; =_{\text{def}} \;\;$ there exist m_1 and m_2 s.t. $m = m_1 \oplus m_2$, $m_1 \vDash P$ and $m_2 \vDash Q$.

Here, $m_1 \oplus m_2$ represents the disjoint union of two pieces of state/memory. The $*$ concisely expresses address (anti)aliasing. For example, if "$p \mapsto v$" is the assertion that data v is stored at address p, then $p \mapsto v * q \mapsto u$ says v is stored at address p, u is stored at address q, and $p \neq q$. Separation logic enables one to verify a Hoare triple locally but use it globally, using the *frame rule:*

$$\textsc{frame}\frac{\{P\}\,c\,\{Q\} \qquad \mathrm{FV}(F) \cap \mathrm{ModV}(c) = \emptyset}{\{P * F\}\,c\,\{Q * F\}}$$

Star has a right adjoint $P \mathrel{-\!\!*} Q$ *separating implication*, a.k.a. "magic wand":

$m \vDash P \mathrel{-\!\!*} Q \;\; =_{\mathrm{def}} \;\;$ for any m_1 and m_2, if $m \oplus m_1 = m_2$ and $m_1 \vDash P$ then $m_2 \vDash Q$.

$$\textsc{wand-adjoint1}\frac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R} \qquad \textsc{wand-adjoint2}\frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}$$

Magic wands are famously difficult to control [15]. In the early days of separation logic, magic wand was used to generate weakest preconditions and verification conditions for automated program verification. However, those verification conditions are not human readable or understandable, and decision procedures for entailment checking with magic wand are quite complex.

Magic wand is rarely used in interactive program verification or pencil-and-paper proofs. (There are some exceptions [11, 12] that we discuss in the related work section (§6).) Authors tend to use forward verification instead of backward verification since it is easier to understand a program correctness proof that goes in the same direction as program execution. "Forward" Hoare logic rules do not generate magic wand expressions; therefore, most authors find that the expressive power of star is already strong enough. For example, we need to define separation logic predicates for different data structures (like records, arrays, linked list and binary trees) in order to verify related programs. Berdine *et al.* [4] and Charguéraud [8] show that these predicates can be defined with separating conjunction only.

In this paper, we propose a new proof technique: *magic wand as frame*, and we show that using magic wand can make program correctness proofs more elegant.

The main content of this paper is a proof pearl. We use our new proof technique to verify the C program in Fig. 1, insertion into a binary search tree (BST). The program uses a while loop to walk down from the root to the location to insert the new element.

A pointer to a tree has type **struct** tree $*$, but we also need the type pointer-to-pointer-to-tree, which we call treebox. The insert function does not return a new tree, it modifies the old tree, perhaps replacing it entirely (if the old tree were the empty tree).

Consider running insert(p_0,8,"h"), where p_0 points to a treebox containing the root of a tree as shown in Fig. 2. After one iteration of the loop or two iterations, variable p contains address p, which is a treebox containing a pointer to a subtree. This subtree t and a partial tree P (shown within the dashed line) form the original BST.

Naturally, we can verify such a program using a loop invariant with the following form: P *is stored in memory* $*$ t *is stored in memory*.

To describe partial trees, most authors [4, 8] would have you introduce a new inductive description of *tree with exactly one hole*—in addition to the inductive description

```
struct tree {int key; void *value;
             struct tree *left, *right;};

typedef struct tree **treebox;

void insert (treebox p, int x, void *v) {
  struct tree *q;
  while (1) {
  q = *p;
  if (q==NULL) {
    q = (struct tree *) surely_malloc (sizeof *p);
    q→key=x; q→value=v;
    q→left=NULL; q→right=NULL;
    *p=q;
    return;
  } else {
    int y = q→key;
    if (x<y)
      p= &q→left;
    else if (y<x)
      p= &q→right;
    else {
      q→value=v;
      return;
} } } }
```

Fig. 1: Binary Search Tree insertion

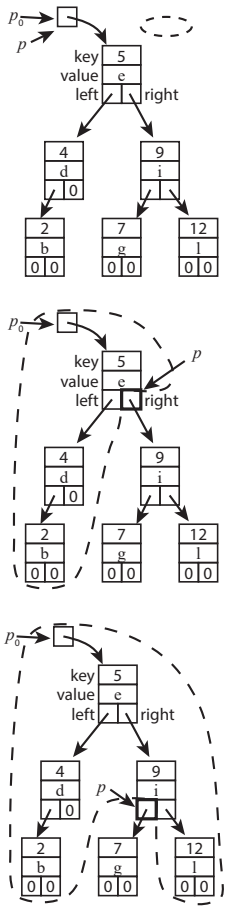


Fig. 2: Execution of insert$(p_0, 8, \text{"h"})$.

of ordinary trees—and define a corresponding recursive separation-logic predicate "partial tree P *is stored in memory*", in addition to the recursive predicate for ordinary trees. (Similarly, "list segment" is inductively defined as a list with one hole.)

That's a lot of duplication. We propose a different approach in this paper: using magic wand to express "P *is stored in memory*". With this approach, we do not even need to define "partial trees" and its corresponding separation logic predicate or prove their domain specific theories.

We organize the rest of this paper as follows:

§3 We verify this C implementation of BST insert using magic-wand-as-frame.

§4 We formalize this correctness proof in Coq using *Verifiable C* [3] and we show that magic-wand-as-frame also works for other implementations of BST insert, other operations of BST, and other data structures such as linked lists.

§5 We compare our proofs with traditional approaches. We discuss the power and limitation of using magic wand and we explain the name of our proof technique *magic-wand-as-frame*.

§6 We discuss related work and summarize our contributions.

3 Proof Pearl: Binary search tree insertion

This section demonstrates the main content of this paper: a magic-wand-as-frame verification of BST insert. Here we use standard mathematical notation; in the next section we give details about the Coq formalization and the proof notation of Verifiable C.

3.1 Specification

Correctness for BSTs means that the insert function—considered as an operation of an abstract data type—implements the *update* operation on finite maps from the key type (in this case, integer) to the range type (in this case **void***). The client of a *finite map* does not need to know that trees are used; we should hide that information in our specifications. For that purpose, we define separation logic predicates for binary trees, and we define map predicates based on tree predicates. Only map predicates show up in the specification of this insert function.

