Int J Softw Tools Technol Transfer (2015) 17:659–676
DOI 10.1007/s10009-014-0310-9

Solving the VerifyThis 2012 challenges with VeriFast

Bart Jacobs · Jan Smans · Frank Piessens

Published online: 20 March 2014

Abstract We describe our experience solving the VerifyThis 2012 challenges with our program verification tool VeriFast, including detailed explanations of our solutions. We also describe some alternative solutions that we developed after the competition. VeriFast is a modular verifier that takes Java or C source code annotated with function/method specifications written in a variant of separation logic, and verifies that the code complies with the annotations through symbolic execution.

Keywords Formal verification · Program correctness · Separation logic

1 Introduction

The first two authors of this article participated as the VeriFast team in the VerifyThis 2012 competition [6] held during the Formal Methods 2012 conference on 30 August 2012 in Paris, France. We used the VeriFast program verification tool [12], a research prototype that we have been developing since 2008. It is a modular verifier that takes Java or C source code annotated with function/method specifications written in a variant of separation logic, and verifies that the code complies with the annotations through symbolic execution.

In this article, we describe our solutions to the competition challenges. In the process of explaining our solutions, we describe several VeriFast features that we have not yet described in earlier published work, including partial fixpoint function applications, parameterized lemma function types, and quantifiers.

B. Jacobs (✉) · J. Smans · F. Piessens
iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium
e-mail: bart.jacobs@cs.kuleuven.be

VeriFast distributions for Windows, MacOS X, and Linux can be downloaded from the VeriFast website [7]. The tool ships with a test suite that includes the solutions described here in the directory examples/fm2012.

While a tutorial and other materials on VeriFast are available via the website, this article intends to be accessible to readers new to VeriFast.

The remainder of this article is structured as follows. In Sect. 2, we provide an overview of the VeriFast approach. In Sects. 3, 4, 5, we describe our solutions to Challenges 1–3, respectively. We offer a conclusion in Sect. 6.

2 Background: the VeriFast approach

In this section, we give an overview of the VeriFast approach. For concreteness, we describe the approach as applied to the C programming language; most concepts carry over straightforwardly to the Java case.

VeriFast performs *modular* verification: each function is verified in isolation, using only the *contracts*, but not the bodies, of other functions. Function contracts are expressed in a variant of *separation logic* [10,13], a logic for reasoning about pointer-manipulating imperative programs. Separation logic is an extension of Hoare logic [5] where assertions are interpreted with respect to a *store* (which maps program variables to values) and a *heap* (a partial function from allocated addresses to values) rather than just a store. It introduces a number of additional operators into the syntax of assertions, the most important of which are **emp**, stating that the heap's domain is empty; the *points-to assertion* $\ell \mapsto v$, stating that the heap contains exactly the mapping of address ℓ to value v and is otherwise empty; and the *separating conjunction* $P * Q$, stating that the heap can be split into two disjoint subheaps such that assertion P holds for one and assertion Q holds for the other. The proof rule for heap mutation then becomes

$$\{\ell \mapsto \_\}\ [\ell] := v\ \{\ell \mapsto v\}$$

where $\ell \mapsto \_$ is shorthand for $\exists v.\ \ell \mapsto v$, and $[\ell] := v$ is the command that assigns the value v to the heap cell at address ℓ. This proof rule is applicable only if the heap contains exactly the heap cell ℓ and no others. Of course, in general, the heap contains multiple heap cells; to accommodate this, the proof rule can be lifted to larger heaps using the *frame rule*:

$$\frac{\{P\}\,c\,\{Q\}}{\{P * R\}\,c\,\{Q * R\}} \text{where } \mathsf{freevars}(R) \cap \mathsf{modifies}(c) = \emptyset$$

This rule states that if a command c runs without failure when started in a state satisfying assertion P and each post-state satisfies assertion Q, then it will also run without failure if started in a state obtained by enlarging the heap with a subheap that satisfies assertion R, and furthermore the command does not modify this subheap. [The side condition states that R does not mention any program variables modified by c. Note that heap locations (looked up in the heap) are not considered to be program variables (looked up in the store)].

Function contracts for functions that manipulate data structures may hide the internal shape of the data structure using an *abstract separation logic predicate* [11], which is essentially a named, parameterized separation logic assertion. Predicates may be recursive; their meaning is defined as the least fixpoint of their definitions. This is well-defined since recursive occurrences must occur in positive positions. Another way to look at this is to see a predicate definition as an inductive definition with a single inference rule. For example, the following separation logic predicate describes a null-terminated linked list that stores the mathematical list of values α:

$$\begin{aligned}&\textbf{predicate } \mathsf{list}(\ell, \alpha) \ \equiv \\ &\quad \ell = 0 \wedge \mathbf{emp} \wedge \alpha = \epsilon \\ &\quad \vee\ \exists v, t, \alpha'.\ \ell \mapsto v * \ell + 1 \mapsto t * \mathsf{list}(t, \alpha') \wedge \alpha = v \cdot \alpha'\end{aligned}$$

where $\cdot$ denotes prepending an element to a list.

Encoded into VeriFast syntax, this becomes:

```
struct node { int value; struct node *next };
/*@
predicate list(struct node *l; list<int>values)
   = l == 0 ?
        values == nil
    :
        l->value |-> ?v &*& l->next |-> ?t &*&
        malloc_block_node(l) &*&
        list(t, ?values0) &*&
        values == cons(v, values0);
@*/
```

VeriFast does not support disjunctions at the level of separation logic assertions, because this would entail the need for backtracking in the symbolic execution engine (see below), which we do not do for the sake of performance, predictability, and diagnosability. [However, it does support disjunctions at the level of boolean expressions, i.e. assertions which do not mention separation logic constructs, because these are passed to the underlying SMT solver (see below)]. Therefore, we use a conditional assertion (of the form `cond ? asn1 : asn2`) as the body of the predicate. We model the contents of the data structure using the mathematical inductive datatype `list<t>` defined as follows in the library that ships with VeriFast:

```
inductive list<t> = nil | cons(t, list<t>);
```

Separating conjunction is written as `&*&`. Points-to assertions are written as `|->`. The built-in predicate `malloc_block_node(l)` denotes the permission to call `free` on the pointer `l`; i.e. it encodes the fact that `l` was allocated using `malloc` (rather than being allocated on the stack, as a global variable, or as part of a larger object). VeriFast does not support general existential quantification; it only supports *pattern matching*: arguments of predicate assertions may be of the form `?x`, in which case the assertion is implicitly existentially quantified over `x`. This construct binds `x`; its scope includes subsequent separating conjuncts. For example, the variable `t` bound to the value of the `next` field of `l` in the assertion `l->next |-> ?t` above is used subsequently in `list(t, ?values0)`. This restricted form of existential quantification enables a simple algorithm for checking assertions (see below).

VeriFast verifies each function against its contract through *symbolic execution*. This is similar to what an interpreter does, except that *symbolic states* are used instead of concrete states. A single symbolic state may represent many (even infinitely many) concrete states, thus allowing a finite number of symbolic executions to "cover" all potentially infinitely many concrete executions. In VeriFast, a symbolic state consists of a *symbolic store*, a *symbolic heap*, and a *path condition*. The symbolic store maps each local variable that is currently in scope to its *symbolic value*, which is a term of first-order logic that may contain *symbols*. The *symbolic heap* is a multiset of *heap chunks* of the form $\mathsf{p}(a_1, \ldots, a_n)$ where p is a separation logic predicate name (either built-in or user-defined) and $a_1, \ldots, a_n$, the *chunk arguments*, are terms of first-order logic. For a given interpretation of the logical symbols that appear in the chunk arguments, a given symbolic heap represents all concrete heaps that satisfy the separation logic assertion obtained by taking the separating conjunction of the heap chunks, interpreted as predicate assertions. The full set of concrete states represented by a symbolic state is obtained by taking the union of the sets of represented concrete states for all logical symbol interpretations that satisfy the *path condition*, which is a set of formulae of first-order logic.

