Bottom-Up Shape Analysis using $\mathcal{LISF}$

BHARGAV S. GULAVANI and SUPRATIK CHAKRABORTY, IIT Bombay
G. RAMALINGAM and ADITYA V. NORI, Microsoft Research India

In this article, we present a new shape analysis algorithm. The key distinguishing aspect of our algorithm is that it is completely compositional, bottom-up and noniterative. 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 ($\mathcal{LISF}$), which uses the iterated separating conjunct of 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 $\mathcal{LISF}$.

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

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

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Compositional analysis, hoare logic, separation logic

ACM Reference Format:
Gulavani, B. S., Chakraborty, S., Ramalingam, G., and Nori, A. V. 2011. Bottom-up shape analysis using $\mathcal{LISF}$. ACM Trans. Program. Lang. Syst. 33, 5, Article 17 (November 2011), 41 pages.
DOI = 10.1145/2039346.2039349 http://doi.acm.org/10.1145/2039346.2039349

1. INTRODUCTION

In this article 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 noniterative. It computes summaries describing the effect of a statement or procedure in a modular, compositional, noniterative 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

The work of B. S. Gulavani was supported by Microsoft Corporation and Microsoft Research India under the Microsoft Research India PhD Fellowship Award.
Authors' addresses: B. S. Gulavani and S. Chakraborty, Department of Computer Science & Engineering, IIT Bombay, Powai, Mumbai 400076, India; email: {bhargav; supratik}@cse.iitb.ac.in; G. Ramalingam and A. V. Nori, Microsoft Research India, 196/36, 2nd Main, Sadashivnagar, Bangalore 560 080, India, email: {grama, adityan}@microsoft.com.

DOI 10.1145/2039346.2039349 http://doi.acm.org/10.1145/2039346.2039349

```
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.

of a particular (abstract) state. Though challenging, bottom-up shape analysis appears worth pursuing because the compositional nature of the analysis promises much better scalability, as illustrated by the recent work of Calcagno et al. [2009]. The algorithm we present is based on ideas introduced by Calcagno et al. [2009].

Motivating Example. Consider the procedure shown in Figure 1. Given a list pointed to by parameter h, this procedure deletes the fragment of the list demarcated 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 $[\varphi]$ S $[\widehat{\varphi}]$ that summarize the procedure. We use the above notation to indicate that the Hoare triples inferred are *total*: the triple $[\varphi]$ S $[\widehat{\varphi}]$ indicates that, given an initial state satisfying φ, the execution of S terminates safely (with no memory errors) in a state satisfying $\widehat{\varphi}$.

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 describing 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 different correct Hoare triples can be produced, differing in the information captured by the postcondition $\widehat{\varphi}$. 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), 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 φ_{pre} and φ_{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: φ_{pre} and φ_{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 φ_2, we refer to the identification of φ_{pre}, φ_{post} such that $\widehat{\varphi_1} * \varphi_{pre} \Leftrightarrow \varphi_{post} * \varphi_2$ 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 this 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 φ_{post} and φ_{pre} such that $\widehat{\varphi} * \varphi_{pre} \Leftrightarrow \varphi_{post} * \varphi$. If we now inductively apply the composition rule, we can then infer a summary of the form $[\varphi * \varphi_{pre}^k]$ $\mathtt{S}^{\mathtt{k}}$ $[\varphi_{post}^k * \widehat{\varphi}]$ 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 φ_{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, that is, 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 article, 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 $\mathcal{S}$ 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 $\mathcal{S}$. 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 $\mathcal{LISF}$ (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. [2009], which derives from the earlier work in Calcagno et al. [2007]. The algorithm described by 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 $\widehat{\varphi}_1$, the state at a callsite, and φ_2, a precondition of a Hoare triple for the called procedure, Calcagno et al. compute φ_{pre} and φ_{post} such that $\widehat{\varphi}_1 * \varphi_{pre} \Rightarrow \varphi_{post} * \varphi_2$. 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 $\mathcal{LISF}$ 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] and 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. 2005a, 2005b] have proposed an interprocedural extension of the basic intraprocedural analysis of TVLA. All these algorithms are top-down and forward. Rinetzky et al. [2005a], 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. [2005a] track precise input-output relations only between finitely many cut-points. Rinetzky et al. [2005b] 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

Program Syntax

$$
\begin{array}{lcl}
\texttt{e} & ::= & \texttt{v} \mid \textbf{null} \\
\texttt{B} & ::= & \texttt{v} = \texttt{e} \mid \texttt{v} \mathrel{!=} \texttt{e} \\
\texttt{S} & ::= & \texttt{v.f} := \texttt{e} \mid \texttt{v} := \texttt{u.f} \mid \texttt{v} := \texttt{new} \mid \texttt{dispose v} \mid \texttt{S;S} \\
 & & \mid \texttt{assert(B)} \mid \texttt{v} := \texttt{e} \mid \texttt{if(B, S, S)} \mid \texttt{while(B) S}
\end{array}
$$

Separation Logic Syntax $(\sim \ \in \{=, \neq\})$

$$
\begin{array}{lcl}
e & ::= & \textbf{null} \mid v \mid \ \ldots \\
P & ::= & e \sim e \mid \textbf{false} \mid \textbf{true} \mid P \wedge P \mid \ \ldots \\
S & ::= & \textbf{emp} \mid e \mapsto (f : e) \mid \textbf{true} \mid S * S \mid \ \ldots \\
\varphi & ::= & P \wedge S \mid \exists v.\ SH
\end{array}
$$

Fig. 2. Program syntax and separation logic syntax.

to by a procedure parameter. In contrast, summaries expressed using $\mathcal{LISF}$ can capture precise input output relationships between an unbounded number of cut-points.

Jeannet et al. [2004] 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, that is, they start abstract computation from the main procedure and analyze each procedure (or reuse its already computed summary, if possible) when it is called.

Yorsh et al. [2006] 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, 2006, 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 intial configuration encoded as an automaton, the goal is to compute the final configuration after the loop exits (i.e., the postcondition). 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. Abdulla et al. [2008] 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 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 != 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 **null**. This language supports heap manipulating operations without address arithmetic.

Semantically, we use a value domain 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 (Locs $\cup$ {**null**}), and a heap h, mapping a finite set of non-null locations to their values, which are n-tuples of (primitive) values.

$$\begin{array}{lcll}
(s,h) & \models & P \wedge S & \text{iff } (s,h) \models P \wedge (s,h) \models S \\
(s,h) & \models & e_1 \sim e_2 & \text{iff } s(e_1) \sim s(e_2) \\
(s,h) & \models & \mathbf{true} & \\
(s,h) & \not\models & \mathbf{false} & \\
(s,h) & \models & P_1 \wedge P_2 & \text{iff } (s,h) \models P_1 \wedge (s,h) \models P_2 \\
(s,h) & \models & \mathbf{emp} & \text{iff } dom(h) = \{\} \\
(s,h) & \models & e_1 \mapsto (f : e_2) & \text{iff } h(s(e_1)) = (f : s(e_2)) \ \wedge dom(h) = \{s(e_1)\} \\
(s,h) & \models & S_1 * S_2 & \text{iff } \exists h_1 h_2 . h_1 \# h_2 \wedge h_1 \sqcup h_2 = h \wedge (s,h_1) \models S_1 \wedge (s,h_2) \models S_2
\end{array}$$

Fig. 3. Separation logic semantics.

Table I. Local Reasoning Rules for Primitive Statements

Mutation	$[v \mapsto (f : \_w; \ldots)]$ `v.f := e` $[v \mapsto (f : e; \ldots)]$
Deallocation	$[v \mapsto (f^1 : \_w^1, \ldots, f^n : \_w^n)]$ `dispose v` $[v \neq \mathbf{null} \wedge \mathbf{emp}]$
Allocation (modifies v)	$[v = \_x]$ `v := new` $[\exists \_w^1 \ldots \_w^n .\ v \mapsto (f^1 : \_w^1, \ldots, f^n : \_w^n)]$
Lookup (modifies v)	$[v = \_x \wedge u \mapsto (f : \_w; \ldots)]$ `v := u.f` $[v = \_w \wedge u \mapsto (f : \_w; \ldots)]$
	$[v = \_x \wedge v \mapsto (f : \_w; \ldots)]$ `v := v.f` $[v = \_w \wedge \_x \mapsto (f : \_w; \ldots)]$
Copy (modifies v)	$[v = \_x]$ `v := e` $[v = e\langle v \rightarrow \_x\rangle]$
Guard	$[v = e]$ `assert(v = e)` $[v = e]$
	$[v \neq e]$ `assert(v!= e)` $[v \neq e]$

Assertion Logic. We illustrate some of the key ideas using standard separation logic, using the syntax shown in Figure 2. The '$\cdots$' 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 $s(\mathbf{null})$ to be **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 $dom(h)$ denote the domain of heap h. **emp** denotes that the heap has no allocated cells, $dom(h) = \{\}$. 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 $*$ operator is called the separating conjunction; $s_1 * s_2$ 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 $h_1 \sqcup h_2$ 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 $[\varphi]$ `S` $[\widehat{\varphi}]$ means that when `S` is run in a state satisfying φ it terminates without any memory error (such as null dereference) in a state satisfying $\widehat{\varphi}$. Thus, we use *total correctness specifications*. Additionally, we call the specification $[\varphi]$ `S` $[\widehat{\varphi}]$ *strong* if $\widehat{\varphi}$ is the strongest postcondition of φ 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 Aux, that do not correspond to the value of any program variable. For the present discussion, we prefix all auxiliary variable names with "_". 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 $\mathbf{true} \wedge S$ and $P \wedge \mathbf{emp}$ in pre- or postconditions are represented simply as S and P, respectively. The notation $\theta : \langle v \rightarrow x\rangle$ refers to a renaming θ that replaces variable v with x, and $e\theta$ refers to the expression obtained by applying renaming θ to e. For sets A and B of variables, we write $\theta : \langle A \hookrightarrow B\rangle$ 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 φ, we use $\mathit{free}|(\varphi)$ to refer to the set of free variables in φ. 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 φ is *independent* of the set of variables A, if $A \cap \mathit{free}(\varphi) = \emptyset$. We use φ^p and φ^s to refer to the pure and spatial parts, respectively, of φ. The notation $\exists X\varphi \ast \exists Y\psi$ is used to denote $\exists X, Y\ \varphi^p \wedge \psi^p \wedge \varphi^s \ast \psi^s$, when φ and ψ 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 *mod*(`S`). For primitive statements, the definition of *mod* is given in Table I. For composite statements, *mod* is defined as follows. *mod*(`S1;S2`) and *mod*(`if(C, S1, S2)`) are both defined as *mod*(`S1`) $\cup$ *mod*(`S2`). On the other hand, *mod*(`while(C) S1`) is defined as *mod*(`S1`).

3.2. Composing Hoare Triples

Given two summaries $[\varphi_1]$ `S1` $[\widehat{\varphi_1}]$ and $[\varphi_2]$ `S2` $[\widehat{\varphi_2}]$, we wish to compute a summary for the composite statement `S1;S2`. If we can compute formulas φ_{pre} and φ_{post} that are independent of *mod*(`S1`) and *mod*(`S2`), respectively, such that $\widehat{\varphi_1} \ast \varphi_{pre} \Leftrightarrow \varphi_{post} \ast \varphi_2$, then by application of frame rule we can infer the summary $[\varphi_1 \ast \varphi_{pre}]$ `S1;S2` $[\varphi_{post} \ast \widehat{\varphi_2}]$. We can compose the two given summaries even under the slightly modified condition $\widehat{\varphi_1} \ast \varphi_{pre} \Leftrightarrow \exists Z.\ (\varphi_{post} \ast \varphi_2)$, if $Z \subseteq \mathsf{Aux}$. The summary inferred in this case is $[\varphi_1 \ast \varphi_{pre}]$ `S1;S2` $[\exists Z.\ (\varphi_{post} \ast \widehat{\varphi_2})]$.

Given $\widehat{\varphi_1}$ and φ_2, we refer to the determination of φ_{pre}, φ_{post} and a set Z of variables such that $\widehat{\varphi_1} \ast \varphi_{pre} \Leftrightarrow \exists Z.\ (\varphi_{post} \ast \varphi_2)$ 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 $\widehat{\varphi_1} \ast \varphi_{pre} \Rightarrow \varphi_{post} \ast \varphi_2$, 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 this composition rule is sound even if we use bi-abduction, bi-abduction may not yield good postconditions. Specifically, if we disallow the deallocation operation, it can be shown that the composition of strong Hoare triples using strong bi-abudction 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, [**true**] `v` := **null** [v = **null**] and [**true**] `v` := **null** [v = **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 $v \mapsto (\mathit{next} : w)$. Let us compose two summaries, $[v = \_a]$ `v := new` $[\exists \_b.\ v \mapsto \_b]$ and $[v = \_c \wedge \_c \mapsto \_d]$ `v := v.next` $[v = \_d \wedge \_c \mapsto \_d]$. 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 $(\exists \_b.\ v \mapsto \_b) \ast \mathbf{emp} \Leftrightarrow \exists \_c, \_d.\ \mathbf{emp} \ast (v = \_c \wedge \_c \mapsto \_d)$ we can compose the two summaries and deduce $[v = \_a]$ `v := new; v := v.next` $[\exists \_c, \_d.\ v = \_d \wedge \_c \mapsto \_d]$. As an aside, note that the program fragment `v:=new; v:=v.next` introduces a memory leak.

