Bottom-up Shape Analysis using \textit{LISF}

Bhargav S. Gulavani and Supratik Chakraborty  
IIT Bombay  
and  
G. Ramalingam and Aditya V. Nori  
Microsoft Research India

In this paper we present a new shape analysis algorithm. The key distinguishing aspect of our algorithm is that it is completely compositional, bottom-up and non-iterative. We present our algorithm as an inference system for computing Hoare triples summarizing heap manipulating programs. Our inference rules are compositional: Hoare triples for a compound statement are computed from the Hoare triples of its component statements. These inference rules are used as the basis for bottom-up shape analysis of programs.

Specifically, we present a Logic of Iterated Separation Formulae (\textit{LISF}), which uses the iterated separating conjunct of Reynolds [Reynolds 2002] to represent program states. A key ingredient of our inference rules is a strong bi-abduction operation between two logical formulas. We describe sound strong bi-abduction and satisfiability procedures for \textit{LISF}.

We have built a tool called \textit{SpInE} that implements these inference rules and have evaluated it on standard shape analysis benchmark programs. Our experiments show that \textit{SpInE} can generate expressive summaries, which are complete functional specifications in many cases.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification—Formal Methods; Programming by contract; D.2.1 [Software Engineering]: Requirements/Specifications

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Compositional Analysis, Hoare Logic, Separation Logic

1. INTRODUCTION

In this paper we present a new shape analysis algorithm: an algorithm for analyzing programs that manipulate dynamic data structures such as lists. The key distinguishing aspect of our algorithm is that it is completely bottom-up and non-iterative. It computes summaries describing the effect of a statement or procedure in a modular, compositional, non-iterative way: the summary for a compound statement is computed from the summaries of simpler statements that make up the compound statement.

Shape analysis is intrinsically challenging. Bottom-up shape analysis is particularly challenging because it requires analyzing complex pointer manipulations when nothing is known about the initial state. Hence, traditional shape analyses are based on an iterative top-down (forward) analysis, where the statements are analyzed in the context of a particular (abstract) state. Though challenging, bottom-up shape analysis appears worthwhile pursuing because the compositional nature of the analysis promises much better scalability, as illustrated by the recent work of Calcagno \textit{et al.} [Calcagno et al. 2009]. The algorithm we present is based on ideas introduced by Calcagno \textit{et al.} [Calcagno et al. 2009].

\textit{Motivating Example.} Consider the procedure shown in Figure 1. Given a list
delete(struct node *h, *a, *b)
1. y=h;
2. while (y!=a && y!=0) {
3. y=y->next;
}
4. x=y;
5. if (y!=0) {y=y->next;}
6. while (y!=b && y!=0) {
7. t=y;
8. y=y->next;
9. delete(t);
}
10. if (x !=0) {
11. x->next=y;
12. if (y!=0) y->prev=x;
}

Fig. 1. Motivating example – deletion of list segment

pointed to by parameter h, this procedure deletes the fragment of the list demar-
cated by parameters a and b. Our goal is to devise an analysis that, given a
procedure S such as this, computes a set of Hoare triples [ϕ] S [φ] that summarize
the procedure. We use the above notation to indicate that the Hoare triples inferred
are total: the triple [ϕ] S [φ] indicates that, given an initial state satisfying ϕ, the
execution of S terminates safely (with no memory errors) in a state satisfying φ.

Inferring Preconditions. There are several challenges in meeting our goal. First,
note that there are a number of interesting cases to consider: the list pointed to by
h may be an acyclic list, or a complete cyclic list, or a lasso (an acyclic fragment
followed by a cycle). The behavior of the code also depends on whether the pointers
a and b point to an element in the list or not. Furthermore, the behavior of the
procedure also depends on the order in which the elements pointed to by a and b
occur in the list.

With traditional shape analyses, a user would have to supply a precondition
descriving the input to enable the analysis of the procedure delete. Alternatively,
an analysis of the calling procedure would identify the abstract state σ in
which the procedure delete is called, and delete would be analyzed in an initial state σ.
In contrast, a bottom-up shape analysis automatically infers relevant preconditions
and computes a set of Hoare triples, each triple describing the procedure’s behavior
for a particular case (such as the cases described in the previous paragraph).

Inferring Postconditions. However, even for a given precondition ϕ, many differ-
cent correct Hoare triples can be produced, differing in the information captured by
the postcondition φ. As an example consider the case where h points to an acyclic
list, and a and b point to elements in the list, with a pointing to an element that
occurs before the element that b points to. In this case, the following are all valid
properties that can be expressed as suitable Hoare triples: (a) The procedure is
memory-safe: it causes no pointer error such as dereferencing a null pointer. (b)
Finally, h points to an acyclic list. (c) Finally, h points to an acyclic list, which is
the same as the list h pointed to at procedure entry, with the fragment from a to
b deleted. Clearly, these triples provide increasingly more information.

A distinguishing feature of our inference algorithm is that it seeks to infer triples
describing properties similar to (c) above, which yield a functional specification
for the analyzed procedure. One of the key challenges in shape analysis is relating the
value of the final data-structure to the value of the initial data-structure. We utilize an extension of separation logic, described later, to achieve this.

Composition via Strong Bi-Abduction. We now informally describe how summaries \([\varphi_1] S1 [\widehat{\varphi}_1]\) and \([\varphi_2] S2 [\widehat{\varphi}_2]\) in separation logic can be composed to obtain summaries for \(S1; S2\). The intuition behind the composition rule, which is similar to the composition rule in [Calcagno et al. 2009], is as follows. Suppose we can identify \(\varphi_{pre}\) and \(\varphi_{post}\) such that \(\widehat{\varphi}_1 * \varphi_{pre}\) and \(\varphi_{post} * \varphi_2\) are semantically equivalent. We can then infer summaries \([\varphi_1 * \varphi_{pre}] S1 [\widehat{\varphi}_1 * \varphi_{pre}]\) and \([\varphi_{post} * \varphi_2] S2 [\varphi_{post} * \widehat{\varphi}_2]\) by application of frame rule [O’Hearn et al. 2001], where * is the separating conjunction of separation logic [Reynolds 2002] (subject to the usual frame rule conditions: \(\varphi_{pre}\) and \(\varphi_{post}\) should not involve variables modified by \(S1\) and \(S2\) respectively). We can then compose these summaries trivially and get \([\varphi_1 * \varphi_{pre}] S1; S2 [\varphi_{post} * \widehat{\varphi}_2]\). Given \(\widehat{\varphi}_1\) and \(\varphi_2\), we refer to the identification of \(\varphi_{pre} \land \varphi_{post}\) such that \(\widehat{\varphi}_1 * \varphi_{pre} \Leftrightarrow \varphi_{post} * \widehat{\varphi}_2\) as strong bi-abduction. Strong bi-abduction also allows for existentially quantifying some auxiliary variables from the right hand side of the equivalence, as discussed later in Section 3.

Iterative Composition. A primary contribution of this paper is to extend the above intuition to obtain loop summaries. Suppose we have a summary \([\varphi] S [\widehat{\varphi}]\), where \(S\) is the body of a loop (including the loop condition). We can apply strong bi-abduction to compose this summary with itself: for simplicity, suppose we identify \(\varphi_{post}\) and \(\varphi_{pre}\) such that \(\widehat{\varphi} \land \varphi_{pre} \Leftrightarrow \varphi_{post} \land \varphi\). If we now inductively apply the composition rule, we can then infer a summary of the form \([\varphi * \varphi_{pre}] S_k [\varphi_{post} \land \widehat{\varphi}]\) that summarizes \(k\) executions of the loop. Here, we have abused notation to convey the intuition behind the idea. If our logic permits a representation of the repetition of a structure \(\varphi_{pre}\) an unspecified number of times \((k)\), we can then directly compute a Hoare triple summarizing the loop from a Hoare triple summarizing the loop body.

Logic Of Iterated Separation Formulae. In order to achieve the above goal, we introduce \(\mathcal{LISF}\), an extension of separation logic, and present sound procedures for strong bi-abduction and satisfiability in \(\mathcal{LISF}\). The logic \(\mathcal{LISF}\) has two key aspects: (i) It contains a variant of Reynolds’ iterated separating conjunct construct that allows the computation of a loop summary from a loop body summary. (ii) It uses an indexed symbolic notation that allows us to give names to values occurring in a recursive (or iterative) data-structure. This is key to meeting the goal described earlier, i.e., computing functional specifications that can relate the value of the final data-structure to that of the initial data-structure. \(\mathcal{LISF}\) gives us a generic ability to define recursive predicates useful for describing certain classes of recursive data-structures. The use of \(\mathcal{LISF}\), instead of specific recursive predicates, such as those describing singly-linked lists or doubly-linked lists, allows us to compute more precise descriptions of recursive data-structures in preconditions. Though we use \(\mathcal{LISF}\) for bottom-up analysis in this paper, its use in not restricted to this. Specifically, it can also be used to represent program states in top down interprocedural analysis.

Empirical Evaluation. We have implemented our inference rules in a bottom-up analyzer SpINE and evaluated it on several shape analysis benchmarks. We say that a set \(S\) of summaries for a program \(P\) is a complete specification for \(P\) if every input configuration starting from which \(P\) terminates without causing errors.
satisfies the precondition of some summary in the set $S$. On most of the examples, we could generate ‘complete’ functional specifications. On the example program in Figure 1, we could generate several summaries with cyclic and lasso structures, although a complete specification was not obtained. As will be explained later, this is due to the incompleteness of our strong bi-abduction algorithm.

Our Contributions. (i) We present a logic of iterated separation formulae $\text{LISF}$ (Section 4), which is a restriction of separation logic with iterated separating conjunction, and give sound algorithms for satisfiability checking and strong bi-abduction in this logic (Sections 6, 7, and 8). (ii) We present inference rules to compute Hoare triples in a compositional bottom-up manner (Section 5). (iii) We have a prototype implementation of our technique. We discuss its performance on several challenging programs (Section 9).

2. RELATED WORK

Our work is most closely related to the recent compositional shape analysis algorithm presented by Calcagno et al. [Calcagno et al. 2009], which derives from the earlier work in [Calcagno et al. 2007]. The algorithm described by Calcagno et al. [Calcagno et al. 2009] is a hybrid algorithm that combines compositional analysis with an iterative forward analysis. The first phase of this algorithm computes candidate preconditions for a procedure, and the second phase utilizes a forward analysis to either discard the candidate precondition, if it is found to potentially lead to a memory error, or find a corresponding sound postcondition. The key idea in this approach, which we borrow and extend, is the use of bi-abduction to handle procedure calls compositionally. Given $\hat{\varphi}_1$, the state at a callsite, and $\varphi_2$, a precondition of a Hoare triple for the called procedure, Calcagno et al. compute $\varphi_{\text{pre}}$ and $\varphi_{\text{post}}$ such that $\hat{\varphi}_1 \ast \varphi_{\text{pre}} \Rightarrow \varphi_{\text{post}} \ast \varphi_2$. Our approach differs from this in several ways. We present a completely bottom-up analysis which does not use any iterative analysis whatsoever. Instead, it relies on a “stronger” form of bi-abduction (where we seek equivalence, instead of implication, but allow some auxiliary variables to be quantified) to compute the post-condition simultaneously. Furthermore, our approach extends the composition rule to treat loops in a similar fashion. Our approach also computes preconditions that guarantee termination. We use $\text{LISF}$ as the basis for our algorithm, while Calcagno et al.’s work uses a set of abstract recursive predicates. We also focus on computing more informative triples that can relate the final value of a data-structure to its initial value.

Several recent papers [Podelski et al. 2008; Abdulla et al. 2008; Lev-Ami et al. 2007] describe techniques to obtain preconditions by going backwards starting from some bad states. Unlike our approach, these techniques are neither compositional nor bottom-up.

Extrapolation techniques proposed in [Touili 2001; Boigelot et al. 2003] compute sound overapproximations of postconditions by identifying the growth in successive applications of transducers and by iterating that growth. Similarly, [Guo et al. 2007] proposes a technique to guess the recursive predicates characterizing a data structure by identifying the growth in successive iterations of the loop and by repeating that growth. In contrast, we identify the growth in both the pre and postconditions by strong bi-abduction and iterate it to compute Hoare triples that are guaranteed...
to be sound. Furthermore, our analysis is bottom-up and compositional in contrast to these top-down (forward) analyses.

TVLA [Sagiv et al. 1999] is a 3-valued predicate logic analyzer with transitive closure. It generates an abstraction of the shape of the program heap at runtime in the form of 3-valued structure descriptors. It performs a top-down analysis within a procedure starting from the given shape of input heap. Several works [Rinetzky and Sagiv 2001; Rinetzky et al. 2005; Rinetzky et al. 2005] have proposed an interprocedural extension of the basic intraprocedural analysis of TVLA. All these algorithms are top-down and forward. In [Rinetzky et al. 2005], Rinetzky et al. compute partially functional summaries. They define a cut-point as a node in the heap graph that is simultaneously reachable from some input parameter of the procedure and some other program variable that is not a parameter to the procedure. The summaries computed in [Rinetzky et al. 2005] track precise input-output relations only between finitely many cut-points. In [Rinetzky et al. 2005], the authors design a global analysis to determine if the program is cut-point free. The summarization algorithm generates summaries only for cut-point free programs. These summaries do not relate the input and output heap cells, except those heap cells that are directly pointed to by a procedure parameter. In contrast, summaries expressed using LISF can capture precise input-output relationships between an unbounded number of cut-points.

In [Jeannet et al. 2004], Jeannet et al. propose an algorithm to generate relational summaries in TVLA. They use instrumentation predicates that relate the input value of a predicate with its output value. Additionally, they also use lemmas specific to the novel instrumentation predicates to avoid loss of information during the abstract computation. Their algorithm is top-down and forward, i.e., they start abstract computation from the main procedure and analyze each procedure (or reuse its already computed summary, if possible) when it is called.

In [Yorsh et al. 2006], Yorsh et al., present a decidable logic of reachable patterns (LRP) in linked data-structures. This logic uses regular patterns to characterize the reachable heap structure. As such, using symbolic variables to represent the initial and final values of the procedure parameters, it is possible to relate the reachable heap cells in the input and output of the procedure. But in this work, the focus is on having a decidable logic for verifying programs annotated with preconditions, postconditions, and loop invariants. They do not provide an algorithm to compute procedure summaries in LRP.

The work on regular model-checking [Abdulla et al. 2004; Bouajjani et al. 2005; Bouajjani et al. 2006; Bouajjani et al. 2004] represents input-output relations by a transducer, which can be looked upon as a functional specification. Given the transducer for the loop body and initial configuration encoded as an automaton, the goal is to compute the final configuration after the loop exits (i.e., the post-condition). This problem is undecidable in general, since the iterated loop body transducer could encode a Turing machine. The authors therefore use abstraction-refinement to compute over-approximations of the postcondition. In [Abdulla et al. 2008], Abdulla et al. propose algebraic structures richer than finite state automata for representing shape of the program heap. Their method allows heap graphs to be directly represented as graphs, and the operational semantics to be represented
Program Syntax

\[ \begin{align*}
    e & ::= v \mid null \\
    B & ::= v = e \mid v \neq e \\
    S & ::= v := e \mid \text{new } \mid \text{dispose } v \mid S; S \\
    & \mid \text{assert}(B) \mid v := \text{if}(B, S, S) \mid \text{while}(B) S
\end{align*} \]

Separation Logic Syntax

\[ \begin{align*}
    e & ::= \text{null } \mid v \mid \ldots \\
    P & ::= e \sim e \mid \text{false } \mid \text{true } \mid P \land P \mid \ldots \\
    S & ::= \text{emp } \mid e \mapsto (f : e) \mid \text{true } \mid S \ast S \mid \ldots \\
    \varphi & ::= P \land S \mid \exists v. SH
\end{align*} \]

Fig. 2. Program syntax and separation logic syntax

\[(s, h) \models P \land S \quad \text{iff} \quad (s, h) \models P \land (s, h) \models S\]

\[(s, h) \models e_1 \sim e_2 \quad \text{iff} \quad s(e_1) \sim s(e_2)\]

\[(s, h) \models \text{true}\]

\[(s, h) \not\models \text{false}\]

\[(s, h) \models P_1 \land P_2 \quad \text{iff} \quad (s, h) \models P_1 \land (s, h) \models P_2\]

\[(s, h) \models \text{emp} \quad \text{iff} \quad \text{dom}(h) = \{\}\]

\[(s, h) \models e_1 \mapsto (f : e_2) \quad \text{iff} \quad h(s(e_1)) = (f : s(e_2)) \land \text{dom}(h) = \{s(e_1)\}\]

\[(s, h) \models S_1 \ast S_2 \quad \text{iff} \quad \exists h_1, h_2. h_1 \# h_2 \land h_1 \cup h_2 = h \land (s, h_1) \models S_1 \land (s, h_2) \models S_2\]

Fig. 3. Separation logic semantics.

as relations on graphs. All the analyses proposed above proceed top-down, and the authors do not leverage compositional techniques to compute the transducer for loops.

3. COMPOSITION VIA STRONG BI-ABDUCTION

In this section we introduce the idea of composing Hoare triples using strong bi-abduction.

3.1 Preliminaries

Programming language. We address a simple language whose syntax appears in Figure 2. The primitives assert(v = e) and assert(v \neq e) are used primarily to present inference rules for conditionals and loops (as will be seen later). Here \(v\), \(u\) are program variables, and \(e\) is an expression which could either be a variable or the constant \(\text{null}\). This language supports heap manipulating operations without address arithmetic.

Semantically, we use a value domain \textbf{Locs} (which represents an unbounded set of locations). Each location in the heap represents a cell with \(n\) fields, where \(n\) is statically fixed. A computational state contains two components: a stack \(s\), mapping program variables to their values (\textbf{Locs} \(\cup\) \{\text{null}\}), and a heap \(h\), mapping a finite set of non-null locations to their values, which are \(n\)-tuples of (primitive) values.

Assertion Logic. We illustrate some of the key ideas using standard separation logic, using the syntax shown in Figure 2. The ‘...’ in Figure 2 refer to constructs and extensions we will introduce in Section 4. discussion. We assume the reader is familiar with basic ideas in separation logic. Every expression \(e\) in separation logic evaluates to a location. Given a stack \(s\), a variable \(v\) evaluates to a location \(s(v)\). We define \(s(\text{null})\) to be \(\text{null}\). A symbolic heap representation consists of a pure part \(P\) and a spatial part \(S\). The pure part \(P\) consists of equalities and disequalities of expressions. The spatial part \(S\) describes the shape of the graph in the heap. Let \text{dom}(h) denote the domain of heap \(h\). \textbf{emp} denotes that the heap

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
has no allocated cells, i.e., \( \text{dom}(h) = \{\} \). The predicate \( x \mapsto (f : l) \) denotes a heap consisting of a single allocated cell pointed to by \( x \), and the \( f \) field of this cell has value \( l \). In general, for objects having \( n \) fields \( f^1, \ldots, f^n \), the general version of the \( \mapsto \) predicate is \( e \mapsto (f^1 : e_1, \ldots, f^n : e_n) \). The \( \ast \) operator is called the separating conjunction; \( s_1 \ast s_2 \) denotes that \( s_1 \) and \( s_2 \) refer to disjoint portions of the heap and the current heap is the disjoint union of these sub-heaps. We use the notation \( h_1 \# h_2 \) to denote that \( h_1 \) and \( h_2 \) have disjoint domains, and use \( h_1 \cup h_2 \) to denote the disjoint union of such heaps. The meaning of pure assertions depends only on the stack, and the meaning of spatial assertions depends on both the stack and the heap.

**Hoare triples.** The specification \([\varphi] \cdot S \cdot [\hat{\varphi}]\) means that when \( S \) is run in a state satisfying \( \varphi \) it terminates without any memory error (such as null dereference) in a state satisfying \( \hat{\varphi} \). Thus, we use total correctness specifications. Additionally, we call the specification \([\varphi] \cdot S \cdot [\hat{\varphi}]\) strong if \( \hat{\varphi} \) is the strongest postcondition of \( \varphi \) with respect to \( S \). We use the logical variable \( v \) to refer to the value of program variable \( v \) in the pre and postcondition of a statement \( S \). The specification may refer to auxiliary logical variables from a set \( \text{Aux} \), that do not correspond to the value of any program variable. For the present discussion, we prefix all auxiliary variable names with \(^{\prime} \). A Hoare triple with auxiliary variables is said to be valid iff it is valid for any value binding for the auxiliary variables occurring in both the pre and postcondition. The local Hoare triples for reasoning about primitive program statements are given in Table I. These are similar to the small axioms of [O’Hearn et al. 2001].

**Notation.** We use the following short-hand notations for the remainder of the paper. Formulae \( \text{true} \land S \) and \( P \land \text{emp} \) in pre or post conditions are represented simply as \( S \) and \( P \) respectively. The notation \( \theta : \langle v \mapsto x \rangle \) refers to a renaming \( \theta \) that replaces variable \( v \) with \( x \), and \( e\theta \) refers to the expression obtained by applying renaming \( \theta \) to \( e \). For sets \( A \) and \( B \) of variables, we write \( \theta : \langle A \leftrightarrow B \rangle \) to denote renaming of a subset of variables in \( A \) by variables in \( B \), and we write \( \theta : \langle A \rightarrow B \rangle \) to denote renaming of all variables in \( A \) by variables in \( B \). Given a formula \( \varphi \), we use \( \text{free}(\varphi) \) to refer to the set of free variables in \( \varphi \). We denote sets of variables by upper-case letters like \( V, W, X, Y, Z, \ldots \). For every such set \( V \), \( V_i \) denotes the set of \( i \) subscripted versions of variables in \( V \). We say that \( \varphi \) is independent of the set of variables \( A \), if \( A \cap \text{free}(\varphi) = \emptyset \). We use \( \varphi^p \) and \( \varphi^s \) to refer to the pure and spatial parts, respectively, of \( \varphi \). The notation \( \exists X \varphi \ast \exists Y \psi \) is used to denote \( \exists X, Y \varphi^p \land \psi^p \land \varphi^s \ast \psi^s \), when \( \varphi \) and \( \psi \) are quantifier free and do not have free \( Y \) and \( X \) variables, respectively.