The core operations of symbolic execution are *assertion production* and *assertion consumption*. Producing an assertion that is a boolean expression evaluates the expression under the symbolic store to yield a logical formula and adds

this formula to the path condition. Producing a predicate assertion adds a corresponding chunk to the symbolic heap. Any existentially quantified chunk arguments are bound to fresh logical symbols. Producing a separating conjunction first produces the left-hand side, and then produces the right-hand side. Producing a conditional assertion forks the symbolic execution: on one branch, the condition is added to the path condition and the first branch's body is produced; on the other branch, the negation of the condition is added to the path condition and the second branch's body is produced.

The path condition is not actually maintained as a set of formulae; rather, during symbolic execution, VeriFast interacts with an SMT solver. The path condition corresponds to the state of the SMT solver. Adding a formula means "pushing" the formula into the SMT solver. If, after pushing a formula, the SMT solver reports that the resulting set of formulae is inconsistent, VeriFast aborts the current symbolic execution path, since it is infeasible, and continues with the next one. Aborting a symbolic execution path includes "popping" the state of the SMT solver to restore the state that existed at the most recent branch point.

Consuming an assertion that is a boolean expression evaluates the expression under the symbolic store to yield a formula and then asks the SMT solver to prove that formula from the path condition; if it fails, VeriFast reports an error. Consuming a predicate assertion looks for a matching chunk in the symbolic heap. If no match is found, an error is reported. Otherwise, existentially quantified variables in the predicate assertion are bound to the corresponding arguments of the matching chunk and the chunk is removed from the symbolic heap. Consuming a separating conjunction first consumes the left-hand side, and then the right-hand side. Consuming a conditional assertion proceeds analogously to production.

VeriFast verifies a function by first producing its precondition, then symbolically executing its body, and then consuming its postcondition. If any heap chunks are left in the symbolic heap after this last step, VeriFast reports that the function leaks heap chunks: this indicates a potential memory leak in the C program. Symbolic execution of an assignment to a struct field first consumes the heap chunk corresponding to the struct field, and then produces the same chunk with an updated value argument. When a struct s is allocated using `malloc`, for each field f of s a chunk $s\_f(\ell, v_f)$ is produced, where ℓ is a fresh symbol denoting the address where the struct was allocated, and v_f is a fresh symbol denoting the (unspecified) initial value of the field. In addition, a malloc_block_$s(\ell)$ chunk is produced, denoting that the struct was allocated using `malloc` and therefore may be freed using `free`. The points-to notation `l->next |-> ?t` in the example above is syntactic sugar for the predicate assertion `node_next(l, ?t)`.

Symbolic execution of function calls proceeds by first consuming the callee's precondition and then producing its postcondition. Notice that this approach implicitly applies separation logic's frame rule: any heap chunks that exist at the call site and that are not consumed by the precondition remain available to the caller in their original state. In our experience, this approach, involving mostly just pattern matching, performs much better than verification condition generation-based approaches where the frame condition must be encoded into the input to the SMT solver using quantifiers that range over all memory locations, stating that all locations not declared as modified by the callee remain unchanged.

Sometimes, the symbolic heap must be rewritten into an equivalent form. For example, if the symbolic heap contains just the chunk `list(l, values)`, and we wish to access the `value` field of the first node of `l`, we need to *open* the `list` chunk. This means replacing the chunk with its definition. The converse operation is *closing* a chunk, which consumes the chunk's definition and produces the chunk itself. In general, VeriFast does not open or close chunks automatically, again to preserve predictable and diagnosable performance. Rather, the user must insert **`open`** and **`close`** ghost commands into the code. However, if a predicate is marked *precise*, by inserting a semicolon between its *input parameters* and its *output parameters*, VeriFast will attempt to automatically open and close it when appropriate, provided the values of the input arguments are known. A predicate is precise if, for a given valuation of the input arguments and for a given heap, there is at most one subheap and at most one valuation of the output arguments that satisfy the predicate.

For modeling and reasoning about data, such as the contents of the linked list in the example above, the VeriFast approach is oriented towards inductive datatypes and primitive recursive functions (called *fixpoint functions*) over these. An example of the former is the `list<t>` datatype, and an example of the latter is the `length` function, defined as follows in the library that ships with VeriFast:

```
fixpoint int length<t>(list<t> xs) {
    switch (xs) {
       case nil: return 0;
       case cons(x, xs0): return 1+length(xs0);
    }
}
```

Like in proof assistants that support primitive recursive functions such as Isabelle/HOL [9] and Coq [1], a fixpoint function's body must be a case split over one of its arguments (called its *inductive argument*), and each recursive call must structurally decrease this argument; i.e. the callee's value for this argument must be a component of the caller's. In the example, the recursive call `length(xs0)` is allowed because `xs0` is a component of `xs`. Inductive datatypes and fixpoint functions have a simple and clear mathematical meaning; they are expressive enough for most purposes; and we can offer well-defined and predictable automated reasoning for them: in principle, only *reduction* occurs automatically.

Reduction equates a fixpoint function call whose inductive argument is a constructor with the body of the corresponding case in the function's definition. VeriFast is purposely incomplete: it does not automatically perform case splits or induction. The goal is to preserve predictable performance. The reasoning is implemented by encoding the inductive datatypes and fixpoint functions into the background theory of the SMT solver using uninterpreted function symbols and axioms for injectiveness and disjointness (but not exhaustiveness) of constructors and for reduction of fixpoint functions.

While no automatic induction occurs for reasoning about inductive datatypes or separation logic predicates, manual induction is supported by writing *lemma functions*: a lemma function is like a regular C function, except that it is defined within an annotation and VeriFast checks that it has no side-effects and that it terminates. VeriFast supports a number of termination arguments; the most important ones are structural induction on an inductive argument and induction on the derivation of a separation logic predicate. The latter is based on the fact that each non-built-in chunk was produced from built-in chunks by a finite number of *close* operations. For example, an inductive proof of the fact that the length of a list is always nonnegative can be given in VeriFast as follows:

```
lemma void length_nonnegative<t>(list<t> xs)
    requires true;
    ensures 0 <= length(xs);
{
    switch (xs) {
        case nil:
        case cons(x, xs0):
            length_nonnegative(xs0);
    }
}
```

Then, to obtain this fact for a given specific inductive list value, one can either call this lemma explicitly as a ghost command, or one can declare the lemma as a **lemma_auto**, in which case the lemma is passed as an axiom to the SMT solver and applied automatically. This introduces a risk of degraded performance or nontermination of the SMT solver; we leave this responsibility to the user. Note: only autolemmas whose contract does not involve separation logic constructions are passed to the SMT solver; others are applied by the symbolic execution engine during chunk production and consumption.

As will be illustrated by the challenge solutions below, the VeriFast approach for verifying properties of an imperative program often comes down to using separation logic to establish a correspondence between the imperative program and a pure functional version of it, and then separately verifying the desired properties with respect to the pure functional program, the latter in ways very similar to how one develops theories in proof assistants like Coq and Isabelle/HOL. This stands in some contrast with approaches that are oriented less towards inductive datatypes and fixpoint functions and more towards quantifier-rich specifications, which tend to lead to more direct statements and proofs of the desired properties, but for which it might be harder to offer a predictable, diagnosable, well-performing proof authoring experience. We believe no one particular approach currently being proposed in the verification community is a clear winner, and each has important benefits and contributions, and as the community grows to understand better the various trade-offs involved we will learn how to combine the various approaches to best tackle particular types of problems.

3 Challenge 1: longest common prefix

3.1 Challenge

Essentially, the challenge was to specify and verify the functional correctness of the following C function, which, given an array of integers `a` of length `N`, returns the length of the longest common prefix of the subarrays starting at indices `x` and `y`.

```
int lcp(int *a, int N, int x, int y) {
    int l = 0;
    while (x + l < N && y + l < N
           && a[x + l] == a[y + l])
        l++;
    return l;
}
```

(We did not, either during or after the competition, attempt to solve the part marked *Advanced* of the challenge, which was to verify a program that computes the longest repeated substring of a given string.)