$$\textsc{Compose}\quad \frac{\begin{array}{c}[\varphi_1]\ \texttt{S1}\ [\widehat{\varphi}_1]\\ [\varphi_2]\ \texttt{S2}\ [\widehat{\varphi}_2]\\ \widehat{\varphi}_1 * \varphi_{pre} \Leftrightarrow \exists Z.\ (\varphi_{post} * \varphi_2)\end{array}}{[\varphi_1 * \varphi_{pre}]\ \texttt{S1;S2}\ [\exists Z.\ (\varphi_{post} * \widehat{\varphi}_2)]}\quad \begin{array}{l} free(\varphi_{pre}) \cap mod(\texttt{S1}) = \emptyset\\ free(\varphi_{post}) \cap mod(\texttt{S2}) = \emptyset\\ Z \subseteq \mathsf{Aux}\end{array}$$

$$\textsc{Branch}\quad \frac{\begin{array}{c}[\varphi \wedge B]\ \texttt{S1}\ [\widehat{\varphi}]\\ [\varphi \wedge !B]\ \texttt{S2}\ [\widehat{\varphi}]\end{array}}{[\varphi]\ \texttt{if(B, S1, S2)}\ [\widehat{\varphi}]}$$

$$\textsc{Exit}\quad \frac{[\varphi]\ \texttt{assert(!B)}\ [\widehat{\varphi}]}{[\varphi]\ \texttt{while(B) S}\ [\widehat{\varphi}]} \qquad \textsc{While}\quad \frac{[\varphi]\ (\texttt{assert(B);S})^{+}\ [\psi'],\quad [\psi']\ \texttt{assert(!B)}\ [\widehat{\varphi}]}{[\varphi]\ \texttt{while(B) S}\ [\widehat{\varphi}]}$$

$$\textsc{Then}\quad \frac{[\varphi]\ \texttt{assert(B);S1}\ [\widehat{\varphi}]}{[\varphi]\ \texttt{if(B, S1, S2)}\ [\widehat{\varphi}]} \qquad \textsc{Else}\quad \frac{[\varphi]\ \texttt{assert(!B);S2}\ [\widehat{\varphi}]}{[\varphi]\ \texttt{if(B, S1, S2)}\ [\widehat{\varphi}]}$$

Fig. 4. Inference rules for sequential composition, loops, and branch statements.

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 COMPOSE rule captures the above idea of using strong bi-abduction for the sequential composition of statements. The rules WHILE, THEN and ELSE use the COMPOSE rule to derive the fact in their antecedent.

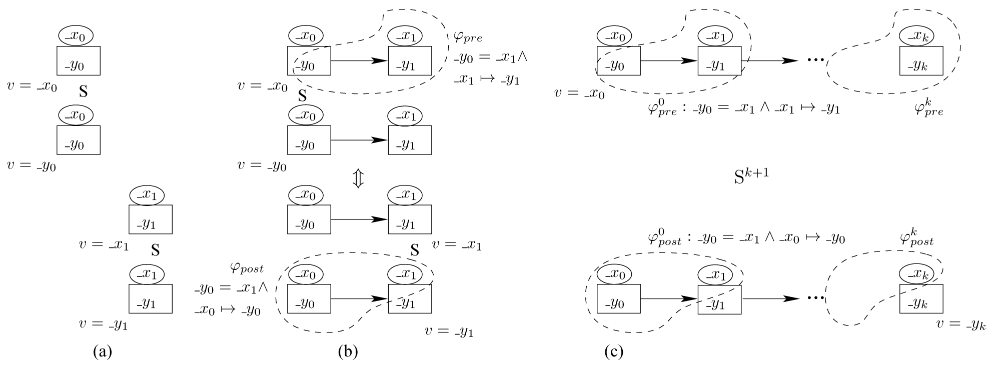
The rules EXIT and WHILE are straightforward rules that decompose analysis of loops into two cases. Rule EXIT handles the case where the loop executes zero times, while rule WHILE applies when the loop executes one or more times. Rule WHILE leaves the bulk of the work to the computation of $[\varphi]$ $\texttt{S}^+$ $[\widehat{\varphi}]$. The notation $[\varphi]$ $\texttt{S}^+$ $[\widehat{\varphi}]$ does not represent a Hoare triple in the standard sense, since $\texttt{S}^+$ is not a statement in our programming language. However, $[\varphi]$ $\texttt{S}^+$ $[\widehat{\varphi}]$ is the key idiom we will use in the remainder of this article. Hence, we overload the notation of Hoare triples, and also call $[\varphi]$ $\texttt{S}^+$ $[\widehat{\varphi}]$ a Hoare triple. The notation $[\varphi]$ $\texttt{S}^+$ $[\widehat{\varphi}]$ means that for every initial state satisfying φ, there exists a $k \geq 1$ such that the state resulting after k executions of $\texttt{S}$ 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 $[\varphi]$ $\texttt{S}^+$ $[\widehat{\varphi}]$.

4. LOGIC OF ITERATED SEPARATION FORMULAE ($\mathcal{LISF}$)

Let $\texttt{S}_\text{L}$ denote the following loop in our programming language: `while (v!=`**null**`) v := v.next`. Let $\odot_{i=0}^{k}\ \psi^i$ informally denote the iterated separating conjunction $\psi^0 * \cdots * \psi^k$ [Reynolds 2002]. We would like to infer the following summary for $\texttt{S}_\text{L}$: $[v = \_x_0 \wedge \_x_k = \textbf{null} \wedge \odot_{i=0}^{k-1} \_x_i \mapsto \_x_{i+1}]$ $\texttt{S}_\text{L}$ $[v = \_x_k \wedge \_x_k = \textbf{null} \wedge \odot_{i=0}^{k-1} \_x_i \mapsto \_x_{i+1}]$. 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 this one.

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

$$[\varphi^0 * \big(\odot_{i=0}^{k-1}\ \varphi^i_{pre}\big)]\texttt{S}^{k+1}\big[\big(\odot_{i=0}^{k-1}\ \varphi^i_{post}\big) * \widehat{\varphi}^k\big] \tag{4.1}$$



φ_{pre}
$\_y_0 = \_x_1 \wedge \_x_1 \mapsto \_y_1$
φ_{post}
$\_y_0 = \_x_1 \wedge \_x_0 \mapsto \_y_0$
$\varphi^0_{pre} : \_y_0 = \_x_1 \wedge \_x_1 \mapsto \_y_1$
φ^k_{pre}
$\varphi^0_{post} : \_y_0 = \_x_1 \wedge \_x_0 \mapsto \_y_0$
φ^k_{post}
$v = \_x_0$
$v = \_y_0$
$v = \_x_1$
$v = \_y_1$
$v = \_y_k$
S
S^{k+1}
(a)
(b)
(c)

Fig. 5. (a) Given summaries, (b) application of COMPOSE, and (c) application of acceleration. Each box represents a heap cell, its contents represents the value of the next field. A circled variable above a box denotes the name of the cell.

$$
\begin{aligned}
ae &::= arr \mid ae[\cdot] \mid ae[\cdot+1] \mid ae[c] \mid ae[\$c]\\
e &::= \ldots \mid ae[\cdot] \mid ae[\cdot+1] \mid ae[c] \mid ae[\$c]\\
P &::= \ldots \mid \mathsf{RP}(P,l,u)\\
S &::= \ldots \mid \mathsf{RS}(S,l,u)\\
SH &::= P \wedge S \mid \exists v\ SH \mid \exists arr\ SH
\end{aligned}
$$

Fig. 6. $\mathcal{LISF}$ assertion syntax.

We call the inference of the Hoare triple in equation (4.1) as *acceleration* of $[\varphi]$ S $[\widehat{\varphi}]$. The following example illustrates acceleration of Hoare triples.

Example 2. Let S be the sequence of statements assert(v! = **null**); v := v.next. Suppose we wish to compose the two summaries $[v = \_x_0 \wedge \_x_0 \mapsto \_y_0]$ S $[v = \_y_0 \wedge \_x_0 \mapsto \_y_0]$ and $[v = \_x_1 \wedge \_x_1 \mapsto \_y_1]$ S $[v = \_y_1 \wedge \_x_1 \mapsto \_y_1]$, which are identical, except for renaming of auxiliary variables. Let φ_{pre} denote $\_x_1 = \_y_0 \wedge \_x_1 \mapsto \_y_1$ and φ_{post} denote $\_x_1 = \_y_0 \wedge \_x_0 \mapsto \_y_0$. Applying the COMPOSE rule results in the following summary: $[(v = \_x_0 \wedge \_x_0 \mapsto \_y_0) * (\_x_1 = \_y_0 \wedge \_x_1 \mapsto \_y_1)]$ S; S $[(\_x_1 = \_y_0 \wedge \_x_0 \mapsto \_y_0) * (v = \_y_1 \wedge \_x_1 \mapsto \_y_1)]$. This is pictorially depicted in Figures 5(a) and 5(b). Iterative application of COMPOSE, or acceleration, yields the summary: $[v = \_x_0 \wedge \_x_0 \mapsto \_y_0 * \odot_{i=0}^{k-1}(\_x_{i+1} = \_y_i \wedge \_x_{i+1} \mapsto \_y_{i+1})]$ $\mathtt{S}^{k+1}$ $[\ \odot_{i=0}^{k-1}(\_x_{i+1} = \_y_i \wedge \_x_i \mapsto \_y_i) * (v = \_y_k \wedge \_x_k \mapsto \_y_k)]$. This is pictorially depicted in Figure 5(c).

4.1. $\mathcal{LISF}$ Syntax and Informal Semantics

We now introduce an extension of separation logic, called Logic of Iterated Separation Formulae (or $\mathcal{LISF}$), that allows us to formally express the restricted form of iterated separating conjunction previously alluded. The syntax of $\mathcal{LISF}$ 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 $(v = \_x_0) \wedge (\_x_k = \mathbf{null}) \wedge (\odot_{i=0}^{k-1} \_x_i \mapsto \_x_{i+1})$ to describe an acyclic singly linked list pointed to by v. Instead, we can use the $\mathcal{LISF}$ formula $\varphi \equiv (v = \mathbf{A}[0]) \wedge (\mathbf{A}[\$0] = \mathbf{null}) \wedge \mathsf{RS}(\mathbf{A}[\cdot] \mapsto \mathbf{A}[\cdot+1], 0, 0)$, where $\mathbf{A}$ is a new type of logical variable and RS is a new predicate.

Variables like $\mathbf{A}$ in the formula φ represent a new type of logical variables, called *array variables*, that may be referenced in $\mathcal{LISF}$ formulae. Intuitively, an array variable represents a sequence of locations corresponding to the "nodes" of a recursive data structure like a linked list. A $\mathcal{LISF}$ formula may specify properties of the ith node in such a data structure, or specify a relation between the ith 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 $\mathcal{LISF}$ also allows references to multidimensional 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 boldface upper-case letters.

The semantics of $\mathcal{LISF}$ uses a mapping from each array variable to a *sequence* of values $(v_0, \ldots, v_k)$. For unidimensional arrays, the values v_i represent locations in the heap, whereas for multidimensional arrays, the v_i's may themselves be sequences of locations or sequences of sequences of locations, and so on. Expressions are extended to allow indexed array references, also called *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) $arr[c]$, (ii) $arr[\$c]$, (iii) $arr[\cdot]$, or (iv) $arr[\cdot + 1]$, where c is a nonnegative integer constant, and arr is either an array name or an array expression. Array expressions with *fixed indices* include array references of the form $arr[c]$ or $arr[\$c]$. These refer to the element at an offset c from the beginning or end, respectively, of the sequence represented by arr. For example, if $\mathbf{A}$ is mapped to the sequence $(v_0, \ldots, v_k)$, then the array expressions $\mathbf{A}[0]$ and $\mathbf{A}[\$0]$ evaluate to v_0 and v_k respectively in $\mathcal{LISF}$ semantics. The semantics of array expressions with *iterated indices*, which include references of the form $arr[\cdot]$ and $arr[\cdot + 1]$, will be explained later.

In addition to array variables, $\mathcal{LISF}$ extends pure and spatial formulae with a pair of new predicates, called RP and 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 iterated index, such as $arr[\cdot]$ or $arr[\cdot + 1]$, then $\mathsf{RS}(S, l, u)$ corresponds to our informal notation $\odot_{i=l}^{k-1-u} S$. Note, however, that the index variable i and bound k are not explicitly represented in $\mathsf{RS}(S, l, u)$. Instead, the values of i and k are provided by the evaluation context. The "dot" in $arr[\cdot]$ or $arr[\cdot + 1]$ intuitively refers to the implicit index variable i. Thus, $arr[\cdot]$ refers to the element at offset i, while $arr[\cdot + 1]$ refers to the element at offset $i+1$. To see how the RS predicate is used, consider the formula $\mathsf{RS}(\mathbf{A}[\cdot] \mapsto \mathbf{A}[\cdot + 1], 0, 0)$, where $\mathbf{A}$ is mapped to a sequence of length $k+1$. This formula asserts that for all $i \in [0, k-1]$, the ith element of $\mathbf{A}$ is the location of a heap cell whose *next* field has the same value as the $i+1$st element of $\mathbf{A}$. In addition, the predicate also asserts that the heap cells represented by elements $\mathbf{A}[0]$ through $\mathbf{A}[k-1]$ are distinct. The usage and intuitive interpretation of RP is similar to that of RS, with the exception that RP is used with a pure subformula P (as in $\mathsf{RP}(P, l, u)$) instead of the spatial subformula S in $\mathsf{RS}(S, l, u)$. For notational convenience, we will henceforth denote $\mathsf{RP}(P, l, u)$ and $\mathsf{RS}(S, l, u)$ simply by $\mathsf{RP}(P)$ and $\mathsf{RS}(S)$, respectively, when both l and u are 0.

While the RP and 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 RP and RS. Specifically, the iteration bounds in $\mathsf{RS}(S, l, u)$ and $\mathsf{RP}(P, l, u)$ are provided by the lengths of sequences mapped to array variables with iterated indices in the sub-formulae S and 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 $\mathcal{LISF}$ (with the shorthand RS(S) for RS(S, 0, 0)) as

$$\mathsf{RS}\left(\begin{array}{l} \mathbf{x}[\cdot] \mapsto (\mathit{nlist} : \mathbf{A}[\cdot][0], \mathit{next} : \mathbf{x}[\cdot+1]) \\ \wedge\ \ (\mathbf{A}[\cdot][\$0] = \mathbf{null}) \\ \wedge\ \ \mathsf{RS}\left(\mathbf{A}[\cdot][\cdot] \mapsto (\mathbf{A}[\cdot][\cdot+1])\right) \end{array}\right) \wedge (\mathbf{x}[\$0] = (\mathit{nlist} : \mathbf{A}[\$0][0], \mathit{next} : \mathbf{null}),$$

where $\mathbf{x}$ is a unidimensional array representing elements (with *nlist* and next fields) of the outer linked list, and $\mathbf{A}$ is a two-dimensional array representing elements (with a *next* field) of the nested linked lists. The semantics of this formula will become clear once we discuss the formal semantics of $\mathcal{LISF}$ 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 to keep these bounds implicit. Another way in which the usage of RP and 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 RP and 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. $\mathcal{LISF}$ Semantics

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

If e is a variable that is not an array, $\mathcal{E}$ simply looks up the stack and returns $s(e)$ as the value of e. If e is the constant **null**, $\mathcal{E}$ returns **null**. However, if e is an array expression, $\mathcal{E}$ uses the list L' of integers and the mapping $\mathcal{V}$ 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, $[\cdot]$ and $[\cdot + 1]$, 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 $\mathcal{E}(e, L', s, \mathcal{V})$, shown in Figure 7, where the function NumIterInd(e) gives the number of iterated indices in e. Formally, NumIterInd(e) is defined as follows: If *array_var* denotes an array variable, *ae* denotes an array expression and v denotes a non-array variable, then NumIterInd(*array_var*) = 0, NumIterInd($ae[\cdot]$) = NumIterInd($ae[\cdot + 1]$) = NumIterInd(ae) + 1, NumIterInd($ae[c]$) = NumIterInd($ae[\$c]$) = NumIterInd($ae$), and NumIterInd($v$) = NumIterInd(**null**) = 0. If e is an array expression of the form *array_var* followed by k (fixed or iterated) indices, then $\mathcal{V}$ must map *array_var* to a k-dimensional sequence of locations in order to avoid indexing errors during evaluation of e and to ensure that $\mathcal{E}(e, L', s, \mathcal{V})$ evaluates to a unique location. This is formalized in the second precondition of $\mathcal{E}(e, L', s, \mathcal{V})$.

input:	e	expression
	L'	list of integers
	s	stack
	$\mathcal{V}$	mapping of array variables to uni- or multi-dimensional sequence(s) of locations
output:	location	

requires:

(1) Number of elements in $L' \geq \mathsf{NumIterInd}(e))$
(2) If e is an array expression of the form $array\_var$ followed by k (fixed or iterated) indices then the dimension of $\mathcal{V}(array\_var)$ equals k