We denote the set of logical variables corresponding to the program variables modified by \( S \) as \( \text{mod}(S) \). For primitive statements, the definition of \( \text{mod} \) is given in Table I. For composite statements, \( \text{mod} \) is defined as follows. \( \text{mod}(S_1; S_2) \) and \( \text{mod}(\text{if}(C, S_1, S_2)) \) are both defined as \( \text{mod}(S_1) \cup \text{mod}(S_2) \). On the other hand, \( \text{mod}(\text{while}(C) \ S_1) \) is defined as \( \text{mod}(S_1) \).

### 3.2 Composing Hoare Triples

Given two summaries \([\varphi_1] \cdot S_1 \cdot [\hat{\varphi}_1] \) and \([\varphi_2] \cdot S_2 \cdot [\hat{\varphi}_2] \), we wish to compute a summary for the composite statement \( S_1; S_2 \). If we can compute formulas \( \varphi_{\text{pre}} \) and \( \varphi_{\text{post}} \) that are independent of \( \text{mod}(S_1) \) and \( \text{mod}(S_2) \), respectively, such that \( \hat{\varphi}_1 \ast \varphi_{\text{pre}} \Rightarrow \hat{\varphi}_2 \ast \varphi_{\text{post}} \), then \( [\varphi_1] \cdot S_1 \cdot [\hat{\varphi}_1] \Rightarrow [\varphi_2] \cdot S_2 \cdot [\hat{\varphi}_2] \).


<table>
<thead>
<tr>
<th>Mutation</th>
<th>( { v \leftarrow (f_1, \ldots, f_n) } \iff { v \leftarrow (f_1, \ldots, f_n) } )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deallocation</td>
<td>( { v \leftarrow (f_1, \ldots, f_n) } \iff { v \leftarrow \text{null} } )</td>
</tr>
<tr>
<td>Allocation (modifies ( v ))</td>
<td>( { v \leftarrow (f_1, \ldots, f_n) } \iff { v \leftarrow (f_1, \ldots, f_n) } )</td>
</tr>
<tr>
<td>Lookup (modifies ( v ))</td>
<td>( { v = u \iff (f_1, \ldots, f_n) } \iff { v = u \iff (f_1, \ldots, f_n) } )</td>
</tr>
<tr>
<td>Copy (modifies ( v ))</td>
<td>( { v = e } \iff { v = e } )</td>
</tr>
<tr>
<td>Guard</td>
<td>( { v \neq e } \iff { v \neq e } )</td>
</tr>
</tbody>
</table>

Table I. Local reasoning rules for primitive statements

\( \varphi_{\text{pre}} \iff \varphi_{\text{post}} \iff \varphi_2 \), then by application of frame rule we can infer the summary \( [\varphi_1 \iff \varphi_{\text{pre}}] S1; S2 [\varphi_{\text{post}} \iff \varphi_2] \). We can compose the two given summaries even under the slightly modified condition \( \varphi_1 \iff \varphi_{\text{pre}} \iff \exists Z. (\varphi_{\text{post}} \iff \varphi_2) \), if \( Z \subseteq \text{Aux} \). The summary inferred in this case is \( [\varphi_1 \iff \varphi_{\text{pre}}] S1; S2 [\exists Z. (\varphi_{\text{post}} \iff \varphi_2)] \).

Given \( \varphi_1 \) and \( \varphi_2 \), we refer to the determination of \( \varphi_{\text{pre}}, \varphi_{\text{post}} \) and a set \( Z \) of variables such that \( \varphi_1 \iff \varphi_{\text{pre}} \iff \exists Z. (\varphi_{\text{post}} \iff \varphi_2) \) as strong bi-abduction. The concept of strong bi-abduction is similar to that of bi-abduction presented in [Calcagno et al. 2009] (in the context of using a Hoare triple computed for a procedure at a particular callsite to the procedure). Key differences are that bi-abduction requires the condition \( \varphi_1 \iff \varphi_{\text{pre}} \iff \exists Z. (\varphi_{\text{post}} \iff \varphi_2) \), whereas we seek equivalence (instead of implication) while allowing some auxiliary variables to be existentially quantified in the right hand side of the equivalence. While the above composition rule is sound even if we use bi-abduction, bi-abduction may not yield good post-conditions. Specifically, if we disallow the deallocation operation, it can be shown that the composition of strong Hoare triples using strong bi-abduction yields strong Hoare triples (refer to the Appendix for a proof). The ‘strong’ property is not preserved under composition using bi-abduction, although the composition is sound. A drawback of using strong bi-abduction, however, is that there exist Hoare triples that cannot be composed using strong bi-abduction but can be composed using bi-abduction. For example, \( [\text{true}] v := \text{null} \iff [v = \text{null}] \) and \( [\text{true}] v := \text{null} \iff [v = \text{null}] \) cannot be composed using strong bi-abduction but can be composed using bi-abduction. However, even with this drawback our tool could generate complete functional specifications for most of the benchmark programs using strong bi-abduction in a bottom-up analysis.

Example 1. In this and subsequent examples, we will use \( v \mapsto w \) as a short-hand for \( v \mapsto (\text{next} : w) \). Let us compose two summaries, \( [v = \l] \iff \text{new } [\exists \l. v \mapsto \l] \) and \( [v = \l \land \l \mapsto \l] \iff v := v \text{next } [v = \l \land \l \mapsto \l] \). Note that all variables other than \( v \) are distinct in the two summaries, as they represent implicitly existentially quantified auxiliary variables in each of the two summaries. Since \( (\exists \l. v \mapsto \l) \iff (\exists \l. v \mapsto \l) \iff (\exists \l. v := \text{new } v \iff v \text{next } \exists \l. v = \l \land \l \mapsto \l) \). As an aside, note that the program fragment \( v := \text{new } v \); v := v \text{next } \) introduces a memory leak.

We now present a set of Hoare inference rules in separation logic for our programming language. The rules are formally presented in Figure 4. The \( \text{COMPOSE} \) rule captures the above idea of using strong bi-abduction for the sequential composition of statements. The rules \( \text{WHILE}, \text{THEN} \) and \( \text{ELSE} \) use the \( \text{COMPOSE} \) rule to derive the fact in their antecedent.
The rules EXIT and WHILE are straightforward rules that decompose analysis of loops into two cases. Rule EXIT handles the case where the loop executes zero times, while rule WHILE applies when the loop executes one or more times. Rule WHILE leaves the bulk of the work to the computation of \([\varphi] \cdot S^+ [\widehat{\varphi}]\). The notation \([\varphi] \cdot S^+ [\widehat{\varphi}]\) does not represent a Hoare triple in the standard sense, since \(S^+\) is not a statement in our programming language. However, \([\varphi] \cdot S^+ [\widehat{\varphi}]\) is the key idiom we will use in the remainder of this paper. Hence, we overload the notation of Hoare triples, and also call \([\varphi] \cdot S^+ [\widehat{\varphi}]\) a Hoare triple. The notation \([\varphi] \cdot S^+ [\widehat{\varphi}]\) means that for every initial state satisfying \(\varphi\), there exists a \(k \geq 1\) such that the state resulting after \(k\) executions of \(S\) satisfies \(\widehat{\varphi}\). Note that this Hoare triple is used only in the WHILE rule. In this rule, the second premise ensures that the state obtained after \(k\) iterations does not satisfy the loop condition, and hence the loop terminates. In next two sections we present a technique for computing triples of the form \([\varphi] \cdot S^+ [\widehat{\varphi}]\).

### 4. LOGIC OF ITERATED SEPARATION FORMULAE (LISF)

Let \(S_L\) denote the following loop in our programming language: \textbf{while } (v != \textbf{null}) v := v . next. Let \(\otimes_k \varphi^i\) informally denote the iterated separating conjunction \(\varphi^0 \cdot \cdots \cdot \varphi^k\) [Reynolds 2002]. We would like to infer the following summary for \(S_L\): \(v = \varphi_0 \wedge \varphi_k = \textbf{null} \wedge \otimes_{i=0}^{k-1} \varphi_i \mapsto \varphi_{i+1}\) \(S_L [v = \varphi_k \wedge \varphi_k = \textbf{null} \wedge \otimes_{i=0}^{k-1} \varphi_i \mapsto \varphi_{i+1}]\). The objective of this section is to present a formal extension of separation logic that lets us express such triples using a restricted form of iterated separating conjunction. We begin by giving an overview of how we intend to infer loop summaries like the one above.

Assume that we have a Hoare triple \([\varphi] S [\widehat{\varphi}]\), where \(\varphi\) and \(\widehat{\varphi}\) are quantifier-free formulae. We can compute a Hoare triple for \(k\) executions of \(S\) by repeated applications of the \textbf{COMPOSE} rule as follows. Let \(\varphi^i\) (resp. \(\widehat{\varphi}^i\)) denote \(\varphi\) (resp. \(\widehat{\varphi}\)) with every variable \(x \in \text{Aux}\) replaced by a corresponding indexed variable \(x_i\). Consider the Hoare triples \([\varphi^i] S [\widehat{\varphi}^i]\) and \([\varphi^{i+1}] S [\widehat{\varphi}^{i+1}]\), obtained from \([\varphi^i] S [\widehat{\varphi}^i]\) by replacing variables in \(\text{Aux}\) by indexed variables as described above. Let \(\varphi^i_{\text{pre}}\) and \(\varphi^i_{\text{post}}\) be such that both \(\text{free}(\varphi^i_{\text{pre}}) \cap \text{mod}(S)\) and \(\text{free}(\varphi^i_{\text{post}}) \cap \text{mod}(S)\) are empty, and \(\varphi^i \cdot \varphi^i_{\text{pre}} \Leftrightarrow \varphi^i_{\text{post}} \cdot \varphi^{i+1}\). Note that unlike \(\varphi^i\) or \(\widehat{\varphi}^i\), we allow \(\varphi^i_{\text{pre}}\) and \(\varphi^i_{\text{post}}\) to have free variables with indices \(i\) as well as \(i + 1\). We can now inductively apply

<table>
<thead>
<tr>
<th>\textbf{COMPOSE}</th>
<th>\textbf{BRANCH}</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\varphi_1] S_1 \ [\widehat{\varphi}_1] ]</td>
<td>[\varphi \wedge B] \ S_1 \ [\widehat{\varphi}]</td>
</tr>
</tbody>
</table>
| \[\varphi_2] S_2 \ [\widehat{\varphi}_2] \] | \[\varphi \wedge B\] S_2 \ [\widehat{\varphi}]

where \(\varphi\) and \(\widehat{\varphi}\) are quantifier-free formulae. We can compute a Hoare triple for \(k\) executions of \(S\) by repeated applications of the \textbf{COMPOSE} rule as follows. Let \(\varphi^i\) (resp. \(\widehat{\varphi}^i\)) denote \(\varphi\) (resp. \(\widehat{\varphi}\)) with every variable \(x \in \text{Aux}\) replaced by a corresponding indexed variable \(x_i\). Consider the Hoare triples \([\varphi^i] S [\widehat{\varphi}^i]\) and \([\varphi^{i+1}] S [\widehat{\varphi}^{i+1}]\), obtained from \([\varphi^i] S [\widehat{\varphi}^i]\) by replacing variables in \(\text{Aux}\) by indexed variables as described above. Let \(\varphi^i_{\text{pre}}\) and \(\varphi^i_{\text{post}}\) be such that both \(\text{free}(\varphi^i_{\text{pre}}) \cap \text{mod}(S)\) and \(\text{free}(\varphi^i_{\text{post}}) \cap \text{mod}(S)\) are empty, and \(\varphi^i \cdot \varphi^i_{\text{pre}} \Leftrightarrow \varphi^i_{\text{post}} \cdot \varphi^{i+1}\). Note that unlike \(\varphi^i\) or \(\widehat{\varphi}^i\), we allow \(\varphi^i_{\text{pre}}\) and \(\varphi^i_{\text{post}}\) to have free variables with indices \(i\) as well as \(i + 1\). We can now inductively apply
The Compose rule and conclude the following Hoare triple.

\[
[\varphi^0 \land (\bigcirc_{i=0}^{k-1} \varphi^i_{\text{pre}})]S^{k+1}[(\bigcirc_{i=0}^{k-1} \varphi^i_{\text{post}}) \land \varphi^k]
\]  

(4.1)

We call the inference of the Hoare triple in equation (4.1) as acceleration of \([\varphi] S \hat{=} \hat{\varphi}\).

The following example illustrates acceleration of Hoare triples.

**Example 2.** Let \(S\) be the sequence of statements assert(\(v! = \text{null}\)); \(v := v.n\ ext\).

Suppose we wish to compose the two summaries \([v = x_0 \land x_0 \mapsto y_1] S [v = x_0 \land x_0 \mapsto y_0]\) and \([v = x_1 \land x_1 \mapsto y_1] S [v = y_1 \land x_1 \mapsto y_1]\), which are identical, except for renaming of auxiliary variables. Let \(\varphi_{\text{pre}}\) denote \(x_1 = x_0 \land x_0 \mapsto y_0\) and \(\varphi_{\text{post}}\) denote \(x_1 = y_1 \land x_1 \mapsto y_1\). Applying the Compose rule results in the following summary: \([v = x_0 \land x_0 \mapsto y_0] \land (x_1 = y_0 \land x_1 \mapsto y_0)\). This is pictorially depicted in Figures 5 (a) and (b). Iterative application of Compose, or acceleration, yields the summary: \([v = x_0 \land x_0 \mapsto y_0] \land (x_1 = y_0 \land x_1 \mapsto y_0)\). This is pictorially depicted in Figure 5(c).

## 4.1 LISF Syntax and Informal Semantics:

We now introduce an extension of separation logic, called Logic of Iterated Separation Formulae (or LISF), that allows us to formally express the restricted form of iterated separating conjunction alluded to above. The syntax of LISF is given in Figure 6, where “…” represents standard constructs of separation logic from Figure 2.

As we will soon see, we no longer need the informal notation \((v = x_0) \land (x_k = \text{null}) \land \bigcirc_{i=0}^{k-1} (v_i \mapsto w_{i+1})\) to describe an acyclic singly linked list pointed to by \(v\). Instead, we can use the LISF formula \(\varphi \equiv (v = A[0]) \land (A[0] = \text{null}) \land RS(A[1] \mapsto A[1] + 1, 0, 0),\) where \(A\) is a new type of logical variable and \(RS\) is a new predicate, as explained below.

Variables like \(A\) in the formula \(\varphi\) represent a new type of logical variables, called array variables, that may be referenced in LISF formulae. Intuitively, an array variable represents a sequence of locations corresponding to the “nodes” of a re-
cursive data structure like a linked list. A \textit{LISF} formula may specify properties of the \(i^{th}\) node in such a data structure, or specify a relation between the \(i^{th}\) and \(i + 1^{st}\) nodes of the same (or even different) data structure(s), by referring to elements of the corresponding arrays. In general, the syntax of \textit{LISF} also allows references to multi-dimensional array variables. This is particularly useful for describing nested recursive data structures, such as a linked list of linked lists. As a matter of convention, we will henceforth denote array variables with bold-face upper case letters.

\[ \text{ae ::= arr} \mid \text{ae[\ldots]} \mid \text{ae[i]} \mid \text{ae[i+1]} \mid \text{ae[\ldots]} \mid \text{ae[\ldots]} \]

\[ e ::= \ldots \mid \text{ae[\ldots]} \mid \text{ae[i]} \mid \text{ae[i+1]} \mid \text{ae[\ldots]} \mid \text{ae[\ldots]} \]

\[ P ::= \ldots \mid \text{RP(P,l,u)} \]

\[ S ::= \ldots \mid \text{RS(S,l,u)} \]

\[ SH ::= P \land S \mid \exists e \ SH \mid \exists \text{arr} \ SH \]

Fig. 6. \textit{LISF} assertion syntax to allow indexed array references, also called \textit{array expressions}, which consist of an array variable name followed by a sequence of one or more indices. An array expression can take one of four forms: (i) \text{arr}[c], (ii) \text{arr}[$c$], (iii) \text{arr}[::], or (iv) \text{arr}[$::$ + 1], where \(c\) is a non-negative integer constant, and \text{arr} is either an array name or an array expression. Array expressions with \textit{iterated indices} include array references of the form \text{arr}[c] or \text{arr}[$c$]. These refer to the element at an offset \(c\) from the beginning or end, respectively, of the sequence represented by \text{arr}. For example, if \(A\) is mapped to the sequence \((v_0, \ldots, v_k)\), then the array expressions \(A[0]\) and \(A[00]\) evaluate to \(v_0\) and \(v_k\) respectively in \textit{LISF} semantics. The semantics of array expressions with \textit{iterated indices}, which include references of the form \text{arr}[$\ldots$] and \text{arr}[$::$ + 1], will be explained later.

In addition to array variables, \textit{LISF} extends pure and spatial formulae with a pair of new predicates, called \textit{RP} and \textit{RS}. These predicates are intended to be used for describing pure and spatial properties, respectively, that repeat across nodes of recursive data structures. Loosely speaking, if \(S\) denotes a spatial formula containing an array expression with \textit{iterated indices}, such as \text{arr}[$\ldots$] or \text{arr}[$::$ + 1], then \text{RS}(S, l, u)\) corresponds to our informal notation \(\copyright k^{1 \ldots u} S\). Note, however, that the index variable \(i\) and bound \(k\) are not explicitly represented in \text{RS}(S, l, u). Instead, the values of \(i\) and \(k\) are provided by the evaluation context. The “dot” in \text{arr}[$\ldots$] or \text{arr}[$::$ + 1] intuitively refers to the implicit index variable \(i\). Thus, \text{arr}[$\ldots$] refers to the element at offset \(i\), while \text{arr}[$::$ + 1] refers to the element at offset \(i + 1\). To see how the \text{RS} predicate is used, consider the formula \(\text{RS}(A[::] \rightarrow A[::] + 1, 0, 0)\), where \(A\) is mapped to a sequence of length \(k + 1\). This formula asserts that for all \(i \in [0, k - 1]\), the \(i^{th}\) element of \(A\) is the location of a heap cell whose \textit{next} field has the same value as the \(i + 1^{st}\) element of \(A\). In addition, the predicate also asserts that the heap cells represented by elements \(A[0]\) through \(A[k-1]\) are distinct. The usage and intuitive interpretation of \textit{RP} is similar to that of \textit{RS}, with the exception that \textit{RP} is used with a pure sub-formula \(P\) (as in \text{RP}(P, l, u)) instead of the spatial sub-formula \(S\) in \text{RS}(S, l, u). For notational convenience, we will henceforth denote \text{RP}(P, l, u) and \text{RS}(S, l, u) simply by \text{RP}(P) and \text{RS}(S), respectively, when both \(l\)
and \( u \) are 0.

While the \( \text{RP} \) and \( \text{RS} \) predicates are clearly motivated by Reynolds’ iterated separating conjunction operator [Reynolds 2002], there are some differences as well. Most important among these is the absence of an explicit iteration bound in the syntax of \( \text{RP} \) and \( \text{RS} \). Specifically, the iteration bounds in \( \text{RS}(\mathcal{S}, l, u) \) and \( \text{RP}(\mathcal{P}, l, u) \) are provided by the lengths of sequences mapped to array variables with iterated indices in the sub-formulae \( \mathcal{S} \) and \( \mathcal{P} \), respectively. This implicit encoding of bounds allows us to uniformly represent simple and nested data structures in a size-independent manner. To see this, consider a linked list in which every element itself points to a distinct nested linked list. Suppose further that the nested linked lists have different lengths. If we were to represent this data structure using iterated separating conjunctions, we would need a formula with two iterated separating conjunctions, one nested within the scope of the other. Furthermore, the upper bound of the inner iterated separating conjunction would need to be expressed as a function of the index of the outer iterated separating conjunction. Clearly, this poses additional complications for algorithms that reason about and manipulate such formulae. In contrast, the same data structure can be expressed in \( \text{LISF} \) (with the shorthand \( \text{RS}(\mathcal{S}) \) for \( \text{RS}(\mathcal{S}, 0, 0) \)) as

\[
\begin{aligned}
\text{RS} & \left( x \mapsto (\text{nlist} : A[\cdot][0], \text{next} : x[\cdot] + 1) \right) \\
\land & \ (A[\cdot][0] = \text{null}) \\
\land & \ \text{RS} (A[\cdot][\cdot] \mapsto (A[\cdot][\cdot] + 1))
\end{aligned}
\land (x[0] = (\text{nlist} : A[0][0], \text{next} : \text{null}),
\]

where \( x \) is a uni-dimensional array representing elements (with \( \text{nlist} \) and \( \text{next} \) fields) of the outer linked list, and \( A \) is a two-dimensional array representing elements (with a \( \text{next} \) field) of the nested linked lists. The semantics of this formula will become clear once we discuss the formal semantics of \( \text{LISF} \) in the next section. However, notice that the formula is syntactically independent of the sizes of individual linked lists. As we will see later, our bi-abduction and acceleration algorithms also do not require explicit bounds of iterated separating conjunctions. Consequently, we choose to keep these bounds implicit. Another way in which the usage of \( \text{RP} \) and \( \text{RS} \) predicates differs from that of iterated separating conjunctions is that the lower and upper bounds of iteration are expressed as offsets from the start and end, respectively, of the sequences mapped to array variables. This allows us to refer to elements at a fixed offset from the beginning or end of a linked list, for example, without explicitly referring to the length of the list. In summary, the \( \text{RP} \) and \( \text{RS} \) predicates may be viewed as variants of Reynolds’ iterated separating conjunction operator, in which iteration bounds and indices are implicitly represented, and are provided by the evaluation context.