3.2 Competition solution 1

The first solution that we came up with during the competition is shown in Fig. 1.[1]

The solution consists of the C function definition given in the challenge (transformed slightly; see below), augmented with VeriFast annotations. These take the form of C comments marked with an `@` sign.

The specification of the C function consists of a precondition, given by the **requires** clause, and a postcondition, given by the **ensures** clause. The precondition states that the function must receive read permission to elements `0`, inclusive, through `N`, exclusive, of the array `a`, and that arguments `x` and `y` are within appropriate bounds. In VeriFast, a function's precondition must require access permissions for all

[1] The only differences with the actual submitted code are (1) that this version uses an improved notation for array permissions which was introduced in the December 2012 release of VeriFast, and (2) that we removed some ghost statements that turned out to be superfluous.

```
/*@
fixpoint int lcp_<t>(list<t> xs, list<t> ys) {
    switch (xs) {
        case nil: return 0;
        case cons(x, xs0): return
            switch (ys) {
                case nil: return 0;
                case cons(y, ys0): return x == y ? 1 + lcp_(xs0, ys0) : 0;
            };
    }
}
@*/

int lcp(int *a, int N, int x, int y)
    //@ requires [?f]a[0..N] |-> ?elems &*& 0 <= x &*& x < N &*& 0 <= y &*& y < N;
    //@ ensures [f]a[0..N] |-> elems &*& result == lcp_(drop(x, elems), drop(y, elems));
{
    int l = 0;
    for (;;)
        //@ requires [f]a[0..N] |-> elems &*& 0 <= l &*& x + l <= N &*& y + l <= N;
        /*@
        ensures
            [f]a[0..N] |-> elems &*& l - old_l == lcp_(drop(x + old_l, elems), drop(y + old_l, elems));
        @*/
        //@ decreases N - l;
    {
        if (!(x + l < N && y + l < N && a[x+l] == a[y+l])) {
            /*@
            if (x + l == N) { drop_length(elems); }
            else if (y + l == N) {
                assert drop(x + l, elems) == cons(_, _);
                drop_length(elems);
            } else {
                drop_n_plus_one(x + l, elems);
                drop_n_plus_one(y + l, elems);
            }
            @*/
            break;
        }
        //@ drop_n_plus_one(x + l, elems);
        //@ drop_n_plus_one(y + l, elems);
        l++;
    }
    return l;
}
```

Fig. 1 Our first competition solution to Challenge 1

memory locations that the function accesses, except for the ones that it allocates itself or for which it obtains permission through some other means (such as acquiring a mutual exclusion lock). In particular, to write to a memory location, full permission is required; to only read from a memory location, some *fractional permission* [2] is sufficient. A full permission can be split into a number of fractional permissions and later reassembled into a full permission. Each fractional permission has a *coefficient*, a rational number between 0, exclusive, and 1, inclusive. (A coefficient of 1 denotes a full permission.) The coefficient, if not 1, is written inside a pair of brackets preceding the permission. The notation `[?f]` denotes that an arbitrary fraction is sufficient, and binds the actual coefficient to ghost variable `f`.

The precondition also binds the contents of the specified array elements, represented as a value of the ghost type `list<int>`, to the ghost variable `elems`. The ghost variables bound in the precondition are in scope in the postcondition. This allows the postcondition to specify that the function returns the same fractional permission `f` that it received, and

that the return value of the function is the longest common prefix of the sublists of `elems` obtained by dropping the first `x` and `y` elements, respectively. The longest common prefix of two lists is specified using the fixpoint function `lcp_` defined recursively in the annotation preceding the C function. This function uses an extended **`switch`** notation to perform a case analysis on the `list` values passed in as arguments. Type `list` is defined inductively in the built-in VeriFast header file `list.h` as follows:

```
inductive list<t> =
    nil | cons(t head, list<t> tail);
```

A list of elements of type `t` is either empty (denoted by constructor `nil`) or non-empty, with a first element (called the *head*) of type `t` and a remainder (called the *tail*) of type `list<t>`. As another example of a fixpoint function, the function `drop` used in the postcondition is defined in `list.h` as follows:

```
fixpoint list<t> drop<t>(int n, list<t> xs) {
    switch (xs) {
        case nil: return nil;
        case cons(x, xs0): return
            n == 0 ? xs : drop(n - 1, xs0);
    }
}
```

Since the `list` type that we use to model the contents of the array is defined inductively, and the functions `lcp_` and `drop` used in the specification are defined recursively, a recursive implementation of the C function would have been closer to the specification and therefore easier to verify. However, the implementation given in the challenge is iterative. Applying the classical loop verification proof rule here, with a loop invariant, would be somewhat painful. However, an alternative proof rule for loops was proposed recently [14] that allows a loop to be verified as if it was a tail-recursive function. We have implemented this proof rule in VeriFast; specifically, for annotating a loop, the user can choose to provide a *loop contract* instead of a loop invariant. For example, the loop

```
while (C)
    //@ requires P;
    //@ ensures Q;
{ B }
```

is verified by VeriFast as if it was a call of a local function:

```
void iter()
    //@ requires P;
    //@ ensures Q;
{ if (C) { B iter(); } }
iter();
```

We have applied this feature in the solution. However, applying this feature means that VeriFast checks that the loop postcondition holds when the loop ends (i.e., when `C` is false). This means that the postcondition must follow from the conjunction of the precondition and the negation of the loop condition. In general, however, VeriFast cannot prove this implication without help. Helping VeriFast means inserting extra ghost statement annotations, such as lemma applications and case splits. But in the case of a regular **`while`** loop, there is no way for the user to insert ghost code between the start of the loop iteration and the exit from the loop. For our solution, to work around this problem we rewrote the given function body slightly: we moved the condition check into the body of the loop using an **`if`** statement and a **`break`** statement; this gave us the opportunity to insert ghost statements between the point where the negation of the loop condition is established and the point where the loop is exited (and where VeriFast checks that the postcondition holds). Schematically, we rewrote our loop into the form

```
for (;;)
    //@ requires P;
    //@ ensures Q;
{
    if (!C) {
        //@ ... ghost code ...
        break;
    }
    B
}
```

which is verified as if it was of the form

```
void iter()
    //@ requires P;
    //@ ensures Q;
{
    if (!C) {
        //@ ... ghost code ...
        return;
    }
    B
    iter();
}
iter();
```

The loop contract in our solution is similar to the function contract, except that the postcondition states that the loop "returns" the length of the longest common prefix of the subarrays starting at `x + old_l` and `y + old_l`, where `old_l` denotes the value of variable `l` at the start of the current loop "call" (i.e. the imaginary call of the tail-recursive function). The "return value" is embodied in the difference between the new value of `l` and the old value of `l`. Notice that this contract is indeed appropriate: (a) it is established by the base case, i.e. when the loop exits; (b) given that the contract holds for a "tail-recursive call" of the loop, it holds for the "caller"; (c) the loop contract implies the function contract, considering that the initial value of `l` is 0 and that the function returns the final value of `l`.

Proving the cases (a) and (b) required a number of ghost statements. For case (a), the case distinction that happens in the loop condition had to be repeated in the annotations, and for each case, additional ghost code was

necessary to establish the loop postcondition. Specifically, if `x + l == N`, the postcondition follows from the fact that `drop(x + l, elems)` equals `nil`, but the latter fact is not immediately obvious to VeriFast since it requires an inductive proof, which VeriFast does not attempt to construct. The solution is to call a lemma function that provides this fact in its postcondition. Such a lemma function, called `drop_length`, is declared in header file `list.h` as follows:

```
lemma void drop_length<t>(list<t> xs);
    requires true;
    ensures drop(length(xs), xs) == nil;
```

The second case of the loop condition, where `x + l != N` but `y + l == N`, can be dealt with similarly, by another call of `drop_length` to show that `drop(y + l, elems)` equals `nil`, except that we also need to perform a case split on `drop(x + L, N)`, since otherwise VeriFast will not reduce the first **`switch`** expression in the body of `lcp_`. (VeriFast never performs case splits implicitly, since this can easily lead to performance degradation and even non-termination of the verifier.) This case split is provoked by asserting that `drop(x + l, elems)` matches the pattern `cons(_, _)`. Such a pattern-matching assertion is checked by doing a case split and checking that all non-matching cases lead to absurdity.