```
$\mathcal{E}(e, L', s, \mathcal{V}) =$
$\mathbf{let}\ L = \mathsf{suffix}(L', \mathsf{NumIterInd}(e))\ \mathbf{in}$
$\mathbf{match}\ e\ \mathbf{with}$
$\mid \mathbf{null} \rightarrow \mathbf{null}$
$\mid v \rightarrow s(v)$
$\mid ae \rightarrow \mathcal{E}_a(ae, L, \mathcal{V})$
```

input:	$aexpr$	array expression
	L	list of integers
	$\mathcal{V}$	mapping of array variables to uni- or multi-dimensional sequence(s) of locations
output:	unique location, or uni-/multi-dimenional sequence of locations	

requires:

(1) Number of elements in $L = \mathsf{NumIterInd}(aexpr)$
(2) If $aexpr$ is of the form $array\_var$ followed by k (fixed or iterated) indices then the dimension of $\mathcal{V}(array\_var)$ is at least k

```
$\mathcal{E}_a(aexpr, L, \mathcal{V}) = \mathbf{match}\ aexpr\ \mathbf{with}$
$\mid array\_var \rightarrow \mathcal{V}(array\_var)$
$\mid ae[\cdot] \rightarrow \mathcal{E}_a(ae, tl(L), \mathcal{V})[hd(L)]$
$\mid ae[\cdot + 1] \rightarrow \mathcal{E}_a(ae, tl(L), \mathcal{V})[1 + hd(L)]$
$\mid ae[c] \rightarrow \mathcal{E}_a(ae, L, \mathcal{V})[c]$
$\mid ae[\$c] \rightarrow \mathbf{let}\ a = \mathcal{E}_a(ae, L, \mathcal{V})\ \mathbf{in}$
        $a[length(a) - 1 - c]$
```

Fig. 7. Semantics of expressions, $\mathcal{E}$.

In general, a list L' satisfying the first precondition of $\mathcal{E}(e, L', s, V)$ may contain more integers than $\mathsf{NumIterInd}(e)$. Therefore, we use the function suffix to extract a suffix of L' of the same length as $\mathsf{NumIterInd}(e)$. 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 $\mathcal{E}_a$ 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 $\mathcal{V}$ of array variables to sequences of locations. The instantiation of iterated indices in *aexpr* with integers from L is done recursively. Specifically, each recursive call instantiates the current rightmost un-instantiated iterated index of *aexpr* with the integer at the head of L, and passes the rest of L, that is, 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 $\mathcal{V}(array\_var)$ is allowed to be greater than the number of indices (fixed or iterated) following *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 $\mathsf{NumInterInd}(aexpr)$ 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 *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 *aexpr* be of the form *array_var* followed by k' (fixed or iterated) indices. Let the dimension of $\mathcal{V}(array\_var)$ be k. The second precondition of $\mathcal{E}_a(aexpr, L, \mathcal{V})$ 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 $hd(L)$ and $tl(L)$ 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 $length(a)$, used in the definition of $\mathcal{E}_a$, returns the number of elements in a.

$$
\begin{array}{lcll}
m & \models & P \wedge S & \text{iff } m \models P \wedge m \models S \\
m & \models & e_1 \sim e_2 & \text{iff } \mathcal{E}(e_1, L, s, \mathcal{V}) \sim \mathcal{E}(e_2, L, s, \mathcal{V}) \\
m & \models & \mathbf{true} & \\
m & \not\models & \mathbf{false} & \\
m & \models & \mathsf{RP}(P, l, u) & \text{iff } \exists k\ k+1 = len(\mathcal{V}, L, P) \wedge \forall l \leq i \leq k-1-u.(s, h, \mathcal{V}, i :: L) \models P \\
m & \models & P_1 \wedge P_2 & \text{iff } m \models P_1 \wedge m \models P_2 \\
m & \models & \mathbf{emp} & \text{iff } dom(h) = \{\} \\
m & \models & e_1 \mapsto (f : e_2) & \text{iff } h(\mathcal{E}(e_1, L, s, \mathcal{V})) = (f : \mathcal{E}(e_2, L, s, \mathcal{V}))\ \wedge dom(h) = \{\mathcal{E}(e_1, L, s, \mathcal{V})\} \\
m & \models & \mathsf{RS}(S, l, u) & \text{iff } \exists k, u', h_l, \ldots, h_{u'}\ k+1 = len(\mathcal{V}, L, S) \wedge u' = k-1-u \wedge h = \bigsqcup_{i=l}^{u'} h_i\ \wedge \\
 & & & \quad \forall l \leq i, j \leq u'.\ i \neq j \Rightarrow h_i \# h_j \wedge \forall l \leq i \leq u'.\ (s, h_i, \mathcal{V}, i :: L) \models S \\
m & \models & S_1 * S_2 & \text{iff } \exists h_1, h_2\ h_1 \# h_2 \wedge h_1 \sqcup h_2 = h \wedge (s, h_1, \mathcal{V}, L) \models S_1 \wedge (s, h_2, \mathcal{V}, L) \models S_2 \\
m & \models & \exists v\ P \wedge S & \text{iff } \exists n \in \mathsf{Locs} \cup \{\mathbf{null}\}\ ([s|v : n], h, \mathcal{V}, L) \models (P \wedge S) \\
m & \models & \exists arr\ P \wedge S & \text{iff } \exists k \in \mathbb{N}, a \in \mathbb{N}^k \rightarrow (\mathsf{Locs} \cup \{\mathbf{null}\})\ (s, h, [\mathcal{V}|arr : a], L) \models (P \wedge S)
\end{array}
$$

Fig. 8. Semantics of $\mathcal{LISF}$, m is $(s, h, \mathcal{V}, L)$, and len is as explained in text.

We now define a class of well-formed $\mathcal{LISF}$ formulae or (*wff*). The semantics is non-trivially defined only for well-formed formulae. A $\mathcal{LISF}$ formula that is not well-formed does not have a model. For notational convenience, we overload the function 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 NumIterInd is defined over predicates as follows. NumIterInd$(e_1 \sim e_2) = \max($ NumIterInd(e_1), NumIterInd$(e_2))$, NumIterInd$(P_1 \wedge P_2)$ = NumIterInd(P_1), NumIterInd(RP$(P, _, _)$) = NumIterInd$(P) - 1$, NumIterInd$(e \mapsto (f_i : l_i))$ = NumIterInd(e), NumIterInd$(S_1 * S_2)$ = NumIterInd(S_1), NumIterInd(RS$(S, _, _)$) = NumIterInd$(S) - 1$. An $\mathcal{LISF}$ formula $P \wedge S$ is then said to be well-formed iff (i) NumIterInd(P) = NumIterInd$(S) = 0$, (ii) for every sub-formula $P_1 \wedge P_2$ of P, we have NumIterInd(P_1) = NumIterInd(P_2), (iii) for every sub-formula $S_1 * S_2$ of S, we have NumIterInd(S_1) = NumIterInd(S_2), and (iv) for every sub-formula $e_1 \mapsto (f : e_2)$ of S, we have NumIterInd$(e_1) \geq$ NumIterInd(e_2).

Structures modeling well-formed $\mathcal{LISF}$ formulae are tuples $(s, h, \mathcal{V})$, where s is a stack, h is a heap, and $\mathcal{V}$ is a mapping of array variables to uni- or multidimensional 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 φ. The list of integers facilitates evaluation of array expressions by the function $\mathcal{E}$ described above. The formal definition of $(s, h, \mathcal{V}, L) \models \varphi$ 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 $[\mathcal{V}|arr : a]$ denotes the mapping $\mathcal{V}'$ defined by $\mathcal{V}'(arr) = a$, and $\mathcal{V}'(\mathbf{x}) = \mathcal{V}(\mathbf{x})$ for all array variables $\mathbf{x}$ different from arr. We say that $(s, h, \mathcal{V})$ is a model of φ iff $(s, h, \mathcal{V}, []) \models \varphi$.

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

(1) Let *ae* be a *maximally indexed* array expression in φ, that is, an array expression that is not a subexpression of another array expression in φ. Let the underlying array variable in *ae* be *array_var*, and let *ae* be of the form *array_var* indexed by a sequence of k (iterated and fixed) indices. Then, the dimension of $\mathcal{V}(array\_var)$ equals k.

(2) The lengths of sequences accessed by array expressions in φ are such that no out-of-bounds exception occurs when function $\mathcal{E}$ 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 φ, every sequence to which e evaluates to during evaluation of φ is of length at least $c + 1$.
   (b) Let ψ be a subformula nested within $n(\geq 1)$ RP (or RS) predicates in φ. In general, ψ may refer to one or more array expressions. For every pair of array expressions e_1 and e_2 in ψ that have at least n iterated indices, the sequences accessed by the nth iterated index of e_1 and e_2 always have the same length.

(3) All sequences mapped to array variables by $\mathcal{V}$ have non-zero lengths.

Let φ be a well-formed $\mathcal{LISF}$ formula, (s, h, V) be a structure in wfs_φ, and L be a list of r integers, where $r \geq \mathsf{NumIterInd}(ae)$ for all array expressions *ae* in φ. From the semantics of $(s, h, \mathcal{V}, L) \models \varphi$ given in Figure 8, we find that for all constructs borrowed from standard separation logic, the semantics remains unchanged. The semantics of predicates RS and RP, which are novel to $\mathcal{LISF}$, however, deserve some explanation. Consider a $\mathsf{RP}(P, l, u)$ (or $\mathsf{RS}(S, l, u)$) predicate nested inside $n - 1$ other RP(or RS) predicates. The length of the sequence accessed by the nth 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 RP(or RS) predicates, function $len(\mathcal{V}, L, P)$ (or $len(\mathcal{V}, L, S)$) determines the length, say $k + 1$, of the sequence accessed by the nth iterated index of an array expression in P (or S). The semantics of $\mathsf{RP}(P, l, u)$ then requires that P holds for each array index i ranging from l to $k - 1 - u$. Similarly, the semantics of $\mathsf{RS}(S, l, u)$ 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 pairwise disjoint. Note also that the definition of wff ensures that whenever $\mathcal{E}(ae, L, s, \mathcal{V})$ 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 `traverse` containing the loop $\mathtt{S_L}$: `while(v! =` **null**`) v := v.next` , that traverses a singly linked list. Let each element of the list have two fields named `Next` and `D`. A summary in $\mathcal{LISF}$ is $[v = \mathbf{x}[0] \wedge \mathsf{RS}(\mathbf{x}[\cdot] \mapsto (\mathtt{Next} : \mathbf{x}[\cdot+1];\ \mathtt{D} : \mathbf{Y}[\cdot]) \wedge \mathbf{x}[\$0] = \mathbf{null}]$ `traverse(v)` $[v = \mathbf{x}[\$0] \wedge \mathsf{RS}(\mathbf{x}[\cdot] \mapsto (\mathtt{Next} : \mathbf{x}[\cdot + 1];\ \mathtt{D} : \mathbf{Y}[\cdot]) \wedge \mathbf{x}[\$0] = \mathbf{null}]$. This summary states that `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], and Calcagno et al. [2007, 2009] would generate the summary [$\mathsf{list}(v, \texttt{next})$] `traverse(v)` [$\mathsf{list}(v, \texttt{next})$], using the recursive predicate $\mathsf{list}(v, \texttt{next})$. This summary does not indicate whether the input list or the contents of any of its elements are modified.

Consider the composite statement `traverse(v); check(v)`, where the procedure `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 list recursive predicate. Let $\mathsf{clist}(v, \texttt{next}, h)$ be the recursive predicate that captures the desired precondition. The two statements cannot be composed unless we have a summary for `traverse` that describes the data structure using the clist predicate. This is because the postcondition of [$\mathsf{list}(v, \texttt{next})$] `traverse(v)` [$\mathsf{list}(v, \texttt{next})$] does not indicate whether the content of any element of the list is modified by `traverse`. 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], and Calcagno et al. [2007, 2009].

In $\mathcal{LISF}$, the precondition for `check` can be expressed as $v = \mathbf{x}[0] \wedge \mathsf{RS}(\mathbf{x}[\cdot] \mapsto (\texttt{Next} : \mathbf{x}[\cdot + 1];\ \texttt{D} : h) \wedge \mathbf{x}[\$0] = \mathbf{null}$. The summaries for `traverse` and `check` can indeed be composed using strong bi-abduction. For this composition, both the formulas φ_{pre} and φ_{post} can be set to $\mathsf{RP}(\mathbf{y}[\cdot] = h)$. Thus, we can use the $\mathcal{LISF}$ 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 $\mathcal{LISF}$, 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 $\mathcal{LISF}$ 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 $\mathcal{LISF}$ are quantifier free, (ii) Hoare triples are always expressed as $[\varphi]$ `S` $[\exists X.\, \widehat{\varphi}]$, (iii) $free(\varphi) = V \cup W$ and $free(\widehat{\varphi}) = V \cup W \cup X$, 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 φ and in $\exists X.\, \widehat{\varphi}$.

5.1. Inference Rule Induct

Let $[\varphi]$ `S` $[\exists X.\, \widehat{\varphi}]$ be a Hoare triple. We wish to compute a strong summary for $\texttt{S}^+$. In Figure 5 and Example 2, we have presented the intuition of acceleration that computes summaries of the form $[\varphi]$ $\texttt{S}^+$ $[\widehat{\varphi}]$ 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 φ^i (respectively, $\widehat{\varphi}^i$) to denote φ (respectively, $\widehat{\varphi}$) with every free auxiliary variable $w \in W$

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

```
INDUCT
Given
1.  $[\varphi]$ S $[\exists X.\ \widehat{\varphi}]$
2.  $\widehat{\varphi}^0$ : $\widehat{\varphi}$ with every $w \in W$ replaced by $w_0$
3.  $\varphi^1$ : $\varphi$ with every $w \in W$ replaced by $w_1$
4.  $free(\varphi^0_{pre}) \cap mod(\mathsf{S}) = \emptyset$
5.  $free(\varphi^0_{post}) \cap mod(\mathsf{S}) = \emptyset$
6.  $(\exists X.\ \widehat{\varphi}^0) * \varphi^0_{pre} \Leftrightarrow \varphi^0_{post} * \varphi^1$
7.  $\alpha : \langle x \rightarrow \mathbf{X}[0]\rangle$, for each $x$ in $W$
8.  $\beta : \langle x \rightarrow \mathbf{X}[\$0]\rangle$, for each $x$ in $W$
9.  Function Iter as explained in following text
Infer
    $[\varphi\alpha * \mathsf{Iter}(\varphi^0_{pre})]$  $\mathsf{S}^+$  $[\exists X.\ \mathsf{Iter}(\varphi^0_{post}) * \widehat{\varphi}\beta]$
```

```
INDUCTQ
Given
1.  $[\varphi]$ S $[\exists X.\ \widehat{\varphi}]$
2.  $\widehat{\varphi}^0$ : $\widehat{\varphi}$ with every $w \in W$ and $x \in X$
    replaced by $w_0$ and $x_1$, resp.
3.  $\varphi^1$ : $\varphi$ with every $w \in W$ replaced by $w_1$
4.  $free(\varphi^0_{pre}) \cap mod(\mathsf{S}) = \emptyset$
5.  $free(\varphi^0_{post}) \cap mod(\mathsf{S}) = \emptyset$
6.  $(\exists X_1.\ \widehat{\varphi}^0) * \varphi^0_{pre} \Leftrightarrow \exists Z_1.\ (\varphi^0_{post} * \varphi^1)$
7.  $Z_1 \subseteq W_1 \cup X_1 \subseteq \mathsf{Aux}$ and $|Z_1| = r$
8.  $free(\varphi^0_{pre}) \cap Z_0 = \emptyset$
9.  $\alpha : \langle x \rightarrow \mathbf{X}[0]\rangle$, for each $x$ in $W \setminus Z$
10. $\beta$, Iter, same as described in INDUCT
Infer
                 $[\varphi\alpha * \mathsf{Iter}(\varphi^0_{pre})]$
                              $\mathsf{S}^+$
    $[\exists X, \mathbf{Z}^1, \ldots, \mathbf{Z}^r.\ \mathsf{Iter}(\varphi^0_{post}) * \widehat{\varphi}\beta]$
```

Fig. 9. Inference rule for acceleration INDUCT and INDUCTQ.

```
Iter($\psi$)
1: $\psi_{ren} \leftarrow$ warp($\psi$)
2: return RP($\psi^p_{ren}$) $\wedge$ RS($\psi^p_{ren}$)

warp($\psi$)
1: Replace every indexed variable $x_0 \in W$ (resp.
   $x_1 \in W$) by $\mathbf{X}[\cdot]$ (resp. $\mathbf{X}[\cdot + 1]$)
2: if $\psi^p$ and $\psi^s$ do not have any newly introduced
   array variables in common then
3:     return $\psi^p \wedge$ pass2($\psi^s$)
4: else
5:     return $\psi$
```