Binary trees: $t = \ E \mid T(t_1, k, v, t_2)$

Representation predicates:

$$\begin{array}{l} \mathrm{treebox\_rep}(E, p) \ =_{\mathrm{def}} \ p \mapsto \mathrm{null} \\ \mathrm{treebox\_rep}(T(t_1, k, v, t_2), p) \ =_{\mathrm{def}} \ \exists q.\ p \mapsto q \ * \ q.\mathsf{key} \mapsto k \ * \ q.\mathsf{value} \mapsto v \ * \\ \quad \mathrm{treebox\_rep}(t_1, q.\mathsf{left}) \ * \ \mathrm{treebox\_rep}(t_2, q.\mathsf{right}) \end{array} \quad (1)$$

$$\mathrm{Mapbox\_rep}(m, p) \ =_{\mathrm{def}} \ \exists t.\ \mathrm{Abs}(t, m) \wedge \mathrm{SearchTree}(t) \wedge \mathrm{treebox\_rep}(t, p)$$

Representation invariant: $\mathrm{SearchTree}(t)$, the *search-tree property*. That is, at any node of the tree t, the keys in the left subtree are strictly less than the key at the node, and the keys in the right subtree are strictly greater.

Abstraction relation: $\mathrm{Abs}(t, m)$, says that m is an abstraction of tree t, i.e., the key-value pairs in map m and tree t are identical.

High-level separation-logic specification of insert function:

$$\begin{array}{rl} \text{Precondition}: & \{[\![\mathsf{p}]\!] = p_0 \wedge [\![\mathsf{x}]\!] = x \wedge [\![\mathsf{v}]\!] = v \wedge \mathrm{Mapbox\_rep}(m_0, p_0)\} \\ \text{Command}: & \mathsf{insert}(\mathsf{p}, \mathsf{x}, \mathsf{v}) \\ \text{Postcondition}: & \{\mathrm{Mapbox\_rep}(\mathrm{update}(m_0, x, v), p_0)\} \end{array}$$

We use $[\![\mathsf{p}]\!]$ to represent the value of program variable p. In the postcondition, $\mathrm{update}(m_0, x, v)$ is the usual update operation on maps. $\mathrm{SearchTree}(t)$ and $\mathrm{Abs}(t, m)$ are formally defined in the SearchTree chapter of *Verified Functional Algorithms* [2]. Their exact definitions are not needed in our proof here.

3.2 Two-level proof strategy

One could directly prove the correctness of the C-language insert function, using the *search-tree* property as an invariant. But it is more modular and scalable to do a two-level proof instead [1, 10]: First, prove that the C program (imperatively, destructively) implements the (mathematical, functional) ins function on binary search trees; then prove that the (pure functional) binary search trees implement (mathematical) finite

maps, that ins implements update, and that ins preserves the search-tree property. So let us define insertion on pure-functional tree structures:

$$\begin{array}{rcl} \mathrm{ins}(E,x,v) & =_{\mathrm{def}} & T(E,x,v,E) \\ \mathrm{ins}(T(t_1,x_0,v_0,t_2),x,v) & =_{\mathrm{def}} & \text{If } x<x_0,\ T(\mathrm{ins}(t_1,x,v),x_0,v_0,t_2) \\ & & \text{If } x=x_0,\ T(t_1,x,v,t_2) \\ & & \text{If } x>x_0,\ T(t_1,x_0,v_0,\mathrm{ins}(t_2,x,v)) \end{array}$$

The *SearchTree* chapter of *VFA* [2] proves (via the Abs relation) that this implements update on abstract finite maps. Next, we'll prove that the C program refines this functional program; then compose the two proofs to show that the C program satisfies its specification given at the end of §3.1.

For that refinement proof, we give a low-level separation-logic specification of the insert function, i.e., the C program refines the functional program:

$$\begin{array}{l} \{[\![\mathsf{p}]\!] = p_0 \wedge [\![\mathsf{x}]\!] = x \wedge [\![\mathsf{v}]\!] = v \wedge \mathrm{treebox\_rep}(t_0,p_0)\} \\ \quad \mathsf{insert}(\mathsf{p},\mathsf{x},\mathsf{v}) \\ \{\mathrm{treebox\_rep}(\mathrm{ins}(t_0,x,v),p_0)\} \end{array} \tag{2}$$

3.3 Magic wand for partial trees

The function body of insert is just one loop. We will need a loop invariant! As shown in Fig. 2, the original binary tree can always be divided into two parts after every loop body iteration: one is a subtree t whose root is tracked by program variable p (that is, $[\![*\mathsf{p}]\!]$ is the address of t's root node) and another part is a partial tree P whose root is identical with the original tree and whose hole is marked by address $[\![\mathsf{p}]\!]$.

The separation logic predicate for trees (also subtrees) is $\mathrm{treebox\_rep}$. We define the separation logic predicate for partial trees as follows. Given a partial tree P, which is a function from binary trees to binary trees:

$$\mathrm{partial\_treebox\_rep}(P,r,i) =_{\mathrm{def}} \forall t.\ \ (\mathrm{treebox\_rep}(t,i) \mathrel{-\!\!*} \mathrm{treebox\_rep}(P(t),r))$$

This predicate has some important properties and we will use these properties in the verification of insert. (3a) and (3b) show how single-layer partial trees are constructed. (3c) shows the construction of empty partial trees. (3d) shows that a subtree can be filled in the hole of a partial tree. And (3e) shows the composition of partial trees.