The final subcase of case (a) is the case where the corresponding array elements are unequal. From this fact, which is expressed in terms of the `elems` list as `nth(x + l, elems) != nth (y + l, elems)`, VeriFast must derive that the `lcp_` function goes into the last branch of its body. For this to happen, VeriFast must see that `drop(x + l, elems)` and `drop(y + l, elems)` are nonempty (i.e. they match `cons(_, _)`) and that their first elements are `nth(x + , elems)` and `nth(y + l, elems)`. This is what the calls of the `drop_n_plus_one` lemma provide, which is declared in `list.h` as follows:

```
lemma void drop_n_plus_one<t>
       (int n, list<t> xs);
    requires 0 <= n &*& n < length(xs);
    ensures drop(n, xs) ==
        cons(nth(n, xs), drop(n + 1, xs));
```

For the case (b) where the loop does not end, i.e. the recursive case, we need to prove that the call of `lcp_` also hits its recursive case. Similarly to the last subcase of (a), this is accomplished with two calls of lemma `drop_n_plus_one`.

3.3 Competition solution 2

The second solution we came up with during the competition is shown in Fig. 2.[2]

As in our first solution, the precondition of `lcp` demands read permission to the first `N` elements of `a`, and requires that `x` and `y` lie within proper bounds. The postcondition returns the requested permissions to the caller. However, the relation between the function's return value and the elements of the array is described in a more *declarative* style. More specifically, the postcondition states (1) that the return value lies within proper bounds, (2) that the subarrays of `a` starting at `x` and `y` share a common prefix of length `result`, and (3) that no longer common prefix exists. In the first solution, these properties can be derived from the definition of the function `lcp_` (for example, via a lemma).

The second property of `result` is expressed via a universal quantifier: for all integers `i` in the range 0 (inclusive) to `result` (exclusive), the element at index `x + i` is equal to the one at `y + i`. Interestingly, this form of bounded quantification is not built into VeriFast. Instead, the specification library `quantifiers.gh` partially shown in Fig. 3 declares the higher-order function `forall`. This function takes a list `vs` and a property `p` (of type **`fixpoint`**`(t, bool)`, which is VeriFast notation for the function type `t` → **`bool`** of functions that take a value of type `t` as an argument and return a value of type **`bool`**) as arguments, and returns whether each element of `vs` satisfies `p`. In addition to the function itself, the library provides two lemmas, `get_not_forall_witness` and `apply_forall`, that can be used by clients to reason about `forall`. The former lemma returns an element of `vs` that does not satisfy `p`, provided `forall(vs, p)` does not hold. The latter lemma states that each member of the list satisfies `p` if `forall(vs, p)` holds.

One might reasonably try to express the fact that the sublists of list `elems` at offsets `x` and `y` have a common prefix of length `result` through an expression like

```
forall(range(0, result), fixpoint(int i)
 {return nth(x + i elems)== nth(y+ i,elems);})
```

where **`fixpoint`**`(T x){`**`return`** `E;}` would denote an anonymous fixpoint function that maps arguments `x` to results `E`. However, VeriFast currently does not support anonymous fixpoint functions. Therefore, the quantifier in the specification of `lcp` uses the fixpoint `sublists_equal_at_offset` to state the desired property. This function is partially applied to `elems`, `x` and `y` to obtain a function from integers to booleans (as required by `forall`).

A disadvantage of providing quantification as a library is that VeriFast is often unable to automatically discharge proof obligations involving quantifiers. In such cases, the tool relies on the user to complete the proof (by inserting ghost code, which typically calls lemmas). For example, consider the body of the loop in Fig. 2. The ghost statements in the

[2] The only differences with the actual submitted code are (1) that this version is written in C instead of Java, (2) that it uses the improved notation for array permissions as before, and (3) that quantifiers are expressed via `forall` instead of `forall_nth`.

```
/*@
fixpoint bool sublists_equal_at_offset(list<int> elems, int x, int y, int i) {
    return nth(x + i, elems) == nth(y + i, elems);
}
@*/

int lcp(int *a, int N, int x, int y)
    //@ requires [?f]a[0..N] |-> ?elems &*& 0 <= x &*& x < N &*& 0 <= y &*& y < N;
    /*@ ensures [f]a[0..N] |-> elems &*&
            0 <= result && result <= N - x && result <= N - y &&
            forall(range(0, result), (sublists_equal_at_offset)(elems, x, y)) == true &*&
            x + result == N || y + result == N || nth(x + result, elems) != nth(y + result, elems); @*/
{
    int l = 0;
    while (x + l < N && y + l < N && a[x + l] == a[y + l])
        /*@ invariant [f]a[0..N] |-> elems &*& 0 <= l &*& x + l <= N &*& y + l <= N &*&
                forall(range(0, l), (sublists_equal_at_offset)(elems, x, y)) == true; @*/
        //@ decreases N - l;
    {
        l++;
        /*@
        if (!forall(range(0, l), (sublists_equal_at_offset)(elems, x, y))) {
            int i = get_not_forall_witness(range(0, l), (sublists_equal_at_offset)(elems, x, y));
            if (i < l - 1)
                apply_forall(range(0, l - 1), (sublists_equal_at_offset)(elems, x, y), i);
        }
        @*/
    }
    return l;
}
```

Fig. 2 Our second competition solution to Challenge 1

Fig. 3 Part of `quantifiers.gh`, a specification library for quantification. Fixpoint function `mem(x, xs)` checks if list `xs` contains element `x`

```
fixpoint bool forall<t>(list<t> xs, fixpoint(t, bool) fp) {
    switch (xs) {
        case nil: return true;
        case cons(x, xs0): return fp(x) && forall(xs0, fp);
    }
}

lemma t get_not_forall_witness<t>(list<t> vs, fixpoint(t, bool) p);
    requires !forall(vs, p);
    ensures mem(result, vs) == true &*& !p(result);

lemma void apply_forall<t>(list<t> vs, fixpoint(t, bool) p, t x);
    requires forall(vs, p) == true &*& mem(x, vs) == true;
    ensures p(x) == true;
```

body are crucial for VeriFast to be able to deduce that the loop preserves the loop invariant. In particular, they prove that the quantifier holds for `range(0, l)`. They do so by contradiction: assume that the quantifier does not hold, and derive absurdity. The absurdity proof in the solution first obtains the index of an element for which the quantifier does not hold, and then proceeds by case analysis on whether the element is the last one or not. If not, absurdity follows from the fact that the quantifier holds for `range(0, l -1)`; otherwise, it follows from the fact that the loop condition holds.

VeriFast also supports *native* quantifiers, in the form of `forall_(int i; E)` expressions. By native, we mean that these quantifiers are encoded as quantifiers in the underlying SMT solver (instead of as applications of the uninterpreted function symbol `forall`). After the competition, we also solved the first challenge making use of such native quantifiers. In particular, the call of `forall` in the loop invariant in Fig. 2 is replaced in this solution by:

```
forall_(int i; i < 0 || l <= i ||
    sublists_equal_at_offset(elems, x, y, i))
```

and similarly in the postcondition. The key advantage of native quantifiers is that the SMT solver can automatically discharge many proof obligations involving such quantifiers, and therefore that the developer needs to provide fewer annotations to help the tool complete the proof. For example, the ghost statements in the body of the loop in Fig. 2 can be omitted when using native quantifiers. However, in our experience relying on native quantifiers also has a number of disadvantages. In particular, the time needed by the SMT solver to discharge proof obligations involving general quantifiers (and even whether it can discharge the proof obligation at all) can be extremely sensitive to minor changes to the program and the specifications. As a consequence, it can be hard to diagnose (and remedy) the exact cause of a verification failure. For example, when we tried to make use of native quantifiers to solve the first challenge during the competition (the postcondition was expressed directly in terms of function `nth`, without using auxiliary function `sublists_equal_at_offset`), verification failed; later diagnosis indicated that the SMT solver's heuristics for instantiating quantifiers failed to produce the required instantiations. Rewriting the quantifier using function `sublists_equal_at_offset` was sufficient to make the proof go through.