```
pass2($\psi$)
match $\psi^s$ with
 | emp $\rightarrow$ true $\wedge$ emp
 | $e_1 \mapsto e_2 \rightarrow e_1 \neq$ null $\wedge\ e_1 \mapsto e_2$
 | $s_1 * s_2 \rightarrow$ pass2($s_1$) $*$ pass2($s_2$)
 | RS($s, l, u$) $\rightarrow$ let $\varphi \leftarrow$ pass2($s$) in
                RP($\varphi^p, l, u$) $\wedge$ RS($\varphi^s, l, u$)
```

Fig. 10. Definition of Iter(ψ).

replaced by an indexed variable w_i. Let φ^0_{pre}, φ^0_{post} be formulas such that $free(\varphi^0_{pre})$ and $free(\varphi^0_{post})$ are disjoint from $mod(\mathsf{S})$ and $(\exists X.\ \widehat{\varphi}^0) * \varphi^0_{pre} \Leftrightarrow \varphi^0_{post} * \varphi^1$. Note that the premises 4, 5, and 6 of INDUCT imply that $free(\varphi^i_{pre})$ and $free(\varphi^i_{post})$ are disjoint from $mod(\mathsf{S})$, and that $(\exists X.\ \widehat{\varphi}^i) * \varphi^i_{pre} \Leftrightarrow \varphi^i_{post} * \varphi^{i+1}$ for any i. Given these conditions, the COMPOSE rule can be iteratively applied to obtain an accelerated summary similar to that in (4.1).

We use α, β, and Iter to express φ^0, $\widehat{\varphi}^k$ and the iterated separating conjunction of accelerated summary (4.1) in $\mathcal{LISF}$. The renaming α replaces every variable $x \in W$ in φ by $\mathbf{x}[0]$. Similarly, β replaces every $x \in W$ in $\widehat{\varphi}$ by $\mathbf{x}[\$0]$.

The function Iter in premise 9 takes an $\mathcal{LISF}$ formula ψ, computes an intermediate formula ψ_{ren}, and returns $\mathsf{RP}(\psi^p_{ren}) \wedge \mathsf{RS}(\psi^s_{ren})$ as defined in Figure 10. The formula ψ_{ren} is computed by applying a function called warp to ψ. warp makes at most two passes over the syntax tree of ψ in a bottom-up manner. In the first pass, it renames every indexed auxiliary variable x_0 (respectively, x_1) by a fresh array with iterated index $\mathbf{x}[\cdot]$ (respectively, $\mathbf{x}[\cdot + 1]$). If ψ^p_{ren} and ψ^s_{ren} do not have any common array variable, it performs a second pass (formalized in algorithm pass2, Figure 10) in which every sub-formula $e_1 \mapsto e_2$ in ψ^s_{ren} is replaced by $e_1 \neq \mathbf{null} \wedge e_1 \mapsto e_2$. All resulting sub-formulas of the form $\mathsf{RS}(P \wedge S, l, u)$ are finally replaced by $\mathsf{RP}(P, l, u) \wedge \mathsf{RS}(S, l, u)$. This ensures that ψ^p_{ren} and ψ^s_{ren} always have at least one common array variable, unless ψ^s is **emp**. The length of these common arrays determines the implicit upper bound in the universal quantifier of RPand RSpredicates in $\mathsf{Iter}(\psi)$.

Example 3. Recall Example 2 where two instances of the summary $[v = \_x \wedge \_x \mapsto \_y]$ S $[v = \_y \wedge \_x \mapsto \_y]$ are composed using $\varphi^0_{pre} : (\_x_1 = \_y_0 \wedge \_x_1 \mapsto \_y_1)$ and φ^0_{post} : $(\_x_1 = \_y_0 \wedge \_x_0 \mapsto \_y_0)$. For this example, $\mathsf{Iter}(\varphi^0_{pre})$ generates the $\mathcal{LISF}$ formula $\mathsf{RP}(\mathbf{x}[\cdot + 1] = \mathbf{y}[\cdot]) \wedge \mathsf{RS}(\mathbf{x}[\cdot + 1] \mapsto \mathbf{y}[\cdot + 1])$, and $\mathsf{Iter}(\varphi^0_{post})$ generates the formula $\mathsf{RP}(\mathbf{x}[\cdot + 1] = \mathbf{y}[\cdot]) \wedge \mathsf{RS}(\mathbf{x}[\cdot] \mapsto \mathbf{y}[\cdot])$. In this representation, the arrays $\mathbf{x}$ and $\mathbf{y}$ represent the sequences $\_x_0, \ldots, \_x_k$ and $\_y_0, \ldots, \_y_k$, respectively. The renamed formulas $\varphi\alpha$ and $\widehat{\varphi}\beta$ correspond to the formulas $v = \mathbf{x}[0] \wedge \mathbf{x}[0] \mapsto \mathbf{y}[0]$ and $v = \mathbf{y}[\$0] \wedge \mathbf{x}[\$0] \mapsto \mathbf{y}[\$0]$ respectively. The application of INDUCT thus generates the summary: $[v = \mathbf{x}[0] \wedge \mathsf{RP}(\mathbf{x}[\cdot + 1] = \mathbf{y}[\cdot]) \wedge \mathbf{x}[0] \mapsto \mathbf{y}[0] * \mathsf{RS}(\mathbf{x}[\cdot + 1] \mapsto \mathbf{y}[\cdot + 1])]$ $\mathtt{S}^+$ $[v = \mathbf{y}[\$0] \wedge \mathsf{RP}(\mathbf{x}[\cdot + 1] = \mathbf{y}[\cdot]) \wedge \mathsf{RS}(\mathbf{x}[\cdot] \mapsto \mathbf{y}[\cdot]) * \mathbf{x}[\$0] \mapsto \mathbf{y}[\$0]]$.

5.2. Inference Rule INDUCTQ

In general, the strong bi-abduction of $\exists X.\ \widehat{\varphi}^0$ and φ^1 in premise 6 may require variables to be existentially quantified on the right hand side. The INDUCT rule needs to be slightly modified in this case. However, the basic intuition of acceleration remains the same, as is illustrated in the Figure 5. The modified rule INDUCTQ is presented in Figure 9. We use a refined notation in INDUCTQ where φ^i (respectively, $\widehat{\varphi}^i$) denotes φ (resp. $\widehat{\varphi}$) with every variable $w \in W$ replaced by an indexed variable w_i and every variable $x \in X$ replaced by x_{i+1}. Let the strong bi-abduction between $\widehat{\varphi}^0$ and φ^1 be $(\exists X_1.\ \widehat{\varphi}^0) * \varphi^0_{pre} \Leftrightarrow \exists Z_1.\ (\varphi^0_{post} * \varphi^1)$, where $Z_1 \subseteq W_1 \cup X_1$ is the set of auxiliary variables. If the additional side-condition $free(\varphi^0_{pre}) \cap Z_0 = \emptyset$ holds, we can infer the accelerated summary in the conclusion of INDUCTQ.

Let Z_i be the set of variables $\{z^1_i, \ldots, z^r_i\}$. The values of variables in $Z_0 = \{z^1_0, \ldots z^r_0\}, \ldots, Z_k = \{z^1_k, \ldots, z^r_k\}$ are represented as elements of r arrays $\mathbf{z}^1 = \{z^1_0, \ldots, z^1_k\}, \ldots, \mathbf{z}^r = \{z^r_0, \ldots, z^r_k\}$ in the postcondition of conclusion of INDUCTQ. 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 $Z_1 \cup \cdots \cup Z_k$ need to be existentially quantified in the postcondition of the conclusion of INDUCTQ because of the existential quantification of Z_1 in strong bi-abduction. Hence, we existentially quantify the array variables $\mathbf{z}^1, \ldots, \mathbf{z}^r$ in the conclusion of INDUCTQ.

By existentially quantifying the array variables $\mathbf{z}^1, \ldots, \mathbf{z}^r$ in the conclusion of 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 correspondance between the Z_0 variables in 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* INDUCT *and* INDUCTQ *are sound.*

PROOF. We use induction on number of compositions to prove INDUCTQ. COMPOSE proves the base case, $[\varphi^0 * \varphi^0_{pre}]$ S; S $[\exists X_2, Z_1.\ (\varphi^0_{post} * \widehat{\varphi}^1)]$. The induction case can be proved as follows.

1. $[\varphi^i]$ S $[\exists X_{i+1},\ \widehat{\varphi}^i]$ Premise 1, Aux. variable renaming
2. $[\varphi^0 * \odot^{k-1}_{j=0} \varphi^j_{pre}]$ $\mathtt{S}^{k+1}$ $[\exists X_{k+1}, Z_1, \ldots, Z_k.$ $\odot^{k-1}_{i=0}\ \varphi^i_{post} * \widehat{\varphi}^k]$ Induction case assumption Premise 6
3. $(\exists X_{k+1}.\ \widehat{\varphi}^k) * \varphi^k_{pre} \Leftrightarrow \exists Z_{k+1}.\ (\varphi^k_{post} * \varphi^{k+1})$

4. $\left(\exists X_{k+1}, Z_1, \ldots, Z_k.\ \odot_{i=0}^{k-1} \varphi^i_{post} * \widehat{\varphi}^k\right) * \varphi^k_{pre}$

$\Updownarrow$ — $\odot_{i=0}^{k-1}\varphi^i_{post}$ depends on $W_0, \ldots, W_k$ and $Z_1, \ldots, Z_k$, it is indep. of X_{k+1}

$\left(\exists Z_1, \ldots, Z_k.\ \odot_{i=0}^{k-1} \varphi^i_{post} * (\exists X_{k+1}.\ \widehat{\varphi}^k)\right) * \varphi^k_{pre}$

$\Updownarrow$ — By premise 8, $Z_i \cap free(\varphi^k_{pre}) = \emptyset$ for any $i \in \{1..k\}$

$\left(\exists Z_1, \ldots, Z_k.\ \odot_{i=0}^{k-1} \varphi^i_{post} * (\exists X_{k+1}.\ \widehat{\varphi}^k) * \varphi^k_{pre}\right)$

$\Updownarrow$ — From 3

$\left(\exists Z_1, \ldots, Z_k.\ \odot_{i=0}^{k-1} \varphi^i_{post} * \exists Z_{k+1}.\ (\varphi^k_{post} * \varphi^{k+1})\right)$

$\Updownarrow$ — $\odot_{i=0}^{k-1}\varphi^i_{post}$ is independent of Z_{k+1}

$\left(\exists Z_1, \ldots, Z_{k+1}.\ \odot_{i=0}^{k} \varphi^i_{post} * \varphi^{k+1}\right)$

5. $[\varphi^{k+1}]$ S $[\exists X_{k+2},\ \widehat{\varphi}^{k+1}]$ — Premise 1, Aux. var. renaming

6. $\left[\varphi^0 * \odot_{i=0}^{k-1}\varphi^i_{pre} * \varphi^k_{pre}\right]$
S^{k+2}
$\left[(\exists Z_1, \ldots, Z_{k+1}.\ \odot_{i=0}^{k} \varphi^i_{post} * (\exists X_{k+2},\ \widehat{\varphi}^{k+1}))\right]$ — Apply Compose to 2 and 5, using strong bi-abduction between first and last formulas of 4

7. $[\varphi^0 * \odot_{i=0}^{k}\varphi^i_{pre}]$
S^{k+2}
$\left[\exists X_{k+2}, Z_1, \ldots, Z_{k+1}.\ \odot_{i=0}^{k} \varphi^i_{post} * \widehat{\varphi}^{k+1}\right]$ — from 6; X_{k+2} is disjoint from $Z_1 \cup \cdots \cup Z_k$

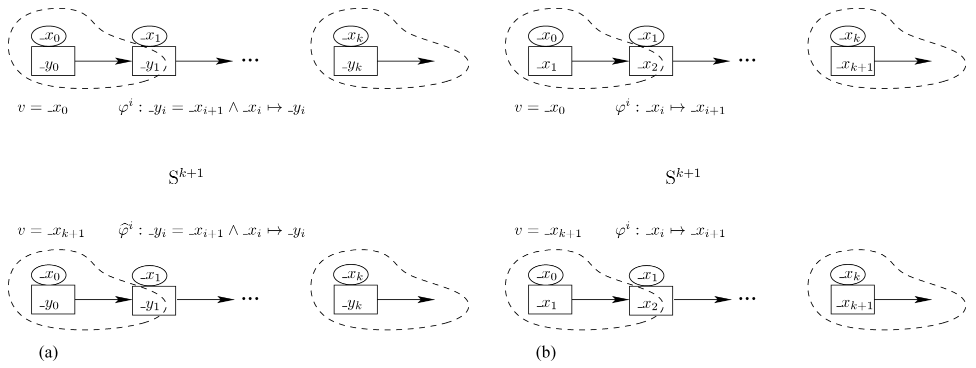
The Hoare triple in 7 in this section is expressed in the conclusion of InductQ as $[\varphi\alpha * \mathsf{Iter}(\varphi^0_{pre})]$ S$^+$ $[\exists X, \mathbf{z}^1, \ldots, \mathbf{z}^r.\ \mathsf{Iter}(\varphi^0_{post}) * \widehat{\varphi}\beta]$. The formulas $\odot_{i=0}^{k}\varphi^i_{pre}$ and $\odot_{i=0}^{k}\varphi^i_{post}$ are expressed in $\mathcal{LISF}$ as $\mathsf{Iter}(\varphi^i_{pre})$ and $\mathsf{Iter}(\varphi^i_{post})$, respectively. The parameter k in the pre- and postcondition of 7 is implicitly is hidden in the semantics of RS and RP predicates output by Iter. Every free array variable in $\mathsf{Iter}(\varphi^i_{post})$ is guaranteed to be free in $\mathsf{Iter}(\varphi^i_{pre})$ 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 $\mathsf{Iter}(\varphi^i_{post})$ are existentially quantified and hence $\mathsf{Iter}(\varphi^i_{pre})$ and $\mathsf{Iter}(\varphi^i_{post})$ 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 $\mathsf{Iter}(\varphi^i_{post})$, where e is an expression from $\mathsf{Iter}(\varphi^i_{pre})$ involving an array variable not present in $\mathsf{Iter}(\varphi^i_{post})$. □

5.3. Inference Rule InductSymm

The inference rule 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 InductQ which capture the effect of executing S one or more times. Additionally, InductSymm also enables us to eliminate some variables from the pre- and postcondition of the inferred summary, thus simplifying it.

If, in Eq. (4.1) φ^i_{pre} (respectively, φ^i_{post}) is same as φ^0 (respectively, $\widehat{\varphi}^k$) modulo variable renaming, then we can infer the following summary: $[(\odot_{i=0}^{k} \varphi^i)]\mathrm{S}^{k+1}[(\odot_{i=0}^{k} \widehat{\varphi}^i)]$. Recall the accelerated summary inferred in Example 2, which is depicted in Figure 5(c). In this example, the shape of φ^0 (respectively, $\widehat{\varphi}^0$) and φ^i_{pre} (respectively, φ^i_{post}) are the same. Hence we can rewrite the accelerated summary as follows. $[v = \_x_0 \wedge \odot_{i=0}^{k}(\_x_i \mapsto \_y_i \wedge \_y_i = \_x_{i+1})]$ S^{k+1} $[v = \_x_{k+1} \wedge \odot_{i=0}^{k}(\_x_i \mapsto \_y_i \wedge \_y_i = \_x_{i+1})]$. This is depicted in Figure 11(a).

The equalities $x_{i+1} = y_i$, 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



$v = \_x_0$
$\varphi^i : \_y_i = \_x_{i+1} \wedge \_x_i \mapsto \_y_i$
S^{k+1}
$v = \_x_{k+1}$
$\widehat{\varphi}^i : \_y_i = \_x_{i+1} \wedge \_x_i \mapsto \_y_i$
$v = \_x_0$
$\varphi^i : \_x_i \mapsto \_x_{i+1}$
S^{k+1}
$v = \_x_{k+1}$
$\varphi^i : \_x_i \mapsto \_x_{i+1}$
(a)
(b)

Fig. 11. (a) Alternate representation of summary in Figure 5-c, and (b) Summary resulting from application of InductSymm. Each box represents a heap cell, its contents represents the value of `next` field. A circled variable above a box denotes the name of the cell.

InductSymm

Given

1. $[\varphi]$ S $[\exists X.\ \widehat{\varphi}]$, $|X| = t$
2. $\widehat{\varphi}^0 : \widehat{\varphi}$ 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. $(\exists X_1.\ \widehat{\varphi}^0) * \varphi^1$ is satisfiable
5. $\tau_i : \langle mod(\mathrm{S}) \cap free(\exists X.\ \widehat{\varphi}) \to free(\varphi^i) \setminus mod(\mathrm{S})\rangle$, for $i \in \{0, 1\}$,
   s.t. Pure part of φ^i implies $(\mathsf{Eq}\ \tau_i)$
6. $\gamma_0 : \langle mod(\mathrm{S}) \cap free(\varphi) \to free(\exists X_1.\widehat{\varphi}^0) \setminus mod(\mathrm{S})\rangle$, s.t. Pure part of $\exists X_1.\widehat{\varphi}^0$ implies $(\mathsf{Eq}\ \gamma_0)$
7. $\alpha : \langle y_0 \to \mathbf{Y}[0]\rangle$, for each y in W
8. $\beta : \langle y_1 \to \mathbf{Y}[\$0]\rangle$, for each y in W
9. Iter same as described in Induct
10. $\delta_0^1 : \langle W_0 \hookrightarrow W_1\rangle$, s.t. $a_0 \to b_1 \in \delta_0^1$ iff $(\mathsf{Eq}\ \gamma_0)\tau_1 \Rightarrow a_0 = b_1$, and $a_0 \notin range(\tau_0)$

Infer

$$[(\mathsf{Eq}\ \tau_0)\alpha \wedge \mathsf{Iter}(\varphi^0\tau_0\delta_0^1 \wedge (\mathsf{Eq}\ \gamma_0)\tau_1\delta_0^1)]$$
$$\mathrm{S}^\star$$
$$[\exists \mathbf{X}^1, \ldots, \mathbf{X}^t.\ (\mathsf{Eq}\ \tau_1)\beta \wedge \mathsf{Iter}(\widehat{\varphi}^0\gamma_0\delta_0^1 \wedge (\mathsf{Eq}\ \gamma_0)\tau_1\delta_0^1)]$$

Fig. 12. Variant of Inductq, InductSymm.

$\_y_i$ by x_{i+1} from both the pre- and postcondition, and thus eliminate all the $\_y_i$'s. We obtain the following simplified summary from this renaming (depicted in Figure 11(b)). $[v = \_x_0 \wedge \odot_{i=0}^{k} \_x_i \mapsto \_x_{i+1}]$ S^{k+1} $[v = \_x_{k+1} \wedge \odot_{i=0}^{k} \_x_i \mapsto \_x_{i+1}]$. The corresponding summary in $\mathcal{LISF}$ is $[v = \mathbf{x}[0] \wedge \mathsf{RS}(\mathbf{x}[\cdot] \mapsto \mathbf{x}[\cdot + 1])]$ $\mathrm{S}^\star$ $[v = \mathbf{x}[\$0] \wedge \mathsf{RS}(\mathbf{x}[\cdot] \mapsto \mathbf{x}[\cdot + 1])]$. In this specification, if the length of $\mathbf{x}$ is $\lambda + 1$ (where $\lambda \geq 0$), then it summarizes λ iterations of S. Hence, it is a summary for zero or more iterations of S, denoted as $[\varphi]$ $\mathrm{S}^\star$ $[\widehat{\varphi}]$. The notation $[\varphi]$ $\mathrm{S}^\star$ $[\widehat{\varphi}]$ means that for every initial state satisfying φ, there exists a $k \geq 0$ such that the state resulting after k executions of S satisfies $\widehat{\varphi}$. These ideas are captured formally by the rule InductSymm in Figure 12.