4.2 \( \text{LISF} \) Semantics

We now extend the semantics of separation logic and formally define the semantics of \( \text{LISF} \). Since an \( \text{LISF} \) expression may be an array reference with one or more iterated indices, we require the mapping of array variables to uni- or multi-dimensional sequences of locations, and a list of integers, one for every iterated index, to evaluate an \( \text{LISF} \) expression in general. Formally, the semantics of an \( \text{LISF} \) expression \( e \) is given by the function \( \mathcal{E}(e, L', s, V) \), shown in Figure 7. This function takes as inputs an \( \text{LISF} \) expression \( e \), a list \( L' \) of non-negative integer
values, a stack $s$, and a mapping $V$ of array variables to uni- or multi-dimensional sequences of locations, and returns a location as the value of $e$.

If $e$ is a variable that is not an array, $E$ simply looks up the stack and returns $s(e)$ as the value of $e$. If $e$ is the constant null, $E$ returns null. However, if $e$ is an array expression, $E$ uses the list $L'$ of integers and the mapping $V$ of array variables to sequences of locations to determine the value of $e$. Intuitively, integers from the list $L'$ are used to instantiate the iterated indices, $\lfloor \cdot \rfloor$ and $\lfloor \cdot + 1 \rfloor$, appearing in $e$. Thus, we need at least as many integers in $L'$ as the number of iterated indices in $e$. This is ensured by the first precondition of function $E(e, L', s, V)$, shown in Figure 7, where the function $\text{NumIterInd}(e)$ gives the number of iterated indices in $e$. Formally, $\text{NumIterInd}(e)$ is defined as follows: If $\text{array.var}$ denotes an array variable, $ae$ denotes an array expression and $v$ denotes a non-array variable, then $\text{NumIterInd}(\text{array.var}) = 0$, $\text{NumIterInd}(ae[\cdot]) = \text{NumIterInd}(ae[\cdot] + 1) = \text{NumIterInd}(ae) + 1$, $\text{NumIterInd}(ae[\cdot]) = \text{NumIterInd}(ae[\cdot]v) = \text{NumIterInd}(ae)$, and $\text{NumIterInd}(v) = \text{NumIterInd}(\text{null}) = 0$. If $e$ is an array expression of the form $\text{array.var}$ followed by $k$ (fixed or iterated) indices, then $V$ must map $\text{array.var}$ to a $k$-dimensional sequence of locations in order to avoid indexing errors during evaluation of $e$ and to ensure that $E(e, L', s, V)$ evaluates to a unique location. This is formalized in the second precondition of $E(e, L', s, V)$.

In general, a list $L'$ satisfying the first precondition of $E(e, L', s, V)$ may contain more integers than $\text{NumIterInd}(e)$. Therefore, we use the function suffix to extract a suffix of $L'$ of the same length as $\text{NumIterInd}(e)$. The “match $e$” construct used in Figure 7 implements a case split based on the structure of the expression $e$ (analogous to the match expression of functional programming languages like ML). The helper function $E_a$ implements evaluation of an array expression, as outlined above. It takes as inputs an array expression $aexpr$, a list $L$ of integers and a mapping $V$ of array variables to sequences of locations. The instantiation of iterated indices in $aexpr$ with integers from $L$ is done recursively. Specifically, each

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
recursive call instantiates the current rightmost un-instantiated iterated index of \( \texttt{aexpr} \) with the integer at the head of \( L \), and passes the rest of \( L \), i.e. its tail, as argument to the next recursive call. Function \( \mathcal{E}_a \) has preconditions similar to those of \( \mathcal{E} \), except that the dimension of \( \mathcal{V}(\texttt{array_var}) \) is allowed to be greater than the number of indices (fixed or iterated) following \( \texttt{array_var} \) in \( e \). Initially, function \( \mathcal{E}_a \) is called from function \( \mathcal{E} \). The preconditions of \( \mathcal{E} \) and the fact that \( L \) is set to a suffix of \( L' \) of length \( \text{NumIterInd}(\texttt{aexpr}) \) ensure that the preconditions of \( \mathcal{E}_a \) are satisfied when it is called from within \( \mathcal{E} \). Subsequently, each recursive call of \( \mathcal{E}_a \) reduces the number of (fixed or iterated) indices of \( \texttt{aexpr} \) by exactly 1. Moreover, the number of iterated indices is reduced by 1 in exactly those cases where the length of the list \( L \) is also reduced by 1. This ensures that once the preconditions of \( \mathcal{E}_a \) are satisfied in the initial call, they will continue to be satisfied in every subsequent recursive call.

Let \( \texttt{aexpr} \) be of the form \( \texttt{array_var} \) followed by \( k' \) (fixed or iterated) indices. Let the dimension of \( \mathcal{V}(\texttt{array_var}) \) be \( k \). The second precondition of \( \mathcal{E}_a(\texttt{aexpr}, L, \mathcal{V}) \) ensures that \( k \geq k' \). It is an easy exercise to see that \( \mathcal{E}_a(e, L, \mathcal{V}) \) returns a \((k-k')\)-dimensional sequence of locations. Therefore, if \( k = k' \), function \( \mathcal{E}_a(e, L, \mathcal{V}) \) returns a unique location. Note that the second precondition of function \( \mathcal{E}(e, L', s, \mathcal{V}) \) ensures that whenever \( \mathcal{E}_a \) is called from within \( \mathcal{E} \), we have \( k = k' \). Therefore, every call of \( \mathcal{E}_a \) from within \( \mathcal{E} \) returns a unique location. The functions \( \text{hd}(L) \) and \( \text{tl}(L) \) used in the definition of \( \mathcal{E}_a \) in Figure 7 return the head and tail, respectively, of the list \( L \). Similarly, if \( \mathcal{E}_a(e, L, \mathcal{V}) \) returns a sequence \( a \), the function \( \text{length}(a) \), used in the definition of \( \mathcal{E}_a \), returns the number of elements in \( a \).

We now define a class of well-formed LIS\(F \) formulae or \( \text{(wff)} \). The semantics is non-trivially defined only for well-formed formulae. A LIS\(F \) formula that is not well-formed does not have a model. For notational convenience, we overload the function \( \text{NumIterInd} \), used in the definition of \( \mathcal{E}(e, L', s, \mathcal{V}) \) above, to operate over expressions as well as predicates. Specifically, the function \( \text{NumIterInd} \) is defined over predicates as follows. \( \text{NumIterInd}(e_1 \sim e_2) = \max(\text{NumIterInd}(e_1), \text{NumIterInd}(e_2)) \), \( \text{NumIterInd}(P_1 \land P_2) = \text{NumIterInd}(P_1) \), \( \text{NumIterInd}(\text{RP}(P, \_)) = \text{NumIterInd}(P) - 1 \), \( \text{NumIterInd}(e \mapsto (f_1 : l_i)) = \text{NumIterInd}(e) \), \( \text{NumIterInd}(S_1 * S_2) = \text{NumIterInd}(S_1) \), \( \text{NumIterInd}(\text{RS}(S, \_)) = \text{NumIterInd}(S) - 1 \). An LIS\(F \) formula \( P \land S \) is then said to be well-formed iff (i) \( \text{NumIterInd}(P) = \text{NumIterInd}(S) = 0 \), (ii) for every subformula \( P_1 \land P_2 \) of \( P \), we have \( \text{NumIterInd}(P_1) = \text{NumIterInd}(P_2) \), (iii) for every subformula \( S_1 * S_2 \) of \( S \), we have \( \text{NumIterInd}(S_1) = \text{NumIterInd}(S_2) \), and (iv) for every subformula \( e_1 \mapsto (f : e_2) \) of \( S \), we have \( \text{NumIterInd}(e_1) \geq \text{NumIterInd}(e_2) \).

Structures modeling well-formed LIS\(F \) formulae are tuples \((s, h, V)\), where \( s \) is a stack, \( h \) is a heap, and \( V \) is a mapping of array variables to un- or multi-dimensional sequences of locations. The semantics of assertions is given by the satisfaction relation \( (\models) \) between a structure augmented with a list of integers \( L \), and an assertion \( \varphi \). The list of integers facilitates evaluation of array expressions by the function \( \mathcal{E} \) described above. The formal definition of \( (s, h, V, L) \models \varphi \) is given in Figure 8. Here, the notation \( i : L \) denotes the list \( L' \) obtained by inserting \( i \) at the head of an already existing list \( L \). Similarly, the notation \( [V][\text{arr}: a] \) denotes the mapping \( V' \) defined by \( V'(\text{arr}) = a \), and \( V'(x) = V(x) \) for all array variables \( x \) different from \( \text{arr} \). We say that \( (s, h, V, []) \models \varphi \).
The lengths of sequences accessed by array expressions in $\varphi$ have non-zero lengths.

Let $\varphi$ be a well-formed $\textit{LISF}$ formula containing array expression(s), and let $(s, h, V)$ be a structure over which we wish to evaluate $\varphi$. It follows from the definition of the semantics (Figure 8) that in order to determine if $(s, h, V, []) \models \varphi$, we must evaluate all array expressions in $\varphi$ in general. In order to avoid indexing errors when evaluating array expressions, certain restrictions must be imposed on the mapping $V$, and hence on the structure $(s, h, V)$. This motivates us to define the set of well-formed structures for a given well-formed $\textit{LISF}$ formula $\varphi$. For notational convenience, we will denote this set by $wfs_\varphi$. Intuitively, a structure $(s, h, V)$ in $wfs_\varphi$ avoids indexing errors during the evaluation of array expressions in $\varphi$ by ensuring that whenever function $E$ is called, the corresponding preconditions (see Figure 7) are satisfied, and no out-of-bounds exception occurs. Formally, a structure $(s, h, V)$ is said to be in $wfs_\varphi$ if $s$ and $h$ are a stack and heap, in the usual sense of semantics of separation logic, and the mapping $V$ satisfies the following conditions.

1. Let $ae$ be a maximally indexed array expression in $\varphi$, i.e. an array expression that is not a sub-expression of another array expression in $\varphi$. Let the underlying array variable in $ae$ be $\text{array\_var}$, and let $ae$ be of the form $\text{array\_var}$ indexed by a sequence of $k$ (iterated and fixed) indices. Then the dimension of $V(\text{array\_var})$ equals $k$.

2. The lengths of sequences accessed by array expressions in $\varphi$ are such that no out-of-bounds exception occurs when function $E$ is used to evaluate these expressions in the definition of the semantics (Figure 8). Specifically:
   (a) If $e[c]$ or $e[\$c]$ is an array expression in $\varphi$, every sequence to which $e$ evaluates to during evaluation of $\varphi$ is of length at least $c + 1$.
   (b) Let $\psi$ be a sub-formula nested within $n \geq 1$ RP (or RS) predicates in $\varphi$. In general, $\psi$ may refer to one or more array expressions. For every pair of array expressions $e_1$ and $e_2$ in $\psi$ that have at least $n$ indexed indices, the sequences accessed by the $n^{th}$ iterated index of $e_1$ and $e_2$ always have the same length.

3. All sequences mapped to array variables by $V$ have non-zero lengths.
Let $\varphi$ be a well-formed $\mathcal{LISF}$ formula, $(s,h,V)$ be a structure in $\mathcal{NF}_\varphi$, and $L$ be a list of $r$ integers, where $r \geq \text{NumIterInd}(ae)$ for all array expressions $ae$ in $\varphi$. From the semantics of $(s,h,V,L) \models \varphi$ given in Figure 8, we find that for all constructs borrowed from standard separation logic, the semantics remains unchanged. The semantics of predicates $\mathcal{RS}$ and $\mathcal{RP}$, which are novel to $\mathcal{LISF}$, however, deserve some explanation. Consider a $\mathcal{RP}(P,l,u)$ (or $\mathcal{RS}(S,l,u)$) predicate nested inside $n - 1$ other $\mathcal{RP}$ (or $\mathcal{RS}$) predicates. The length of the sequence accessed by the $n^{\text{th}}$ iterated index of every array expression in $P$ (or $S$) is guaranteed to be identical by the requirement of well-formed structures of a formula. Given a list $L$ of $n - 1$ index values corresponding to the evaluation context arising from the outer $\mathcal{RP}$ (or $\mathcal{RS}$) predicates, function $\text{len}(V,L,P)$ (or $\text{len}(V,L,S)$) determines the length, say $k + 1$, of the sequence accessed by the $n^{\text{th}}$ iterated index of an array expression in $P$ (or $S$). The semantics of $\mathcal{RP}(P,l,u)$ then requires that $P$ holds for each array index $i$ ranging from $l$ to $k − 1 − u$. Similarly, the semantics of $\mathcal{RS}(S,l,u)$ requires that $S$ holds over a sub-heap $h_i$ of $h$ for each array index $i$ ranging from $l$ to $k − 1 − u$, with the additional constraint that the $h_i$’s are also pair-wise disjoint. Note also that the definition of $\text{wff}$ ensures that whenever $E(ae,L,s,V)$ is invoked in the definition of the semantics, then $ae$ is a maximally indexed array expression.

4.3 Comparison with summaries generated by separation logic based automated shape analysis tools

In $\mathcal{LISF}$ we represent the values of variables in successive instances of a repeated formula by using an array instead of hiding them under an existential quantifier of a recursive predicate. This enables us to relate the data-structures before and after the execution of a loop. This is crucial for generating succinct specifications. In the following, we illustrate how more succinct specifications can be generated using $\mathcal{LISF}$ compared to those generated using recursive predicates by recent shape analysis algorithms [Distefano et al. 2006; Berdine et al. 2007; Calcagno et al. 2007; 2009].

Consider a procedure $\text{traverse}$ containing the loop $S_L$: $\textbf{while}(v! = \text{null}) \; v := v.\text{next}$, that traverses a singly linked list. Let each element of the list have two fields named $\text{next}$ and $D$. A summary in $\mathcal{LISF}$ is $[v = x[0] \land \mathcal{RS}(x[\cdot]) \Rightarrow (\text{next} : x[\cdot] + 1); \; D : v[\cdot] \land x[80] = \text{null}] \; \text{traverse}(v) \; [v = x[80] \land \mathcal{RS}(x[\cdot]) \Rightarrow (\text{next} : x[\cdot] + 1); \; D : v[\cdot] \land x[80] = \text{null}]$. This summary states that $\text{traverse}$ neither modifies the elements of the linked list nor the relative links between them. The shape analysis algorithms presented in [Distefano et al. 2006; Berdine et al. 2007; Calcagno et al. 2007; 2009] would generate the summary $[\text{list}(v.\text{next})] \; \text{traverse}(v) \; [\text{list}(v.\text{next})]$, using the recursive predicate $\text{list}(v,\text{next})$. This summary does not indicate whether the input list or the contents of any of its elements are modified.

Consider the composite statement $\text{traverse}(v); \text{check}(v)$, where the procedure $\text{check}$ requires, as precondition, a linked list pointed to by $v$ with the $D$ field of each element pointing to $h$. This precondition cannot be expressed using the $\text{list}$ recursive predicate. Let $\text{clist}(v,\text{next},h)$ be the recursive predicate that captures the desired precondition. The above two statements cannot be composed unless we have a summary for $\text{traverse}$ that describes the data structure using the $\text{clist}$ predicate. This is because the postcondition of $[\text{list}(v.\text{next})] \; \text{traverse}(v) \; [\text{list}(v.\text{next})]$ does not indicate whether the content of any element of the list is modified by $\text{traverse}$. 

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
Thus, either (i) we need to generate summaries for traverse using all possible recursive predicates (e.g. list, clist, dll) that may be required in some part of the code, leading to an explosion of summaries, or (ii) we need to reanalyze traverse with new recursive predicates, making the analysis non-modular. Note that even if we use the generic predicates defined in [Berdine et al. 2007] to capture both the predicates list and clist in a common framework, the summary for traverse computed using such predicates does not assert that none of the list elements are modified by traverse. Hence it is not possible to generate a succinct set of summaries for traverse that can be used in modular analysis using the recursive predicates and shape analysis algorithms presented in [Distefano et al. 2006; Berdine et al. 2007; Calcagno et al. 2007; 2009].

In LISF, the precondition for check can be expressed as \( v = x[0] \land RS(x[:]) \mapsto (\text{Next} : x[+1]; D : h) \land x[80] = \text{null} \). The summaries for traverse and check can indeed be composed using strong bi-abduction. For this composition, both the formulas \( \varphi_{\text{pre}} \) and \( \varphi_{\text{post}} \) can be set to \( \text{RP}(y[:]) = h \). Thus, we can use the LISF summary for traverse in any context that requires the postcondition of traverse to satisfy some properties in addition to the singly linked list structure, thereby facilitating modular analysis. Note that relational summaries can be expressed using higher order recursive predicates other than LISF, as illustrated in [Biering et al. 2005]. However, we do not know of any other automated tool that generates relational summaries using higher order recursive predicates.

5. INDUCTIVE COMPOSITION

The rules introduced in Figure 4 are valid even with LISF extension of separation logic. The set of auxiliary variables, Aux, includes the array variables in this extension. For clarity, we adopt the following convention in the remainder of the paper: (i) unless explicitly stated, all formulas in LISF are quantifier free, (ii) Hoare triples are always expressed as \( [\varphi] \ S [\exists X. \vec{\varphi}] \), (iii) \( \text{free}(\vec{\varphi}) = V \cup W \) and \( \text{free}(\varphi) = V \cup W \cup X \), where \( V \) denotes the set of logical variables representing values of program variables, and \( W, X \) are sets of auxiliary variables, including array variables\(^1\). Thus \( W \) is the set of free auxiliary variables occurring in \( \varphi \) and in \( \exists X. \vec{\varphi} \).

5.1 Inference rule Induct

Let \( [\varphi] \ S [\exists X. \vec{\varphi}] \) be a Hoare triple. We wish to compute a strong summary for \( S^+ \). In Figure 5 and Example 2 we have presented the intuition of acceleration that computes summaries of the form \( [\varphi] \ S^+ [\vec{\varphi}] \) from the summary of \( S \). We formalize this intuition in the inference rule Induct as shown in Figure 9. As in the previous Section, we use \( \varphi^1 \) (resp. \( \vec{\varphi}^1 \)) to denote \( \varphi \) (resp. \( \vec{\varphi} \)) with every free auxiliary variable \( w \in W \) replaced by an indexed variable \( w_i \). Let \( \varphi^0_{\text{pre}}, \varphi^0_{\text{post}} \) be formulas such that \( \text{free}(\varphi^0_{\text{pre}}) \) and \( \text{free}(\varphi^0_{\text{post}}) \) are disjoint from \( \text{mod}(S) \) and \( (\exists X. \varphi^0) * \varphi^0_{\text{pre}} \equiv \varphi^0_{\text{post}} * \varphi^1 \). Note that the premises 4, 5, and 6 of Induct imply that \( \text{free}(\varphi^1_{\text{pre}}) \) and \( \text{free}(\varphi^1_{\text{post}}) \) are disjoint from \( \text{mod}(S) \), and that \( (\exists X. \varphi^1) * \varphi^1_{\text{pre}} \equiv \varphi^1_{\text{post}} * \varphi^{i+1} \)

\(^1\)By restricting preconditions to quantifier free formulas we do not sacrifice expressiveness. Indeed, the Hoare triple \( [\exists Y. \psi(V,W,Y)] \ S [\exists X. \vec{\psi}(V,W,X)] \) is valid iff \( [\psi(V,W,Y)] \ S [\exists X. \vec{\psi}(V,W,X)] \) is valid, where \( W, X, Y \) are disjoint sets of auxiliary variables (see defn. 124 in [Cousot 1990]).

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
the indexed auxiliary variables.

\[ \text{pass2}(\psi) \]

\begin{align*}
&\text{match } \psi^i \text{ with } \\
&\quad| \text{emp } \rightarrow \text{true } \land \text{emp} \\
&\quad| \ s_1 \rightarrow e_2 \rightarrow e_1 \neq \text{null } \land \ e_1 \rightarrow e_2 \\
&\quad| \ s_1 \ast s_2 \rightarrow \text{pass2}(s_1) \ast \text{pass2}(s_2) \\
&\quad| \ \text{RS}(s, l, u) \rightarrow \text{let } \varphi \leftarrow \text{pass2}(s) \text{ in } \\
&\quad\quad\text{RP}(\psi^i, l, u) \land \text{RS}(\varphi^i, l, u)
\end{align*}

for any \( i \). Given these conditions, the \textsc{compose} rule can be iteratively applied to obtain an accelerated summary similar to that in (4.1).

We use \( \alpha, \beta \), and \textit{iter} to express \( \varphi^0, \varphi^k \) and the iterated separating conjunction of accelerated summary (4.1) in \textsc{lisf}. The renaming \( \alpha \) replaces every variable \( x \in W \) in \( \varphi \) by \( x^i \). Similarly \( \beta \) replaces every \( x \in W \) in \( \varphi \) by \( x^i \).