4 Challenge 2: prefix sum

4.1 Challenge

The challenge was essentially to specify and verify functional correctness of an algorithm for computing the prefix sums of a given array of integers, expressed in Java in Fig. 4: When method `prefixSums` returns, the element of array `a` at index i should contain the sum of the initial values of elements 0 through $i-1$, for all i, provided that the length of the array is a power of two greater than 1.

4.2 Competition solution

A cleaned version of the solution we submitted during the competition is shown in Figs. 5 and 6.

Our specification for method `prefixSums` requires in its precondition that the array can be described using the separation logic predicate `tree0`. `tree0`(a, l, r, t) says that $l < r$ and array a stores the leaves of the complete binary tree t in elements s through r with $s = r + 1 - 2(r - l)$. That is, l is the index of the element storing the rightmost leaf of the left subtree of t and r is the index of the element storing the rightmost leaf of the right subtree of t. For the precondition of method `prefixSums`, this means that the length of array `a` must be a power of two greater than 1, and its contents are bound to ghost variable `values` in the form of a value of type

```
void upsweep(int[] a, int left, int right) {
    if (right > left+1) {
        int space = right - left;
        upsweep(a, left-space/2, left);
        upsweep(a, right-space/2, right);
    }
    a[right] = a[left]+a[right];
}

void downsweep(int[] a, int left, int right) {
    int tmp = a[right];
    a[right] = a[right] + a[left];
    a[left] = tmp;
    if (right > left+1) {
        int space = right - left;
        downsweep(a, left-space/2, left);
        downsweep(a, right-space/2, right);
    }
}

void prefixSums(int[] a) {
    upsweep(a, a.length/2-1, a.length-1);
    a[a.length - 1] = 0;
    downsweep(a, a.length/2-1, a.length-1);
}
```

Fig. 4 Prefix sum algorithm of Challenge 2

`tree` that is complete (i.e. all leaves are at the same depth). Type `tree` is defined at the top of our solution.

The postcondition states that when the method returns, the array contents are described by the fixpoint function `prefix_sums` applied to `values`, with initial value 0 of the accumulator. This function, defined earlier in the solution, returns the list of prefix sums of the values stored in the given binary tree, incremented by the given accumulator value.

To explain the proof of method `prefixSums`, we first explain the specification and proof of methods `upsweep` and `downsweep`, and of lemma function `tree2_prefix_sums`.

The specification of method `upsweep` requires that when the method is called, the array must be described by predicate `tree0`, as described above. It ensures that when the method returns, the array is described by predicate `tree1` with the same arguments, except that the rightmost element is separately specified to contain the sum of the initial values of the relevant elements of the array. `tree1`(a, l, r, t) states that $l < r$, that t is a complete binary tree with $2(r - l)$ leaves, and that the array element at index $s + i - 1$ contains the sum of the leaves of the largest subtree of t for which t's i'th leaf is the rightmost leaf, where $s = r + 1 - 2(r - l)$ as before, for all $1 \leq i < 2(r - l)$. The rightmost element (the element at index r) is not described by the `tree1` predicate; this prevents the predicate from encapsulating the permission required for the array element assignments in `upsweep` and `prefixSums`.

```
/*@

inductive tree = leaf(int) | node(tree, tree);

predicate tree0(int[] a, int left, int right, tree values) =
    right > left + 1 ?
        tree0(a, left - (right - left) / 2, left, ?lvalues) &*&
        tree0(a, right - (right - left) / 2, right, ?rvalues) &*&
        values == node(lvalues, rvalues) &*&
        right - left == (right - left) / 2 * 2
    :
        a[left] |-> ?l &*& a[right] |-> ?r &*& values == node(leaf(l), leaf(r)) &*&
        right == left + 1;

fixpoint int sum(tree t) {
    switch (t) {
        case leaf(n): return n;
        case node(n1, n2): return sum(n1) + sum(n2);
    }
}

predicate tree1(int[] a, int left, int right, tree values) =
    right > left + 1 ?
        tree1(a, left - (right - left) / 2, left, ?lvalues) &*& a[left] |-> sum(lvalues) &*&
        tree1(a, right - (right - left) / 2, right, ?rvalues) &*&
        values == node(lvalues, rvalues) &*&
        right - left == (right - left) / 2 * 2
    :
        a[left] |-> ?l &*& values == node(leaf(l), leaf(_)) &*&
        right == left + 1;

predicate tree2(int[] a, int left, int right, int leftSum, tree values) =
    right > left + 1 ?
        tree2(a, left - (right - left) / 2, left, leftSum, ?lvalues) &*&
        tree2(a, right - (right - left) / 2, right, leftSum + sum(lvalues), ?rvalues) &*&
        values == node(lvalues, rvalues) &*&
        right - left == (right - left) / 2 * 2
    :
        a[left] |-> leftSum &*& a[right] |-> ?r &*& values == node(leaf(r - leftSum), leaf(_)) &*&
        right == left + 1;

fixpoint list<int> prefix_sums(int leftSum, tree b) {
    switch (b) {
        case leaf(n): return cons(leftSum, nil);
        case node(t1, t2): return append(prefix_sums(leftSum, t1), prefix_sums(leftSum + sum(t1), t2));
    }
}

lemma void tree2_prefix_sums()
    requires tree2(?a, ?left, ?right, ?leftSum, ?values);
    ensures a[left + 1 - (right - left)..right + 1] |-> prefix_sums(leftSum, values);
{
    open tree2(_, _, _, _, _);
    if (right > left + 1) {
        tree2_prefix_sums();
        tree2_prefix_sums();
    }
}

@*/
```

Fig. 5 Our competition solution to Challenge 2 (Part 1 of 2)

```
class PrefixSumRec {

    public static void upsweep(int[] a, int left, int right)
        //@ requires tree0(a, left, right, ?values);
        //@ ensures tree1(a, left, right, values) &*& a[right] |-> sum(values);
    {
        //@ open tree0(_, _, _, _);
        if (right > left+1) {
            int space = right - left;
            upsweep(a,left-space/2,left);
            upsweep(a,right-space/2,right);
        }
        a[right] = a[left]+a[right];
        //@ close tree1(a, left, right, values);
    }

    public static void downsweep(int[] a, int left, int right)
        //@ requires tree1(a, left, right, ?values) &*& a[right] |-> ?leftSum;
        //@ ensures tree2(a, left, right, leftSum, values);
    {
        //@ open tree1(_, _, _, _);
        int tmp = a[right];
        a[right] = a[right] + a[left];
        a[left] = tmp;
        if (right > left+1) {
            int space = right - left;
            downsweep(a,left-space/2,left);
            downsweep(a,right-space/2,right);
        }
        //@ close tree2(a, left, right, leftSum, values);
    }

    public static void prefixSums(int[] a)
        //@ requires a != null &*& tree0(a, a.length / 2 - 1, a.length - 1, ?values);
        //@ ensures a[..] |-> prefix_sums(0, values);
    {
        upsweep(a, a.length / 2 - 1, a.length - 1);
        a[a.length - 1]=0;
        downsweep(a, a.length / 2 - 1, a.length - 1);
        //@ tree2_prefix_sums();
    }

    public static void main(String[] args)
        //@ requires true;
        //@ ensures true;
    {
        int [] a = {3,1,7,0,4,1,6,3};
        //@ close tree0(a, 0, 1, _);
        //@ close tree0(a, 2, 3, _);
        //@ close tree0(a, 1, 3, _);
        //@ close tree0(a, 4, 5, _);
        //@ close tree0(a, 6, 7, _);
        //@ close tree0(a, 5, 7, _);
        //@ close tree0(a, 3, 7, _);
        prefixSums(a);
        //@ assert a[..] |-> {0,3,4,11,11,15,16,22};
    }

}
```

Fig. 6 Our competition solution to Challenge 2 (Part 2 of 2)

For the verification of the body of `upsweep`, all that was required was an explicit unfolding of predicate `tree0` and an explicit folding of predicate `tree1`. This is because the definition of the predicates matches closely the structure of the code.