These properties are direct instances of the MAGIC-WAND-AS-FRAME proof rules (§1), which are all derived rules from minimum first-order separation logic (i.e. intuitionistic first order logic + commutativity and associativity of separating conjunction + emp being separating conjunction unit + WAND-AJOINT). WANDQ-FRAME-INTRO proves (3a), (3b) and (3c). WANDQ-FRAME-ELIM proves (3d). WANDQ-FRAME-VER and WANDQ-FRAME-REFINE together prove (3e). The soundness of (3c) (3d) and (3e) do not even depend on the definition of $\mathrm{treebox\_rep}$.

$$p \mapsto q \ * \ q.\mathsf{key} \mapsto k \ * \ q.\mathsf{value} \mapsto v \ * \ \mathrm{treebox\_rep}(t_2, q.\mathsf{right}) \vdash \mathrm{partial\_treebox\_rep}(\lambda t.\ T(t, k, v, t_2),\ p,\ q.\mathsf{left}) \tag{3a}$$

$$p \mapsto q \ * \ q.\mathsf{key} \mapsto k \ * \ q.\mathsf{value} \mapsto v \ * \ \mathrm{treebox\_rep}(t_1, q.\mathsf{left}) \vdash \mathrm{partial\_treebox\_rep}(\lambda t.\ T(t_1, k, v, t),\ p,\ q.\mathsf{right}) \tag{3b}$$

$$\mathrm{emp} \vdash \mathrm{partial\_treebox\_rep}(\lambda t.\ t,\ p,\ p) \tag{3c}$$

$$\mathrm{treebox\_rep}(t, i) * \mathrm{partial\_treebox\_rep}(P, r, i) \vdash \mathrm{treebox\_rep}(P(t), r) \tag{3d}$$

$$\mathrm{partial\_treebox\_rep}(P_1,\ p_1,\ p_2) * \mathrm{partial\_treebox\_rep}(P_2,\ p_2,\ p_3) \vdash \mathrm{treebox\_rep}(P_1 \circ P_2,\ p_1,\ p_3) \tag{3e}$$

3.4 Implementation correctness proof

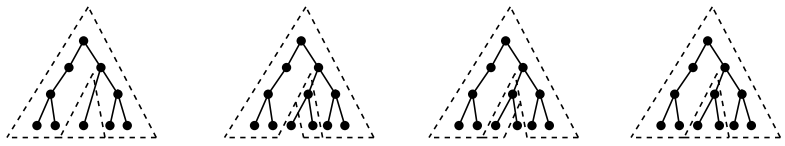
Now, we can verify the insert function with the loop invariant,

$$\exists t\ p\ P.\ P(\mathrm{ins}(t, x, v)) = \mathrm{ins}(t_0, x, v) \ \wedge \ [\![\mathsf{p}]\!] = p \ \wedge \ [\![\mathsf{x}]\!] = x \ \wedge \ [\![\mathsf{v}]\!] = v \ \wedge \ \mathrm{treebox\_rep}(t, p) \ * \ \mathrm{partial\_treebox\_rep}(P, p_0, p)$$

It says, a partial tree P and a tree t are stored in disjoint pieces of memory, and if we apply the ins function to t locally and fill the hole in P with that result, then we will get the same as directly applying ins to the original binary tree t_0.

The correctness of insert is based on the following two facts. First, the precondition of insert implies this loop invariant because we can instantiate the existential variables t, p and P with t_0, p_0 and $\lambda t.\ t$ and apply property (3c). Second, the loop body preserves this loop invariant and every **return** command satisfies the postcondition of the whole C function. Fig. 3 shows our proof (for conciseness, we omit $[\![\mathsf{x}]\!] = x \ \wedge \ [\![\mathsf{v}]\!] = v$ in all assertions).

This loop body has four branches: two of them end with return commands and the other two end normally. In the first branch, the inserted key does not appear in the original tree. This branch ends with a **return** command at line 13. We show that the program state at that point satisfies the postcondition of the whole function body (line 11). The transition from line 10 to line 11 is sound due to rule (3d). The second branch contains only one command at line 18. We re-establish our loop invariant in this branch (line 20). The transition from line 15 to line 19 is due to rule (3a) and the transition from line 19 to line 20 is due to rule (3e). The third branch at line 22 is like the second, and the last branch is like the first one.



In summary, the partial tree P is established as an empty partial tree ($\lambda \hat{t}.\ \hat{t}$) in the beginning. The program merges one small piece of subtree t into the partial tree

1 $\left\{\begin{array}{c} P(\mathrm{ins}(t,x,v)) = \mathrm{ins}(t_0,x,v) \ \wedge\ [\![\mathsf{p}]\!] = p \ \wedge \\ \mathrm{treebox\_rep}(t,p) \ *\ \mathrm{partial\_treebox\_rep}(P,p_0,p) \end{array}\right\}$

2 **q = * p;**

3 **if (q == NULL)** {

4 $\left\{\begin{array}{c} P(\mathrm{ins}(t,x,v)) = \mathrm{ins}(t_0,x,v) \ \wedge\ [\![\mathsf{p}]\!] = p \ \wedge\ [\![\mathsf{q}]\!] = \mathrm{null} \ \wedge\ t = E \ \wedge \\ p \mapsto \mathrm{null} \ *\ \mathrm{partial\_treebox\_rep}(P,p_0,p) \end{array}\right\}$

5 $\left\{\begin{array}{l} P(T(E,x,v,E)) = \mathrm{ins}(t_0,x,v) \ \wedge\ [\![\mathsf{p}]\!] = p \ \wedge \\ \ p \mapsto \mathrm{null} \ *\ \mathrm{partial\_treebox\_rep}(P,p_0,p) \end{array}\right\}$

6 **q = (struct tree *) surely_malloc (sizeof *q);**

7 **q→key=x; q→value=v;**

8 **q→left=NULL; q→right=NULL;**

9 ***p=q;**

10 $\left\{\begin{array}{c} P(T(E,x,v,E)) = \mathrm{ins}(t_0,x,v) \ \wedge\ [\![\mathsf{p}]\!] = p \ \wedge \\ \mathrm{treebox\_rep}(T(E,x,v,E),p) \ *\ \mathrm{partial\_treebox\_rep}(P,p_0,p) \end{array}\right\}$

11 $\{P(T(E,x,v,E)) = \mathrm{ins}(t_0,x,v) \ \wedge\ [\![\mathsf{p}]\!] = p \ \wedge\ \mathrm{treebox\_rep}(P(T(E,x,v,E)),p_0)\}$