The function \textit{iter} in premise 9 takes an \textsc{lisf} formula \( \varphi \), computes an intermediate formula \( \varphi_{ren} \), and returns \( \text{RP}(\varphi_{ren}) \land \text{RS}(\varphi_{ren}) \) as defined in Figure 10. The formula \( \varphi_{ren} \) is computed by applying a function called \textit{warp} to \( \varphi \). \textit{warp} makes at most two passes over the syntax tree of \( \varphi \) in a bottom-up manner. In the first pass it renames every indexed auxiliary variable \( x_0 \) (resp. \( x_1 \)) by a fresh array with indexed index \( x_i \) (resp. \( x_{i+1} \)). If \( \varphi_{ren}^0 \) and \( \varphi_{ren}^i \) do not have any common array variable, it performs a second pass (formalized in algorithm \textit{pass2}, Figure 10) in which every sub-formula \( e_1 \rightarrow e_2 \) in \( \varphi_{ren}^i \) is replaced by \( e_1 \neq \text{null } \land e_1 \rightarrow e_2 \). All resulting sub-formulas of the form \( \text{RS}(P \land S, l, u) \) are finally replaced by \( \text{RP}(P, l, u) \land \text{RS}(S, l, u) \). This ensures that \( \varphi_{ren} \) always have at least one common array variable, unless \( \varphi^i \) is \text{emp}. The length of these common arrays determines the implicit upper bound in the universal quantifier of \( \text{RP} \) and \( \text{RS} \).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure9.png}
\caption{Inference rule for acceleration \textsc{induct} and \textsc{inductq}}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure10.png}
\caption{Definition of \textit{iter}(\psi)}
\end{figure}
Example 3. Recall Example 2 where the instances of a summary $[v = x \land ϕ^0 \Rightarrow \omega] \land [v = \omega \land x \Rightarrow \omega]$ are composed using $ϕ^0_{post} : (x_1 = y_0 \land x_1 \Rightarrow y_1)$ and $ϕ^0_{pre} : (x_1 = y_0 \land x_0 \Rightarrow y_0)$. For this example, it can be seen that $\hat{ϕ}$ generates the LIFS formula $\Box (ϕ_{pre})$. In this representation, the arrays $x$ and $y$ represent the sequences $v_0, \ldots, v_k$ and $y_0, \ldots, y_k$, respectively. The renamed formulas $ϕ(x)$ and $ϕ(y)$ correspond to the formulas $v = x[0] \land x[0] \Rightarrow y[0]$ and $v = y[0] \land x[0] \Rightarrow y[0]$ respectively. The application of $InductQ$ thus generates the summary: $[v = x[0] \land \Box (ϕ_{pre}) \land RS(x[0]) \Rightarrow RS(y[0])]$. For this example, $ψ_{post}$ generates $ϕ_{post}$. In the conclusion.

5.2 Inference rule $\text{InductQ}$

In general, the strong bi-abduction of $\exists X \cdot \hat{ϕ}^0$ and $ϕ^1$ in premise 6 may require variables to be existentially quantified on the right hand side. The $\text{InductQ}$ rule need to be slightly modified in this case. However, the basic intuition of acceleration remains the same, as illustrated in the Figure 5. The modified rule $\text{InductQ}$ is presented in Figure 9. We use a refined notation in $\text{InductQ}$ where $ϕ^1$ (resp. $\hat{ϕ}^1$) denotes $ϕ$ (resp. $\hat{ϕ}$) with every variable $w \in W$ replaced by an indexed variable $w_i$ and every variable $x \in X$ replaced by $x_i$. Let the strong bi-abduction between $ϕ^0$ and $ϕ^1$ be $(\exists X_1 \cdot \hat{ϕ}^0) \ast ϕ^0_{pre} \Rightarrow \exists Z_1. (ϕ^0_{post} \ast ϕ^1)$, where $Z_1 \subseteq W_1 \cup X_1$ is the set of auxiliary variables. If the additional side-condition $free(ϕ^0_{pre}) \cap Z_0 = \emptyset$ holds, we can infer the accelerated summary in the conclusion of $\text{InductQ}$. Let $Z_i$ be the set of variables $\{z_i^1, \ldots, z_i^k\}$. The values of variables in $Z_0 = \{z_0^1, \ldots, z_0^k\}$ are represented as elements of $r$ arrays $z^i = \{z_0^i, \ldots, z_k^i\}$ in the postcondition of $ϕ^0_{post}$, where $Z_1 \subseteq W_1 \cup X_1$. These two representations are analogous to representing elements of the same matrix row-wise and column-wise. The variables representing the values of variables in $Z_1 \cup \ldots \cup Z_k$ need to be existentially quantified in the postcondition of $ϕ^0_{post}$ because of the existential quantification of $Z_1$ in strong bi-abduction. Hence we existentially quantify the array variables $z^1, \ldots, z^k$ in the conclusion of $\text{InductQ}$.

By existentially quantifying the array variables $z^1, \ldots, z^k$ in the conclusion of $\text{InductQ}$, we also quantify the array indices representing values of the variables in $Z_0$, which need not be quantified. Although this is sound, we lose the correspondence between the $Z_0$ variables in the pre and postcondition of the conclusion. We can establish this correspondence by adding extra equalities $z_0 = z$, for every variable $z_0 \in Z_0$, to $ϕ^0_{post}$ in the conclusion.

Lemma 5.1. Inference rules $\text{Induct}$ and $\text{InductQ}$ are sound

Proof. We use induction on number of compositions to prove $\text{InductQ}$. Compose proves the base case, $[ϕ^0 \ast ϕ^0_{pre}]$: $[∃X_2, Z_1_0. (ϕ^0_{post} \ast ϕ^1)]$. The induction case can be proved as follows:
1. \([\varphi^0] \subseteq \exists X_{k+1}, \varphi^k\]
2. \([\varphi^0 \cup \bigwedge_{i=0}^{k} \varphi^i_{pre} \cup \varphi^k] \subseteq [\exists X_{k+1}, Z_1, \ldots, Z_k, \bigwedge_{i=0}^{k} \varphi^i_{post} \cup \varphi^k]\]
3. \((\exists X_{k+1}, \varphi^k) \cup \varphi^k_{pre} \Rightarrow \exists X_{k+1}, (\varphi^k_{post} \cup \varphi^k)\]
4. \((\exists X_{k+1}, Z_1, \ldots, Z_k, \bigwedge_{i=0}^{k} \varphi^i_{post} \cup \varphi^k) \Rightarrow \varphi^k_{pre}\)

The Hoare triple in 7 above is expressed in the conclusion of INDUCTQ as \([\varphi^{k+1}] \subseteq [\exists X_{k+2}, \varphi^{k+1}]\). The formulas \([\bigwedge_{i=0}^{k} \varphi^i_{pre} \cup \varphi^k]\) and \([\bigwedge_{i=0}^{k} \varphi^i_{post} \cup \varphi^k]\) are expressed in \(\text{LISF}\) as \(\text{Iter}(\varphi^i_{pre})\) and \(\text{Iter}(\varphi^i_{post})\), respectively. The parameter \(k\) in the pre and postcondition of 7 is implicitly hidden in the semantics of RS and RP predicates output by \(\text{Iter}\). Every free array variable in \(\text{Iter}(\varphi^i_{post})\) is guaranteed to be free in \(\text{Iter}(\varphi^i_{pre})\) by the strong bi-abduction in the premise of INDUCTQ. This common array variable ensures the same parameter \(k\) in the pre and postcondition of the resulting Hoare triple. However, it is possible that all the array variables in \(\text{Iter}(\varphi^i_{post})\) are existentially quantified and hence \(\text{Iter}(\varphi^i_{pre})\) and \(\text{Iter}(\varphi^i_{post})\) do not share an array variable. This results in an over-approximate postcondition. We can obtain a stronger postcondition in this case by adding a dummy equality \(e = e\) in the RP predicate output by \(\text{Iter}(\varphi^i_{pre})\), where \(e\) is an expression from \(\text{Iter}(\varphi^i_{pre})\) involving an array variable not present in \(\text{Iter}(\varphi^i_{post})\). 

5.3 Inferenece rule \(\text{INDUCTSYMM}\)

The inference rule \(\text{INDUCTSYMM}\) enables us to compute summaries that capture the effect of executing the statement \(S\) zero or more times. This is in contrast with the summaries inferred by \(\text{INDUCTQ}\) which capture the effect of executing \(S\) one or more times. Additionally, \(\text{INDUCTSYMM}\) also enables us to eliminate some variables from the pre and postcondition of the inferred summary, thus simplifying it. If, in equation (4.1) \(\varphi^i_{pre}\) (resp. \(\varphi^i_{post}\)) is same as \(\varphi^0\) (resp. \(\varphi^k\)) modulo variable renaming, then we can infer the following summary: \([\bigwedge_{i=0}^{k} \varphi^i] S [\bigwedge_{i=0}^{k} \varphi^i]\). Recall the accelerated summary inferred in Example 2, which is depicted in Figure 5-c. In this example the shape of \(\varphi^0\) (resp. \(\varphi^0\)) and \(\varphi^i_{pre}\) (resp. \(\varphi^i_{post}\)) are the same. Hence we can re-write the accelerated summary as follows: \([v = x_{t_i} \land \bigwedge_{i=0}^{k} (x_{t_i} \mapsto y_i \land y_i = x_{t_i+1}) S [v = x_{t_{k+1}+1} \land \bigwedge_{i=0}^{k} (x_{t_i} \mapsto y_i \land y_i = x_{t_{i+1}})]\). This is depicted in Figure 12-a.

The equalities \(x_{t_i+1} = y_i\), for each \(i\), in the pre and postcondition identify the folding points [Guo et al. 2007] of the repeated data-structure in the heap. We can replace \(y_i\) by \(x_{t_i+1}\) from both the pre and postcondition, and thus eliminate all the

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
Given
1. \([\varphi] \not\in \{X. \varnothing \}, |X| = t\)
2. \(\varphi^0 : \varnothing \) with every \(w \in W\) and \(x \in X\) replaced by \(w_0\) and \(x_1\), resp.
3. \(\varphi^i : \varphi \) with every \(w \in W\) replaced by \(w_i\), for \(i \in \{0, 1\}\)
4. \((3X_1, \varphi^0) \ast \varphi^i \) is satisfiable
5. \(\tau_i : (\mod(\emptyset) \cap \text{free}(\exists X. \varnothing) \rightarrow \text{free}(\varphi \setminus \mod(\emptyset))), \) for \(i \in \{0, 1\}\),
s.t. Pure part of \(\varphi\) implies \(\text{Eq } \tau_i\)
6. \(\gamma_0 : (\mod(\emptyset) \cap \text{free}(\varphi) \rightarrow \text{free}(\exists X_1, \varphi^0) \setminus \mod(\emptyset)), \) s.t. Pure part of \(\exists X_1, \varphi^0\) implies \(\text{Eq } \gamma_0\)
7. \(\alpha : (y_0 \rightarrow Y[0]), \) for each \(y \in W\)
8. \(\beta : (y_1 \rightarrow Y[0]), \) for each \(y \in W\)
9. \(\text{Iter same as described in } \text{INDUCT}\)
10. \(\delta^0_k : (W_0 \rightarrow W_1), \) s.t. \(a_0 \rightarrow b_1 \in \delta^0_k \) iff \(\text{(Eq } \gamma_0) \tau_1 \Rightarrow a_0 = b_1, \) and \(a_0 \not\in \text{range}(\tau_0)\)

\[\begin{align*}
\text{InductSymm} \\
\text{Infer} \\
\text{Eq } \tau_0, \text{Eq } \gamma_0 \\
\text{Fig. 11. Variant of } \text{INDUCTQ, INDUCTSYM} \\
\end{align*}\]

Fig. 11. Variant of \text{INDUCTQ, INDUCTSYM}

\[\begin{align*}
\text{Eq } \tau_0 & \text{ and } \text{Eq } \gamma_0 \text{ imply: } [\text{Eq } \gamma_0]_{\alpha} \land \text{Iter}(\varphi^0 \gamma_0 \delta^0_k \land (\text{Eq } \gamma_0) \tau_1 \delta^0_k)] \\
\text{and } \text{Eq } \gamma_0 \text{ implies: } [\exists X^1, \ldots, X^t, (\text{Eq } \tau_1)_{\beta} \land \text{Iter}(\varphi^0 \gamma_0 \delta^0_k \land (\text{Eq } \gamma_0) \tau_1 \delta^0_k)] \\
\end{align*}\]

Fig. 12. (a) Alternate representation of summary in Figure 5-c, and (b) Summary resulting from application of \text{INDUCTSYM}. Each box represents a heap cell, its contents represents the value of \text{next} field. A circled variable above a box denotes the name of the cell.

\[\begin{align*}
\text{Eq } \gamma_i \text{'s. We obtain the following simplified summary from this renaming (depicted in Figure 12-b). } [v = x_0 \land \bigotimes_{t=0}^k \tau_i \Rightarrow \varphi_{i+1}] & \text{ sk}^{k+1} [v = x_{k+1} \land \bigotimes_{t=0}^k \tau_i \Rightarrow \varphi_{i+1}]. \\
\text{The corresponding summary in } \text{LISF} \text{ is } [v = x[0] \land \text{RS}(x[1] \rightarrow x[1] + 1)] & \text{ s}^* [v = x[0] \land \text{RS}(x[1] \rightarrow x[1] + 1)]. \text{ In this specification, if the length of } x \text{ is } \lambda + 1 \text{ (where } \lambda \geq 0, \text{ then it summarizes } \lambda \text{ iterations of } S. \text{ Hence it is a summary for zero or more iterations of } S, \text{ denoted as } [\varphi] \text{ s}^* [\varnothing]. \text{ The notation } [\varphi] \text{ s}^* [\varnothing] \text{ means that for every initial state satisfying } \varnothing, \text{ there exists a } k \geq 0 \text{ such that the state resulting after } k \text{ executions of } S \text{ satisfies } \varnothing. \text{ The above ideas are captured formally by the rule } \text{INDUCTSYM} \text{ in Figure 11.} \\
\text{For a renaming } \gamma, \text{ let } (\text{Eq } \gamma) \text{ denote the conjunction of all the equalities } a = b \text{ such that } \gamma \text{ renames } a \text{ to } b. \text{ The premises 5 and 6 of } \text{INDUCTSYM} \text{ in Figure 11 imply } \varphi^0 = \text{Eq } \tau_0 \land \varphi^0 \text{ and } \exists X_1, \varnothing \text{ and } \text{Eq } \gamma_0, \text{ respectively. These premises also imply that } \gamma_0 	ext{ and } \tau_1 \text{ have same domains and their ranges are independent of } \end{align*}\]
mod(S) variables, hence \((\text{Eq } \gamma_0)_{\tau_1}\) is independent of \(\text{mod}(S)\). This fact implies that 
\((\text{Eq } \gamma_0) \land (\text{Eq } \gamma_0)_{\tau_1} \Rightarrow (\text{Eq } \gamma_1) \land (\text{Eq } \gamma_0)_{\tau_1}\). Hence, \([\psi^0] S \exists X_1. \tilde{\psi}^0\) is a valid Hoare triple, where \(\tilde{\psi}^0 \equiv \text{Eq } \gamma_0 \land \psi^0 \land (\text{Eq } \gamma_0)_{\tau_1}\) and \(\tilde{\psi}^0 \equiv \text{Eq } \gamma_1 \land \psi^0 \land (\text{Eq } \gamma_0)_{\tau_1}\). 
Let \(\psi^i\) (resp. \(\psi^1\)) be same as \(\psi^0\) (resp. \(\tilde{\psi}^0\)) except that the variable indices 0 and 1 are replaced by indices \(i\) and \(i + 1\), respectively. By the law of auxiliary variable renaming, it follows that for any \(i\), \([\psi^i] S \exists X_{i+1}. \tilde{\psi}^0\) is a valid Hoare triple. Let us compose the Hoare triples \([\psi^0] S \exists X_1. \tilde{\psi}^0\) and \([\psi^1] S \exists X_2. \tilde{\psi}^1\). From the definitions of \(\tilde{\psi}^0\) and \(\psi^1\), we can infer the following strong bi-abduction between \(\exists X_1. \tilde{\psi}^0\) and \(\psi^1\).

\[
(\exists X_1. \tilde{\psi}^0) * \varphi^1_{\text{pre}} \land (\text{Eq } \gamma_1)_{\tau_2} \Rightarrow \exists X_1. (\varphi^0_{\text{pre}} \land (\text{Eq } \gamma_0)_{\tau_1}) * \psi^1_{\text{post}}) \tag{5.2}
\]

An interesting feature of this strong bi-abduction is that \(\varphi^0_{\text{pre}} \land (\text{Eq } \gamma_1)\) (resp. \(\varphi^1_{\text{post}} \land (\text{Eq } \gamma_1)\)) is same as \(\psi^1\) (resp. \(\tilde{\psi}^0\)). Thus the shape of \(\varphi^0_{\text{pre}}\) (resp. \(\varphi^1_{\text{post}}\)) is same as that of \(\psi^0\) (resp. \(\tilde{\psi}^0\)). Thus from the premises 1-9, by inductively applying COMPOSE to the sequence of Hoare triples, \([\psi^0] S \exists X_1. \tilde{\psi}^0\), \([\psi^1] S \exists X_2. \tilde{\psi}^1\), ..., \([\tilde{\psi}^0] S \exists X_{k+1}. \tilde{\psi}^k\), we obtain the following accelerated summary.

\[
([\text{Eq } \gamma_0] \land \bigcirc_{i=0}^k \varphi^i_{\text{pre}} \land (\text{Eq } \gamma_i)_{\tau_{i+1}}] S^+ [(\text{Eq } \tau_{k+1}] \land \bigcirc_{i=0}^k \exists X_{i+1}. \tilde{\psi}^i_{\text{post}} \land (\text{Eq } \gamma_i)_{\tau_{i+1}}]) \tag{5.3}
\]

\text{INDUCTSYM} uses the premise 10 to existentially quantify some auxiliary variables from the summary \([\psi^1] S \exists X_{i+1}. \tilde{\psi}^0\) and thus simplify the final accelerated summary computed above. For this purpose we define a renaming \(\delta^0_{\text{delta}}\) from variables in \(W_0\) to variables in \(W_1\). It is computed from the equalities in \((\text{Eq } \gamma_0)_{\tau_1}\).

Using the rule for existentially quantifying auxiliary variables, it follows that each of \([\psi^0] S \exists X_1. \tilde{\psi}^0\), \([\psi^1] S \exists X_2. \tilde{\psi}^1\), ..., \([\tilde{\psi}^0] S \exists X_{k+1}. \tilde{\psi}^k\) is a valid Hoare triple. If \(a_0 \rightarrow b_1 \in \delta^0_{\text{delta}}\) then we can eliminate all occurrences of \(a_0\)'s by applying the renaming \(\delta^0_{\text{delta}}\) and \(\delta^1_{\text{delta}}\) to both sides of the the strong bi-abduction in (5.2). The renaming \(\delta^0_{\text{delta}}\) has a property that if \(b_1 \in \text{range}(\delta^0_{\text{delta}})\) then \(b_1 \in \text{range}(\tau_1)\) which in turn implies \(b_1 \notin \text{dom}(\delta^1_{\text{delta}})\). This ensures that (a) \((\text{Eq } \gamma_1)_{\tau_1} \delta^0_{\text{delta}} \equiv (\text{Eq } \gamma_1)\), (b) \((\tilde{\psi}^0_{\text{pre}}) \delta^0_{\text{delta}} \equiv (\tilde{\psi}^0_{\text{pre}})\), and (c) \((\text{Eq } \gamma_0)_{\tau_1} \delta^0_{\text{delta}} \equiv (\text{Eq } \gamma_0)\). Hence \(\tilde{\psi}^0_{\text{pre}} \delta^1_{\text{delta}} \equiv \tilde{\psi}^0_{\text{pre}}\).

Using the renamings \(\delta^0_{\text{delta}}\) and \(\delta^1_{\text{delta}}\) we can therefore infer the following strong bi-abduction between \(\exists X_1. \psi^0_{\text{pre}} \delta^0_{\text{delta}}\) and \(\psi^1_{\text{post}}\).

\[
(\exists X_1. \tilde{\psi}^0_{\text{pre}}) * \varphi^1_{\text{pre}} \tau_1 \delta^2_{\text{delta}} \land (\text{Eq } \gamma_1)_{\tau_2} \delta^2_{\text{delta}} \Rightarrow \exists X_1. (\varphi^0_{\text{pre}} \tau_0 \delta^0_{\text{delta}} \land (\text{Eq } \gamma_0)_{\tau_1} \delta^1_{\text{delta}}) * \psi^1_{\text{post}}) \tag{5.4}
\]

We require \(\delta^0_{\text{delta}}\) to satisfy the constraint \(a_0 \in \text{dom}(\delta^0_{\text{delta}}) \Rightarrow a_0 \notin \text{range}(\gamma_0)\) so that \(\tilde{\psi}^0_{\text{pre}} \delta^1_{\text{delta}}\) is equivalent to \(\tilde{\psi}^0_{\text{pre}}\) and it does not have variables with all indices 0, 1 and 2, otherwise its repetition cannot be expressed by \(\text{LIFS}\) predicates RS and RP.

By inductive application of the compose rule to the sequence of Hoare triples, 
\([\psi^0] S \exists X_1. \tilde{\psi}^0_{\text{pre}}\), \([\psi^1] S \exists X_2. \psi^1_{\text{post}}\), ..., \([\tilde{\psi}^0] S \exists X_{k+1}. \psi^k_{\text{post}}\), we

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
get the following accelerated Hoare triple.

\[
[(Eq \tau_0) \land \circ^i_{k=0} \varphi^i \tau_i \delta^i_{1+1} \land (Eq \gamma_i) \tau_{i+1} \delta^i_{1+1}]^* \tag{5.5}
\]

The conclusion of INDUCTSYM uses the renaming \( \alpha, \beta \) and the function \( \text{iter} \) (which are same as those defined in INDUCT) to represent the above Hoare triple in \( LIFS \).

Example 3 uses the inference rule INDUCT to accelerate the summary \( [v = x \land \mathbf{w} \rightarrow \mathbf{w}^\prime] \ S [v = y \land x \rightarrow y] \). In the following example we apply the inference rule INDUCTSYM to accelerate the same summary.