The specification of `downsweep` requires in its precondition the state as it is left by `upsweep`, except that the value of `a[right]` is not constrained, and ensures that when the method returns, the relevant part of `a` is described by the predicate `tree2`. `tree2`(a, l, r, p, t) states that $l < r$, that t is a complete binary tree with $2(r - l)$ leaves, and that the element of array a at index $s+i-1$ contains the sum of p and the first through $i-1$'th leaf of t, where $s = r+1-2(r-l)$ as before, for all $1 \leq i \leq 2(r-l)$.

Similarly to the proof of `upsweep`, the proof of the function `downsweep` only requires an explicit unfolding of predicate `tree1` and an explicit folding of predicate `tree2`, again because the predicate definitions have been chosen to coincide with the structure of the code.

Finally, the lemma `tree2_prefix_sums` is used to show that predicate `tree2` does indeed imply the prefix sums property. The proof is by straightforward induction on the derivation of the `tree2` predicate.

5 Challenge 3: iterative deletion in a binary search tree

5.1 The challenge

The challenge was to specify and verify functional correctness of the following algorithm that removes the minimum element from a binary search tree:

```
typedef struct tree {
    struct tree *left;
    int data;
    struct tree *right;
} *Tree;

(Tree, int) search_tree_delete_min(Tree t) {
    Tree tt, pp, p;
    int m;
    p = t->left;
    if (p == NULL) {
        m = t->data; tt = t->right; dispose(t);
        t = tt;
    } else {
        pp = t; tt = p->left;
        while (tt != NULL) {
            pp = p; p = tt; tt = p->left;
        }
        m = p->data; tt = p->right; dispose(p);
        pp->left = tt;
    }
    return (t, m);
}
```

5.2 Competition solution

The most challenging part of this task is the specification of the loop: the loop "returns" three pointers that point deep inside a tree data structure, and then the code after the loop modifies the tree through these pointers. The information provided by the loop specification should be sufficient to prove that the piece of code after the loop is safe, establishes the correct function result, and maintains the well-formedness of the binary search tree.

A cleaned-up version of the solution that we submitted during the competition is shown in Figs. 7, 8.

Before we describe how we addressed the loop specification problem, we first explain how we specified the function.

The function's precondition uses separation logic predicate `Tree` (not to be confused with program type `Tree`) to specify that pointer `t` must point to a well-formed tree data structure, and to bind its contents to ghost variable `vs`. Furthermore, it requires that the tree be non-empty, and that it receive permission to write to the pointers `r1` and `r2`. The postcondition states that in the post-state `r1` points to some pointer `tresult` which itself points to a well-formed tree data structure whose contents can be obtained by deleting the leftmost leaf of `vs`; furthermore, `r2` points to the value of this leaf.

Notice that the data structure contents are specified as a value of inductive datatype `tree`, which specifies a precise tree shape, including the address of each node. While this does not break information hiding (since the predicate `Tree` need not comply with the shape information specified by the `tree` value; indeed, it could ignore all information except for the set of data values contained in the nodes), this is not a particularly clean choice. However, given the time constraints, we chose a datatype that would facilitate the proof of the body of the function.

The `Tree` predicate chosen for this particular proof does in fact follow the shape of the given `tree` value.

As for the proof of the body of the function, the case where the root of the tree is the leftmost node is trivial. We concentrate on the other case.

To address the abovementioned loop specification problem, for this solution we chose to have the loop return a "magic wand". A magic wand or separating implication $P \mathrel{-\!\!*} Q$ is a separation logic permission that means as much as "Q except for P", i.e. one can derive Q given both P and $P \mathrel{-\!\!*} Q$:

$$P * (P \mathrel{-\!\!*} Q) \Rightarrow Q$$

Our loop returns a magic wand that says: if you give me the `left` field of `pp` updated with the correct new value, then I will give you the tree data structure, well-formed, in its new state where the minimum element has been deleted. Indeed, when the loop ends, the tree is in the desired state, except

```
/*@
inductive tree = empty | node(Tree, tree, int, tree);

fixpoint tree delete_min(tree t) {
    switch (t) {
        case empty: return empty;
        case node(p, t1, v, t2): return t1 == empty ? t2 : node(p, delete_min(t1), v, t2);
    }
}
fixpoint int min_value(tree vs) {
    switch (vs) {
        case empty: return 0;
        case node(p, t1, v, t2): return t1 == empty ? v : min_value(t1);
    }
}
fixpoint Tree p_(tree t) {
    switch (t) {
        case empty: return 0;
        case node(p, l, v, r): return p;
    }
}
predicate Tree(Tree t; tree vs) =
    t == 0 ?
        vs == empty
    :
        t->left |-> ?l &*& Tree(l, ?vsl) &*& t->right |-> ?r &*& Tree(r, ?vsr) &*&
        t->data |-> ?v &*& malloc_block_tree(t) &*& vs == node(t, vsl, v, vsr);

predicate Tree_r(Tree t; int v, tree rvs) =
    t->data |-> v &*& malloc_block_tree(t) &*& t->right |-> ?r &*& Tree(r, rvs);

lemma_auto void Tree_inv()
    requires Tree(?t, ?vs);
    ensures Tree(t, vs) &*& t == p_(vs);
{ open Tree(_, _); }

typedef lemma void fill_hole_lemma(predicate() pred, Tree t, tree vs, Tree pp, Tree r)();
    requires pred() &*& pp->left |-> r;
    ensures Tree(t, delete_min(vs));
@*/
```

Fig. 7 Our competition solution for Challenge 3 (Part 1 of 2)

that the `left` field of `pp` must still be updated to point to the right-hand child of `p`.

VeriFast does not have built-in support for magic wands. However, it supports *lemma function pointers* and *parameterized lemma function types*. Together, these features allow us to encode magic wands into VeriFast.

Specifically, VeriFast supports *lemma function type definitions* of the following form:

```
typedef lemma rt lft(tpt1 tp1, ..., tptN tpN)
                    (pt1 p1, ..., ptM pM);
    requires P;
    ensures Q;
```

This defines a lemma function type `lft` with return type `rt`, *lemma function type parameters* `tp1` through `tpN`, lemma function parameters `p1` through `pM`, precondition `P`, and postcondition `Q`. Both the lemma function type parameters and the lemma function parameters may appear in the precondition and the postcondition. Given this lemma function type definition, we can assert that some lemma function `lf` complies with the type, instantiated with particular values `v1` through `vN` for the lemma function type parameters, using a ghost command of the form

```
produce_lemma_function_pointer_chunk(lf)
    : lft(v1, ..., vN)(p1, ..., pM)
{ G1 call(); G2 };
```