For a renaming γ, let $(\mathsf{Eq}\ \gamma)$ denote the conjunction of all the equalities $a = b$ such that γ renames a to b. The premises 5 and 6 of InductSymm in Figure 12 imply $\varphi^0 \equiv \mathsf{Eq}\ \tau_0 \wedge \varphi^0\tau_0$ and $\exists X_1.\ \widehat{\varphi}^0 \equiv \exists X_1.\ (\widehat{\varphi}^0\gamma_0 \wedge \mathsf{Eq}\ \gamma_0)$, respectively. These premises also imply that γ_0 and τ_1 have same domains and their ranges are independent of *mod*(S) variables, hence $(\mathsf{Eq}\ \gamma_0)\tau_1$ is independent of *mod*(S). This fact implies that $(\mathsf{Eq}\ \gamma_0) \wedge (\mathsf{Eq}\ \gamma_0)\tau_1 \Leftrightarrow (\mathsf{Eq}\ \tau_1) \wedge (\mathsf{Eq}\ \gamma_0)\tau_1$. Hence, $[\psi^0]$ S $[\exists X_1.\ \widehat{\psi}^0]$ is a valid Hoare triple, where $\psi^0 \equiv \mathsf{Eq}\ \tau_0 \wedge \varphi^0\tau_0 \wedge (\mathsf{Eq}\ \gamma_0)\tau_1$ and $\widehat{\psi}^0 \equiv \mathsf{Eq}\ \tau_1 \wedge \widehat{\varphi}^0\gamma_0 \wedge (\mathsf{Eq}\ \gamma_0)\tau_1$. Let ψ^i (respectively, $\widehat{\psi}^i$)

be same as ψ^0 (respectively, $\widehat{\psi}^0$) 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 $[\exists X_{i+1}.\ \widehat{\psi}^i]$ is a valid Hoare triple. Let us compose the Hoare triples $[\psi^0]$ S $[\exists X_1.\ \widehat{\psi}^0]$ and $[\psi^1]$ S $[\exists X_2.\ \widehat{\psi}^1]$. From the definitions of $\widehat{\psi}^0$ and ψ^1, we can infer the following strong bi-abduction between $\exists X_1.\ \widehat{\psi}^0$ and ψ^1.

$$(\exists X_1.\ \widehat{\psi}^0) \ * \ \underbrace{\varphi^1\tau_1 \wedge (\mathsf{Eq}\ \gamma_1)\tau_2}_{\varphi^0_{pre}} \Leftrightarrow \exists X_1.\ (\underbrace{\widehat{\varphi}^0\gamma_0 \wedge (\mathsf{Eq}\ \gamma_0)\tau_1}_{\varphi^0_{post}} \ * \ \psi^1). \tag{5.2}$$

An interesting feature of this strong bi-abduction is that $\varphi^0_{pre} \wedge (\mathsf{Eq}\ \tau_1)$ (respectively, $\varphi^0_{post} \wedge (\mathsf{Eq}\ \tau_1)$) is same as ψ^1 (respectively, $\widehat{\psi}^0$). Thus, the shape of φ^0_{pre} (respectively, φ^0_{post}) is same as that of ψ^0 (respectively, $\widehat{\psi}^0$). Thus, from the premises 1-9, by inductively applying COMPOSE to the sequence of Hoare triples, $[\psi^0]$ S $[\exists X_1.\ \widehat{\psi}^0]$, $[\psi^1]$ S $[\exists X_2.\ \widehat{\psi}^1], \ldots,$ $[\psi^k]$ S $[\exists X_{k+1}.\ \widehat{\psi}^k]$, we obtain the following accelerated summary.

$$[(\mathsf{Eq}\ \tau_0) \wedge \odot_{i=0}^k \varphi^i\tau_i \wedge (\mathsf{Eq}\ \gamma_i)\tau_{i+1}] \ \mathsf{S}^* \ [(\mathsf{Eq}\ \tau_{k+1}) \wedge \odot_{i=0}^k \exists X_{i+1}.\ \widehat{\varphi}^i\gamma_i \wedge (\mathsf{Eq}\ \gamma_i)\tau_{i+1}]. \tag{5.3}$$

INDUCTSYMM uses the premise 10 to existentially quantify some auxiliary variables from the summary $[\psi^i]$ S $[\exists X_{i+1}.\ \widehat{\psi}^i]$ and thus simplify the final accelerated summary computed above. For this purpose, we define a renaming δ^1_0 from variables in W_0 to variables in W_1. It is computed from the equalities in $(\mathsf{Eq}\ \gamma_0)\tau_1$. Using the rule for existentially quantifying auxiliary variables, it follows that each of $[\psi^0\delta^1_0]$ S $[\exists X_1.\ \widehat{\psi}^0\delta^1_0]$, $[\psi^1\delta^2_1]$ S $[\exists X_2.\ \widehat{\psi}^1\delta^2_1], \ldots, [\psi^k\delta^{k+1}_k]$ S $[\exists X_{k+1}.\ \widehat{\psi}^k\delta^{k+1}_k]$ is a valid Hoare triple. If $a_0 \to b_1 \in \delta^1_0$ then we can eliminate all occurrences of a_i's by applying the renaming δ^1_0 and δ^2_1 to both sides of the the strong bi-abduction in (5.2). The renaming δ^1_0 has a property that if $b_1 \in range(\delta^1_0)$ then $b_1 \in range(\tau_1)$ which in turn implies $b_1 \notin dom(\delta^2_1)$. This ensures that (a) $(\mathsf{Eq}\ \tau_1)\delta^1_0\delta^2_1 \equiv (\mathsf{Eq}\ \tau_1)$, (b) $(\widehat{\varphi}^0\gamma_0)\delta^1_0\delta^2_1 \equiv (\widehat{\varphi}^0\gamma_0)\delta^1_0$, and (c) $(\mathsf{Eq}\ \gamma_0)\tau_1\delta^1_0\delta^2_1 \equiv (\mathsf{Eq}\ \gamma_0)\tau_1\delta^1_0$. Hence, $\widehat{\psi}^0\delta^1_0\delta^2_1 \equiv \widehat{\psi}^0\delta^1_0$. Using the renamings δ^1_0 and δ^2_1, we can therefore infer the following strong bi-abduction between $\exists X_1.\ \widehat{\psi}^0\delta^1_0$ and $\psi^1\delta^2_1$.

$$\big(\exists X_1.\ \widehat{\psi}^0\delta^1_0\big) \ * \ \underbrace{\varphi^1\tau_1\delta^2_1 \wedge (\mathsf{Eq}\ \gamma_1)\tau_2\delta^2_1}_{\varphi^0_{pre}} \Leftrightarrow \exists X_1.\ \big(\underbrace{\widehat{\varphi}^0\gamma_0\delta^1_0 \wedge (\mathsf{Eq}\ \gamma_0)\tau_1\delta^1_0}_{\varphi^0_{post}} \ * \ \psi^1\delta^2_1\big). \tag{5.4}$$

We require δ^1_0 to satisfy the constraint $a_0 \in dom(\delta^1_0) \Rightarrow a_0 \notin range(\tau_0)$ so that $\widehat{\psi}^0\delta^1_0\delta^2_1$ is equivalent to $\widehat{\psi}^0\delta^1_0$ and it does not have variables with all indices 0,1 and 2; otherwise, its repetition cannot be expressed by $\mathcal{LISF}$ predicates RS and RP.

By inductive application of the compose rule to the sequence of Hoare triples, $[\psi^0\delta^1_0]$ S $[\exists X_1.\ \widehat{\psi}^0\delta^1_0]$, $[\psi^1\delta^2_1]$ S $[\exists X_2.\ \widehat{\psi}^1\delta^2_1], \ldots, [\psi^k\delta^{k+1}_k]$ S $[\exists X_{k+1}.\ \widehat{\psi}^k\delta^{k+1}_k]$, we get the following accelerated Hoare triple.

$$\begin{array}{c} \big[(\mathsf{Eq}\ \tau_0) \ \wedge \ \odot_{i=0}^k \ \varphi^i\tau_i\delta^{i+1}_i \wedge (\mathsf{Eq}\ \gamma_i)\tau_{i+1}\delta^{i+1}_i\big] \\ \mathsf{S}^* \\ \big[(\mathsf{Eq}\ \tau_{k+1}) \ \wedge \ \odot_{i=0}^k \ \exists X_{i+1}.\ \widehat{\varphi}^i\gamma_i\delta^{i+1}_i \wedge (\mathsf{Eq}\ \gamma_i)\tau_{i+1}\delta^{i+1}_i\big] \end{array} \tag{5.5}$$

The conclusion of INDUCTSYMM uses the renaming α, β and the function Iter (which are same as those defined in INDUCT) to represent this Hoare triple in $\mathcal{LISF}$.

Example 3 uses the inference rule INDUCT to accelerate the summary $[v = \_x \wedge \_x \mapsto \_y]$ S $[v = \_y \wedge \_x \mapsto \_y]$. In the following example, we apply the inference rule INDUCTSYMM to accelerate the same summary.

Example 4. Recall the acceleration of summary $[v = \_x_0 \wedge \_x_0 \mapsto \_y_0]$ S $[v = \_y_0 \wedge \_x_0 \mapsto \_y_0]$ in Example 3. For this example we can obtain τ_i and γ_0 as $\langle v \to \_x_i\rangle$ and

$\langle v \rightarrow \_y_0\rangle$, respectively. These renamings satisfy the premises 5 and 6 of InductSymm. With these renamings we find that $(\mathsf{Eq}\ \gamma_0)\tau_1$ is equivalent to $\_y_0 = \_x_1$. The expressions $\varphi^0\tau_0$ and $\widehat{\varphi}^0\gamma_0$ are both equivalent to $\_x_0 \mapsto \_y_0$. Hence we can infer the valid Hoare triple $[\psi^0]$ S $[\exists X_1.\ \widehat{\psi}^0]$, where ψ^0 and $\exists X_1.\ \widehat{\psi}^0$ are $v = \_x_0 \wedge \_x_0 \mapsto \_y_0 \wedge \_y_0 = \_x_1$, and $v = \_x_1 \wedge \_x_0 \mapsto \_y_0 \wedge \_y_0 = \_x_1$, respectively.

The renaming $\langle \_y_0 \rightarrow \_x_1\rangle$ satisfies the requirements of δ_0^1 in the premise 10. Hence, we find that both $(\varphi^0\tau_0\delta_0^1 \wedge (\mathsf{Eq}\ \gamma_0)\tau_1\delta_0^1)$ and $(\widehat{\varphi}^0\gamma_0\delta_0^1 \wedge (\mathsf{Eq}\ \gamma_0)\tau_1\delta_0^1)$ are equivalent to $\_x_0 \mapsto \_x_1$.

For composing the two triples $[\psi^0\delta_0^1]$ S $[\exists X_1.\ \widehat{\psi}^0\delta_0^1]$ and $[\psi^1\delta_1^2]$ S $[\exists X_2.\ \widehat{\psi}^1\delta_1^2]$, the following is a valid strong bi-abduction.

$(v = \_x_1 \wedge \_x_0 \mapsto \_x_1) * (\_x_1 \mapsto \_x_2) \Leftrightarrow (v = \_x_1 \wedge \_x_0 \mapsto \_x_1) * (\_x_1 \mapsto \_x_2)$

Thus, the premises of InductSymm guarantee the validity of the following accelerated summary $[v = \_x_0 \wedge \odot_{i=0}^{k} \_x_i \mapsto \_x_{i+1}]$ $\mathsf{S}^\star$ $[v = \_x_{k+1} \wedge \odot_{i=0}^{k} \_x_i \mapsto \_x_{i+1}]$. Hence, by application of InductSymm we obtain the following $\mathcal{LISF}$ summary $[v = \mathbf{x}[0] \wedge \mathsf{RS}(\mathbf{x}[\cdot] \mapsto \mathbf{x}[\cdot + 1])]$ $\mathsf{S}^\star$ $[v = \mathbf{x}[\$0] \wedge \mathsf{RS}(\mathbf{x}[\cdot] \mapsto \mathbf{x}[\cdot + 1])]$

5.4. Discussion

The summary inferred by InductSymm captures the effect of executing the statement S zero or more times. This is in contrast with the summaries inferred by InductQ, which captures 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 that 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 *nonzero* 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 nonzero length inner linked lists. However, the resulting summary after two applications of InductQ is not a complete specification for the program.

In contrast, 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 InductSymm, we can obtain a summary for the outer while loop whose precondition expresses the fact that `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 Figures 4, 9, and 12 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.

Lemma 5.2. *The rule* InductSymm *is sound.*

5.5. Generating Summaries using Combination of Rules

The Compose and 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 Induct, InductQ, and InductSymm rules generate an accelerated summary for use in the While rule. Any pair of accelerated summaries can also be composed to obtain new accelerated summaries.

$$\textsc{Join}\ \frac{\begin{array}{c}[x = 0 \wedge \varphi_1]\ \texttt{assert(e);S1}\ [\widehat{\varphi}_1],\ [x \mapsto (f:y) * \varphi_2]\ \texttt{assert(!e);S2}\ [\widehat{\varphi}_2],\\ \widehat{\varphi}_1\mu \Rightarrow \widehat{\varphi}_2,\ \varphi_1\mu \Leftrightarrow \varphi_2,\ \mathbf{A} \text{ is a fresh auxiliary variable, } mod(\texttt{S1}) = mod(\texttt{S2})\end{array}}{[x = \mathbf{A}[0] \wedge \mathbf{A}[\$0] = \mathbf{null} \wedge \mathsf{RP}(\mathbf{A}[\cdot+1] = \mathbf{null}) \wedge \mathsf{RS}(\mathbf{A}[\cdot] \mapsto (f:y)) * \varphi_2]\ \texttt{if(e, S1, S2)}\ [\widehat{\varphi}_2]}$$

Fig. 13. The rule Join.

We now present a procedure to enumerate all possible accelerated summaries for the while loop `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 $\widehat{\mathcal{S}}$ be the set of summaries for the loop body `assert(B);S`. For the summaries s_1 and s_2, let s_1^+ denote the accelerated summary obtained by applying one of the Induct, InductQ, or InductSymm rules to s_1, and let $s_1 \circ s_2$ denote the summary obtained by applying the Compose rule to s_1 and s_2. Let $\bar{\mathcal{S}}$ be the set of summaries defined as the least fix-point of the following set transformer: $F(\mathcal{S}) = \{s^+ \mid s \in S\} \cup \{s_1 \circ s_2 \mid s_1, s_2 \in S\} \cup \widehat{\mathcal{S}}$. The set $\bar{\mathcal{S}}$ contains all the accelerated summaries – a complete functional specification for the loop `while (B) S` (assuming $\widehat{\mathcal{S}}$ is a complete set of summaries for the loop body `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, nonrecursive procedure calls can be analyzed by the Compose rule, as in Calcagno et al. [2009]. The InductQ rule can be used to compute accelerated summaries of tail recursive procedures having at most one self-recursive call.

5.6. Generating Conscise 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 $[x = 0 \wedge \varphi_1]$ `assert(e);S1` $[\widehat{\varphi}_1]$, and $[x \mapsto (f{:}y) * \varphi_2]$ `assert(!e);S2` $[\widehat{\varphi}_2]$ of two branches of the statement `if (e, S1, S2)` (first two premises of Join). If $\widehat{\varphi}_1\mu \Rightarrow \widehat{\varphi}_2$ and $\varphi_1\mu \Leftrightarrow \varphi_2$, where μ renames auxiliary variables, are valid, then we can infer the concise summary $[(x = 0 \vee x \mapsto (f : y)) * \varphi_2]$ `if(e, S1, S2)` $[\widehat{\varphi}_2]$. Since $\mathcal{LISF}$ does not permit disjunctions, the precondition cannot be directly expressed in $\mathcal{LISF}$. However, we can encode the disjunction $(x = 0 \vee x \mapsto (f : y))$ using a fresh auxiliary array variable $\mathbf{A}$ as: $\psi \equiv x = \mathbf{A}[0] \wedge \mathbf{A}[\$0] = \mathbf{null} \wedge \mathsf{RP}(\mathbf{A}[\cdot+1] = \mathbf{null}) \wedge \mathsf{RS}(\mathbf{A}[\cdot] \mapsto (f : y))$. The formula $\exists \mathbf{A}\, \psi$ is equivalent to $x = \mathbf{null}$ (respectively, $x \mapsto (f : y)$) when the length of $\mathbf{A}$ is 1 (respectively, 2). It in inconsistent when the length of $\mathbf{A}$ is greater than 2. Hence, it is equivalent to $(x = 0 \vee x \mapsto (f : y))$. In Section 6 on strong bi-abduction, we show how to implement the checks $\varphi_1\mu \Leftrightarrow \varphi_2$ and $\widehat{\varphi}_1\mu \Rightarrow \widehat{\varphi}_2$ for quantifier free $\mathcal{LISF}$ 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 $\varphi_1, \widehat{\varphi}_1$ are quantifier free formulas over free variable V, W and $\varphi_2, \widehat{\varphi}_2$ are quantifier free formulas over free variable V, Y.