12 $\{\mathrm{treebox\_rep}(\mathrm{ins}(t_0,x,v),p_0)\}$

13 **return;**

14 } **else** {

15 $\left\{\begin{array}{l} \exists t_1\ t_2\ x_0\ v_0\ q.\ \ P(\mathrm{ins}(t,x,v)) = \mathrm{ins}(t_0,x,v) \ \wedge\ t = T(t_1,x_0,v_0,t_2) \ \wedge\ [\![\mathsf{p}]\!] = p \ \wedge \\ [\![\mathsf{q}]\!] = q \ \wedge\ p \mapsto q \ *\ q.\mathsf{key} \mapsto x_0 \ *\ q.\mathsf{value} \mapsto v_0 \ *\ \mathrm{treebox\_rep}(t_1, q.\mathsf{left}) \ * \\ \mathrm{treebox\_rep}(t_2, q.\mathsf{right}) \ *\ \mathrm{partial\_treebox\_rep}(P,p_0,p) \end{array}\right\}$

16 **int y = q→key;**

17 **if (x<y)**

18 **p= &q→left;**

19 $\left\{\begin{array}{l} \exists t_1\ t_2\ x_0\ v_0\ q.\ \ x < x_0 \ \wedge\ P(\mathrm{ins}(t,x,v)) = \mathrm{ins}(t_0,x,v) \ \wedge \\ t = T(t_1,x_0,v_0,t_2) \ \wedge\ [\![\mathsf{p}]\!] = q.\mathsf{left} \ \wedge \\ \mathrm{partial\_treebox\_rep}(\lambda\hat{t}.\ T(\hat{t},x_0,v_0,t_2),p,q.\mathsf{left}) \ * \\ \mathrm{treebox\_rep}(t_1,q.\mathsf{left}) \ *\ \mathrm{partial\_treebox\_rep}(P,p_0,p) \end{array}\right\}$

20 $\left\{\begin{array}{l} \exists t_1\ t_2\ x_0\ v_0\ q.\ \ P(T(\mathrm{ins}(t_1,x,v),x_0,v_0,t_2)) = \mathrm{ins}(t_0,x,v) \ \wedge \\ [\![\mathsf{p}]\!] = q.\mathsf{left} \ \wedge\ \mathrm{treebox\_rep}(t_1,q.\mathsf{left}) \ * \\ \mathrm{partial\_treebox\_rep}(\lambda\hat{t}.\ P(T(\hat{t},x_0,v_0,t_2)),p_0,q.\mathsf{left}) \end{array}\right\}$

21 **else if (y<x)**

22 **p= &q→right;**

23 $\left\{\begin{array}{l} \exists t_1\ t_2\ x_0\ v_0\ q.\ \ P(t_1,x_0,v_0,T(\mathrm{ins}(t_2,x,v))) = \mathrm{ins}(t_0,x,v) \ \wedge \\ [\![\mathsf{p}]\!] = q.\mathsf{right} \ \wedge\ \mathrm{treebox\_rep}(t_2,q.\mathsf{right}) \ * \\ \mathrm{partial\_treebox\_rep}(\lambda\hat{t}.\ P(T(t_1,x_0,v_0,\hat{t})),p_0,q.\mathsf{right}) \end{array}\right\}$

24 **else** {

25 **p→value=v;**

26 $\left\{\begin{array}{l} \exists t_1\ t_2\ x_0\ v_0\ q.\ \ x = x_0 \ \wedge\ P(\mathrm{ins}(t,x,v)) = \mathrm{ins}(t_0,x,v) \ \wedge\ t = T(t_1,x_0,v_0,t_2) \ \wedge \\ \mathrm{treebox\_rep}(T(t_1,x,v,t_2),p) \ *\ \mathrm{partial\_treebox\_rep}(P,p_0,p) \end{array}\right\}$

27 $\left\{\begin{array}{l} \exists t_1\ t_2\ x_0\ v_0\ q.\ \ P(T(t_1,x,v,t_2)) = \mathrm{ins}(t_0,x,v) \ \wedge \\ \mathrm{treebox\_rep}(T(t_1,x,v,t_2),p) \ *\ \mathrm{partial\_treebox\_rep}(P,p_0,p) \end{array}\right\}$

28 $\{\mathrm{treebox\_rep}(\mathrm{ins}(t_0,x,v),p_0)\}$

29 **return;**

30 } }

Fig. 3: Proof of loop body

in each iteration of the loop body. Finally, when the program returns, it establishes a local insertion result ($\mathrm{ins}(t, x, v)$) and fills it in the hole of that partial tree—we know the result must be equivalent with directly applying insertion to the original binary tree. The diagrams above illustrate the situations of these four branches and our proof verifies this process.

4 Coq formalization in Verifiable C

We machine-check this proof in Coq, using the Verified Software Toolchain's *Verifiable C* program logic [3], which is already proved sound w.r.t. CompCert Clight [6]. We import from *Verified Functional Algorithms* the definition of purely functional search trees and their properties. Readers can find our Coq development online:

https://*github.com/PrincetonUniversity/VST/tree/master/wand_demo*

We formalize our proof using Verifiable C's interactive symbolic execution system in Coq [7]. Until now, Verifiable C had not included much proof theory for wand, except the basic WAND-ADJOINT. Now we add the PROOF RULES OF WAND-FRAME (see wandQ_frame.v) as derived lemmas from Verifiable C's basic separation logic. We use them in the Coq proof of $\mathrm{partial\_treebox\_rep}$'s properties (see §4.1 and bst_lemmas.v).

4.1 Separation logic predicates and properties for BST

Binary trees with keys and values are already formalized in VFA as an inductive data type in Coq. Here, we will formalize the separation logic predicate $\mathrm{treebox\_rep}$.

```
Fixpoint tree_rep (t: tree val) (p: val) : mpred :=
 match t with
 | E => !!(p=nullval) && emp
 | T a x v b =>
    EX pa:val, EX pb:val,
       data_at Tsh t_struct_tree (Vint (Int.repr (Z.of_nat x)),(v,(pa,pb))) p *
       tree_rep a pa * tree_rep b pb
 end.

Definition treebox_rep (t: tree val) (b: val) :=
 EX p: val, data_at Tsh (tptr t_struct_tree) p b * tree_rep t p.

Lemma treebox_rep_spec: forall (t: tree val) (b: val),
  treebox_rep t b =
  EX p: val,
  data_at Tsh (tptr t_struct_tree) p b *
  match t with
  | E => !!(p=nullval) && emp
  | T l x v r =>
      field_at Tsh t_struct_tree [StructField _key] (Vint (Int.repr (Z.of_nat x))) p *
      field_at Tsh t_struct_tree [StructField _value] v p *
      treebox_rep l (field_address t_struct_tree [StructField _left] p) *
      treebox_rep r (field_address t_struct_tree [StructField _right] p)
  end.
```

Instead of defining $\mathrm{treebox\_rep}$ directly as in (1), we first define $\mathrm{tree\_rep}$, then define $\mathrm{treebox\_rep}$ based on that. Finally, we prove that it satisfies the equalities in (1). We choose to do this because C functions for BST operations do not always take arguments with type (**struct** tree $*$ $*$) (or equivalently, treebox). For example, a look-up operation does not modify a BST, so it can just take a BST by an argument with type (**struct** tree $*$).