where `G1` and `G2` are optional sequences of ghost commands (such as **open** or **close** commands or lemma function calls) to aid in deriving the lemma function precondition from the

```
void search_tree_delete_min(Tree t, Tree *r1, int *r2)
    //@ requires Tree(t, ?vs) &*& vs != empty &*& *r1 |-> _ &*& *r2 |-> _;
    //@ ensures *r1 |-> ?tresult &*& Tree(tresult, delete_min(vs)) &*& *r2 |-> min_value(vs);
{
    //@ open Tree(_, _);
    if (t->left == 0) { /*@ open Tree(0, _); @*/ *r2 = t->data; *r1 = t->right; free(t); } else {
        Tree pp = t; Tree p = pp->left; Tree tt = p->left;
        for (;;)
            /*@ requires Tree(pp, ?vs_) &*& vs_ == node(pp, ?lvs, ?v, ?rvs) &*&
                    lvs == node(p, ?llvs, _, ?rlvs) &*& tt == p_(llvs); @*/
            /*@ ensures p->left |-> _ &*& p->data |-> min_value(vs_) &*& p->right |-> ?r &*&
                    malloc_block_tree(p) &*&
                    is_fill_hole_lemma(_, ?pred, old_pp, vs_, pp, r) &*& pred() &*& pp->left |-> p; @*/
        {
            //@ { open Tree(pp, _); open Tree(p, _); open Tree(tt, _); }
            if (tt == 0) {
                /*@ {
                    predicate P() = Tree_r(pp, v, rvs) &*& Tree(p_(rlvs), rlvs);
                    lemma void lem()
                        requires P() &*& pp->left |-> p_(rlvs);
                        ensures Tree(pp, delete_min(vs_));
                    { open P(); }
                    produce_lemma_function_pointer_chunk(lem) : fill_hole_lemma(P, pp, vs_, pp, p_(rlvs))()
                    { call(); };
                    close P();
                } @*/
                break;
            }
            pp = p; p = tt; tt = p->left;
            //@ recursive_call();
            //@ assert is_fill_hole_lemma(?lem0, ?P0, _, _, ?pp_, ?r);
            /*@ {
                predicate P() =
                    Tree_r(old_pp, v, rvs) &*& old_pp->left |-> old_p &*&
                    is_fill_hole_lemma(lem0, P0, p_(lvs), lvs, pp_, r) &*& P0();
                lemma void lem()
                    requires P() &*& pp->left |-> r;
                    ensures Tree(old_pp, delete_min(vs_));
                {
                    open P();
                    lem0();
                    leak is_fill_hole_lemma(_, _, _, _, _, _);
                }
                produce_lemma_function_pointer_chunk(lem) : fill_hole_lemma(P, old_pp, vs_, pp_, r)()
                { call(); };
                close P();
            } @*/
        }
        *r1 = t; *r2 = p->data; tt = p->right; free(p); pp->left = tt;
        //@ assert is_fill_hole_lemma(?lem, _, _, _, _, _);
        //@ lem();
        //@ leak is_fill_hole_lemma(_, _, _, _, _, _);
    }
}
```

Fig. 8 Our competition solution for Challenge 3 (Part 2 of 2)

lemma function type precondition and the lemma function type postcondition from the lemma function postcondition, respectively. This ghost command produces a *lemma function pointer chunk* of the form `is_lft(lfp, v1, ..., vN)` where `lfp` is a pointer to the lemma function. (Of course, since lemma functions do not really exist in memory at run

time, the pointer is not really a memory address but is just a value that identifies the lemma function.) Once we have a lemma function pointer chunk, we can perform a lemma function pointer call:

```
assert is_lft(?lfp, _, ..., _);
lfp(w1, ..., wM);
```

We first bind the lemma function pointer to ghost variable `lfp` and then perform a function call through that pointer, passing the arguments for the lemma function parameters. When encountering a lemma function pointer call, VeriFast checks that a lemma function pointer chunk exists in the symbolic heap. It then uses the values for the lemma function type parameters found in the chunk and the values for the lemma function parameters found in the call itself to instantiate the lemma function type precondition and postcondition, and finally uses the resulting contract to verify the call (by consuming the precondition and then producing the postcondition).

Our challenge solution defines the parameterized lemma function type `fill_hole_lemma`. Our loop returns a pointer to a lemma function that implements this type, instantiated appropriately.

The loop is again specified using a loop contract (see our discussion of our competition solution for Challenge 1 above). A particular "call" of the loop operates on the subtree pointed to by the current value of `pp`: given such a subtree, it returns the permissions for the fields of the node to be deleted (sufficient to deallocate the node) as well as the permission to update the `left` field that needs to be updated, and a magic wand (the combination of a lemma function pointer call permission `is_fill_hole_lemma(...)` and a function-specific permission `pred()` holding the specific permissions required by the lemma to perform its duty) that can be used after the update to obtain the `Tree` permission describing the properly updated subtree at the old value of `pp`.

The code after the loop simply needs to call the returned lemma. To do so, it first uses an **`assert`** statement to bind the lemma function pointer (which is the first argument of the lemma function pointer call permission) to ghost variable `lem`. After the call, we use the **`leak`** command to acknowledge that we are consciously leaking the lemma function

```
/*@
predicate Tree_with_hole(Tree t, Tree hole; tree vs) =
  hole != 0 &*&
  t == hole ?
    vs == empty
  :
    t != 0 &*& t->left |-> ?l &*& Tree_with_hole(l, hole, ?vsl) &*&
    t->right |-> ?r &*& Tree(r, ?vsr) &*&
    t->data |-> ?v &*& malloc_block_tree(t) &*&
    vs == node(t, vsl, v, vsr);

fixpoint tree merge(tree t1, tree t2) {
  switch(t1) {
    case empty: return t2;
    case node(p, l, v, r): return node(p, merge(l, t2), v, r);
  }
}

lemma void move_hole(Tree t);
  requires Tree_with_hole(t, ?hole, ?vs1) &*& Tree(hole, node(hole, ?left, ?v, ?right)) &*& p_(left) != 0;
  ensures Tree_with_hole(t, p_(left), merge(vs1, node(hole, empty, v, right))) &*& Tree(p_(left), left) &*&
          merge(vs1, node(hole, left, v, right)) == merge(merge(vs1, node(hole, empty, v, right)), left);

lemma void plug_hole(Tree t1, Tree t2);
  requires Tree_with_hole(t1, t2, ?vs1) &*& Tree(t2, ?vs2);
  ensures Tree(t1, merge(vs1, vs2));

lemma void delete_min_merge(tree t1, tree t2);
  requires t2 != empty;
  ensures merge(t1, t2) != empty &*& delete_min(merge(t1, t2)) == merge(t1, delete_min(t2)) &*&
          min_value(merge(t1, t2)) == min_value(t2);
@*/
```

Fig. 9 Our post-competition solution for Challenge 3 (Part 1). The bodies of the lemmas have been omitted from this article

```
void search_tree_delete_min(Tree t, Tree* r1, int* r2)
  //@ requires Tree(t, ?vs) &*& vs != empty &*& *r1 |-> _ &*& *r2 |-> _;
  //@ ensures *r1 |-> ?tresult &*& Tree(tresult, delete_min(vs)) &*& *r2 |-> min_value(vs);
{
  Tree tt, pp, p;
  int m;
  //@ open Tree(t, vs);
  p = t->left;
  if (p == 0) {
    //@ open Tree(p, _);
    m = t->data; tt = t->right; free (t); t = tt;
  } else {
    pp = t; tt = p->left;
    while (tt != 0)
      /*@ invariant Tree_with_hole(t, pp, ?vs1) &*& Tree(pp, ?vs2) &*& vs2 == node(pp, ?lvs, _, _) &*&
                    lvs == node(p, ?llvs, _, _) &*& p_(llvs) == tt &*& vs == merge(vs1, vs2);
      @*/
    {
      //@ open Tree(pp, vs2);
      //@ open Tree(pp->left, _);
      pp = p; p = tt; tt = p->left;
      //@ move_hole(t);
    }
    //@ open Tree(pp, _);
    //@ open Tree(pp->left, _);
    //@ open Tree(tt, _);
    m = p->data; tt = p->right; free(p); pp->left = tt;
    //@ plug_hole(t, pp);
    //@ delete_min_merge(vs1, vs2);
  }
  *r1 = t;
  *r2 = m;
}
```

Fig. 10 Our post-competition solution for Challenge 3 (Part 2)

pointer call permission since we will not be using it further; this keeps VeriFast from reporting a leak error.

The body of the loop consists of two cases: either we have found the node to be deleted, or we need to continue our descent. In the former case, we define a lemma `lem` and an accompanying predicate `P` that reassembles the subtree at `pp` given the right-hand section of `pp` (i.e. everything except the `left` field and the left subtree, represented by predicate `Tree_r`) and the right-hand subtree of `p`. We then use a **`produce_lemma_function_pointer_chunk`** command to generate a lemma function pointer call permission. This command generates a proof obligation that the lemma's contract implies the lemma function type's contract, appropriately instantiated. Any ghost commands required to discharge this proof can be written before or after the **`call`**`();` statement inside the ghost code block accompanying the command.