**Example 4.** Recall the acceleration of summary \( [v = x_0 \land x_0 \rightarrow x_0] \ S [v = y_0 \land x_0 \rightarrow y_0] \) in Example 3. For this example we can obtain \( \tau_i \) and \( \gamma_0 \) as \( \langle v \rightarrow x_i \rangle \) and \( \langle v \rightarrow y_0 \rangle \), respectively. These renamings satisfy the premises 5 and 6 of INDUCTSYM. With these renamings we find that \( (Eq \gamma_0) \tau_i \) is equivalent to \( \varphi^0 \tau_0 \). The expressions \( \varphi^0 \tau_0 \) and \( \varphi^0 \gamma_0 \) are both equivalent to \( x_0 \rightarrow y_0 \). Hence we can infer the valid Hoare triple \( \delta^0 \) \( S \ [3X_1, \varphi^0] \), where \( \psi^0 \) and \( \exists X_1, \varphi^0 \) are \( v = x_0 \land x_0 \rightarrow y_0 \land y_0 = x_1 \), and \( v = x_1 \land x_0 \rightarrow y_0 \land y_0 = x_1 \), respectively.

The renaming \( \langle y_0 \rightarrow x_1 \rangle \) satisfies the requirements of \( \delta^0 \) in the premise 10. Hence we find that both \( \langle \varphi^0 \tau_0 \delta^0 \rangle \) and \( \langle \varphi^0 \gamma_0 \delta^0 \rangle \) are equivalent to \( x_0 \rightarrow x_1 \).

For composing the two triples \( \delta^0 \) \( S \ [3X_1, \varphi^0 \delta^0] \) and \( \delta^1 \delta^1 \) \( S \ [3X_2, \varphi^1 \delta^1] \), the following is a valid strong bi-abduction.

\[
(v = x_1 \land x_0 \rightarrow x_1) \cdot (x_1 \rightarrow x_2) \leftrightarrow (v = x_1 \land x_0 \rightarrow x_1) \cdot (x_1 \rightarrow x_2)
\]

Thus the premises of INDUCTSYM guarantee the validity of the following accelerated summary \( [v = x_0 \land \circ^i_{k=0} x_i \rightarrow x_{i+1}] \ S^* \ [v = x_{k+1} \land \circ^i_{k=0} x_i \rightarrow x_{i+1}] \). Hence by application of INDUCTSYM we obtain the following \( LIFS \) summary \( [v = x[0] \land \mathbf{RS}(x[i] \rightarrow x[i+1])] \ S^* \ [v = x[0] \land \mathbf{RS}(x[i] \rightarrow x[i+1])] \).

### 5.4 Discussion.

The summary inferred by INDUCTSYM captures the effect of executing the statement \( S \) zero or more times. This is in contrast with the summaries inferred by INDUCTQ which capture the effect of executing \( S \) one or more times. Summaries that capture the effect of executing \( S \) zero or more times enable us to compute succinct specifications, and in some cases complete specifications which could not have been possible otherwise.

As an illustration, consider a program with a while loop nested within an outer while loop. The outer while loop iterates over a single linked list pointed to by \( h \), whereas the inner while loop deletes the linked list pointed to by the data field of each element of the outer linked list. Using the rule INDUCTQ, the inner while loop is summarized by two Hoare triples one summarizing zero iterations of the loop body (corresponding to zero length inner linked list), and the other summarizing one or more iterations of the loop body (corresponding to non-zero length inner linked list). By one more application of INDUCTQ we can obtain a summary for the outer while loop whose precondition either expresses the fact that all outer linked list elements point to zero length inner linked lists or the fact that all outer linked
list elements point to non-zero length inner linked lists. However, the resulting summary after two applications of \textsc{Inductq} is not a complete specification for the program.

In contrast, \textsc{InductSymm} enables us to compute a single summary for the inner while loop. It captures the deletions of inner linked lists of any length (zero or more). By one more application of \textsc{InductSymm} we can obtain a summary for the outer while loop whose precondition expresses the fact that \texttt{data} field of each outer linked list element points to a linked list of length zero or more. Notice that this is a complete specification for the program.

Note that if any Hoare triple in the premise of inference rules in Figure 4, 9, and 11 is partial (i.e., termination is not guaranteed starting from a state satisfying precondition), then the Hoare triple in the conclusion will also be partial.

\textbf{Lemma 5.2.} The rule \textsc{InductSymm} is sound.

### 5.5 Generating summaries using combination of rules

The \textsc{Compose} and \textsc{Exit} rules can be used to obtain summaries of loop free code fragments and trivial summaries of loops, respectively. Given a loop body summary, the \textsc{Induct}, \textsc{Inductq} and \textsc{InductSymm} rules generate an accelerated summary for use in the \textsc{While} rule. Any pair of accelerated summaries can also be composed to obtain new accelerated summaries.

We now present a procedure to enumerate all possible accelerated summaries for the while loop \texttt{while (B) S}. This enumeration process may not terminate in general. However, when it does terminate, it generates a complete specification for the while loop. Let $\hat{S}$ be the set of summaries for the loop body \texttt{assert(B);S}. For the summaries $s_1$ and $s_2$, let $s_1^+$ denote the accelerated summary obtained by applying one of the \textsc{Induct}, \textsc{Inductq}, or \textsc{InductSymm} rules to $s_1$, and let $s_1 \circ s_2$ denote the summary obtained by applying the \textsc{Compose} rule to $s_1$ and $s_2$. Let $\hat{S}$ be the set of summaries defined as the least fix-point of the following set transformer: $F(S) = \{s^+ | s \in S\} \cup \{s_1 \circ s_2 | s_1, s_2 \in S\} \cup \hat{S}$. The set $\hat{S}$ contains all the accelerated summaries – a complete functional specification for the loop \texttt{while (B) S} (assuming $\hat{S}$ is a complete set of summaries for the loop body \texttt{assert(B);S}). This set can be computed in an iterative fashion, by repeated application of $F$ to the emptyset. However, this iterative fix-point computation may not terminate. Hence, in practice we use heuristics to guide the iterative fix-point computation in order to generate useful summaries. For instance, in practice we could limit the number of applications of $F$ to a small fixed constant to quickly generate a useful set of summaries. As another alternative, heuristics used for acceleration in [Bardin et al. 2005] can be adapted to guide the application of acceleration and composition rules for synthesizing useful summaries.

Given procedure summaries, non-recursive procedure calls can be analyzed by the \textsc{Compose} rule, as in [Calcagno et al. 2009]. The \textsc{Inductq} rule can be used to compute accelerated summaries of tail recursive procedures having at most one self-recursive call.
\[ x = A[0] \land A[8] = \text{null} \land \text{RP}(A[+ 1] = \text{null}) \land \text{RS}(A[i] = (f : y)) \land \varphi_2 \] if\( (e, S_1, S_2) [\varphi_2] \)

Fig. 13. The rule Join.

5.6 Generating concise summaries using the JOIN rule

In order to avoid explosion of summaries for programs with many branching statements, we present the rule JOIN. It facilitates merging the summaries for two branches of if-then-else statement into a single summary. The JOIN rule is presented in Figure 13. Consider two summaries \([x = 0 \land \varphi_1] \text{assert}(e); S_1 [\varphi_1],\) and \([x \mapsto (f : y) \ast \varphi_2] \text{assert}([e); S_2 [\varphi_2],\) of two branches of the statement if \( (e, S_1, S_2) \) (first two premises of JOIN). If \( \varphi_1 \mu \Rightarrow \bar{\varphi}_2 \) and \( \varphi_1 \mu \Leftarrow \varphi_2,\) where \( \mu \) Renames auxiliary variables, are valid then we can infer the concise summary \([x = 0 \lor x \mapsto (f : y)] \ast \varphi_2 \) if\( (e, S_1, S_2) [\varphi_2] \). Since LISF does not permit disjunctions, the precondition cannot be directly expressed in LISF. However, we can encode the disjunction \((x = 0 \lor x \mapsto (f : y))\) using a fresh auxiliary variable \( A \) as: \( \psi \equiv \exists x = A[0] \land A[8] = \text{null} \land \text{RP}(A[+ 1] = \text{null}) \land \text{RS}(A[i] = (f : y)). \) The formula \( \exists \psi \) is equivalent to \( x = \text{null} \) (resp. \( x \mapsto (f : y)\)) when the length of \( A \) is 1 (resp. 2). It is inconsistent when the length of \( A \) is greater than 2. Hence it is equivalent to \( x = 0 \lor x \mapsto (f : y)\). In the section 6 on strong bi-abduction we show how to implement the checks \( \varphi_1 \mu \Rightarrow \varphi_2 \) and \( \varphi_1 \mu \Rightarrow \bar{\varphi}_2 \) for quantifier free LISF formulas, as required by the JOIN rule. Although the JOIN rule is valid even if the postconditions of the two summaries in the premise have existentially quantified variables, in order to implement the checks in the premise using the algorithm that we will present in section 6, we require them to be quantifier free formulas. Hence we assume that \( \varphi_1, \bar{\varphi}_1 \) are quantifier free formulas over free variable \( V,W \) and \( \varphi_2, \bar{\varphi}_2 \) are quantifier free formulas over free variable \( V,Y \).

5.7 Generating summaries with recursive predicates

Instead of translating a recurrence into a LISF formula, we could as well translate it into a recursive predicate in the conclusion of \textsc{Induct}, \textsc{Inductq} or \textsc{InductSymm}. As an illustration, recall the summary \([v = \omega_0 \land \bigotimes_{i=0}^{k} \omega_i \mapsto \omega_{i+1}] S^* [v = \omega_{k+1} \land \bigotimes_{i=0}^{k} \omega_i \mapsto \omega_{i+1}] \) generated by the \textsc{InductSymm} rule in Example 4. The recurrence \( \bigotimes_{i=0}^{k} \omega_i \mapsto \omega_{i+1} \) obtained above can be translated into a recursive predicate \( \text{list}(\omega_0, \omega_{k+1}) \), where \( \text{list}(\omega_0, \omega_{k+1}) \) is the standard recursive predicate that characterizes a linked-list segment [Distefano et al. 2006; Calcagno et al. 2007; 2009]. It is defined recursively as follows, \( \text{list}(\omega_0, \omega_{k+1}) \ifdef \omega_0 \mapsto \omega_{k+1} \lor \exists x_1, \omega_0 \mapsto x_1 \ast \text{list}(x_1, \omega_{k+1}) \). Hence we can generate the summary \([v = \omega_0 \land \text{list}(\omega_0, \omega_{k+1})] S^* [v = \omega_{k+1} \land \text{list}(\omega_0, \omega_{k+1})] \), using recursive predicates as a conclusion of \textsc{InductSymm}. In general, we could either use the acceleration inference rules to generate new recursive predicates, or pick a recursive predicate from the set of predefined predicates to generate the accelerated summary. But summaries with recursive predicates do not relate the input and output data-structures of a procedure and hence are non-functional.
6. A STRONG BI-ABDUCTION ALGORITHM FOR LISF

In this section we present a procedure to compute strong bi-abduction. We first present a solution to a sub-problem of computing LISF formulas $\delta_1$ and $\delta_2$, given two quantifier free LISF formulas $\varphi$ and $\psi$, such that $\varphi \ast \delta_1 \Rightarrow \delta_2 \ast \psi$. The algorithm Decompose given in Figure 14 computes such $\delta_1$ and $\delta_2$ given $\varphi$ and $\psi$ as input.

The key step in Decompose is the Match procedure used in line 2. Match takes two spatial formulas $\varphi^s$ and $\psi^s$ and an integer constant (that corresponds to nesting depth of $\varphi^s$ and $\psi^s$ within $\mathcal{RS}$ predicate) as inputs and returns a set of four-tuples $(M, C, L_1, L_2)$ where $M$ is a pure formula and $C, L_1, L_2$ are spatial formulas. For each such tuple, $M$ describes a constraint under which the heaps defined by $\varphi^s$ and $\psi^s$ can be decomposed into an overlapping part defined by $C$ and non-overlapping parts defined by $L_1$ and $L_2$ respectively.

We present procedure Match as a set of inference rules in Figure 15. The rule No-Match does not find any overlap between $S_1$ and $S_2$, whereas Cell-Match matches the two input mapsto predicates. The rule Recursion recursively finds all possible overlaps between $S_1$ and $S_2$.

The utility of the integer parameter $d$ of Match is in unrolling the $\mathcal{RS}$ predicate

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
in \textsc{UnrollFront} and \textsc{UnrollBack}. The function \texttt{unroll}(\textsc{RS}(S,l,u), d) required by \textsc{UnrollFront} unrolls \textsc{RS} once from the beginning. It returns the formula obtained by replacing every \((d+1)^{th}\) iterated index \([i]\) (resp. \([l+1]\)) in \texttt{S} by the fixed index \([i]\) (resp. \([l+1]\)). Similarly \texttt{unroll}(\textsc{RS}(S,l,u), d), required by \textsc{UnrollBack} unrolls \textsc{RS} once from the end. It returns the formula obtained by replacing every \((d+1)^{th}\) iterated index \([i]\) (resp. \([l+1]\)) in \texttt{S} by the fixed index \([{}$u$+1]\) (resp. \([{}$u$]\)). The rule \textsc{MatchRs} finds an overlapping part of the two \textsc{RS} predicates. This is the only rule that increments \(d\). The function \texttt{separate\_zero\_depth(M)} used in the premise of \textsc{MatchRs} returns a pair of predicates \(M_0\) and \(M_1\). \(M_0\) is the conjunction of predicates in \texttt{M} with depth zero (i.e., those predicates for which \texttt{dim} evaluates to 0, refer definition of the function \texttt{dim} in Section 4.2) and \(M_1\) is the conjunction of remaining predicates in \texttt{M}. For example, \texttt{separate\_zero\_depth(x[i]} = \(h \land \texttt{RP(A[x]} = \texttt{d[i]} \land x = y)\) would return \((\texttt{RP(A[x]} = \texttt{d[i]} \land x = y, x[i]} = h)\). The predicates in \(M_1\) are embedded in an \textsc{RP} predicate in the conclusion of \textsc{MatchRs}, whereas the predicates in \(M_0\) are not embedded in an \textsc{RP} predicate since it would result in a non well-formed formula. This is the main purpose of separating \(M_0\) from \(M_1\).

These inference rules can be easily implemented as a recursive algorithm. Note that in rules \textsc{UnrollFront} and \textsc{UnrollBack}, the size of the formula \(L_1 * \textsc{RS} (...)\) in the conclusion may be larger than the size of formula \(k_1\) in the premise. This may lead to non-termination of the recursion. In practice we circumvent this problem by limiting the number of applications of these rules.

**Lemma 6.1.** Every \((M,C,L_1,L_2)\) computed in line 2 of \texttt{Decompose} satisfies (i) \(M \land \varphi^* \iff (M \land C) \land L_1\), and (ii) \(M \land \psi^* \iff (M \land C) \land L_2\).

**Proof.** We prove the lemma by induction on the depth of the recursion tree of \texttt{Match}. Base case. Single recursive call. Rules \texttt{No-Match} and \texttt{Cell-Match} trivially satisfy the property. Induction step. Assuming that the call to \texttt{Match} in the premise of rules \texttt{Recursion}, \texttt{UnrollFront}, \texttt{UnrollBack} and \textsc{MatchRs} satisfies properties (i) and (ii), we prove that the conclusion of these rules also satisfies properties (i) and (ii). In the following we prove only property (i), property (ii) can be proved symmetrically.

(1) **Recursion**

1. \(M \land k_1 \iff M \land C \land L_1\) assumption
2. \(N \land S'_1 \land L_1 \iff N \land C' \land L'_1\) assumption
3. \(M \land N \land S'_1 \land L_1 \iff M \land N \land C' \land L'_1\) from 1
4. \(M \land N \land S'_1 \land k_1 \iff M \land N \land C' \land L'_1 \land C\) from 1
5. \(M \land N \land S'_1 \land k_1 \iff M \land N \land C' \land L'_1 \land C\) from 1
6. \(M \land N \land S_1 \iff M \land N \land C \land C' \land L'_1\) premise

(2) **UnrollFront**

1. \(M \land k' \iff M \land C \land L_1\) assumption
2. \(M \land \textsc{RS}(S,l,u) \iff M \land k' \land \textsc{RS}(S,l+1,u)\) \texttt{Defn. of unroll}
3. \(M \land \textsc{RS}(S,l,u) \iff M \land C \land L_1 \land \textsc{RS}(S,l+1,u)\) from 1

(3) **MatchRs**

1. \(M \land S_1 \iff M \land C\) assumption
2. \(M \land S_2 \iff M \land C\) assumption
3. \(M_0 \land \texttt{RP}(M_1,l,u) \land \textsc{RS}(S_1,l,u) \iff M_0 \land \texttt{RP}(M_1,l,u) \land \textsc{RS}(C,l,u)\) from 1 and definition of \texttt{separate\_zero\_depth}
4. \(M_0 \land \texttt{RP}(M_1,l,u) \land \textsc{RS}(S_2,l,u) \iff M_0 \land \texttt{RP}(M_1,l,u) \land \textsc{RS}(C,l,u)\) from 2 and definition of \texttt{separate\_zero\_depth}

Note that proof of \textsc{UnrollBack} is similar to that of \textsc{UnrollFront}. □
Given a possible decomposition \((M, C, L_1, L_2)\) of \(\varphi^s\) and \(\psi^s\) as computed by \textbf{Match}(\(\varphi^s, \psi^s, 0\), line 4 of \textbf{Decompose} checks whether this decomposition is consistent with \(\varphi^p\) and \(\psi^p\). This is done by checking the satisfiability of \((\varphi^p \land L_1) \land (\psi^p \land L_2)\). If this formula is found to be satisfiable, \(\delta_1\) and \(\delta_2\) are computed as \(M \land \psi^p \land L_2\) and \(M \land \varphi^p \land L_1\), respectively.

**Lemma 6.2.** Every \((\delta_1, \delta_2)\) pair computed in lines 5 and 6 of \textbf{Decompose} satisfies \(\varphi \land \delta_1 \Rightarrow \delta_2 \land \psi\).

**Proof.** Follows from the following equivalences:

\[
\begin{align*}
A. \quad & \varphi^p \land \varphi^s \land M \Leftrightarrow (M \land C) \land (\varphi^p \land L_1) \quad \text{from Lemma 6.1} \\
B. \quad & \psi^p \land \varphi^s \land M \Leftrightarrow (M \land C) \land (\psi^p \land L_2) \quad \text{from Lemma 6.1} \\
C. \quad & \Delta \Leftrightarrow \Delta \land (M \land \varphi^p \land L_1) \quad \text{defn of} \Delta \text{ and A} \\
D. \quad & \Delta \Leftrightarrow \Delta \land (M \land \varphi^p \land L_2) \quad \text{defn of} \Delta \text{ and B} \\
E. \quad & \Delta \land (M \land \varphi^p \land L_1) \Leftrightarrow \Delta \land (M \land \varphi^p \land L_2) \quad \text{defn of} \Delta \text{ and \textbf{Decompose}} \\
F. \quad & \Delta \land (M \land \varphi^p \land L_1) \Leftrightarrow \Delta \land (M \land \varphi^p \land L_2) \quad \text{defn of} \delta_2 \text{, line 6 of \textbf{Decompose}}
\end{align*}
\]

Note that the \textbf{Match} procedure results in a possibly exponential number of decompositions, many of which could be discarded by the check on line 4 of \textbf{Decompose}.

One of the reasons for this exponential blow-up is the application of \textbf{Recursion} rule which explores all possible overlaps between \(\varphi^s\) and \(\psi^s\). The exponential blow-up can be mitigated by early identification of inconsistent decompositions during the application of the \textbf{Recursion} rule. This can be done by pruning the application of \textbf{Recursion} rule if the partial decomposition indicated in its second premise, \((M, C, L_1, L_2) \in \text{Match}(k_1, k_2, 0)\), is inconsistent with \(\varphi^p \land \psi^p\), i.e., when \(M \land \varphi^p \land \psi^p\) is unsatisfiable.

For a model \((s, h, V)\) of \(\varphi \land \delta_1\) (and also of \(\delta_2 \land \psi\)), let \(h_{\varphi}\) and \(h_{\delta_1}\) be disjoint sub-heaps that partition \(h\), i.e., \(h = h_{\varphi} \cup h_{\delta_1}\), such that \((s, h_{\varphi}, V) \models \varphi\) and \((s, h_{\delta_1}, V) \models \delta_1\). Similarly, let \(h_{\psi}\) and \(h_{\delta_2}\) be disjoint sub-heaps that partition \(h\), i.e., \(h = h_{\psi} \cup h_{\delta_2}\), such that \((s, h_{\psi}, V) \models \psi\) and \((s, h_{\delta_2}, V) \models \delta_2\). It follows from Lemma 6.1 that every pair \((\delta_1, \delta_2)\) computed by \textbf{Decompose} satisfies the following minimality property.

**Definition 6.1.** (Minimality Property) If \(\varphi \land \delta_1 \Rightarrow \delta_2 \land \psi\) then \(\delta_1\) and \(\delta_2\) are said to be minimal if for every model \((s, h, V)\) of \(\varphi \land \delta_1\) (and also of \(\delta_2 \land \psi\), for every \(h_{\varphi}\), \(h_{\psi}\) and every \(h_{\delta_1}, h_{\delta_2}\), we have \(h_{\delta_1} \subseteq h_{\psi}\) and \(h_{\delta_2} \subseteq h_{\varphi}\).

The minimality property ensures that strong bi-abduction does not include any more heap cells in \(\delta_1\) and \(\delta_2\) than those already present in \(\varphi\) and \(\psi\), respectively.