Here, val is CompCert Clight's value type; nullval has type val and represents the value of NULL pointer. The Coq type mpred is the type of Verifiable C's separation logic predicates. "&&", "$*$" and "EX" are notations for conjunction, separating conjunction and existential quantifiers in Verifiable C's assertion language. "!! _" is the notation that injects Coq propositions into the assertion language. The expression (Vint (Int.repr (Z.of_nat x)) injects a natural number x into the integers, then to a 32-bit integer,[3] then to CompCert Clight's value type, val.

Data_at is a mapsto-like predicate for C aggregate types. Here,

```
data_at Tsh t_struct_tree (Vint (Int.repr (Z.of_nat x)),(v,(pa,pb))) p
```

means that x, v, pa, pb are four fields of the "**struct** tree" stored at address p. Tsh means top share (full read/write permission). Verifiable C's field_at is like data_at but permits a field name such as ".right".

Our partial tree predicate partialT *does not care* how treebox_rep works internally. Thus, in defining the proof theory of partial trees, we'll parameterize over the treebox predicate. As claimed in §3, the soundness of rules (3c) (3d) and (3e) do not depend on the definition of $\mathrm{treebox\_rep}$; we prove them sound for arbitrary partial tree predicates. For the sake of space, we only list one of these three here.

```
Definition partialT (rep: tree val → val → mpred) (P: tree val → tree val) (p_root p_in: val) :=
    ALL t: tree val, rep t p_in -∗ rep (P t) p_root.

Definition partial_treebox_rep := partialT treebox_rep.

Lemma rep_partialT_rep: forall rep t P p q,   rep t p * partialT rep P q p ⊢ rep (P t) q.
  Proof. intros. exact (wandQ_frame_elim _ (fun t => rep t p) (fun t => rep (P t) q) t). Qed.
```

As described in §3, we define $\mathrm{Mapbox\_rep}$ based on $\mathrm{treebox\_rep}$, Abs and $\mathrm{SearchTree}$; and Abs and $\mathrm{SearchTree}$ are already defined in VFA. Similarly, we define $\mathrm{Map\_rep}$ based on $\mathrm{tree\_rep}$; application of it can be found in §4.3.

4.2 C program specification and verification

Specification and Coq proof goal. Verifiable C requires users to write C function specification in a canonical form.

The following is the specification of C function insert. The WITH clause there says that this specification is a parameterized Hoare triple—that is, for any p0, x, v, m0, this specific triple is valid. The brackets after PRE hold the C argument list. CompCert Clight turns every C variable into an identifier in the Clight abstract syntax tree defined

[3] Mapping $\mathbb{Z}$ to $\mathbb{Z} \mod 2^{32}$ is not injective; in a practical application the client of this search-tree module should prove that $x < 2^{32}$.

in Coq. In this argument list, _p is the identifier for C variable p, etc. The brackets after POST hold the C function return type.

```
Definition insert_spec :=
 DECLARE _insert
  WITH p0: val, x: nat, v: val, m0: total_map val
  PRE [ _p OF (tptr (tptr t_struct_tree)), _x OF tint, _value OF (tptr Tvoid) ]
    PROP()
    LOCAL(temp _p p0; temp _x (Vint (Int.repr (Z.of_nat x))); temp _value v)
    SEP (Mapbox_rep m0 p0)
  POST [ Tvoid ]
    PROP() LOCAL() SEP (Mapbox_rep (t_update m0 x v) p0).
```

Both precondition and postcondition are written in a PROP/LOCAL/SEP form. PROP clauses are for program-variable-irrelevant pure facts; there happen to be none here. LOCAL clauses talk about the values of program variables. For example, temp _p p0 says $[\![\mathsf{p}]\!] = p_0$. SEP clauses are separating conjuncts. Verifiable C requires users to isolate programs variables in their assertions—SEP conjuncts do not refer directly to C program variables—so we use LOCAL clauses to connect program variables to PROP and SEP clauses.

Theorem insert_body. The C function implements its functional specification, insert_spec.

Reduce to implementation correctness. We split the program-correctness proof into an implementation correctness proof (the C program refines a functional algorithm) and an algorithm correctness proof. To connect these, we prove (in verif_bst.v):

Lemma insert_concrete_to_abstract: LOW-LEVEL SEPARATION-LOGIC SPECIFICATION (Fig. 4d) implies HIGH-LEVEL SEPARATION-LOGIC SPECIFICATION (Fig. 4a).

We state this theorem as an implication between Hoare triples that are derived from the two function-specifications. Each Hoare triple has the form semax Δ P c Q, representing the judgment $\Delta \vdash \{P\}c\{Q\}$ in Verifiable C, where Δ records information like C types of C program variables. The postcondition Q is a quadruple: normal postcondition, break condition, continue condition, return condition. In this lemma—about a function body that must return—only the return condition is nontrivial.

Proof. This lemma takes only a few lines to prove:

```
rewrite !Mapbox_rep_unfold. Intros t0.
apply (semax_post'' (PROP () LOCAL () SEP (treebox_rep (insert x v t0) p0))); auto.
Exists (insert x v t0). entailer!.
split; [apply insert_relate | apply insert_SearchTree]; auto.
```

The first tactic "rewrite !Mapbox_rep_unfold" unfolds the definition of $\mathrm{Mapbox\_rep}$ and gives us the proof goal in Fig. 4b. Then we use Verifiable C's tactic "Intros t0" to extract existentially quantified variables and related pure facts from the precondition to Coq assumptions (see Fig. 4c).