If we have not yet found the node to be deleted, we update the loop variables and perform the imaginary recursive call of the loop. (Note: in our competition solution for Challenge 1, the recursive call was implicitly inserted at the end of the loop body. Here we specify it explicitly so that we can insert ghost commands after the recursive call to derive the caller's postcondition from the callee's postcondition.) After the recursive call, we take the magic wand `lem0` returned by the recursive call (which operates on the subtree at `p`) and use it to build a new magic wand `lem` that operates on the subtree at `pp`.

5.3 Post-competition solution

During the competition, we worked on an alternative approach for the third challenge. However, we did not manage to fully work out this solution in the allotted time, and we completed it only after the competition. It is shown in Figs. 7 (shared with our competition solution), 9 and 10.

The loop in the function `search_tree_delete_min` iterates over the `left` pointers until it finds the leftmost node in the tree. The key idea of this solution is that in each iteration the entire tree can be decomposed into a "tree with a hole" at `pp` (describing the part of the tree up to, but not including, the node at `pp` and its descendants) and another, complete

tree starting at `pp`. Indeed, the loop invariant of Fig. 10 states exactly this property.

To describe trees with a hole, we introduce the predicate `Tree_with_hole` (shown in Fig. 9). The body of this predicate is similar to the body of `Tree` of Fig. 7 except that the recursion stops when `t` equals `hole` (instead of when `t` is zero). The fixpoint `merge` fills the hole in a tree `t1` by plugging in another tree `t2`.

A single iteration of the loop moves `pp`, `p` and `tt` down one node in the tree. The lemma `move_hole` states that doing so preserves the loop invariant. After the loop, the leftmost node is deleted from the tree. The lemma `plug_hole` is then used to recombine the tree with a hole and the tree at `pp` into a single tree, as required by the postcondition.

The predicate `Tree_with_hole` was inspired by the predicate `lseg` which is commonly used in separation logic proofs about linked lists. `lseg(n1, n2, vs)` holds if there exists a sequence of nodes from `n1` up to (but not including) `n2` which contains the data values `vs`. In other words, `lseg(n1, n2, vs)` describes a linked list with a hole at `n2`.

The function contract of `search_tree_delete_min` shown in Fig. 10 only states that the function deletes the leftmost node in the tree and that it returns the value stored in this node. In our actual implementation, we additionally prove that the resulting tree is a binary search tree, if the tree was a binary search tree on entry to the function.

6 Conclusion

For each of the three challenges issued during the competition, we were able to submit a solution. Moreover, the function contracts we came up with for each challenge are *complete* in the sense that they specify full functional correctness of the program at hand.

The annotation overhead and verification time required for each solution are shown in Table 1. While the annotation overhead ranges from 3 up to 7 lines of annotations per line of code, each solution verifies in less than one second.

In hindsight, we believe three key features of VeriFast enabled us to solve the challenges within the limited competition time frame: (1) the existing specification libraries, (2) fast verification and (3) separation logic. First of all, VeriFast ships with a number of specification libraries. In all our solutions, we reused definitions and lemmas provided by these libraries. For example, `list.gh` defines the function `drop` and offers a number of lemmas such as `drop_n_plus_one` and `drop_length` about this function that we reused in our first solution to Challenge 1. Second, VeriFast typically offers feedback on a verification attempt in under a second. This allowed us to stay focussed on the particular sub-problem at hand without losing time waiting for an answer from the verifier. Finally, the third challenge involves specifying a function operating on a tree. Thanks to separation logic, we were able to concisely specify the structure of the tree. Moreover, separation logic allowed us to reason locally: if a code fragment demands access only to a particular sub-tree, client code can safely assume that the fragment does not invalidate properties that hold for contexts containing that sub-tree (provided the context is disjoint from the sub-tree).

Table 1 Table showing the lines of code (LOC), lines of annotations (LOA), annotation overhead, and the verification time required for each solution

Solution	*LOC*	*LOA*	Overhead (%)	Time (s)
LCP sol. 1	7	26	371	0.13
LCP sol. 2	5	16	320	0.03
PrefixSum	29	194	669	0.28
TreeDel sol. 1	20	102	510	0.09
TreeDel sol. 2	18	114	633	0.09

The experiments were executed on a MacBook Pro with a 2.53 GHz Intel Core i5 processor and 4 GB of memory running Ubuntu Linux 12.04.1 LTS. The data in this table was gathered by running VeriFast from the command-line with the `-stats` option using Z3 [3] 2.3 to discharge proof obligations. Note that the lines of code and annotations only include the solution file itself; specification libraries that ship with VeriFast (such as `list.gh`) are not included in the measurements

In the future, we plan to extend the automation in VeriFast such that additional ghost statements (such as lemma calls and open/close statements) can be inferred to reduce the annotation overhead. A key challenge in designing such an extension is keeping verification times low. Also, any inference algorithm will be incomplete, so we will need to provide ways for the user to supply hints.

We have participated in two other verification competitions in the past: the First [8] and Second [4] Verified Software Competitions. We did as follows: in the first, we were tied with all other participants; in the second, we were not among the medalists: we ended up in "Rank 8" with a score of 570/600. In the end, a team's result in a verification competition of course depends not only on the "quality" of the tool and the approach, but also on the "shape" that the team members are in at the time of the competition, and to a large extent on random factors (i.e. "luck"). Still, during this latest competition we benefited from new features recently introduced into VeriFast, most prominently constructor patterns (e.g. `xs == cons(_,_)` and local predicates (e.g. predicates `P` in Fig. 8), which, while not theoretically extending the power of VeriFast's logic, saved us some time, typing, and brain cycles.

We enjoyed the competition very much and from the ensuing discussions we learned a great deal about the other verification approaches; we gratefully acknowledge the significant

efforts by the organizers to bring about this successful outcome.

Acknowledgments This research is partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE). Jan Smans is a postdoctoral fellow of the Fund for Scientific Research Flanders (FWO). We acknowledge support from Microsoft Research Cambridge as part of the Verified Software Initiative.

References

1. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer, Berlin (2004)
2. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. POPL (2005)
3. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS. In: Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer, Berlin (2008)
4. Filliâtre, J.-C., Paskevich, A., Stump, A.: The 2nd verified software competition: experience report. In: Klebanov, V., Grebing, S. (eds.), COMPARE2012: 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems. Manchester, UK, June 2012. EasyChair (2012)
5. Hoare, C.A.R.: An axiomatic basis for computer programming. CACM **12**(10) (1969)
6. Huisman, M., Klebanov, V., Monahan, R.: The VerifyThis 2012 website. http://fm2012.verifythis.org/ (2012)
7. Jacobs, B., Smans, J., Piessens, F.: The VeriFast website. http://distrinet.cs.kuleuven.be/software/VeriFast/ (2013)
8. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Rustan, K., Leino, M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st Verified Software Competition: Experience report. In: Butler, M., Schulte W. (eds) Proceedings, 17th International Symposium on Formal Methods (FM), vol. 6664 of LNCS. Springer, Berlin (2011)
9. Nipkow, Tobias: Interactive proof: Introduction to Isabelle/HOL. In: Grumberg, O., Nipkow, T., Hauptmann, B. (eds.) Software Safety and Security, pp. 254–285. IOS Press, Amsterdam (2012)
10. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. CSL (2001)
11. Parkinson, M., Bierman, G.: Separation logic and abstraction. POPL (2005)
12. Philippaerts, P., Mühlberg, J.T., Penninckx, W., Smans, J., Jacobs, B., Piessens, F.: Software verification with VeriFast: industrial case studies. Sci. Comput. Progr (2013)
13. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. LICS (2002)
14. Tuerk, T.: Local reasoning about while-loops. In: VSTTE Theory Workshop (2010)