```
Decompose($\varphi$, $\psi$)
 1: $res \leftarrow \{\}$
 2: for all $(M, C, L_1, L_2) \in$ Match($\varphi^s, \psi^s, 0$)
    do
 3:     $\Delta \leftarrow (\varphi^p \wedge L_1) * (M \wedge C) * (\psi^p \wedge L_2)$
 4:     if sat($\Delta$) then
 5:         $\delta_1 \leftarrow M \wedge \psi^p \wedge L_2$
 6:         $\delta_2 \leftarrow M \wedge \varphi^p \wedge L_1$
 7:         $res \leftarrow res \cup \{(\delta_1, \delta_2)\}$
 8: return $res$
```

```
BiAbduct($\varphi$, $\psi$, $mod_1$, $mod_2$)
 1: $res \leftarrow \{\}$
 2: for all $(\delta_1, \delta_2) \in$ Decompose($\varphi, \psi$) do
 3:     $\delta'_1 \leftarrow$ RemoveVar($\delta_1, \varphi, mod_1, V \cup W$)
 4:     $\delta'_2 \leftarrow$ RemoveVar($\delta_2, \psi, mod_2, V \cup Y$)
 5:     $\gamma \leftarrow$ ComputeRenaming($\delta'_1, Y, mod_1$)
 6:     $\kappa_1 \leftarrow \delta'_1\gamma$
 7:     $\hat{Z} \leftarrow dom(\gamma)$
 8:     if IsIndep($\kappa_1, mod_1$) and IsIndep($\delta'_2, mod_2$) then
 9:         $\theta \leftarrow$ ComputeRenaming($\kappa_1, Y, X$)
10:         $\widetilde{Z} \leftarrow$ Domain($\theta$)
11:         $\kappa'_1 \leftarrow$ RemoveRedundant($\kappa_1\theta, \varphi^p$)
12:         if IsIndep($\kappa'_1, X$) then
13:             $\kappa_2 \leftarrow$ RemoveRedundant($\delta'_2\bar{\theta}, \psi^p$)
14:             $res \leftarrow res \cup (\kappa'_1, \kappa_2, \hat{Z} \cup \widetilde{Z})$
15: return $res$
```

Fig. 14. Algorithm BiAbduct.

5.7. Generating Summaries with Recursive Predicates

Instead of translating a recurrence into a $\mathcal{LISF}$ formula, we could as well translate it into a recursive predicate in the conclusion of INDUCT, INDUCTQ or INDUCTSYMM. As an illustration, recall the summary $[v = \_x_0 \wedge \odot_{i=0}^{k} \_x_i \mapsto \_x_{i+1}]$ $\mathsf{S}^\star$ $[v = \_x_{k+1} \wedge \odot_{i=0}^{k} \_x_i \mapsto \_x_{i+1}]$ generated by the INDUCTSYMM rule in Example 4. The recurrence $\odot_{i=0}^{k} \_x_i \mapsto \_x_{i+1}$ previously obtained can be translated into a recursive predicate $\mathsf{list}(\_x_0, \_x_{k+1})$, where $\mathsf{list}(\_x_0, \_x_{k+1})$ 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, $\mathsf{list}(\_x_0, \_x_{k+1}) \stackrel{\mathsf{def}}{=} \_x_0 \mapsto \_x_{k+1} \vee \exists \_x_1.\ \_x_0 \mapsto \_x_1 * \mathsf{list}(\_x_1, \_x_{k+1})$. Hence, we can generate the summary $[v = \_x_0 \wedge \mathsf{list}(\_x_0, \_x_{k+1})]$ $\mathsf{S}^\star$ $[v = \_x_{k+1} \wedge \mathsf{list}(\_x_0, \_x_{k+1})]$, using recursive predicates as a conclusion of 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 nonfunctional.

6. A STRONG BI-ABDUCTION ALGORITHM FOR $\mathcal{LISF}$

In this section, we present a procedure to compute strong bi-abduction. We first present a solution to a subproblem of computing $\mathcal{LISF}$ formulas δ_1 and δ_2, given two quantifier free $\mathcal{LISF}$ formulas φ and ψ, such that $\varphi * \delta_1 \Leftrightarrow \delta_2 * \psi$. The algorithm Decompose given in Figure 14 computes such δ_1 and δ_2 given φ and ψ as input.

The key step in Decompose is the Match procedure used in line 2. Match takes two spatial formulas φ^s and ψ^s and an integer constant (that corresponds to nesting depth of φ^s and ψ^s within 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 φ^s and ψ^s can be decomposed into an overlapping part defined by C and nonoverlapping 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 RS predicate in UNROLLFRONT and UNROLLBACK. The function $\mathsf{unroll_f}(\mathsf{RS}(S, l, u), d)$ required by rule UNROLLFRONT unrolls RS once from the beginning. It returns the formula obtained by replacing every $(d+1)$th iterated index $[\cdot]$ (respectively, $[\cdot+1]$) in S by the fixed index $[l]$

No-Match

$$\frac{}{(\mathbf{true}, \mathbf{emp}, S_1, S_2) \in \mathsf{Match}(S_1, S_2, d)}$$

Cell-Match

$$\frac{k_1 \equiv x \mapsto (f^i : x^i),\ \ k_2 \equiv y \mapsto (f^i : y^i) \qquad M \equiv x = y \wedge \bigwedge\{x^i = y^i\}}{(M, x \mapsto (f^i : x^i), \{\}, \{\}) \in \mathsf{Match}(k_1, k_2, d)}$$

Recursion

$$\frac{S_1 = S_1' * k_1,\ S_2 = S_2' * k_2 \qquad (M, C, L_1, L_2) \in \mathsf{Match}(k_1, k_2, d) \qquad (N, C', L_1', L_2') \in \mathsf{Match}(S_1' * L_1, S_2' * L_2, d)}{(M \wedge N, C * C', L_1', L_2') \in \mathsf{Match}(S_1, S_2, d)}$$

UnrollFront

$$\frac{k_1 \equiv \mathsf{RS}(S, l, u),\ \ k_2 \equiv x \mapsto (f : y), \qquad k' \equiv \mathsf{unroll_f}(\mathsf{RS}(S, l, u), d) \qquad (M, C, L_1, L_2) \in \mathsf{Match}(k', k_2, d)}{(M, C, L_1 * \mathsf{RS}(S, l+1, u), L_2) \in \mathsf{Match}(k_1, k_2, d)}$$

UnrollBack

$$\frac{k_1 \equiv \mathsf{RS}(S, l, u),\ \ k_2 \equiv x \mapsto (f : y), \qquad k' \equiv \mathsf{unroll_b}(\mathsf{RS}(S, l, u), d) \qquad (M, C, L_1, L_2) \in \mathsf{Match}(k', k_2, d)}{(M, C, L_1 * \mathsf{RS}(S, l, u+1), L_2) \in \mathsf{Match}(k_1, k_2, d)}$$

MatchRs

$$\frac{k_1 \equiv \mathsf{RS}(S_1, l, u),\ \ k_2 \equiv \mathsf{RS}(S_2, l, u), \qquad (M, C, \{\}, \{\}) \in \mathsf{Match}(S_1, S_2, d+1) \qquad (M_0, M_1) = \mathsf{separate\_zero\_depth}(M)}{(\mathsf{RP}(M_1, l, u) \wedge M_0, \mathsf{RS}(C, l, u), \{\}, \{\}) \in \mathsf{Match}(k_1, k_2, d)}$$

Note: $\mathsf{unroll_f}(\mathsf{RS}(S, l, u), d)$ and $\mathsf{separate\_zero\_depth}(M)$ defined in the text.

Fig. 15. Rules for procedure Match.

(respectively, $[l + 1]$). Similarly $\mathsf{unroll_b}(\mathsf{RS}(S, l, u), d)$, required by UnrollBack unrolls RS once from the end. It returns the formula obtained by replacing every $(d + 1)$th iterated index $[\cdot]$ (respectively, $[\cdot + 1]$) in S by the fixed index $[\$u + 1]$ (respectively, $[\$u]$). The rule MatchRs finds an overlapping part of the two RS predicates. This is the only rule that increments d. The function $\mathsf{separate\_zero\_depth}(M)$ used in the premise of MatchRs returns a pair of predicates M_0 and M_1. M_0 is the conjunction of predicates in M with depth zero (i.e., those predicates for which dim evaluates to 0, refer definition of the function dim in Section 4.2) and M_1 is the conjunction of remaining predicates in M. For example, $\mathsf{separate\_zero\_depth}(\mathbf{x}[\cdot] = h \wedge \mathsf{RP}(\mathbf{A}[\cdot] = \mathbf{D}[\cdot]) \wedge x = y)$ would return $(\mathsf{RP}(\mathbf{A}[\cdot] = \mathbf{D}[\cdot]) \wedge x = y,\ \mathbf{x}[\cdot] = h)$. The predicates in M_1 are embedded in an RP predicate in the conclusion of MatchRs, whereas the predicates in M_0 are not embedded in an 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 UnrollFront and UnrollBack, the size of the formula $L_1 * \mathsf{RS}(\_, \_, \_)$ in the conclusion may be larger than the size of formula k_1 in the premise. This may lead to nontermination 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* Decompose *satisfies (i) $M \wedge \varphi^s \Leftrightarrow (M \wedge C) * L_1$, and (ii) $M \wedge \psi^s \Leftrightarrow (M \wedge C) * L_2$.*

Proof. We prove the lemma by induction on the depth of the recursion tree of Match.

Base Case. Single recursive call. Rules No-Match and Cell-Match trivially satisfy the property.

Induction Step. Assuming that the call to Match in the premise of rules Recursion, UnrollFront, UnrollBack, and 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.	$M \wedge k_1 \Leftrightarrow M \wedge C * L_1$	assumption
2.	$N \wedge S_1' * L_1 \Leftrightarrow N \wedge C' * L_1'$	assumption
3.	$M \wedge N \wedge S_1' * L_1 \Leftrightarrow M \wedge N \wedge C' * L_1'$	
4.	$M \wedge N \wedge S_1' * L_1 * C \Leftrightarrow M \wedge N \wedge C' * L_1' * C$	
5.	$M \wedge N \wedge S_1' * k_1 \Leftrightarrow M \wedge N \wedge C' * L_1' * C$	from 1
6.	$M \wedge N \wedge S_1 \Leftrightarrow M \wedge N \wedge C * C' * L_1'$	premise

(2) UNROLLFRONT

1.	$M \wedge k' \Leftrightarrow M \wedge C * L_1$	assumption
2.	$M \wedge \mathsf{RS}(S,l,u) \Leftrightarrow M \wedge k' * \mathsf{RS}(S,l+1,u))$	Defn. of $\mathsf{unroll_f}$
3.	$M \wedge \mathsf{RS}(S,l,u) \Leftrightarrow M \wedge C * L_1 * \mathsf{RS}(S,l+1,u)$	from 1

(3) MATCHRS

1.	$M \wedge S_1 \Leftrightarrow M \wedge C$	assumption
2.	$M \wedge S_2 \Leftrightarrow M \wedge C$	assumption
3.	$M_0 \wedge \mathsf{RP}(M_1,l,u) \wedge \mathsf{RS}(S_1,l,u) \Leftrightarrow M_0 \wedge \mathsf{RP}(M_1,l,u) \wedge \mathsf{RS}(C,l,u)$	from 1 and definition of separate_zero_depth
4.	$M_0 \wedge \mathsf{RP}(M_1,l,u) \wedge \mathsf{RS}(S_2,l,u) \Leftrightarrow M_0 \wedge \mathsf{RP}(M_1,l,u) \wedge \mathsf{RS}(C,l,u)$	from 2 and definition of separate_zero_depth

Note that proof of UNROLLBACK is similar to that of UNROLLFRONT. □

Given a possible decomposition (M, C, L_1, L_2) of φ^s and ψ^s as computed by Match(φ^s, ψ^s, 0), line 4 of Decompose checks whether this decomposition is consistent with φ^p and ψ^p. This is done by checking the satisfiability of $(\varphi^p \wedge L_1) * (M \wedge C) * (\psi^p \wedge L_2)$. If this formula is found to be satisfiable, δ_1 and δ_2 are computed as $M \wedge \psi^p \wedge L_2$ and $M \wedge \varphi^p \wedge L_1$, respectively.

LEMMA 6.2. *Every* (δ_1,δ_2) *pair computed in lines* 5 *and* 6 *of* Decompose *satisfies* $\varphi * \delta_1 \Leftrightarrow \delta_2 * \psi$

PROOF. Follows from the following equivalences

A.	$\varphi^p \wedge \varphi^s \wedge M \Leftrightarrow (M \wedge C) * (\varphi^p \wedge L_1)$	from Lemma 6.1
B.	$\psi^p \wedge \psi^s \wedge M \Leftrightarrow (M \wedge C) * (\psi^p \wedge L_2)$	from Lemma 6.1
C.	$\Delta \Leftrightarrow \varphi * (M \wedge \psi^p \wedge L_2)$	defn of Δ and A
D.	$\Delta \Leftrightarrow \psi * (M \wedge \varphi^p \wedge L_1)$	defn of Δ and B
5.	$\varphi * (M \wedge \psi^p \wedge L_2) \Leftrightarrow \varphi * \delta_1$	defn. of δ_1, line 5 of Decompose
6.	$\psi * (M \wedge \varphi^p \wedge L_1) \Leftrightarrow \psi * \delta_2$	defn. of δ_2, line 6 of Decompose □

Note that the Match procedure results in a possibly exponential number of decompositions, many of which could be discarded by the check on line 4 of Decompose. One of the reasons for this exponential blow-up is the application of RECURSION rule which explores all possible overlaps between φ^s and ψ^s. The exponential blow-up can be mitigated by early identification of inconsistent decompositions during the application of the RECURSION rule. This can be done by pruning the application of RECURSION rule if the partial decomposition indicated in its second premise, $(M, C, L_1, L_2) \in$ Match(k_1, k_2, 0), is inconsistent with $\varphi^p \wedge \psi^p$, that is, when $M \wedge \varphi^p \wedge \psi^p$ is unsatisfiable.

For a model $(s, h, \mathcal{V})$ of $\varphi*\delta_1$ (and also of $\delta_2*\psi$), let h_φ and h_{δ_1} be disjoint sub-heaps that partition h, that is, $h = h_\varphi \sqcup h_{\delta_1}$, such that $(s, h_\varphi, \mathcal{V}) \models \varphi$ and $(s, h_{\delta_1}, \mathcal{V}) \models \delta_1$. Similarly, let h_ψ and h_{δ_2} be disjoint sub-heaps that partition h, that is, $h = h_\psi \sqcup h_{\delta_2}$, such that $(s, h_\psi, \mathcal{V}) \models \psi$ and $(s, h_{\delta_2}, \mathcal{V}) \models \delta_2$. It follows from Lemma 6.1 that every pair (δ_1, δ_2) computed by Decompose satisfies the following *minimality* property.

Definition 6.1 (*Minimality Property*). If $\varphi * \delta_1 \Leftrightarrow \delta_2 * \psi$ then δ_1 and δ_2 are said to be minimal if for every model $(s, h, \mathcal{V})$ of $\varphi * \delta_1$ (and also of $\delta_2 * \psi$), for every h_{δ_1}, h_φ and every h_{δ_2}, h_ψ, 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 δ_1 and δ_2 than those already present in ψ and φ, respectively.

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

6.1. Algorithm BiAbduct

We now present a sound algorithm for computing φ_{pre}, φ_{post} and Z in the equivalence $(\exists X\, \widehat{\varphi}) * \varphi_{pre} \Leftrightarrow \exists Z\, (\varphi_{post} * \varphi)$ in the premise of the Compose and InductQ rules. Simplifying notation, the problem can be stated as follows: given variable sets mod_1 and mod_2, and two $\mathcal{LISF}$ 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 φ_{pre}, φ_{post}, and a set $Z \subseteq X \cup Y$ such that (i) $(\exists X\, \varphi) * \varphi_{pre} \Leftrightarrow \exists Z\, (\varphi_{post} * \psi)$, (ii) $free(\varphi_{pre}) \cap mod_1 = \emptyset$, and (iii) $free(\varphi_{post}) \cap mod_2 = \emptyset$.

Our strong bi-abduction algorithm, BiAbduct, is presented in Figure 14. We first illustrate the intuition of BiAbduct using our running example: $\varphi \equiv v \mapsto \_b$, $\psi \equiv v = \_c \wedge \_c \mapsto \_d$, $V = \{v\}$, $W = \{\}$, $X = \{\_b\}$, $Y = \{\_c, \_d\}$ and $mod_1 = mod_2 = \{v\}$. As explained previously, Decompose(φ, ψ) returns the decomposition $(v = \_c \wedge \_b = \_d \wedge \mathbf{emp}, v = \_c \wedge \_b = \_d \wedge \mathbf{emp})$. Thus, we have $\varphi * (v = \_c \wedge \_b = \_d \wedge \mathbf{emp}) \Leftrightarrow (v = \_c \wedge \_b = \_d \wedge \mathbf{emp}) * \psi$. We explain the intuition of our strong bi-abduction algorithm in the following three steps.

—We want φ_{pre} and φ_{post} to be independent mod_1 and mod_2, respectively. To do this, we use the equalities involving mod_1 variables in φ (respectively, mod_2 variables in ψ) to eliminate mod_1 (respectively, mod_2) variables from φ_{pre} (respectively, φ_{post}). In our current example, we replace $v \in mod_2$ by $\_c$ in φ_{post} since ψ contains the equality $v = \_c$. Hence, we obtain $\varphi * (v = \_c \wedge \_b = \_d \wedge \mathbf{emp}) \Leftrightarrow (\_b = \_d \wedge \mathbf{emp}) * \psi$. However, using this transformation we cannot make φ_{pre} independent of v, since φ does not have any equalities involving v.

—In order to make φ_{pre} independent of mod_1 variables we existentially quantify the auxiliary variables that are equated to mod_1 variables in φ_{pre} from both sides of the equivalence. In our current example, we existentially quantify $\_c$ from both sides of the equivalence. As a consequence we can drop the equality $v = \_c$ involving the auxiliary variable $\_c$ from φ_{pre}, thus making φ_{pre} independent of v. We now obtain the equivalence $\varphi * (\_b = \_d \wedge \mathbf{emp}) \Leftrightarrow \exists \_c\, (\_b = \_d \wedge \mathbf{emp}) * \psi$.

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

The preceding intuitions are formalized in the procedure BiAbduct given in Figure 14. The key step of bi-abduction is the Decompose procedure described previously. For each pair (δ_1, δ_2) returned by Decompose(φ, ψ), we compute δ'_1 and δ'_2 from δ_1 and δ_2, respectively, using the function RemoveVar (lines 3, 4). The function RemoveVar$(\phi_1, \phi_2, mod_i, B)$ replaces every free variable $v \in mod_i$ in ϕ_1 by e if ϕ_2 implies $v = e$ and $free(e) \in B \setminus mod_i$. After renaming, it also removes any redundant equalities of the form $x = x$, and equalities implied by ϕ_2^p from ϕ_1. For our running example, $\delta_1 \equiv v = \_c \wedge \_b = \_d$ and $\delta_2 \equiv v = \_c \wedge \_b = \_d$. RemoveVar$(\delta_2, \psi, mod_2, V \cup Y)$ renames v by $\_c$ in δ_2, hence $\delta'_2 \equiv \_b = \_d$. RemoveVar$(\delta_1, \varphi, mod_1, V \cup W)$ does not rename any variables from δ_1, hence $\delta_1 \equiv \delta'_1 \equiv v = \_c \wedge \_b = \_d$.

Next, we process the formula δ'_1 so as to make it independent of mod_1. In line 5, we compute a renaming $\gamma : \langle Y \hookrightarrow mod_1 \rangle$ such that $\delta'_1\gamma$ is independent of mod_1 variables. This is done by invoking function ComputeRenaming. The function ComputeRenaming(ϕ, A, B) renames a variable $a \in A$ by $b \in B$ if ϕ^p implies the equality $a = b$. The renaming γ ensures that $\varphi * \kappa_1 \Leftrightarrow \exists\hat{Z}\,(\delta'_2 * \psi)$, where $\kappa_1 \equiv \delta'_1\gamma$ and $\hat{Z} = dom(\gamma)$. If $\delta'_1\gamma$ is not independent of mod_1 or δ'_2 is not independent of mod_2, we discard the pair (δ'_1, δ'_2) (line 8). Note the asymmetry in dealing with δ'_1 and δ'_2, which stems from the asymmetric structure ($\exists Z$ only on right side) of the required solution $(\exists X\, \varphi) * \varphi_{pre} \Leftrightarrow \exists Z\, (\varphi_{post} * \psi)$. For our running example, $\hat{Z} = \{\_c\}$ and $\gamma : \langle \_c \rightarrow v \rangle$ gives a valid renaming, since $\delta'_1\gamma \equiv \_b = \_d$ is independent of v.

LEMMA 6.3. *Every κ_1 and $\hat{Z}$ computed in lines 6 and 7 of* BiAbduct *satisfy* $\varphi * \kappa_1 \Leftrightarrow \exists\hat{Z}\,(\delta'_2 * \psi)$.

PROOF. Follows from the following equivalences.

1. $\exists\hat{Z}\,\varphi * \delta'_1 \Leftrightarrow \exists\hat{Z}\,\delta'_2 * \psi$	Definition of Decompose and RemoveVar, and $\exists$ elimination
2. $\exists\hat{Z}\,\varphi * \delta'_1 \Leftrightarrow \varphi * \delta'_1\gamma$	$\exists\hat{Z}\,\delta'_1 \Leftrightarrow \delta'_1\gamma$, and φ is independent of $\hat{Z}$ variables
3. $\varphi * \delta'_1\gamma \Leftrightarrow \exists\hat{Z}\,\delta'_2 * \psi$	from 1,2 □

For every κ_1, at line 9 we compute a renaming $\theta : \langle \widetilde{Z} \rightarrow X \rangle$, where $\widetilde{Z} \subseteq Y$, so as to render $\kappa_1\theta$ independent of X (lines 9, 10, 11). The function ComputeRenaming(κ_1, Y, X) computes the renaming θ. Let $\bar{\theta} : \langle X \hookrightarrow \widetilde{Z} \rangle$ be a renaming such that $\bar{\theta}(x) = z$ only if $\theta(z) = x$. The function RemoveRedundant(ϕ_1, ϕ_2^p) removes the equalities from ϕ_1 that are implied by ϕ_2^p. It also removes trivial equalities like $x = x$ or RP$(\mathbf{x}[\cdot] = \mathbf{x}[\cdot])$ from ϕ_1. If $\kappa'_1 =$ RemoveRedundant$(\kappa_1\theta, \varphi^p)$ is independent of X then BiAbduct returns $(\kappa'_1, \kappa_2, \widetilde{Z} \cup \hat{Z})$, where κ_2 is the formula returned by RemoveRedundant$(\delta'_2\bar{\theta}, \psi^p)$, as a solution of strong bi-abduction.