The lemma semax_post'' is a special form of the rule of consequence. Applying that leaves us two proof goals, see Fig. 4d and Fig. 4e. The former is the low level

(a) High-level spec. (b) After unfold. (c) After Intros.

(d) Low-level spec. (e) Entailment. (f) After Entailer!.

Fig. 4: Coq proof goals

specification—the premise of this lemma, *implementation correctness*. The latter can be easily solved: Verifiable C provides "Exists" to instantiate the existentially quantified variables on the right side of entailments. Verifiable C also provides "entailer!", an automatic simplifier for separation logic entailments. These two tactics leave the proof goal in Fig. 4f—*the algorithm correctness*. That goal can be proved by two theorems about pure-functional BST insert imported from VFA.

```
Theorem body_insert: semax_body Vprog Gprog f_insert insert_spec.
 (* The C function f_insert (Fig. 1) implements its specification *)
Proof.
  start_function.
  apply insert_concrete_to_abstract; intros.
  . . . (* 62 lines of forward proof in separation logic *)
Qed.
```

Implementation correctness. After we apply the insert_concrete_to_abstract lemma, the rest of the proof is symbolic execution in separation logic. It is here that we use the PROOF RULES OF WAND-AS-FRAME specialized as equations (3a)–(3e).

From the proof goal in Fig. 4d, we apply the forward_loop tactic with our loop invariant (§3.4). This leaves two subgoals:

1. The function precondition implies the loop invariant (easy, by instantiating the three existentials, applying the entailment solver, and using (3c)).
2. The loop body preserves the invariant. The main structure of this Coq proof is very similar to the decorated program shown in Fig. 3. This is done by 13 steps of forward symbolic execution (e.g., simple invocations of the forward tactic and the forward_if tactic provided by Verifable C) interspersed with 35 lines of interactive proofs of side conditions, introductions of existentials that appear in preconditions, and so on. These proofs use equations (3a)–(3e).

The forward tactic generates strongest postconditions of assignment commands. For example, we omitted an assertion after line 16 in Fig. 3 for simplicity. The tactic forward generates that assertion, so we never need to write it explicitly. In this case, it adds a conjunct $[\![\mathsf{y}]\!] = x_0$ to the assertion in line 15.

The forward_if tactic generates preconditions for the then and else branches. For example, we omit an assertion before line 18 in Fig. 3 for conciseness. The tactic forward_if generates that assertion. Specifically, since we know $[\![\mathsf{x}]\!] = x$ and $[\![\mathsf{y}]\!] = x_0$ before that **if** command, forward_if adds $x < x_0$ to the precondition of if-then branch.

In summary, we do not need to manually type those long assertions in our interactive proof. Verifiable C generates most of them for us. We only provide function pre/postcondition and the loop invariant.

4.3 Other data structures, programs and proofs

Magic-wand-as-frame is a pretty flexible proof technique. We briefly introduce some other possibilities in magic-wand-as-frame proofs here. Interested readers can download our Coq development for more details.

Alternative magic-wand-as-frame proofs for BST insert. Universal quantifiers are not necessary for magic-wand-as-frame proofs. In the BST insert example, we can also use

$$\exists t\ p.\ [\![\mathsf{p}]\!] = p\ \wedge\ [\![\mathsf{x}]\!] = x\ \wedge\ [\![\mathsf{v}]\!] = v\ \wedge \\ \mathrm{treebox\_rep}(t, p)\ *\ (\mathrm{treebox\_rep}(\mathrm{ins}(t, x, v), p) \mathrel{-\!\!*} \mathrm{treebox\_rep}(\mathrm{ins}(t_0, x, v), p_0))$$

as loop invariant. The proofs are very similar except that we can use WAND-FRAME rules instead of WANDQ-FRAME rules to generate properties of partial tree predicates.

QUANTIFIER-FREE PROOF RULES OF WAND-FRAME(wand_frame.v) :

WAND-FRAME-INTRO: $Q \vdash P \mathrel{-\!\!*} P * Q$

WAND-FRAME-ELIM: $P * (P \mathrel{-\!\!*} Q) \vdash Q$

WAND-FRAME-VER: $(P \mathrel{-\!\!*} Q) * (Q \mathrel{-\!\!*} R) \vdash P \mathrel{-\!\!*} R$

WAND-FRAME-HOR: $(P_1 \mathrel{-\!\!*} Q_1) * (P_2 \mathrel{-\!\!*} Q_2) \vdash P_1 * P_2 \mathrel{-\!\!*} Q_1 * Q_2$

Other BST operations. We also verify C implementations of BST delete and look-up operation with the magic-wand-as-frame technique. In the verification of BST delete, we also use $\mathrm{partial\_treebox\_rep}$ to describe partial trees and use rules (3a-3e) to complete the proof. In the verification of BST look-up, we define $\mathrm{partial\_tree\_rep}$ using parameterized partialT and prove similar proof rules for it.

```
Definition partial_tree_rep := partialT tree_rep.
```

Especifically, we get the counterparts of (3c-3e) for free because we have already proved them for general partialT predicates. Proofs of the other two are also very straightforward using WANDQ-INTRO.

Another data structure: linked list. We also use magic-wand-as-frame to verify linked list append (see verif_list.v). In that proof, we use the following separation logic predicates and proof rules (see list_lemmas.v). These proof rules are direct instances of WANDQ-FRAME rules. Here, we use $l_1 l_2$ to represent the list concatenation of l_1 and l_2.

$$
\begin{array}{ll}
p \rightsquigarrow [] & =_{\mathrm{def}}\ p = \mathrm{null} \wedge \mathrm{emp} \\
p \rightsquigarrow (h :: t) & =_{\mathrm{def}}\ p.\mathsf{head} \mapsto h\ *\ \exists q.\ p.\mathsf{tail} \mapsto q\ *\ q \rightsquigarrow t \\
p \overset{l}{\rightsquigarrow\!\!\!\shortmid} q & =_{\mathrm{def}}\ \forall l'.\ (q \rightsquigarrow l'\ \mathrel{-\!\!*}\ p \rightsquigarrow ll')
\end{array}
$$

$$
p.\mathsf{head} \mapsto h\ *\ p.\mathsf{tail} \mapsto q\ \vdash\ p \overset{[h]}{\rightsquigarrow\!\!\!\shortmid} q \qquad \mathrm{emp}\ \vdash\ p \overset{[]}{\rightsquigarrow\!\!\!\shortmid} p
$$

$$
p \overset{l_1}{\rightsquigarrow\!\!\!\shortmid} q\ *\ q \rightsquigarrow l_2\ \vdash\ p \rightsquigarrow l_1 l_2 \qquad p \overset{l_1}{\rightsquigarrow\!\!\!\shortmid} q\ *\ q \overset{l_2}{\rightsquigarrow\!\!\!\shortmid} r\ \vdash\ p \overset{l_1 l_2}{\rightsquigarrow\!\!\!\shortmid} r \tag{4}
$$