As an example, suppose we wish to compose the two summaries \([v = \mathbf{w}] v := \mathbf{new} \exists \mathfrak{B} v \mapsto j\) and \([v = \mathfrak{C} \land \mathfrak{C} \mapsto \mathfrak{H}]) v := v.\mathbf{next} [v = \mathfrak{J} \land \mathfrak{C} \mapsto \mathfrak{H}]\) used for illustrations in Example 1. In order to compose these summaries we need to compute a strong bi-abduction between \(\exists \mathfrak{B} v \mapsto j\) and \(v = \mathfrak{C} \land \mathfrak{C} \mapsto \mathfrak{H}\). We use this as a running example to demonstrate our implementation of strong bi-abduction. Let \(\varphi \equiv v \mapsto j, \psi \equiv v = \mathfrak{C} \land \mathfrak{C} \mapsto \mathfrak{H}\). One of the two decompositions returned by \textbf{Match}(\(\varphi^s, \psi^s\)) is \((\mathbf{true}, \mathbf{emp}, v \mapsto \mathfrak{H}, \mathfrak{C} \mapsto \mathfrak{H})\). This decomposition indicates that \(v \mapsto j\) and \(\mathfrak{C} \mapsto \mathfrak{H}\) belong to disjoint portions of the heap, thus implying \(v \neq \mathfrak{C}\). However, since \(\psi^p\) asserts that \(v = \mathfrak{C}\), this decomposition is inconsistent with \(\varphi^p \land \psi^p\). Hence it is discarded. The other decomposition is \((v = \mathfrak{C} \land j = \mathfrak{J}, v \mapsto j, \mathbf{emp})\). This decomposition is consistent with \(\varphi^p \land \psi^p\), and hence \((v = \mathfrak{C} \land j = \mathfrak{J} \land emp, v = \mathfrak{C} \land j = \mathfrak{J} \land emp)\) is returned as a solution of \textbf{Decompose}(\(\varphi, \psi\)).

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
6.1 Algorithm BiAbduct

We now present a sound algorithm for computing $\varphi_{\text{pre}}$, $\varphi_{\text{post}}$ and $Z$ in the equivalence $(\exists X \varphi) \ast \varphi_{\text{pre}} \Leftrightarrow \exists Z (\varphi_{\text{post}} \ast \varphi)$ in the premise of the COMPOSE and INDUCTQ rules. Simplifying notation, the problem can be stated as follows: given variable sets $\text{mod}_1$ and $\text{mod}_2$, and two $\text{LISF}$ formulas $\exists X \varphi(V, W, X)$ and $\psi(V, Y)$ where $V, W, X, Y$ are disjoint sets of variables, we wish to compute $\varphi_{\text{pre}}$, $\varphi_{\text{post}}$, and a set $Z \subseteq X \cup Y$ such that (i) $(\exists X \varphi) \ast \varphi_{\text{pre}} \Leftrightarrow \exists Z (\varphi_{\text{post}} \ast \psi)$, (ii) $\text{free}(\varphi_{\text{pre}}) \cap \text{mod}_1 = \emptyset$, and (iii) $\text{free}(\varphi_{\text{post}}) \cap \text{mod}_2 = \emptyset$.

Our strong bi-abduction algorithm, BiAbduct, is presented in Figure 14. We first illustrate the intuition of BiAbduct using our running example: $\varphi \equiv v \mapsto j$, $\psi \equiv v = \emptyset \land \varphi \mapsto \emptyset, V = \{v\}, W = \{\}, X = \{j\}, Y = \{\emptyset, d\}$ and $\text{mod}_1 = \text{mod}_2 = \{v\}$. As explained before, Decompose$(\varphi, \psi)$ returns the decomposition $(v = \emptyset \land \varphi = d \land \text{emp}, v = \emptyset \land \varphi = d \land \text{emp})$. Thus we have $\varphi \ast (v = \emptyset \land \varphi = d \land \text{emp}) \Leftrightarrow (v = \emptyset \land \varphi = d \land \text{emp}) \ast \psi$. We explain the intuition of our strong bi-abduction algorithm in the following three steps.

---

We want $\varphi_{\text{pre}}$ and $\varphi_{\text{post}}$ to be independent $\text{mod}_1$ and $\text{mod}_2$, respectively. To do this we use the equalities involving $\text{mod}_1$ variables in $\varphi$ (respectively, $\text{mod}_2$ variables in $\psi$) to eliminate $\text{mod}_1$ (respectively, $\text{mod}_2$) variables from $\varphi_{\text{pre}}$ (respectively, $\varphi_{\text{post}}$). In our current example, we replace $v \in \text{mod}_2$ by $\emptyset$ in $\varphi_{\text{post}}$ since $\psi$ contains the equality $v = \emptyset$. Hence we obtain $\varphi \ast (v = \emptyset \land \varphi = d \land \text{emp}) \Leftrightarrow (v = \emptyset \land \varphi = d \land \text{emp}) \ast \psi$. However, using this transformation we cannot make $\varphi_{\text{pre}}$ independent of $v$, since $\varphi$ does not have any equalities involving $v$.

---

In order to make $\varphi_{\text{pre}}$ independent of $\text{mod}_1$ variables we existentially quantify the auxiliary variables that are equated to $\text{mod}_1$ variables in $\varphi_{\text{pre}}$ from both sides of the equivalence. In our current example, we existentially quantify $\emptyset$ from both sides of the equivalence. As a consequence we can drop the equality $v = \emptyset$ involving the auxiliary variable $\emptyset$ from $\varphi_{\text{pre}}$, thus making $\varphi_{\text{pre}}$ independent of $v$.

We now obtain the equivalence $\varphi \ast (\emptyset = d \land \text{emp}) \Leftrightarrow \emptyset = d \land \text{emp} \ast \psi$.

---

Our goal is to compute a strong bi-abduction between $\exists \emptyset \varphi$ and $\psi$. Since the current $\varphi_{\text{pre}}$ has free $\emptyset$, $\exists \emptyset (\varphi \ast \varphi_{\text{pre}})$ is not equivalent to $(\exists \emptyset \varphi) \ast \varphi_{\text{pre}}$. However, if we can make $\varphi_{\text{pre}}$ independent of $\emptyset$ then the equivalence would hold. In order to make $\varphi_{\text{pre}}$ independent of $\emptyset$, we existentially quantify the auxiliary variables that are equated to $\emptyset$ in $\varphi_{\text{pre}}$ from both sides of the equivalence. In our current example, since $\varphi_{\text{pre}}$ contains $\emptyset$ only in the equality $\emptyset = d$, we existentially quantify $\emptyset$ from both sides of the equivalence, thus giving $\varphi \ast (\text{true} \land \text{emp}) \Leftrightarrow \emptyset = d \land \text{emp} \ast \psi$. The right-hand side can be further simplified by eliminating $\emptyset$ to obtain $\emptyset = (\text{true} \land \text{emp}) \ast \psi$. Now we can existentially quantify $\emptyset$ from both sides of the equivalence and obtain $(\exists \emptyset \varphi) \ast (\text{true} \land \text{emp}) \Leftrightarrow \emptyset = (\text{true} \land \text{emp}) \ast \psi$.

The above intuitions are formalized in the procedure BiAbduct given in Figure 14. The key step of bi-abduction is the Decompose procedure described above. For each pair $(\delta_1, \delta_2)$ returned by Decompose$(\varphi, \psi)$, we compute $\delta_1'$ and $\delta_2'$ from $\delta_1$ and $\delta_2$, respectively, using the function RemoveVar (lines 3, 4). The function RemoveVar$(\phi_1, \phi_2, \text{mod}_1, B)$ replaces every free variable $v \in \text{mod}_1$ in $\phi_1$ by $e$ if $\phi_2$ implies $v = e$ and $\text{free}(e) \in B \setminus \text{mod}_1$. After renaming, it also removes any redundant equalities.
of the form $x = x$, and equalities implied by $\phi^p$ from $\phi_1$. For our running example, $\delta_1 \equiv v = x \land \psi = \omega$ and $\delta_2 \equiv v = x \land \psi = \omega$. The function $\text{RemoveVar}(\delta_2, \psi, \text{mod}_2, V \cup Y)$ renames $x$ by $x$ in $\delta_2$, hence $\delta_2' \equiv x = \omega$. The function $\text{RemoveVar}(\delta_1, \varphi, \text{mod}_1, V \cup W)$ does not rename any variables from $\delta_1$, hence $\delta_1 \equiv \delta_1' \equiv v = x \land \psi = \omega$.

Next, we process the formula $\delta_1'$ so as to make it independent of $\text{mod}_1$. In line 5, we compute a renaming $\gamma : (Y \mapsto \text{mod}_1)$ such that $\delta_1' \gamma$ is independent of $\text{mod}_1$. This is done by invoking function $\text{ComputeRenaming}$. The function $\text{ComputeRenaming}(\phi, A, B)$ renames a variable $a \in A$ by $b \in B$ if $\phi^p$ implies the equality $a = b$. The renaming $\gamma$ ensures that $\varphi \ast \kappa_1 \Leftrightarrow \exists Z(\delta_2' \ast \psi)$, where $\kappa_1 \equiv \delta_1' \gamma$ and $Z = \text{dom}(\gamma)$. If $\delta_1' \gamma$ is not independent of $\text{mod}_1$ or $\delta_2' \ast \psi$ is not independent of $\text{mod}_2$, we discard the pair $(\delta_1', \delta_2')$ (line 8). Note the asymmetry in dealing with $\delta_1'$ and $\delta_2'$, which stems from the asymmetric structure (3Z only on right side) of the required solution $(\exists X \varphi) \ast \varphi^p \Leftrightarrow \exists Z(\varphi^p \ast \psi)$. For our running example, $Z = \{x\}$ and $\gamma : (x \mapsto v)$ gives a valid renaming, since $\delta_2' \gamma \equiv \omega = \omega$ is independent of $v$.

**Lemma 6.3.** Every $\kappa_1$ and $\hat{Z}$ computed in lines 6 and 7 of $\text{BiAbduct}$ satisfy $\varphi \ast \kappa_1 \Leftrightarrow \exists Z(\delta_2' \ast \psi)$.

**Proof.** Follows from the following equivalences.

1. $\exists Z \varphi \ast \delta_1' \Leftrightarrow \exists Z \delta_2' \ast \psi$ Definition of $\text{Decompose}$ and $\text{RemoveVar}$, and 3 elimination
2. $\exists Z \varphi \ast \delta_1' \Leftrightarrow \varphi \ast \delta_1' \gamma \Leftrightarrow \exists Z \delta_2' \ast \psi$ and $\varphi$ is independent of $\hat{Z}$ variables
3. $\varphi \ast \delta_1' \gamma \Leftrightarrow \exists Z \delta_2' \ast \psi$ from 1,2

For every $\kappa_1$ at line 9 we compute a renaming $\theta : (\hat{Z} \mapsto X)$, where $\hat{Z} \subseteq Y$, so as to render $\kappa_1 \theta$ independent of $X$ (lines 9, 10, 11). The function $\text{ComputeRenaming}(\kappa_1, Y, X)$ computes the renaming $\theta$. Let $\hat{\theta} : (X \mapsto \hat{Z})$ be a renaming such that $\hat{\theta}(x) = z$ only if $\theta(z) = x$. The function $\text{RemoveRedundant}(\phi_1, \phi^p_2)$ removes the equalities from $\phi_1$ that are implied by $\phi^p_2$. It also removes trivial equalities like $x = x$ or $\text{RP}(x[i] = x[i])$ from $\phi_1$. If $\kappa_1' \equiv \text{RemoveRedundant}(\kappa_1 \theta, \phi^p)$ is independent of $X$ then $\text{BiAbduct}$ returns $(\kappa_1', \kappa_2, \hat{Z} \cup \hat{Z})$, where $\kappa_2$ is the formula returned by $\text{RemoveRedundant}(\delta_2' \theta, \phi^p)$, as a solution of strong bi-abduction.

The invocations of $\text{ComputeRenaming}$ in lines 5 and 9 have one important difference: in line 5 only non-array variables in $\text{mod}_1$ are renamed, whereas in line 9 array variables in $Y$ may be renamed. The function $\text{ComputeRenaming}(\phi, A, B)$ renames array variables as follows. An array variable $a \in A$ is renamed to another array variable $b \in B$ if $\phi^p$ implies one of the following facts: (i) $\text{RP}(A[i] = D[i]) \land A[80] = D[80]$, or (ii) $\text{RP}(A[i] = D[i] + 1) \land A[0] = D[0]$, or (iii) $\text{RP}(A[i] = D[i] \land A[i] + 1 = D[i] + 1)$. Higher dimensional arrays can be renamed by performing similar checks for each dimension. For our running example, we have $X = \{\omega\}, \hat{Z} = \{x\}$ and $\theta : (\omega \mapsto \omega)$. It is evident that $(\exists \omega v \mapsto \omega) \ast (\text{true} \land \text{emp}) \Leftrightarrow \exists \omega \omega v \mapsto \omega) \ast (\text{true} \land \text{emp}) \ast (v = \omega \land x \mapsto \omega)$. Thus $\varphi^p \equiv \kappa_1' \equiv \text{RemoveRedundant}((\kappa_1 \theta, \phi^p) \equiv \text{true} \land \text{emp}$, $\varphi^p \ast \kappa_2 \equiv \text{RemoveRedundant}(\delta_2' \theta, \phi^p) \equiv \text{true} \land \text{emp}$, and and $Z = \{x, \omega\}$ is a solution of strong bi-abduction between $\exists \omega \phi \equiv \exists \omega v \mapsto \omega)$ and $\psi \equiv v = x \land x \mapsto \omega)$.

**Lemma 6.4.** Every $\theta$ and $\hat{Z}$ at line 12 of $\text{BiAbduct}$ satisfy $(\exists X \varphi) \ast \kappa_1 \theta \Leftrightarrow \exists Z, \hat{Z} (\delta_2' \ast \psi)$

**Proof.** Follows from the following equivalences.

ACM Transactions on Programming Languages and Systems, Vol. X, No. N, Month 20YY.
Example 5. Let us compute strong bi-abduction between $\exists X \varphi \equiv \exists x h = x[0] \wedge RS(x[1] \mapsto x[1] + 1) \wedge x[80] = \text{null}$ and $\psi \equiv h = y[0] \wedge RS(y[1] \mapsto y[1] + 1) \wedge y[80] = \text{null}$. Let the sets $\text{mod}_1$ and $\text{mod}_2$ be empty.

—The Match procedure finds the following overlap between $\varphi$ and $\psi$: $(M, C, \text{emp}, \text{emp})$ where $M$ is $\text{RP}(x[1] \mapsto y[1] \wedge x[1] + 1 = y[1] + 1)$ and $C$ is $\text{RS}(x[1] \mapsto x[1] + 1)$. Hence $\delta_1$ is computed as $M \wedge h = y[0] \wedge y[80] = \text{null} \wedge \text{emp}$ and $\delta_2$ is computed as $M \wedge h = x[0] \wedge x[80] = \text{null} \wedge \text{emp}$, thus giving the equivalence $\varphi \equiv \delta_1 \Leftrightarrow \delta_2 \wedge \psi$.

—Since the mod set is empty, $\gamma$ is an empty renaming and $\hat{Z}$ is an empty set.

—The set of quantified variables $X$ contains the array variable $x$. We compute the renaming $\theta$ as $(y \mapsto x)$, from the predicate $\text{RP}(x[1] \mapsto y[1] \wedge x[1] + 1 = y[1] + 1)$ present in $\delta_1$. $\delta_\theta$ is the formula $\text{RP}(x[1] \mapsto x[1] \wedge x[1] + 1 = x[1] + 1) \wedge h = x[0] \wedge x[80] = \text{null} \wedge \text{emp}$. $\text{RemoveRedundant}(\delta_\theta, \varphi)$ eliminates the redundant equalities from $\delta_\theta$ and returns the formula $\text{true} \wedge \text{emp}$ which is independent of $x$. $\theta$ is $(x \mapsto y)$ and $\delta_\theta$ is the formula $\text{RP}(y[1] \mapsto y[1] \wedge y[1] + 1 = y[1] + 1) \wedge h = y[0] \wedge y[80] = \text{null} \wedge \text{emp}$, and $\text{RemoveRedundant}(\delta_\theta, \psi)$ removes the redundant equalities and returns the formula $\text{true} \wedge \text{emp}$. Hence the result of strong bi-abduction is $(\exists x \varphi) \equiv \text{true} \wedge \text{emp} \Leftrightarrow \exists y (\text{true} \wedge \text{emp} \wedge \psi)$.

6.2 Implementation of the JOIN rule

In section 5.6 we presented the JOIN rule to merge summaries for two branches of the statement if (e, $S_1$, $S_2$). The premises of JOIN require us to check whether $\varphi_1 \mu \Leftrightarrow \varphi_2$ and $\varphi_1 \mu \Leftrightarrow \hat{\varphi}_2$ for quantifier free LIFS formulas $\varphi_1, \hat{\varphi}_2, \varphi_1$ and $\varphi_2$. We now show how the BiAbduct can be used to implement these checks. We will use the observations in the Proposition 6.1.

Proposition 6.1. Given $\psi$ and $\hat{\psi}$.

(1) if $\psi \equiv (\text{true} \wedge \text{emp}) \Leftrightarrow (\text{true} \wedge \text{emp}) \Leftrightarrow (P \wedge \text{emp}) \equiv (P \wedge \text{emp}) \equiv (P \wedge \text{emp}) \Leftrightarrow (P \wedge \text{emp}) \Leftrightarrow (P \wedge \text{emp}) \equiv \hat{\psi}$ then $\psi \equiv \hat{\psi}$

(2) if $\psi \equiv (\text{true} \wedge \text{emp}) \Leftrightarrow (P \wedge \text{emp}) \equiv (P \wedge \text{emp}) \equiv \hat{\psi}$ then $\psi \Rightarrow \hat{\psi}$

In order to check whether $\varphi_1 \mu \Leftrightarrow \varphi_2$, where $\varphi_1$ is a formula over free variables $V, W$ and $\varphi_2$ is a formula over free variables $V, Y$, we call $\text{BiAbduct}(3W, \varphi_1, \varphi_2, V, Y)$. The following lemma gives sufficient conditions under which we can infer $\varphi_1 \mu \Leftrightarrow \varphi_2$.

Lemma 6.5. If $\hat{Z}$ computed at line 7 of $\text{BiAbduct}$ (Figure 14) is $\emptyset$, and $\delta$ computed at line 9 of $\text{BiAbduct}$ is such that $\kappa_1 \theta$ and $\delta_2 \theta$ are both equivalent to $\text{true} \wedge \text{emp}$ then we can infer $\varphi_1 \mu \Leftrightarrow \varphi_2$.

Proof. Follows from the following equivalences.
In order to implement the check $\hat{\varphi}_1 \mu \Rightarrow \hat{\varphi}_2$, where $\hat{\varphi}_1$ is a quantifier free formula over free variables $V, W$ and $\hat{\varphi}_2$ is a quantifier free formula over free variables $V, Y$, we use the renaming $\bar{\theta}$ computed in the previous step and call $\text{BiAbduct}(\hat{\varphi}_1 \bar{\theta}, \hat{\varphi}_2, V, V)$. The following lemma characterizes sufficient conditions for validity of $\hat{\varphi}_1 \bar{\theta} \Rightarrow \hat{\varphi}_2$.

**Lemma 6.6.** If $\delta'_1$ computed at line 3 of $\text{BiAbduct}$ is equivalent to $\text{true} \land \text{emp}$ and $\delta'_2$ computed at line 4 of $\text{BiAbduct}$ is equivalent to $P \land \text{emp}$ then we can infer $\hat{\varphi}_1 \bar{\theta} \Rightarrow \hat{\varphi}_2$.

**Proof.** If $\delta'_1$ is $\text{true} \land \text{emp}$ then $\gamma$ computed at line 5 of $\text{BiAbduct}$ is an empty renaming (by the definition of $\text{ComputeRenaming}$). Hence the set $\hat{Z}$ computed at line 7 of $\text{BiAbduct}$ is an empty set. Therefore by Lemma 6.3 we have $\hat{\varphi}_1 \bar{\theta} \Rightarrow (\text{true} \land \text{emp}) \Rightarrow P \land \text{emp}$ by Proposition 6.1.

### 6.3 A note on incompleteness of BiAbduct

A strong bi-abduction procedure can be said to be complete if, whenever there exists $\text{LISF}$ formulas $\varphi_{\text{pre}}$ and $\varphi_{\text{post}}$ and a set $Z$ of auxiliary variables for input $\text{LISF}$ formulas $\exists X \varphi$ and $\psi$ such that $\exists X \varphi \land \varphi_{\text{pre}} \Leftrightarrow \exists Z (\varphi_{\text{post}} \land \psi)$, the procedure finds such $\varphi_{\text{pre}}, \varphi_{\text{post}}$ and $Z$. For the $\text{LISF}$ formulas $\varphi: h = x[0] \land \text{RS}(x[\cdot]) \Rightarrow x[\cdot] + 1, 0, 0 \land x[0] = \text{null}$ and $\psi: h = y[0] \land \text{RS}(y[\cdot] + 1) \Rightarrow y[\cdot], 0, 0 \land y[0] = \text{null}$, the fact that $\exists X \varphi \land (\text{true} \land \text{emp}) \Leftrightarrow \exists Y ((\text{true} \land \text{emp}) \land \psi)$ is valid. However, $\text{BiAbduct}$ will not be able to compute this strong bi-abduction. This is because the $\text{Match}$ procedure cannot find the correct overlap between $\varphi^s$ and $\psi^s$. Hence $\text{BiAbduct}$ is not a complete strong bi-abduction procedure. The pure constraint expressing the correct overlap between $\varphi^s$ and $\psi^s$ is not expressible in $\text{LISF}$. In the next section we present techniques to do sophisticated matching.