The invocations of ComputeRenaming in lines 5 and 9 have one important difference: in line 5 only non-array variables in mod_1 are renamed, whereas in line 9 array variables in Y may be renamed. The function ComputeRenaming(ϕ, A, B) renames array variables as follows. An array variable $a \in A$ is renamed to another array variable $b \in B$ if ϕ^p implies one of the following facts: (i) RP$(\mathbf{A}[\cdot] = \mathbf{D}[\cdot]) \wedge \mathbf{A}[\$0] = \mathbf{D}[\$0]$, or (ii) RP$(\mathbf{A}[\cdot + 1] = \mathbf{D}[\cdot + 1]) \wedge \mathbf{A}[0] = \mathbf{D}[0]$, or (iii) RP$(\mathbf{A}[\cdot] = \mathbf{D}[\cdot] \wedge \mathbf{A}[\cdot + 1] = \mathbf{D}[\cdot + 1])$. Higher dimensional arrays can be renamed by performing similar checks for each dimension. For our running example, we have $X = \{\_b\}$, $\widetilde{Z} = \{\_d\}$ and $\theta : \langle \_d \rightarrow \_b \rangle$. It is evident that $(\exists \_b\ v \mapsto \_b) * (\mathbf{true} \wedge \mathbf{emp}) \Leftrightarrow \exists \_c, \_d\ (\mathbf{true} \wedge \mathbf{emp}) * (v = \_c \wedge \_c \mapsto \_d)$. Thus, $\varphi_{pre} \equiv \kappa'_1 \equiv$ RemoveRedundant$(\kappa_1\theta, \varphi^p) \equiv \mathbf{true} \wedge \mathbf{emp}$, $\varphi_{post} \equiv \kappa_2 \equiv$ RemoveRedundant$(\delta'_2\bar{\theta}, \psi^p) \equiv \mathbf{true} \wedge \mathbf{emp}$, and and $Z = \{\_c, \_d\}$ is a solution of strong bi-abduction between $\exists \_b\ \varphi \equiv \exists \_b\ v \mapsto \_b$ and $\psi \equiv v = \_c \wedge \_c \mapsto \_d$.

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

PROOF. Follows from the following equivalences.

1.	$\varphi * \kappa_1 \Leftrightarrow \exists \hat{Z}\, \delta_2' * \psi$	from previous step
2.	$\exists \widetilde{Z}(\varphi * \kappa_1) \Leftrightarrow \exists \hat{Z}, \widetilde{Z}\,(\delta_2' * \psi)$	quantify $\widetilde{Z}$
3.	$\exists \widetilde{Z}\,(\varphi * \kappa_1) \Leftrightarrow \varphi * \kappa_1\theta$	φ independent of $\widetilde{Z}$, and $\exists \widetilde{Z}\,\kappa_1 \Leftrightarrow \kappa_1\theta$
4.	$\varphi * \kappa_1\theta \Leftrightarrow \exists \hat{Z}, \widetilde{Z}\,\delta_2' * \psi$	from 2,3
5.	$\varphi * \kappa_1\theta \Leftrightarrow \varphi * \text{RemoveRedundant}(\kappa_1\theta, \varphi^p)$	definition of RemoveRedundant
6.	$\varphi * \kappa_1\theta \Leftrightarrow \varphi * \kappa_1'$	$\kappa_1' \Leftrightarrow \text{RemoveRedundant}(\kappa_1\theta, \varphi^p)$
7.	$\exists X\,\varphi * \kappa_1' \Leftrightarrow \exists \hat{Z}, \widetilde{Z}, X\,(\delta_2' * \psi)$	from 4 and 6
8.	κ_1' *and* ψ *are independent of* X	*assumption*
9.	$(\exists X\,\varphi) * \kappa_1' \Leftrightarrow \exists \hat{Z}, \widetilde{Z}\,(\delta_2'\bar{\theta} * \psi)$	$\exists X\,\delta_2' \Leftrightarrow \delta_2'\bar{\theta}$
10.	$(\exists X\,\varphi) * \kappa_1' \Leftrightarrow \exists \hat{Z}, \widetilde{Z}\,(\kappa_2 * \psi)$	from 9 and definition of κ_2 □

Example 5. Let us compute strong bi-abduction between $\exists X\ \varphi \equiv \exists \mathbf{x}\ h = \mathbf{x}[0] \wedge \text{RS}(\mathbf{x}[\cdot] \mapsto \mathbf{x}[\cdot+1]) \wedge \mathbf{x}[\$0] = \mathbf{null}$ and $\psi \equiv h = \mathbf{Y}[0] \wedge \text{RS}(\mathbf{Y}[\cdot] \mapsto \mathbf{Y}[\cdot+1]) \wedge \mathbf{Y}[\$0] = \mathbf{null}$. Let the sets mod_1 and mod_2 be empty

—The Match procedure finds the following overlap between φ and ψ: $(M, C, \mathbf{emp}, \mathbf{emp})$ where M is $\text{RP}(\mathbf{x}[\cdot] = \mathbf{Y}[\cdot] \wedge \mathbf{x}[\cdot+1] = \mathbf{Y}[\cdot+1])$ and C is $\text{RS}(\mathbf{x}[\cdot] \mapsto \mathbf{x}[\cdot+1])$. Hence δ_1 is computed as $M \wedge h = \mathbf{Y}[0] \wedge \mathbf{Y}[\$0] = \mathbf{null} \wedge \mathbf{emp}$ and δ_2 is computed as $M \wedge h = \mathbf{x}[0] \wedge \mathbf{x}[\$0] = \mathbf{null} \wedge \mathbf{emp}$, thus giving the equivalence $\varphi * \delta_1 \Leftrightarrow \delta_2 * \psi$.
—Since the *mod* set is empty, γ is an empty renaming and $\hat{Z}$ is an empty set.
—The set of quantified variables X contains the array variable $\mathbf{x}$. We compute the renaming θ as $\langle \mathbf{Y} \rightarrow \mathbf{x} \rangle$, from the predicate $\text{RP}(\mathbf{x}[\cdot] = \mathbf{Y}[\cdot] \wedge \mathbf{x}[\cdot+1] = \mathbf{Y}[\cdot+1])$ present in δ_1. $\delta_1\theta$ is the formula $\text{RP}(\mathbf{x}[\cdot] = \mathbf{x}[\cdot] \wedge \mathbf{x}[\cdot+1] = \mathbf{x}[\cdot+1]) \wedge h = \mathbf{x}[0] \wedge \mathbf{x}[\$0] = \mathbf{null} \wedge \mathbf{emp}$. RemoveRedundant$(\delta_1\theta, \varphi^p)$ eliminates the redundant equalities from $\delta_1\theta$ and returns the formula $\mathbf{true} \wedge \mathbf{emp}$ that is independent of $\mathbf{x}$. $\bar{\theta}$ is $\langle \mathbf{x} \rightarrow \mathbf{Y} \rangle$ and $\delta_2\bar{\theta}$ is the formula $\text{RP}(\mathbf{Y}[\cdot] = \mathbf{Y}[\cdot] \wedge \mathbf{Y}[\cdot+1] = \mathbf{Y}[\cdot+1]) \wedge h = \mathbf{Y}[0] \wedge \mathbf{Y}[\$0] = \mathbf{null} \wedge \mathbf{emp}$, and RemoveRedundant$(\delta_2\bar{\theta}, \psi^p)$ removes the redundant equalities and returns the formula $\mathbf{true} \wedge \mathbf{emp}$. Hence, the result of strong bi-abduction is $(\exists \mathbf{x}\,\varphi) * \mathbf{true} \wedge \mathbf{emp} \Leftrightarrow \exists \mathbf{Y}\,(\mathbf{true} \wedge \mathbf{emp} * \psi)$.

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, S1, S2)`. The premises of JOIN require us to check whether $\varphi_1\mu \Leftrightarrow \varphi_2$ and $\widehat{\varphi}_1\mu \Rightarrow \widehat{\varphi}_2$ for quantifier free $\mathcal{LISF}$ formulas $\widehat{\varphi}_1, \widehat{\varphi}_2, \varphi_1$, and φ_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* ψ *and* $\widehat{\psi}$.

(1) *if* $\psi\ *\ (\mathbf{true} \wedge \mathbf{emp}) \Leftrightarrow (\mathbf{true} \wedge \mathbf{emp})\ *\ \widehat{\psi}$, *then* $\psi \Leftrightarrow \widehat{\psi}$
(2) *if* $\psi\ *\ (\mathbf{true} \wedge \mathbf{emp}) \Leftrightarrow (P \wedge \mathbf{emp})\ *\ \widehat{\psi}$, *then* $\psi \Rightarrow \widehat{\psi}$

In order to check whether $\varphi_1\mu \Leftrightarrow \varphi_2$, where φ_1 is a formula over free variables V, W and φ_2 is a formula over free variables V, Y, we call BiAbduct$(\exists W\ \varphi_1, \varphi_2, V, V)$. 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* BiAbduct *(Figure* 14*) is* $\emptyset$, *and* θ *computed at line 9 of* BiAbduct *is such that* $\kappa_1\bar{\theta}$ *and* $\delta_2'\bar{\theta}$ *are both equivalent to* $\mathbf{true} \wedge \mathbf{emp}$, *then we can infer* $\varphi_1\bar{\theta} \Leftrightarrow \varphi_2$.

PROOF. Follows from the following equivalences.

1. $\varphi_1 * \kappa_1 \Leftrightarrow \delta'_2 * \varphi_2$	From Lemma 6.3 and since $\hat{Z}$ is $\emptyset$
2. $\varphi_1\bar{\theta} * \kappa_1\bar{\theta} \Leftrightarrow \delta'_2\bar{\theta} * \varphi_2\bar{\theta}$	Apply renaming $\bar{\theta}$
3. $\varphi_1\bar{\theta} * (\mathbf{true} \wedge \mathbf{emp}) \Leftrightarrow (\mathbf{true} \wedge \mathbf{emp}) * \varphi_2$	φ_2 is indep. of $dom(\bar{\theta})$ and $\kappa_1\bar{\theta} \equiv \delta'_2\bar{\theta} \equiv$ $\mathbf{true} \wedge \mathbf{emp}$
4. $\varphi_1\bar{\theta} \Leftrightarrow \varphi_2$	Proposition 6.1 □

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

LEMMA 6.6. *If δ'_1 computed at line 3 of* BiAbduct *is equivalent to* $\mathbf{true} \wedge \mathbf{emp}$ *and* δ'_2 *computed at line 4 of* BiAbduct *is equivalent to* $P \wedge \mathbf{emp}$*, then we can infer* $\widehat{\varphi}_1\bar{\theta} \Rightarrow \widehat{\varphi}_2$.

PROOF. If δ'_1 is $\mathbf{true} \wedge \mathbf{emp}$, then γ computed at line 5 of BiAbduct is an empty renaming (by the definition of ComputeRenaming). Hence, the set $\hat{Z}$ computed at line 7 of BiAbduct is an empty set. Therefore, by Lemma 6.3, we have $\widehat{\varphi}_1\bar{\theta} * (\mathbf{true} \wedge \mathbf{emp}) \Leftrightarrow (P \wedge \mathbf{emp}) * \widehat{\varphi}_2$. The proof now follows from 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 $\mathcal{LISF}$ formulas φ_{pre} and φ_{post} and a set Z of auxiliary variables for input $\mathcal{LISF}$ formulas $\exists X\, \varphi$ and ψ such that $\exists X\, \varphi * \varphi_{pre} \Leftrightarrow \exists Z\, (\varphi_{post} * \psi)$, the procedure finds such φ_{pre}, φ_{post} and Z. For the $\mathcal{LISF}$ formulas $\varphi : h = \mathbf{x}[0] \wedge \mathsf{RS}(\mathbf{x}[\cdot] \mapsto \mathbf{x}[\cdot+1], 0, 0) \wedge \mathbf{x}[\$0] = \mathbf{null}$ and $\psi : h = \mathbf{y}[\$0] \wedge \mathsf{RS}(\mathbf{y}[\cdot+1] \mapsto \mathbf{y}[\cdot], 0, 0) \wedge \mathbf{y}[0] = \mathbf{null}$, the fact that $\exists \mathbf{x}\, \varphi * (\mathbf{true} \wedge \mathbf{emp}) \Leftrightarrow \exists \mathbf{y}\, ((\mathbf{true} \wedge \mathbf{emp}) * \psi)$ is valid. However, BiAbduct will not be able to compute this strong bi-abduction. This is because the Match procedure cannot find the correct overlap between φ^s and ψ^s. Hence BiAbduct is not a complete strong bi-abduction procedure. The pure constraint expressing the correct overlap between φ^s and ψ^s is not expressible in $\mathcal{LISF}$. In the next section, we present techniques to do sophisticated matching.

7. AN EXTENSION OF $\mathcal{LISF}$

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.

$$\varphi \;:\; h = \mathbf{x}[0] \wedge \mathsf{RS}(\mathbf{x}[\cdot] \mapsto \mathbf{x}[\cdot+1], 0, 0) \wedge \mathbf{x}[\$0] = \mathbf{null}$$
$$\psi \;:\; h \mapsto \mathbf{y}[0] * \mathsf{RS}(\mathbf{y}[\cdot] \mapsto \mathbf{y}[\cdot+1], 0, 0) \wedge \mathbf{y}[\$0] = \mathbf{null}.$$

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

Now consider the same formula φ as described previously and ϕ defined here.

$$\phi \;:\; h = \mathbf{z}[\$0] \wedge \mathsf{RS}(\mathbf{z}[\cdot+1] \mapsto \mathbf{z}[\cdot], 0, 0) \wedge \mathbf{z}[0] = \mathbf{null}$$

The formulas φ and ϕ are different representations for the linked list of any length (including zero) pointed to by h. The length of array $\mathbf{x}$ (respectively, $\mathbf{z}$) in φ (respectively,

ϕ) is one greater than the length of the linked list pointed to by h. The strong bi-abduction of φ and ϕ returns a solution $(\varphi_{pre}, \varphi_{post})$ that restricts the length of linked list in $\varphi * \varphi_{pre}$ (or $\varphi_{post} * \phi$) to one, although both φ and ϕ model linked lists of arbitrary lengths. The reason for this "too restrictive" solution is that $\mathcal{LISF}$ 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 φ and ϕ we enrich $\mathcal{LISF}$ with rev predicate. We describe this enhancement in Section 7.2.

7.1. Enhancement of $\mathcal{LISF}$ with *sub* Predicate

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

To remedy this problem, we introduce a new pure predicate $sub(e, l, u, e')$ where e and e' are two $\mathcal{LISF}$ 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, $sub(e, l, u, e')$ 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 $sub(e, l, u, e')$ is formally defined as follows. Note that we overload the function len defined in Section 4.2 and used in Figure 8 to operate over single expressions instead of pure or spatial formulas.

$$\begin{aligned}(s, h, \mathcal{V}, L) \models sub(e, l, u, e') \text{ iff } & \exists k\; k+1 = len(\mathcal{V}, L, e') \wedge len(\mathcal{V}, L, e) > l + u \wedge \\ & len(\mathcal{V}, L, e') = len(\mathcal{V}, L, e) - l - u \wedge \\ & \forall 0 \leq i \leq k.\; \mathcal{E}_a(e, (i+l) :: L, s, \mathcal{V}) = \mathcal{E}_a(e', i :: L, s, \mathcal{V}).\end{aligned} \tag{7.6}$$

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

$$\textsc{MatchRsA}\;\frac{k_1 :\ \mathsf{RS}(S_1, 0, 0),\ k_2 :\ \mathsf{RS}(S_2, l, u), \quad (M, C, \{\}, \{\}) \in \mathsf{Match}(S_1, \mathsf{SubS}(S_2, l, u), 1)}{(\mathsf{RP}(M, 0, 0) \wedge \mathsf{SubP}(S_2, l, u), \mathsf{RS}(C, 0, 0), \{\}, \{\}) \in \mathsf{Match}(k_1, k_2, 0)}$$

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 RP predicate we allow introduction of *sub* predicate only while matching RS predicate that 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 $\mathbf{A}$, 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 $\mathbf{A}'$. 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 $\mathbf{A}'$, with the old ones, like $\mathbf{A}$. 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 $sub(\mathsf{lb}(e), l, u, \mathsf{lb}(e'))$ 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$(\mathbf{x}[\cdot] \mapsto \mathbf{x}[\cdot+1], 1, 0)$ returns the spatial formula $\mathbf{x}'[\cdot] \mapsto \mathbf{x}'[\cdot+1]$ and SubP$(\mathbf{x}[\cdot] \mapsto \mathbf{x}[\cdot+1], 1, 0)$ returns the pure formula $sub(\mathbf{x}[\cdot], 1, 0, \mathbf{x}'[\cdot]) \wedge sub(\mathsf{lb}(\mathbf{x}[\cdot+1]), 1, 0, \mathsf{lb}(\mathbf{x}'[\cdot+1]))$. By definition of lb, $sub(\mathsf{lb}(\mathbf{x}[\cdot+1]), 1, 0, \mathsf{lb}(\mathbf{x}'[\cdot+1])) \equiv sub(\mathbf{x}[\cdot], 1, 0, \mathbf{x}'[\cdot])$.

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 MATCHRS 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$(\mathbf{x}[\cdot] \mapsto \mathbf{x}[\cdot+1], 1, 0)$ and RS$(\mathbf{Y}[\cdot] \mapsto \mathbf{Y}[\cdot+1], 0, 0)$, and thus compute Match$(\varphi^s, \psi^s)$ as a set consisting of $(M, \psi^s, \{\}, \{\})$, where M is $h = \mathbf{x}[0] \wedge \mathbf{Y}[0] = \mathbf{x}[1] \wedge \mathsf{RP}(\mathbf{x}'[\cdot] = \mathbf{Y}[\cdot] \wedge \mathbf{x}'[\cdot+1] = \mathbf{Y}[\cdot+1], 0, 0) \wedge sub(\mathbf{x}[\cdot], 1, 0, \mathbf{x}'[\cdot])$. This match is consistent with φ^p and ψ^p. Hence, the procedure Decompose computes δ_1 as $M \wedge \mathbf{Y}[\$0] = \mathbf{null} \wedge \mathbf{emp}$ and δ_2 as $M \wedge h = \mathbf{x}[0] \wedge \mathbf{x}[\$0] = \mathbf{null} \wedge \mathbf{emp}$, 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 that is not expressible in $\mathcal{LISF}$. $\mathcal{LISF}$ can express only equality of array lengths.

7.2. Enhancement of $\mathcal{LISF}$ with *rev* Predicate

Consider the formulas φ and ϕ defined at the start of Section 7. The Match algorithm will match the RS predicates in φ^s and ϕ^s and return the four-tuple $(M, \varphi^s, \{\}, \{\})$ as the only solution, where M is the pure formula RP$(\mathbf{x}[\cdot] = \mathbf{z}[\cdot+1] \wedge \mathbf{z}[\cdot] = \mathbf{x}[\cdot+1], 0, 0)$. But this too-restrictive constraint restricts the length of the matched list to be ≤ 1,

$$\text{MATCHRSB}\frac{\begin{array}{c} k_1: \mathsf{RS}(S_1, l, u), k_2: \mathsf{RS}(S_2, u, l), \\ (M, C, \{\}, \{\}) \in \mathsf{Match}(S_1, \mathsf{RevS}(S_2), 1) \end{array}}{(\mathsf{RP}(M, l, u) \wedge \mathsf{RevP}(S_2), \mathsf{RS}(C, l, u), \{\}, \{\}) \in \mathsf{Match}(k_1, k_2, 0)}.$$

Although φ and ϕ represent a same set of structures in the heap, bi-abduction of φ and ϕ generates constraints that reduce this set of structures. This is because the pure constraint describing the overlap of a list expressed as RS$(\mathbf{x}[\cdot] \mapsto \mathbf{x}[\cdot+1], 0, 0)$ and the same list expressed as RS$(\mathbf{z}[\cdot+1] \mapsto \mathbf{z}[\cdot], 0, 0)$ cannot be expressed in $\mathcal{LISF}$ without restricting the lengths of $\mathbf{x}$ and $\mathbf{z}$. To remedy this problem, we introduce a new predicate $rev(e, e')$ where e and e' are $\mathcal{LISF}$ expressions that differ only in the array name. The semantics of $rev(e, e')$ is defined as follows

$$(s, h, \mathcal{V}, L) \models rev(e, e') \text{ iff } \exists k.\ k+1 = len(\mathcal{V}, L, e') = len(\mathcal{V}, L, e) \wedge \forall 0 \leq i \leq k.\ \mathcal{E}_a(e, i :: L, s, \mathcal{V}) = \mathcal{E}_a(e', (k-i) :: L, s, \mathcal{V}). \quad (7.7)$$

For example, the predicate $rev(\mathbf{x}[\cdot], \mathbf{z}[\cdot])$ asserts that $\mathbf{x}$ and $\mathbf{z}$ are arrays of same lengths and that the sequence $\mathbf{x}[0], \mathbf{x}[1], \ldots, \mathbf{x}[\$0]$ is same as $\mathbf{z}[\$0], \mathbf{z}[\$1], \ldots, \mathbf{z}[0]$.

The *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 *rev* predicate to match an array with the reverse of another array. To avoid nesting of the *rev* predicate within a RP predicate, we allow introduction of *rev* predicate only while matching RS predicates that are not nested within another RS predicate.

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

PROPOSITION 7.2. *For a predicate* $\mathsf{RS}(S, l, u)$ *not embedded in any* RS *predicate,* $\mathsf{RS}(S, l, u) \wedge \mathsf{RevP}(S) \Leftrightarrow \mathsf{RS}(\mathsf{RevS}(S), u, l) \wedge \mathsf{RevP}(S)$.

We extend the rule MATCHRS in Match algorithm to the rule MATCHRSB that uses *rev* predicate to match two RS predicates. We can now use the rule MATCHRSB to match $\mathsf{RS}(\mathbf{x}[\cdot] \mapsto \mathbf{x}[\cdot+1], 0, 0)$ and $\mathsf{RS}(\mathbf{z}[\cdot+1] \mapsto \mathbf{z}[\cdot], 0, 0)$, and thus compute $\mathsf{Match}(\varphi^s, \phi^s)$ as $(M, \varphi^s, \{\}, \{\})$, where M is $\mathsf{RP}(\mathbf{x}[\cdot] = \mathbf{z}'[\cdot] \wedge \mathbf{x}[\cdot+1] = \mathbf{z}'[\cdot+1], 0, 0) \wedge rev(\mathbf{z}[\cdot], \mathbf{z}'[\cdot+1])$. This match is consistent with φ^p and ϕ^p. Hence, the procedure Decompose computes δ_1 as $M \wedge h = \mathbf{z}[\$0] \wedge \mathbf{z}[0] = \mathbf{null} \wedge \mathbf{emp}$ and δ_2 as $M \wedge h = \mathbf{x}[0] \wedge \mathbf{x}[\$0] = \mathbf{null} \wedge \mathbf{emp}$, such that $\varphi * \delta_1 \Leftrightarrow \delta_2 * \phi$.

The use of *rev* predicates allows us to equate array elements that are arbitrary distance apart (e.g., i and $k - i$ in Eq. (7.7)). $\mathcal{LISF}$ 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) $\mathcal{LISF}$ formulas, and (b) $\mathcal{LISF}$ extended with *sub* and *rev* predicates. Any $\mathcal{LISF}$ formula is of the form $P \wedge S$ or $\exists X.\ P \wedge S$. Since $\exists X.\ P \wedge S$ is equisatisfiable with $P \wedge S$, we present satisfiability procedures only for quantifier free $\mathcal{LISF}$ formulas.

8.1. Satisfiability Checking Procedure for $\mathcal{LISF}$

The basic idea of the satisfiability checking procedure is to convert a $\mathcal{LISF}$ 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 RP and 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.