5 Magic-wand-as-frame vs. traditional proofs

We have used magic wand to define partial tree (tree-with-a-hole) predicates and list segment (list-with-a-hole) predicates. Berdine *et al.* [5] first defined *list segments* and demonstrated a proof of imperative list append; Charguéraud defined tree-with-holes for a proof of BST operations.

These authors defined partial tree (and also list segment) by an explicit inductive definition, roughly as follows:

Partial trees:

$$P = H\ \mid\ L(P, k, v, t_2)\ \mid\ R(t_1, k, v, P)$$

Representation predicates for partial trees:

$$
\begin{array}{l}
\mathrm{partial\_treebox\_rep}^{\mathrm{R}}(H, r, i)\ =_{\mathrm{def}}\ r = i \wedge \mathrm{emp} \\
\mathrm{partial\_treebox\_rep}^{\mathrm{R}}(L(P, k, v, t_2), r, i)\ =_{\mathrm{def}} \\
\quad \exists q.\ r \mapsto q\ *\ q.\mathsf{key} \mapsto k\ *\ q.\mathsf{value} \mapsto v\ * \\
\quad\quad \mathrm{partial\_treebox\_rep}^{\mathrm{R}}(P, q.\mathsf{left}, i)\ *\ \mathrm{treebox\_rep}(t_2, q.\mathsf{right}) \\
\mathrm{partial\_treebox\_rep}^{\mathrm{R}}(R(t_1, k, v, P), r, i)\ =_{\mathrm{def}}\ \ldots
\end{array}
$$

That is: a partial tree is either one single hole or a combination of a partial tree and a complete tree; the partial tree can act as either the left subtree or the right subtree. And $\mathrm{partial\_treebox\_rep}^{\mathrm{R}}$ is defined as a *recursive* predicate over that structure.

5.1 One comparison

With this alternative definition, proof rules (3a)–(3e) are still sound and our proof in Fig. 3 still holds. However, our magic wand approach is better than that in three aspects.

Parameterized definition and proofs. Using magic wand, we can define partialT as a parameterized predicate for partial trees and proof rules (3c)–(3e) are sound in that parameterized way. Both $\mathrm{partial\_treebox\_rep}$ and $\mathrm{partial\_tree\_rep}$ are its instances.

Domain-specific theories for free. When a partial tree is defined as a function from trees to trees, we get the definition of "filling the hole in P with tree t" and "shrinking the hole in P_1 with another partial tree P_2" for free. They are just $P(t)$ and $P_1 \circ P_2$. In contrast, when partial trees are defined as a Coq inductive type, we would have to define these two combinators by Coq recursive functions and we would have to prove the following properties by induction:

$$(P_1 \circ P_2)(t) = P_1(P_2(t)) \qquad P_1 \circ (P_2 \circ P_3) = (P_1 \circ P_2) \circ P_3$$

Avoiding brittle and complex separation logic proofs. Using magic wand and quantifiers, rule (3d) and (3e) are direct corollaries of WANDQ-FRAME rules. However, proving them from recursively defined $\mathrm{partial\_treebox\_rep}^{\mathrm{R}}$ needs induction over the partial tree structure.

In some situation, these induction proofs can be very complicated and even annoying to formalize in Coq. The separation logic predicate for C aggregate types is such an example. The data_at predicate is already dependently typed. Proof rules that substitute a single field's data are now described by magic-wand-involved expressions. Their soundness proofs are significantly shorter (although still quite long) than using hole-related predicates.

Even worse, those inductive proofs are actually very brittle beside their length and complexity. Using linked-list predicates as an example, different authors had proposed different recursive definitions for list segments. Here is Smallfoot's [5] definition:

$$\begin{aligned}
&p \overset{[]}{\rightsquigarrow} q \;\; =_{\mathrm{def}} \;\; p = q \wedge \mathrm{emp} \\
&p \overset{(h::t)}{\rightsquigarrow} r \;\; =_{\mathrm{def}} \exists q.\, p \neq r \;\wedge\; p.\mathsf{head} \mapsto h \;*\; p.\mathsf{tail} \mapsto q \;*\; q \overset{t}{\rightsquigarrow} r \\
&p \rightsquigarrow l \;\; =_{\mathrm{def}} \;\; p \overset{l}{\rightsquigarrow} \mathrm{null}
\end{aligned}$$

And here is the definition from Charguéraud [8]:

$$\begin{aligned}
&p \overset{[]}{\rightsquigarrow} q \;\; =_{\mathrm{def}} \;\; p = q \wedge \mathrm{emp} \\
&p \overset{(h::t)}{\rightsquigarrow} r \;\; =_{\mathrm{def}} \;\; \exists q.\, p.\mathsf{head} \mapsto h \;*\; p.\mathsf{tail} \mapsto q \;*\; q \overset{t}{\rightsquigarrow} r \\
&p \rightsquigarrow l \;\; =_{\mathrm{def}} \;\; p \overset{l}{\rightsquigarrow} \mathrm{null}
\end{aligned}$$

These two definitions look similar, but their proof theories are surprisingly different. Proof rules in (4) are unsound with respect to Smallfoot's definition but sound with respect to Charguéraud's definition. Specifically, SmallFoot's list segment only satisfies weaker rules like the following one:

$$p \overset{l_1}{\rightsquigarrow} q \ * \ q \overset{l_2}{\rightsquigarrow} r \ * \ r \rightsquigarrow l_3 \ \vdash \ p \overset{l_1 l_2}{\rightsquigarrow} r \ * \ r \rightsquigarrow l_3$$

5.2 Another comparison

In the execution of BST insert's loop body, the memory that the magic wand expression $\mathrm{partial\_treebox\_rep}(P, p_0, p)$ describes is never touched by any C command. Also, this expression is preserved as a separating conjunct in the assertions in the proof until getting merged with another conjunct in the end. Specifically, it gets merged with another $\mathrm{partial\_treebox\_rep}$ predicate by rule (3e) in two normal branches and it gets merged with a $\mathrm{treebox\_rep}$ by rule (3d) in two **return** branches. In other words, although we do not explicitly use the frame rule in the proof, this magic wand expression acts as a frame. Thus we call our proof a magic-wand-as-frame verification.