### 7. AN EXTENSION OF \text{LISF}

In this section, we describe a couple of limitations of the strong bi-abduction technique presented so far and present extensions to overcome these limitations.

\[
\varphi: h = x[0] \land \text{RS}(x[\cdot]) \Rightarrow x[\cdot] + 1, 0, 0 \land x[0] = \text{null} \\
\psi: h \Rightarrow y[0] \land \text{RS}(y[\cdot] + 1) \Rightarrow y[\cdot], 0, 0 \land y[0] = \text{null}
\]

Consider the formulas $\varphi$ and $\psi$ defined above. The formula $\varphi$ represents a linked list of any length (including zero) pointed to by $h$. The length of array $x$ in $\varphi$ is one greater than the length of the linked list pointed to by $h$. Whereas, the formula $\psi$ characterizes a linked list of non-zero length pointed to by $h$. In $\psi$, the length of array $y$ is same as the length of the list pointed to by $h$. The strong bi-abduction of $\varphi$ and $\psi$, however, does not have a valid solution since the constructs of $\text{LISF}$ do not allow us to relate arrays of different lengths ($x$ and $y$ in this case). In order to overcome this shortcoming and enable computation of strong bi-abduction between $\varphi$ and $\psi$ we enrich $\text{LISF}$ with $\text{sub}$ predicate. Section 7.1 describes this enhancement.
Now consider the same formula \( \varphi \) as described above and \( \phi \) defined below.

\[
\phi : h = z[0] \land RS[z[1] \mapsto z[\cdot], 0, 0] \land z[0] = \text{null}
\]

The formulas \( \varphi \) and \( \phi \) are different representations for the linked list of any length (including zero) pointed to by \( h \). The length of array \( x \) (resp. \( z \)) in \( \varphi \) (resp. \( \phi \)) is one greater than the length of the linked list pointed to by \( h \). The strong bi-abduction of \( \varphi \) and \( \phi \) returns a solution \((\varphi_{\text{pre}}, \varphi_{\text{post}})\) that restricts the length of linked list in \( \varphi \ast \varphi_{\text{pre}} \) (or \( \varphi_{\text{post}} \ast \phi \)) to one, although both \( \varphi \) and \( \phi \) model linked lists of arbitrary lengths. The reason for this ‘too restrictive’ solution is that \( \text{LISF} \) does not allow us to compare array elements at equal offsets from opposite ends. In order to overcome this shortcoming and enable computation of strong bi-abduction between \( \varphi \) and \( \phi \) we enrich \( \text{LISF} \) with \( \text{rev} \) predicate. We describe this enhancement in section 7.2.

7.1 Enhancement of \( \text{LISF} \) with \( \text{sub} \) predicate

The Match algorithm can match the \( \text{RS} \) predicates of \( \varphi^s \) and \( \psi^s \) and return the four-tuple \((\text{RP}(x[\cdot]) = y[\cdot] \land x[\cdot] + 1 = y[\cdot] + 1), \varphi^s, \{\}, h \mapsto \psi(0)\). But this overlap is not consistent with \( \varphi^p \) and \( \psi^p \). The Match algorithm returns another solution for the pair \( \varphi^s \) and \( \psi^s \). It first unrolls the predicate \( \text{RS}(x[\cdot]) \mapsto x[\cdot] + 1, 0, 0) \) to give \( x[0] \mapsto x[1] \ast \text{RS}(x[\cdot]) \mapsto x[\cdot] + 1, 1, 0) \) and matches \( x[0] \mapsto x[1] \) with \( h \mapsto \psi(0) \). The residual RS-predicate in \( \varphi^p \) cannot be matched with the one in \( \psi^p \) because of the different offsets in the two RS predicates. The solution returned by Match, in this case, is the four-tuple \((h = x[0] \land y[0] = x[1], h \mapsto y(0), \text{RS}(x[\cdot]) \mapsto x[\cdot] + 1, 1, 0), \text{RS}(y[\cdot]) \mapsto y[\cdot] + 1, 0, 0)) \). For this decomposition, \( M \land \varphi^s \ast L_2 \) (and also \( M \land \psi^s \ast L_1 \)) is inconsistent since \( M \) implies \( y[0] = x[1] \) whereas the spatial parts have predicates \( y[0] \mapsto \_ \ast x[1] \mapsto \_ \) and hence imply \( y[0] \neq x[1] \). Due to the inability to relate arrays of different lengths in \( \text{LISF} \), Match cannot find the right overlap between \( \varphi^p \) and \( \psi^p \). Hence the strong bi-abduction of \( \varphi \) and \( \psi \) fails, although they represent structures for which strong bi-abduction should be possible.

To remedy this problem we introduce a new pure predicate \( \text{sub}(e, l, u, e') \) where \( e \) and \( e' \) are two \( \text{LISF} \) expressions that differ only in the array name and \( l, u \) are non-negative integers. Let \( a \) and \( a' \) be the arrays accessed by the first iterated index of expressions \( e \) and \( e' \), respectively. Intuitively, \( \text{sub}(e, l, u, e') \) establishes the equality of all elements of array \( a' \) and the elements of array \( a \) between the offsets \( l \) and \( u \) from its start and end, respectively. Thus, it implicitly constrains the lengths of arrays \( a \) and \( a' \). The semantics of \( \text{sub}(e, l, u, e') \) is formally defined as follows. Note that we overload the function \( \text{len} \) defined in section 4.2 and used in Figure 8 to operate over single expressions instead of pure or spatial formulas.

\[
(s, h, V, L) \models \text{sub}(e, l, u, e') \iff \exists k \ k + 1 = \text{len}(V, L, e') \land \text{len}(V, L, e) > l + u \land
\text{len}(V, L, e') \land \text{len}(V, L, e) - l - u \land
\forall 0 \leq i \leq k. \text{Eq}(e, (i + l) :: L, s, V) = \text{Eq}(e', i :: L, s, V)
\]

For example, the pure predicate \( \text{sub}(x[i], 1, 0, y[i]) \), represents the fact that length of array \( x \) is one more than that of array \( y \) and that the sequence \( x[1], \ldots, x[80] \) is same as the sequence \( y[0], \ldots, y[80] \). It may seem that we could have used just array names in the \( \text{sub} \) predicate and written the above fact as \( \text{sub}(x, 1, 0, y) \). However, we wish to express \( \text{sub} \) relationships among the nested arrays in a uniform manner, e.g., the predicate \( \text{sub}(a[1][i], 1, 0, d[2][i]) \) expresses the \( \text{sub} \) relationship between

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
the arrays A[1] and D[2]. Hence we use array expressions instead of array names.

The \( sub \) predicate provides us with the vocabulary to relate arrays of different lengths. We now introduce new match rule that uses this predicate to match arrays of different lengths. To avoid nesting of the \( sub \) predicate within a \( RS \) predicate we allow introduction of \( sub \) predicate only while matching \( RS \) predicate which are not nested within another \( RS \) predicate.

For notational convenience we introduce two macros \( SubS \) and \( SubP \), which are defined as follows. \( SubS(S, l, u) \) is defined as the spatial formula obtained by replacing the array variable, say \( A \), in every expression \( e \) in \( S \) having at least one iterated index with an expression \( e' \) which is same as \( e \) but the array variable is replaced with a primed version, say \( A' \). Intuitively, \( SubS(S, l, u) \) returns a spatial formula over the primed versions of the array names that will be related to the original unprimed names by the \( sub \) predicates. \( SubP(S, l, u) \) generates a pure fact relating the newly introduced array variables, like \( A' \), with the old ones, like \( A \). Let the function \( lb(e) \) replace the first iterated index in \( e \) by the index \([ \cdot ]\). \( SubP(S, l, u) \) returns a conjunction of facts of the form \( sub(lb(e), l, u, lb(e')) \) for every expression \( e \) in \( S \) replaced with \( e' \) by \( SubS(S, l, u) \). The macro \( SubP(S, l, u) \) generates the conjunction of such \( sub \) predicates. For example, \( SubS(x[i] \mapsto x'[i+1], 1, 0) \) returns the spatial formula \( x'[i] \mapsto x'[i+1] \) and \( SubP(x[i] \mapsto x'[i+1], 1, 0) \) returns the pure formula \( sub(x[i], 1, 0, x'[i]) \wedge sub(lb(x[i+1]), 1, 0, lb(x'[i+1])). \) By definition of \( lb \), \( sub(lb(x[i+1]), 1, 0, lb(x'[i+1])) \equiv sub(x[i], 1, 0, x'[i]). \)

**Proposition 7.1.** For a predicate \( RS(S, l, u) \) not embedded in any \( RS \) predicates, \( RS(S, l, u) \wedge SubP(S, l, u) \Leftrightarrow RS(SubS(S, l, u), 0, 0) \wedge SubP(S, l, u). \)

We extend the rule \( MatchRsA \) in \( Match \) algorithm to the rule \( MatchRsA \) that uses \( sub \) predicate to match two \( RS \) predicates. We can now use the rule \( MatchRsA \) to match \( RS(x[i] \mapsto x'[i+1], 1, 0) \) and \( RS(y[i] \mapsto y'[i+1], 0, 0) \), and thus compute \( Match(\varphi^*, \psi^*) \) as a set consisting of \( (M, \varphi^*, \psi^*, \{\}, \{\}) \), where \( M \) is \( h = x[0] \wedge y[0] = x[1] \wedge \Box RP(x'[i] = y'[i] \wedge x'[i+1] = y'[i+1], 0, 0) \). This match is consistent with \( \varphi^* \) and \( \psi^* \). Hence the procedure \( Decompose \) computes \( \delta_1 \) as \( \delta_1 = M \wedge \Box \varphi[80] = \text{null} \wedge \text{emp} \) and \( \delta_2 \) as \( \delta_2 = M \wedge h = x[0] \wedge x[80] = \text{null} \wedge \text{emp} \), such that \( \varphi^* \delta_1 \Leftrightarrow \delta_2 \psi^* \).

The use of \( sub \) predicate allows us to express equality constraints between arrays of different lengths. Implicitly this allows to express difference constraints between lengths of array variables which is not expressible in \( LISF \). \( LISF \) can express only equality of array lengths.

### 7.2 Enhancement of \( LISF \) with \textit{rev} predicate

Consider the formulas \( \varphi \) and \( \phi \) defined at the start of section 7. The \( Match \) algorithm will match the \( RS \) predicates in \( \varphi \) and \( \phi \) and return the four-tuple \( (M, \varphi^*, \psi^*, \{\}, \{\}) \) as the only solution, where \( M \) is the pure formula \( RP(x[i] = z[i+1] \land z[i] = x[i+1], 0, 0) \). But this too restrictive constraint restricts the length of the matched list to be \( \leq 1 \).
Although $\varphi$ and $\phi$ represent a same set of structures in the heap, bi-abduction of $\varphi$ and $\phi$ generates constraints that reduce this set of structures. This is because the pure constraint describing the overlap of a list expressed as $RS(x[::]) \Rightarrow x[::+1], 0, 0$ and the same list expressed as $RS(z[::+1]) \Rightarrow z[::], 0, 0$ cannot be expressed in $\text{LISF}$ without restricting the lengths of $x$ and $z$. To remedy this problem we introduce a new predicate $\text{rev}(e, e')$ where $e$ and $e'$ are $\text{LISF}$ expressions that differ only in the array name. The semantics of $\text{rev}(e, e')$ is defined as follows.

$$
(s, h, V, L) \models \text{rev}(e, e') \text{ iff } \exists k. k + 1 = \text{len}(V, L, e') = \text{len}(V, L, e) \land 
\forall 0 \leq i \leq k. \exists_a (e, i : L, s, V) = \exists_a (e', (k - i) : L, s, V)
$$

(7.7)

For example the predicate $\text{rev}(x[::], z[::])$ asserts that $x$ and $z$ are arrays of same lengths and that the sequence $x[0], x[1], \ldots, x[80]$ is same as $z[0], z[1], \ldots, z[0]$. The $\text{rev}$ predicate provides us with the vocabulary to relate array elements that are at the same offsets from the opposite ends. We now introduce new match rule that uses $\text{rev}$ predicate to match an array with the reverse of another array. To avoid nesting of the $\text{rev}$ predicate within a $\text{RP}$ predicate, we allow introduction of $\text{rev}$ predicate only while matching $\text{RS}$ predicates that are not nested within another $\text{RS}$ predicate.

For notational convenience we introduce two macros $\text{RevS}$ and $\text{RevP}$, which are defined as follows. $\text{RevS}(S)$ is the spatial formula obtained as follows. Initially, we replace the first indexed term $[]$ (resp. $[: + 1]$) in every expression $e$ in $S$ with an indexed term $[::]$ (resp. $[::+1]$). Then we replace the array variable in such expressions, say $A$, with a primed variable, say $A'$. The function $\text{RevP}(S)$ denotes a pure fact relating the newly introduced array variables, like $A'$, with the old ones, like $A$. Recall from previous section that $\text{lb}(e)$ returns the expression same as $e$ but with its first indexed index switched to $[]$. $\text{RevP}(S)$ returns a conjunction of facts of the form $\text{rev}(\text{lb}(e), \text{lb}(e'))$ for every expression $e$ in $S$ replaced with $e'$ by $\text{RevS}(S)$. Intuitively, $\text{RevS}(S)$ returns a spatial formula over the primed versions of the array names that are related to the original unprimed names through the $\text{rev}$ predicates. The macro $\text{RevP}(S)$ generates the conjunction of such $\text{rev}$ predicates. For example, $\text{RevS}(z[::+1]) \Rightarrow z[::]$ returns the spatial formula $z'[::] \Rightarrow z'[::+1]$ and $\text{RevP}(z[::+1]) \Rightarrow z[::]$ returns the pure formula $\text{rev}(\text{lb}(z[::+1]), \text{lb}(z'[::+1]))$ (7.7).

Note that by definition of $\text{lb}$ the above formula reduces to $\text{rev}(z[::], z'[::])$.

**Proposition 7.2.** For a predicate $\text{RS}(S, l, u)$ not embedded in any $\text{RS}$ predicate, $\text{RS}(S, l, u) \land \text{RevP}(S) \iff \text{RS}(\text{RevS}(S), u, l) \land \text{RevP}(S)$.

We extend the rule $\text{MatchRS}$ in $\text{Match}$ algorithm to the rule $\text{MatchRSB}$ that uses $\text{rev}$ predicate to match two $\text{RS}$ predicates. We can now use the rule $\text{MatchRSB}$ to match $\text{RS}(x[::]) \Rightarrow x[::+1], 0, 0$ and $\text{RS}(z[::+1]) \Rightarrow z[::], 0, 0$, and thus compute $\text{Match}(\varphi^*, \phi^*)$ as $(M, \varphi^*, \{\}, \{\})$, where $M$ is $\text{RP}(x[::]) = z'[::] \Rightarrow z'[::+1] = z'[::+1], 0, 0) \land \text{rev}(z[::], z'[::+1])$. This match is consistent with $\varphi^*$ and $\phi^*$. Hence the procedure $\text{Decompose}$ computes $\delta_1$ as $M \land h = z[0] \land z[0] = \text{null} \land \text{emp}$ and...
δ₂ as \( M \land h = x[0] \land x[0] = \text{null} \land \text{emp} \), such that \( \varphi \land \delta_1 \iff \delta_2 \land \phi \).

The use of \textit{rec} predicates allows us to equate array elements which are arbitrary distance apart (e.g. \( i \) and \( k-i \) in Equation 7.7). \textit{LISF} does not allow us to express this fact.

8. SATISFIABILITY CHECKING ALGORITHMS

In this section we provide a sound procedures for checking satisfiability of (a) \textit{LISF} formulas, and (b) \textit{LISF} extended with \textit{sub} and \textit{rec} predicates. Any \textit{LISF} formula is of the form \( P \land S \) or \( \exists X. P \land S \). Since \( \exists X. P \land S \) is equisatisfiable with \( P \land S \), we present satisfiability procedures only for quantifier free \textit{LISF} formulas.

8.1 Satisfiability checking procedure for \textit{LISF}

The basic idea of the satisfiability checking procedure is to convert a \textit{LISF} formula to a formula in separation logic without iterated predicates (satisfiability checking of these formulas can be reduced to satisfiability checking of formulas in the theory of equality and is hence efficiently decidable). This is achieved by instantiating the lengths of all dimensions of all arrays to fixed constants, and by soundly unrolling the \textit{RP} and \textit{RS} predicates. The array lengths are so chosen that the offsets specified in the fixed indices of all expressions in the formula are within the respective array bounds. We illustrate the algorithm through an example before presenting it formally.

\textbf{Example 6.} Consider a \textit{LISF} formula \( \varphi \equiv (h = x[0]) \land (g = y[0]) \land (t = x[1]) \land (x[0] = y[0]) \land (y[0] = \text{null}) \land \text{RS}(x[0] \mapsto x[0] + 1) \land y[1] \mapsto y[1] + 1, 0, 0). \)

The \textit{RS} predicate in \( \varphi \) requires that \( x \) and \( y \) have same lengths. The expressions \( x[0] \) and \( x[0] \) (respectively \( x[0] \varphi \) and \( y[0] \) require that the length of array \( x \) (respectively, array \( y \) be at least 2. A sound way of checking the satisfiability of \( \varphi \) is to guess the lengths of the arrays and expand the \textit{RS} and \textit{RP} predicates for these array lengths so as to obtain a standard separation logic formula (one without \textit{RS} or \textit{RP} predicates).

For the current example, setting the lengths of both arrays \( x \) and \( y \) to 2 satisfies the constraints imposed on their lengths by \( \varphi \). If the length of array \( x \) is 2, we have \( x[0] = x[1] \) and \( x[0] = x[0] \). Similarly, if length of \( y \) is 2, we have \( y[0] = y[1] \).

Moreover, the predicate \( \text{RS}(x[0] \mapsto x[0] + 1) \land y[1] \mapsto y[1] + 1, 0, 0) \) can be written as \( x[0] \mapsto x[0] + 1 \). By applying the semantic definition of \( \text{RS} \) given in Figure 8). Hence, if we set the lengths of \( x \) and \( y \) to 2, we can rewrite \( \varphi \) as

\[ \psi \equiv h = x[0] \land g = y[0] \land t = x[0] \land x[1] = y[1] \land y[1] = \text{null} \land x[0] \mapsto x[1] + 1 \land y[0] \mapsto y[1]. \]

The only array expressions in \( \psi \) are of the form \( x[i] \) or \( y[i] \), \( i \in \{0, 1\} \). It has no \textit{RS} or \textit{RP} predicates. Hence it is a standard separation logic formula. It is evident that if \( \psi \) is satisfiable then so is \( \varphi \).

The formula \( \psi \) can be satisfied by having \( x[0] = h = t = l_1 \), \( y[0] = g = l_2 \) and \( x[1] = y[1] = \text{null} \), \( l_1 \neq l_2 \), \( h(l_1) = \text{null} \), and \( h(l_2) = \text{null} \). Hence \( \varphi \) is satisfiable.

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
The function `GetLengths(\varphi)` computes these lengths. Any model of the flattened formula $\psi$ is also a model of \textit{LISF} formula $\varphi$. The function `$\texttt{sat}_\varphi(\psi)$` determines the satisfiability of a separation logic formula $\psi$.

The predicates $\textit{RS}$, $\textit{RP}$ and the expressions with fixed indices in $\varphi$ impose restrictions on the length of different dimensions of array variables. The function `GetLengths` encodes these constraints in the formula $F$. The variables in $F$ are represented as $\langle X, i \rangle$, where $X$ is a free $k$-dimensional array variable in $\varphi$ and $1 \leq i \leq k$. The variable $\langle X, i \rangle$ represents a safe length for the $i^{th}$ dimension of $X$ that avoids indexing errors. Lines 2-7 add constraints to $F$ so that evaluation of fixed indices in the expressions of $\varphi$ does not cause an array indexing error. The function $\textit{LB}(\varphi)$ returns a set of tuples $\langle X, i, l \rangle$ such that there is an expression in $\varphi$ accessing the $i^{th}$ dimension of array $X$ with a fixed index $l$. Similarly, $\textit{UB}(\varphi)$ returns a set of tuples $\langle X, i, u \rangle$ such that there is an expression in $\varphi$ accessing the $i^{th}$ dimension of array $X$ with a fixed index $u$. The function `IterConst(\varphi)` returns a set of pairs $\langle \langle X, i \rangle, \langle (X, j) \rangle \rangle$ such that there exist expressions $e_1$ and $e_2$ embedded in an $\textit{RS}$ (or $\textit{RP}$) predicate such that $\textit{free}(e_1) = X$, $\textit{free}(e_2) = Y$ and $i$ and $j$ are the dimensions of $X$ and $Y$, respectively, over which the $\textit{RS}$ (or $\textit{RP}$) predicate iterates. Lines 6 and 7 capture constraints imposed by $\textit{RS}$ and $\textit{RP}$ predicates on the lengths of array dimensions. The function `IterConst` is defined in Figure 18. The function `\textit{iterDim}(e, i)` used in Figure 18 returns the dimension number corresponding to the $i^{th}$ iterated index in $e$ if $e$ has at least $i$ iterated indices, otherwise it returns $-1$. 

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig16.png}
\caption{Satisfiability procedure: $\texttt{sat}(\varphi)$}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig17.png}
\caption{Unroll functions}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig18.png}
\caption{Function `IterConst(\varphi)`}
\end{figure}
The formula $F$ is always satisfiable as the only constraints it has are of the form $c \leq \langle x, i \rangle$ or $\langle x, i \rangle = \langle y, j \rangle$ ($c$ is a constant). To construct a satisfying assignment to the variables in $F$ we first compute the equivalence classes of variables (implied by equality constraints) in $F$. We set the value of each variable in an equivalence class to the largest constant among all the inequality constraints involving those variables. The function $\text{Solve}(F)$ returns such an assignment to the variables in $F$. Any structure having array sizes conforming to $\text{lentbl}$ returned by $\text{GetLengths}(\varphi)$ (line 1 of $\text{sat}$) is a well-formed structure for $\varphi$.