The proof theory of magic wand supports this conjuncts-merging quite well. The derived rule WANDQ-ELIM enables us to fill the hole of a partial data structure and get a complete one. The rule WANDQ-VER enables us to shrink the holes of partial data structures. The rule WANDQ-HOR simply merges two holes into a larger one. The diagrams below illustrate these merging operations.

Fill the hole: 

Vertical composition: 

Horizontal composition: 

In contrast, the recursively defined $\mathrm{partial\_treebox\_rep}^{\mathrm{R}}$ reveals more information about that partial tree but offers less support for merging. For example, we know that

$$\mathrm{partial\_treebox\_rep}^{\mathrm{R}}(L(P, x, v, t), r, i) \ \vdash \ \exists p.\ r \mapsto p \ * \ p.\mathsf{key} \mapsto x \ * \ \top$$

But we cannot prove any corresponding property about $\mathrm{partial\_treebox\_rep}$.

After all, $\mathrm{partial\_treebox\_rep}$ is a weaker predicate—we can even prove:

$$\mathrm{partial\_treebox\_rep}^{\mathrm{R}}(P, r, i) \vdash \mathrm{partial\_treebox\_rep}(P, r, i)$$

But that magic wand expression precisely reveals the properties that we need in verification: the hole in it can be filled with another tree and it can also get merged with aother partial tree.

6 Related work and conclusion

Previous work with magic wand: Hobor and Villard [11] use magic wand in their ramification theory in graph algorithm verification. Their proof rule RAMIF can be treated as a special instance of magic-wand-as-frame proof. RAMIF can be proved by WAND-FRAME-INTRO, WAND-FRAME-ELIM, FRAME and Hoare logic's consequence rule.

$$\text{RAMIF}\ \frac{\{L\}\,c\,\{L'\} \qquad G \vdash L * (L' \mathrel{-\!\!*} G') \qquad \mathrm{FV}(L' \mathrel{-\!\!*} G') \cap \mathrm{ModV}(c) = \emptyset}{\{G\}\,c\,\{G'\}}$$

Iris has used magic wand heavily since Iris 3.0 [12]. They use magic wand and weakest-precondition (wp) to define their Hoare triple:

$$\{P\}c\{Q\} =_{\mathrm{def}} \quad (\vdash P \mathrel{-\!\!*} \mathrm{wp}(Q))$$

In principle, they do not limit the use of magic wand in a structural way. They develop *Iris Proof Mode* [13] for proving such separation logic entailments in Coq, which simplifies the process of applying the adjoint property in the object language.

Charguéraud [8] also mentions in his paper that if the purpose of a partial tree is to fold back with the *original* subtree (e.g. in BST look-up), magic wand can be used to describe that piece of memory. Our method shows that *even if the subtree is modified,* we can use a magic wand expression to describe a partial tree.

Separation logic for trees and lists: SmallFoot [5] verifies a shape analysis of a few linked list operations and tree operations. Charguéraud [8] formalizes a series of separation logic verifications for high order linked lists and trees. They use recursively defined list segment and tree-with-a-hole instead of magic wand. We have discussed their work in §5. Chlipala's Bedrock paper presents a proof of imperative list append [9, Figure 2], in a different style from our proof. He uses a "double-barreled" loop invariant with both pre- and postconditions for the loop, and uses neither list-segments nor magic wand. However, the double-barreled loop invariant is not a panacea: if applied to our BST insert of Fig. 3, without the use of any program transformations to turn it into some sort of tail recursion, one would require some sort of tree-with-a-hole, and we would recommend the use of wand-frames.

In this paper, we demonstrate a Coq formalized verification of BST insert. Compared to the work of previous authors, our contributions are:

1. We present a new proof technique: magic-wand-as-frame, with its four rules (intro, elim, ver(tical composition), hor(izontal composition)).
2. We discover the power of magic wand in merging partial data structure together.
3. We show that defining magic-wand-involved predicates for partial data structure permits elegant soundness proofs of their critical properties. It avoids us writing brittle, less general and complex inductive proofs.
4. We formalize our proof in Coq and that formalization successfully uses those projects developed by other authors.
5. Thanks to CompCert, Verifiable C and VFA, our Coq proof is actually an end-to-end correctness proof from top level specification to compiled assembly code.

References

1. Andrew W. Appel. Modular verification for computer security. In *CSF 2016: 29th IEEE Computer Security Foundations Symposium*, pages 1–8, June 2016.
2. Andrew W. Appel. *Verified Functional Algorithms*, volume 3 of *Software Foundations*. 2017.
3. Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge, 2014.
4. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 97–109. Springer, 2004.
5. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects*, pages 115–135, 2005.
6. Sandrine Blazy and Xavier Leroy. Mechanized semantics for the clight subset of the c language. *Journal of Automated Reasoning*, 43(3):263–288, Oct 2009.
7. Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning*, (to appear), 2018.
8. Arthur Charguéraud. Higher-order representation predicates in separation logic. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 3–14. ACM, 2016.
9. Adam Chlipala. The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier. In *ICFP'13: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, September 2013.
10. Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *42nd ACM Symposium on Principles of Programming Languages (POPL'15)*, pages 595–608. ACM Press, January 2015.
11. Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. In *POPL'13: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 523–536. ACM, January 2013.
12. Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *European Symposium on Programming*, pages 696–723. Springer, 2017.
13. Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 205–217. ACM, 2017.
14. John Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002: IEEE Symposium on Logic in Computer Science*, pages 55–74, July 2002.
15. J. K. Rowling. *Harry Potter and the Philosopher's Stone*. Bloomsbury Childrens, London, 1997.