$\text{Flatten}$ uses an intermediate function $\text{isFlat}(\varphi)$ which returns true if $\varphi$ does not have any $\text{RS}$ or $\text{RP}$ predicate, otherwise it returns false. The function $\text{FindLength}(t, \text{lentbl})$, where $t$ is $\text{RP}(P, l, u)$ (resp. $\text{RS}(S, l, u)$), returns the length of array dimension corresponding to the first iterated index of any array expression in $P$ (resp. $S$). $\text{Flatten}$ then eliminates the iterated predicates $t$ by the function $\text{iter_unroll}(t, cnt)$, which is a repeated application of $\text{unroll}(t, 0)$ as defined in Figure 17. Recall that $\text{unroll}(\text{RS}(S, l, u), d)$ is defined in Section 6 as the formula obtained by replacing the $(d + 1)^{\text{th}}$ iterated index $[i]$ (resp. $[i + 1]$) of every expression in $S$ by the fixed index $[l]$ (resp. $[l + 1]$). The function $\text{unroll}(\text{RP}(P, l, u), d)$ is analogously defined. Finally, all expressions that access a dimension, say $i$, of an array, say $X$, with a fixed index $\$u$ are modified by replacing $\langle \$u \rangle$ with $\langle \text{lentbl}(X, i) - 1 - u \rangle$. The function $\text{ModifyUB}(\varphi, \text{lentbl})$ does this transformation.

**Lemma 8.1.** For a $\text{LISF}$ formula $\varphi$, if $\text{sat}(\varphi)$ returns true then $\varphi$ is satisfiable.

### 8.2 Satisfiability checking procedure for $\text{LISF}$ extended with sub and rev predicates

With the use of $\text{sub}$ and $\text{rev}$ lemmas in bi-abduction, the pure part of $\text{LISF}$ formulas can have additional conjunction of constraints of the form $\text{sub}(e, l, u, e')$ and $\text{rev}(e, e')$. We need to modify the $\text{Flatten}$ and $\text{GetLengths}$ algorithms for checking satisfiability of $\text{LISF}$ formulas in the presence of these additional constraints. The modified algorithms $\text{FlattenL}$ and $\text{GetLengthsL}$ are presented in Figure 19. The algorithm $\text{satL}(\varphi)$ uses these modified algorithms to flatten $\varphi$.

Algorithm $\text{GetLengthsL}$ takes into account the constraints imposed on array lengths by $\text{sub}(e, l, u, e')$ and $\text{rev}(e, e')$ in addition to the constraints considered in $\text{GetLengths}$ to calculate the array lengths.

Let $\text{arr}(e)$ give the array name used to build the array expression $e$ and $\text{idim}(e)$ give the dimension number corresponding to first iterated index in $e$. The predicate $\text{sub}(e, l, u, e')$ requires that the length, $\text{len}$, of dimension $\text{idim}(e')$ of $\text{arr}(e')$ be equal to length of dimension $\text{idim}(e)$ of $\text{arr}(e) - (l + u)$ (as defined in Eq. 7.6). Lines 2-5 add such constraints to $F$. The predicate $\text{rev}(e, e')$ requires that the length of dimension $\text{idim}(e)$ of $\text{arr}(e)$ be same as the length of dimension $\text{idim}(e')$ of $\text{arr}(e')$ (as defined in Eq. 7.7). Lines 7-10 of $\text{GetLengthsL}$ add these constraints to $F$.

Suppose for a predicate $\text{sub}(e, l, u, e')$ (or $\text{rev}(e, e'))$, the number of dimensions of $\text{arr}(e)$ and $\text{arr}(e')$ are $k$ and $k'$, respectively. The definition of $\text{sub}$ (resp. $\text{rev}$) requires that for every $0 \leq j \leq k - \text{idim}(e)$, the length of dimension $\text{idim}(e) + j$ of $\text{arr}(e)$ is same as the length of dimension $\text{idim}(e') + j$ of $\text{arr}(e')$. The function $\text{EquateHigher}(e, e', F)$ adds such constraints to $F$ (lines 6 and 11). Lines 12-17 add constraints imposed on array lengths by $\text{RS}$ and $\text{RP}$ predicates and expressions with fixed indices. In contrast to constraints obtained in $\text{GetLengths}$, constraints
in \text{GetLengths}(\varphi)\) may have difference constraints. This is due to the constraints imposed by the predicate \(\text{sub}(e,l,u,e')\) in line 5. Hence the formula \(F\) may be unsatisfiable. The function \text{sat\_dc}(F)\) at line 18 checks whether \(F\) is satisfiable. If \(F\) is satisfiable \text{GetLengths}(\varphi)\) returns the model constructed by \text{SolveDiff}(F)\) (line 19), otherwise it raises an an error indicating unsatisfiability of \(\varphi\) (line 21). Any structure having array sizes confirming to \(\text{lentbl}\) returned by \text{GetLengths}(\varphi)\) is a well-formed structure for \(\varphi\).

The function \text{FlattenL} first soundly eliminates the predicates \(\text{sub}(e,l,u,e')\) (line 2) and \(\text{rev}(e,e')\) (line 3) from \(\varphi\). It replaces the predicates \(\text{sub}(e,l,u,e')\) (resp. \(\text{rev}(e,e')\)) with a pure constraint given in the defining equation 7.6 (resp. 7.7) by calling \text{AddSubConstrs} (resp \text{AddRevConstrs}) at line 2 (resp. line 3). Finally it soundly eliminates the iterative predicates in \(\varphi\) by calling \text{Flatten}(\varphi,\text{lentbl})\).

**Lemma 8.2.** Given a LISF formula \(\varphi\) with \text{sub} and \text{rev} predicates, if \text{sat\_dc}(\varphi)\) returns true then \(\varphi\) is satisfiable.

The satisfiability procedures presented in the previous subsections are sound but incomplete. This is because \text{GetLengths}(\varphi)\) and \text{GetLengthsL}(\varphi)\) return only one of the many (possibly infinite) mappings from array dimensions to their lengths. The formula \(\varphi\) may be satisfiable, but not for the array length mappings returned by the function \text{GetLengths} or \text{GetLengthsL}. In [Gulavani et al. 2009] we show that satisfiability checking of a subclass of LISF having only single dimensional arrays is decidable. Any formula \(\varphi\) belonging to this subclass is satisfiable iff it is satisfiable for some array length mapping in the finite set \(M_{\varphi}\) of array length mappings. This means that if \(\varphi\) is satisfiable then there exists a model of bounded size. Hence satisfiability checking is decidable for this subclass of LISF. Unfortunately, the size of the finite set is doubly exponential in the size of \(\varphi\) in the worst case. However, the efficient but incomplete procedures of the previous two subsections and the inefficient but complete decision procedure given in [Gulavani et al. 2009] are two extremes of the satisfiability checking procedures. The insights in these contrasting
Fig. 20. Experimental results on (a) list manipulating examples from [Calcagno et al. 2007], (b) examples from [Abdulla et al. 2008; Møller and Schwartzbach 2001], (c) functions from Firewire Windows Device Drivers, and (d) a miscellaneous set of programs. For a program in each row, Column ‘size’ indicates its size in terms of lines of code, Column ‘time(s)’ indicates time in seconds taken by the SpInE to calculate the number of triples indicated in Column IV, and Column V indicates whether the discovered triples give a complete specification for the program. Experiments performed on Pentium 4 CPU, 2.66GHz, 1 GB RAM.

procedures can be exploited for tuning the efficiency and precision of satisfiability checking procedure as suitable for a specific application domain.

9. IMPLEMENTATION

We have implemented the inference rules to generate specifications of programs in a tool SpInE. It takes as input a C program and outputs summaries for each procedure in the program. SpInE analyzes the program in a bottom-up manner, i.e., a procedure is analyzed before analyzing its callers. We tabulate the procedure summaries in a central repository. Currently SpInE cannot generate accelerated summaries for (mutually) recursive procedures. Analysis of pointer arithmetic is also beyond its current scope. SpInE takes two optional input arguments – -lemmas and -join – to guide the application of heuristics for generating useful summaries.

Option -lemmas. With this option the strong bi-abduction algorithm uses the predicates sub and rev, described in Section 7, to generate more expressive summaries. The algorithm Match uses the rules MATCHRSA and MATCHRSB described in Section 7 in addition to the rules outlined in Figure 15.

Option -join. With this option turned on SpInE tries to merge summaries for two branches of the if-then-else statement by using the rule Join presented in Figure 13. This helps generate concise specifications for branching constructs and potentially complete specifications when such constructs are embedded in loops.

9.1 Experimental Evaluation of SpInE

The results of running SpInE on a set of challenging programs, without -lemmas or -join option, are tabulated in Table 20. Programs in Table 20(a) are adopted from [Calcagno et al. 2007]. Program delete is the same as the motivating example in Section 1. Programs in Table 20(b) are adopted from [Abdulla et al. 2008; Møller and Schwartzbach 2001].
Table 20. Results of running SpInE with -lemmas and -join option. Columns are same as in Table 20.

<table>
<thead>
<tr>
<th>Progs</th>
<th>size</th>
<th>time(s)</th>
<th>IV</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete</td>
<td>42</td>
<td>0.082</td>
<td>21</td>
<td>No</td>
</tr>
<tr>
<td>rev-rev</td>
<td>30</td>
<td>0.025</td>
<td>4</td>
<td>No</td>
</tr>
<tr>
<td>off-trav</td>
<td>31</td>
<td>0.016</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>dll-trav-2</td>
<td>24</td>
<td>0.014</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>PnpRemove</td>
<td>460</td>
<td>23.800</td>
<td>32</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Fig. 21. Experimental results of running SpInE with -lemmas and -join option. Columns are same as in Table 20.

These programs manipulate singly or doubly linked lists. In each of these tables, the fourth column indicates the number of summaries inferred by SpInE. The last column indicates whether the inferred summaries provide a complete specification for the corresponding program. SpInE inferred richer summaries than those inferred by the tool in [Calcagno et al. 2007]. For example, for the programs delete and reverse, SpInE infers preconditions with cyclic lists (indicated by * in fourth column). For the program delete some of the inferred preconditions even have a lasso structure.

The examples in Table 20(c) are program fragments modifying linked structures in the Firewire Windows Device Driver. We report only the summaries discovered for the main procedures in these programs. A complete set of summaries is discovered for all the other procedures in these programs. The original programs and data structures have been modified slightly so as to remove pointer arithmetic. These programs perform selective deletion or search through doubly linked lists. The program PnpRemove iterates over five different cyclic lists and deletes all of them; it has significant branching structure. All programs except CancelIrp refer to only the next field of list nodes. The program CancelIrp also refers to the prev field of list nodes. The increased number of inferred summaries for CancelIrp is due to the exploration of different combinations of prev and next fields in the the pre and postconditions. We have checked whether the computed summaries form a complete specification for the corresponding programs by manually going through the summaries output by SpInE. We found that the summaries inferred for all programs except PnpRemove are complete. These summaries capture the transformations on an unbounded number of heap cells, although they constrain only the next fields of list nodes. Hence these summaries can be plugged in contexts where richer structural invariants involving both next and prev fields are desired.

The program PnpRemove iterates over five different cyclic lists and deletes all of them; it has significant branching structure. All programs except CancelIrp refer to only the next field of list nodes. The program CancelIrp also refers to the prev field of list nodes. The increased number of inferred summaries for CancelIrp is due to the exploration of different combinations of prev and next fields in the the pre and postconditions. We have checked whether the computed summaries form a complete specification for the corresponding programs by manually going through the summaries output by SpInE. We found that the summaries inferred for all programs except PnpRemove are complete. These summaries capture the transformations on an unbounded number of heap cells, although they constrain only the next fields of list nodes. Hence these summaries can be plugged in contexts where richer structural invariants involving both next and prev fields are desired.

Programs in Table 20(d) is a miscellaneous collection of singly or doubly linked list manipulating routines. Program nested deletes a nested linked list, rev-rev reverses a linked list twice. Program off-trav has two loops – the first loop traverses all elements except the head and the second loop traverses all elements of the list. Program dll-trav-2 also has two loops – the first loop traverses the double linked list from head to tail following the next field and the second loop traverses the same list from tail to head following the prev field. SpInE is unable to generate a complete specification for any of these programs, except the program nested.

We repeated the experiments by running SpInE with -lemmas and -join option. SpInE can now generate richer specifications for the program tabulated in

3available to the interested readers at http://www.cfdvs.iitb.ac.in/~bhargav/spine.html
Table 21. Complete specification can now be generated for programs \texttt{off-trav} and \texttt{PnpRemove}. The use of \texttt{rev} (resp. \texttt{sub}) predicate was instrumental for generating richer specifications for \texttt{rev-rev} and \texttt{dll-trav-2} (resp. \texttt{off-trav}). The use of \texttt{JOIN} rule was instrumental for generating complete specification for \texttt{PnpRemove}. \texttt{PnpRemove} has several nested branching constructs of the form \texttt{if (v \neq \texttt{null}) delete(v)} inside while loops. The use of \texttt{JOIN} rule enabled \textsc{SpInE} to generate a single, complete summary for such branching constructs. This facilitated the generation of complete specification for each while loop in the program \texttt{PnpRemove}. With the options \texttt{-lemma} and \texttt{-join}, \textsc{SpInE} neither produced any new summaries nor did it take more time while analyzing the remaining programs.

10. CONCLUSION

We have presented inference rules for bottom-up and compositional shape analysis. Strong bi-abduction and satisfiability checking form the basis of our inference rules. The novel insight of inductive composition is captured by the inference rule \textsc{Induct}. This rule enables us to hoist the Hoare triple of a loop body outside the loop. This enables uniform application of the compositional analysis to entire program, albeit without recursive procedures.

We have introduced a new logic called \textsc{LISF} to express the Hoare triples. \textsc{LISF} provides a uniform framework to express recursive predicates characterizing list-like and nested list-like data-structures. This logic enables us to relate the data-structures in the pre and postcondition of the program. We illustrate the advantages of Hoare triples expressed using \textsc{LISF} over those expressed using recursive predicates with respect to succinctness and composability.

We have presented sound procedures for strong bi-abduction and satisfiability checking of \textsc{LISF} formulas. Although neither of these procedures are complete, we identify a fragment of \textsc{LISF} that has a small model property. Hence checking satisfiability of this fragment is decidable. But, its worst case complexity is doubly exponential. Secondly, we do not yet know whether the satisfiability checking of entire \textsc{LISF} is decidable. Hence we use the sound procedure \texttt{sat} in our implementation for checking satisfiability of \textsc{LISF} formulas.

One possible direction for future work is to enhance the strong bi-abduction procedure to make it complete for an expressive fragment of \textsc{LISF}. Another possibility is to have a fall-back mechanism to compute only a bi-abduction, whenever strong bi-abduction cannot be computed (or strong bi-abduction does not exist). Identifying a class of programs for which our inference rules can generate complete specification is also an interesting problem to solve. In future we would like to extend our technique to generate expressive specifications for programs having recursive procedures and those manipulating tree-like data-structures.

Acknowledgment. We thank Hongseok Yang and Dino Distefano for introducing us to the idea of abduction and for providing us with benchmark programs. We also thank the anonymous reviewers for their insightful and critical comments. The ideas in the appendix are motivated by the suggestions made by one of the reviewers of earlier draft. The first author was supported by Microsoft Corporation and Microsoft Research India under the Microsoft Research India PhD Fellowship Award.
REFERENCES


ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
A. COMPOSITION OF STRONG HOARE TRIPLES USING STRONG BI-ABDUCTION

Let Post(S, (s, h)) denote the set of states resulting from the execution of S starting from the initial state (s, h). We say that a program statement S satisfies domain expansion property if for any state (s', h') ∈ Post(S, (s, h)), we have dom(h') ⊇ dom(h). A program statement S satisfies minimal resource property if (s', h') ∈ Post(S, (s, h)) implies that for all h₀ disjoint from h and h', (s', h' ⊔ h₀) ∈ Post(S, (s, h ∪ h₀)). It is straightforward to see that all the primitive program statements given in Figure 2, except the deallocation statement dispose, satisfy the domain expansion and minimal resource properties.

Consider (s', h') ∈ Post(S, (s, h)), where s'(x) = s(y) = l', s(x) = l, s(y) = l', and dom(h') = dom(h) = {0}. Let dom(h₀) = {l'}. Starting from a state (s, h ∪ h₀), execution of S cannot result in a state (s', h' ⊔ h₀) because the statement x := new cannot allocate a new object at an already allocated location l’ ∈ dom(h ∪ h₀). Hence, (s', h' ⊔ h₀) ∉ Post(S, (s, h ∪ h₀)), although h₀ is disjoint from h and h'.

In the following, we first show that programs without the deallocation statement satisfy the domain expansion and minimal resource properties. Later, we prove that if the deallocation statement is disallowed then the composition of strong Hoare triples using strong bi-abduction yields strong Hoare triples.

Lemma A.1. If statements S₁ and S₂ satisfy domain expansion and minimal resource properties then their composition S₁; S₂ also does.

Proof. Consider (s'', h'') ∈ Post(S₁; S₂, (s, h)). Let (s', h') be an intermediate state such that (s', h') ∈ Post(S₁, (s, h)) and (s'', h'') ∈ Post(S₂, (s', h')). Since S₁ and S₂ both satisfy domain expansion property, it follows that dom(h'') ⊇ dom(h'). Hence S₁; S₂ satisfies the domain expansion property.

Consider a trace starting from (s, h) such that (s', h') ∈ Post(S₁, (s, h)) and (s'', h'') ∈ Post(S₂, (s', h')). By the domain expansion property, we have dom(h'') ⊇ dom(h'). Hence for all h₀ such that h₀ is disjoint from h', we have h₀ is disjoint from h₀ ∪ h'. Combining these with the fact that both S₁ and S₂ satisfy minimal resource property, we obtain that for all h₀ such that h₀ is disjoint from h', (s'', h₀ ∪ h'') ∈ Post(S₂, (s', h₀ ∪ h')) and (s', h₀ ∪ h') ∈ Post(S₁, (s, h₀ ∪ h')). Hence S₁; S₂ satisfies minimal resource property.

Lemma A.2. If assert(B); S satisfies domain expansion and minimal resource properties then while(B) S also does.

Proof. This can be proved by induction on the number of times the loop body iterates using Lemma A.1 as the base case.
LEMMA A.3. If $S_1$ satisfies domain expansion and minimal resource properties, $[\phi_1] S_1 [\hat{\phi}_1]$ is a strong Hoare triple, and $\phi_{pre} \cap \text{mod}(S_1) = \emptyset$ then $[\phi_1*\phi_{pre}] S_1 [\hat{\phi}_1*\phi_{pre}]$ is a strong Hoare triple.

PROOF. By frame rule, it is evident that $[\phi_1*\phi_{pre}] S_1 [\hat{\phi}_1*\phi_{pre}]$ is a valid Hoare triple.

We now show that $[\phi_1*\phi_{pre}] S_1 [\hat{\phi}_1*\phi_{pre}]$ is strong. Consider $(s,h) \models [\hat{\phi}_1*\phi_{pre}]$. Let $h = h_1\#h_2$ such that $(s,h_1) \models \hat{\phi}_1$ and $(s,h_2) \models \phi_{pre}$. Since $[\phi_1] S_1 [\hat{\phi}_1]$ is a strong Hoare triple, there exists $(s',h'_1)$ such that $(s,h_1) \in \text{Post}(S_1,(s',h'_1))$. Since $s$ and $s'$ map variables other than $\text{dom}(S_1)$ to same values, and since $\phi_{pre}$ is independent of $\text{mod}(S_1)$, it follows that $(s',h_2) \models \phi_{pre}$. Moreover, since $S_1$ satisfies domain expansion property, $\text{dom}(h'_1) \subseteq \text{dom}(h_1)$ and hence $h'_1\#h_2$. Consequently, $(s',h'_1 \cup h_2) \models \phi_1*\phi_{pre}$. Furthermore, since $S_1$ satisfies minimal resource property, $(s,h_1 \cup h_2) \in \text{Post}(S_1,(s',h'_1 \cup h_2))$. Thus for every $(s,h) \models [\hat{\phi}_1*\phi_{pre}$ there exists $(s',h') \models \phi_1*\phi_{pre}$ such that $(s,h) \in \text{Post}(S_1,(s',h'))$. Hence $[\phi_1*\phi_{pre}] S_1 [\hat{\phi}_1*\phi_{pre}]$ is a strong Hoare triple. □

LEMMA A.4. If statements $S_1$ and $S_2$ satisfy domain expansion and minimal resource properties, $[\hat{\phi}_1] S_1 [\hat{\phi}_2]$ and $[\phi_2] S_2 [\hat{\phi}_2]$ are strong Hoare triples, $\phi_1*\phi_{pre} \Leftrightarrow \exists Z. (\phi_{post}*\phi_2)$, and $\phi_{pre} \cap \text{mod}(S_1) = \phi_{post} \cap \text{mod}(S_2) = \emptyset$ then $[\phi_1*\phi_{pre}] S_1 [\phi_2] [\exists Z. (\phi_{post}*\phi_2)]$ is a strong Hoare triple.

PROOF. Given the assumptions and using the frame rule, it is straightforward to show that $[\phi_1*\phi_{pre}] S_1 [\exists Z. (\phi_{post}*\phi_2)]$ is a valid Hoare triple.

From Lemma A.3 it follows that $[\phi_1*\phi_{pre}] S_1 [\hat{\phi}_1*\phi_{pre}]$ and $[\phi_{post}*\phi_2] S_2 [\hat{\phi}_2]$ are strong Hoare triples. Hence $[\exists Z. (\phi_{post}*\phi_2)] S_2 [\exists Z. (\phi_{post}*\phi_2)]$ is a strong Hoare triple.

Since $\hat{\phi}_1*\phi_{pre} \Leftrightarrow \exists Z. (\phi_{post}*\phi_2)$ it follows that $[\phi_1*\phi_{pre}] S_1 [\exists Z. (\phi_{post}*\phi_2)]$ is a strong Hoare triple. □