```
sat(φ)
1: $lentbl \leftarrow$ GetLengths($\varphi$)
2: $\psi \leftarrow$ Flatten($\varphi, lentbl$)
3: return sat_sep($\psi$)

GetLengths(φ)
1: $F \leftarrow 0 = 0$
2: for all $(\mathbf{X}, i, l) \in$ LB($\varphi$) do
3:     $F \leftarrow F \wedge (l + 1 \leq \langle \mathbf{X}, i \rangle)$
4: for all $(\mathbf{X}, i, u) \in$ UB($\varphi$) do
5:     $F \leftarrow F \wedge (u + 1 \leq \langle \mathbf{X}, i \rangle)$
6: for all $(\langle \mathbf{X}, i \rangle, \langle \mathbf{Y}, j \rangle) \in$ IterConstr($\varphi$) do
7:     $F \leftarrow F \wedge (\langle \mathbf{X}, i \rangle = \langle \mathbf{Y}, j \rangle)$
8: return Solve($F$)

Flatten(φ, lentbl)
1: while ¬ isFlat($\varphi$) do
2:     for all top-level terms $t$ : RP($\ldots, l, u$) or
       RS($\ldots, l, u$) in $\varphi$ do
3:         $len \leftarrow$ FindLength($t, lentbl$)
4:         $cnt \leftarrow max(\{len - 1 - l - u, 0\})$
5:         $t' \leftarrow$ iter_unroll$_f$($t, cnt$)
6:         replace $t$ with $t'$ in $\varphi$
7: end while
8: ModifyUB($\varphi, lentbl$)
9: return $\varphi$
```

Fig. 16. Satisfiability procedure: sat(φ).

Example 6. Consider a $\mathcal{LISF}$ formula $\varphi \equiv (h = \mathbf{x}[0]) \wedge (g = \mathbf{y}[0]) \wedge (t = \mathbf{x}[\$1]) \wedge (\mathbf{x}[\$0] = \mathbf{y}[\$0]) \wedge (\mathbf{y}[\$0] = \mathbf{null}) \wedge \mathsf{RS}(\mathbf{x}[\cdot] \mapsto \mathbf{x}[\cdot + 1] * \mathbf{y}[\cdot] \mapsto \mathbf{y}[\cdot + 1], 0, 0)$. The RS predicate in φ requires that **x** and **y** have same lengths. The expressions $\mathbf{x}[0]$ and $\mathbf{x}[\$0]$ (respectively, $\mathbf{y}[0]$ and $\mathbf{y}[\$0]$) require that the length of array **x** (respectively, array **y**) be at least 1. Similarly, the expression $\mathbf{x}[\$1]$ requires that the length of **x** be at least 2. A sound way of checking the satisfiability of φ is to guess the lengths of the arrays and expand the RS and RP predicates for these array lengths so as to obtain a standard separation logic formula (one without RS or 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 φ. If the length of array **x** is 2, we have $\mathbf{x}[0] = \mathbf{x}[\$1]$ and $\mathbf{x}[1] = \mathbf{x}[\$0]$. Similarly, if length of **y** is 2, we have $\mathbf{y}[\$0] = \mathbf{y}[1]$. Moreover, the predicate $\mathsf{RS}(\mathbf{x}[\cdot] \mapsto \mathbf{x}[\cdot + 1] * \mathbf{y}[\cdot] \mapsto \mathbf{y}[\cdot + 1], 0, 0)$ can be written as $\mathbf{x}[0] \mapsto \mathbf{x}[1] * \mathbf{y}[0] \mapsto \mathbf{y}[1]$, by applying the semantic definition of RS (given in Figure 8). Hence, if we set the lengths of **x** and **y** to 2, we can rewrite φ as $\psi \equiv h = \mathbf{x}[0] \wedge g = \mathbf{y}[0] \wedge t = \mathbf{x}[0] \wedge \mathbf{x}[1] = \mathbf{y}[1] \wedge \mathbf{y}[1] = \mathbf{null} \wedge \mathbf{x}[0] \mapsto \mathbf{x}[1] * \mathbf{y}[0] \mapsto \mathbf{y}[1]$. The only array expressions in ψ are of the form $\mathbf{x}[i]$ or $\mathbf{y}[i]$, $i \in \{0, 1\}$. It has no RS or RP predicates. Hence, it is a standard separation logic formula. It is evident that if ψ is satisfiable then so is φ. The formaula ψ can be satisfied by having $\mathbf{x}[0] = h = t = l_1, \mathbf{y}[0] = g = l_2$ and $\mathbf{x}[1] = \mathbf{y}[1] = \mathbf{null}$, $l_1 \neq l_2$, $h(l_1) = \mathbf{null}$, and $h(l_2) = \mathbf{null}$. Hence, φ is satisfiable.

This intuition is formalized in the satisfiability procedure sat given in Figure 16. The key step of sat procedure is the conversion of an $\mathcal{LISF}$ formula φ to a formula ψ in separation logic without iterated predicates using the Flatten procedure. In order to soundly eliminate iterated predicates from an $\mathcal{LISF}$ formula φ, Flatten requires the lengths of all dimensions of all the array variables in φ. The function GetLengths(φ) computes these lengths. Any model of the flattened formula ψ is also a model of $\mathcal{LISF}$ formula φ. The function sat_sep(ψ) determines the satisfiability of a separation logic formula ψ.

The predicates RS, RP and the expressions with fixed indices in φ 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 $\langle \mathbf{x}, i \rangle$, where **x** is a free k-dimensional array variable in φ and $1 \leq i \leq k$. The variable $\langle \mathbf{x}, i \rangle$ represents a safe length for the ith dimension of **x** that avoids indexing errors. Lines 2–7 add constraints to F so that evaluation of fixed indices in the expressions of φ does not cause an array indexing error. The function LB(φ) returns a set of tuples $(\mathbf{x}, i, l)$ such that there is an expression in φ accessing the ith dimension of array **x** with a fixed index l. Similarly, UB(φ) returns a set of tuples $(\mathbf{x}, i, u)$ such that there is an expression in φ accessing the ith dimension of array **x** with a fixed index $\$u$. The function IterConstr(φ) returns a set of pairs $(\langle \mathbf{x}, i \rangle, \langle \mathbf{y}, j \rangle)$ such that there exist expressions e_1 and e_2 embedded

```
$\mathsf{IterConstr}(\varphi) \stackrel{\text{def}}{=} \mathsf{IterExpr}(\varphi, 1)$
$\mathsf{IterExpr}(\varphi, i) \stackrel{\text{def}}{=}$ match $\varphi$ with
        | $\mathsf{RS}(\psi, l, u)$
        | $\mathsf{RP}(\psi, l, u) \rightarrow \{\langle \mathbf{X}, j\rangle = \langle \mathbf{Y}, k\rangle$  |  $\mathbf{X} = free(e_1), \mathbf{Y} = free(e_2), e_1, e_2 \in \psi$ and
                                                    $j = iterDim(e_1, i), k = iterDim(e_2, i)$ and
                                                    $j, k \geq 0$, and
                  $\} \cup \mathsf{IterExpr}(\psi, i + 1)$
        | _ $\rightarrow \{\}$
```

Fig. 17. Function $\mathsf{IterConstr}(\varphi)$.

```
$\mathsf{iter\_unroll_f}(\mathsf{RP}(P, l, u), c)$ =
if $(c = 0)$ then true else
  $\mathsf{unroll_f}(\mathsf{RP}(P, l, u), 0) \wedge \mathsf{iter\_unroll_f}(\mathsf{RP}(P, l{+}1, u), c{-}1)$

$\mathsf{iter\_unroll_f}(\mathsf{RS}(S, l, u), c)$ =
if $(c = 0)$ then emp else
  $\mathsf{unroll_f}(\mathsf{RS}(S, l, u), 0) * \mathsf{iter\_unroll_f}(\mathsf{RS}(S, l{+}1, u), c{-}1)$
```

Fig. 18. Unroll functions.

in an RS (or RP) predicate such that $free(e_1) = \mathbf{x}$, $free(e_2) = \mathbf{y}$ and i and j are the dimensions of $\mathbf{x}$ and $\mathbf{y}$, respectively, over which the RS (or RP) predicate iterates. Lines 6 and 7 capture constraints imposed by RS and RP predicates on the lengths of array dimensions. The function IterConstr is defined in Figure 17. The function $iterDim(e, i)$ used in Figure 17 returns the dimension number corresponding to the ith iterated index in e if e has at least i iterated indices, otherwise it returns -1. The formula F is always satisfiable as the only constraints it has are of the form $c \leq \langle \mathbf{x}, i\rangle$ or $\langle \mathbf{x}, i\rangle = \langle \mathbf{y}, j\rangle$ (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 Solve(F) returns such an assignment to the variables in F. Any structure having array sizes conforming to *lentbl* returned by GetLengths(φ) (line 1 of sat) is a well-formed structure for φ.

Flatten uses an intermediate function isFlat(φ), which returns true if φ does not have any RS or RP predicate; otherwise, it returns false. The function FindLength(t, *lentbl*), where t is RP(P, l, u) (respectively, RS(S, l, u)), returns the length of array dimension corresponding to the first iterated index of any array expression in P (respectively, S). Flatten then eliminates the iterated predicates t by the function $\mathsf{iter\_unroll_f}(t, cnt)$, which is a repeated application of $\mathsf{unroll_f}(t, 0)$ as defined in Figure 18. Recall that $\mathsf{unroll_f}(\mathsf{RS}(S, l, u), d)$ is defined in Section 6 as the formula obtained by replacing the $(d+1)$th iterated index $[\cdot]$ (respectively, $[\cdot + 1]$) of every expression in S by the fixed index $[l]$ (respectively, $[l + 1]$). The function $\mathsf{unroll_f}(\mathsf{RP}(P, l, u), d)$ is analogously defined. Finally, all expressions that access a dimension, say i, of an array, say $\mathbf{x}$, with a fixed index $\$u$ are modified by replacing $[\$u]$ with $[lentbl(\mathbf{x}, i) - 1 - u]$. The function ModifyUB($\varphi$, *lentbl*) does this transformation.

LEMMA 8.1. *For a $\mathcal{LISF}$ formula φ, if* sat(φ) *returns true, then φ is satisfiable.*

8.2. Satisfiability Checking Procedure for $\mathcal{LISF}$ Extended with *sub* and *rev* Predicates

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

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

```
satL($\varphi$)
  1: $lentbl \leftarrow$ GetLengthsL($\varphi$)
  2: $\psi \leftarrow$ FlattenL($\varphi, lentbl$)
  3: return sat_sep($\psi$)

FlattenL($\varphi, lentbl$)
  1: $lentbl \leftarrow$ GetLengthsL($\varphi$)
  2: $p_1 \leftarrow$ AddRevConstrs($\varphi$)
  3: $p_2 \leftarrow$ AddSubConstrs($\varphi$)
  4: return $p_1 \wedge p_2 \wedge$ Flatten($\varphi, lentbl$)

GetLengthsL($\varphi$)
  1: $F \leftarrow 0 = 0$
  2: for all predicates $sub(e, l, u, e')$ in $\varphi$ do
  3:     $v \leftarrow \langle arr(e), idim(e) \rangle$
  4:     $v' \leftarrow \langle arr(e'), idim(e') \rangle$
  5:     $F \leftarrow F \wedge v' = v - l - u \wedge v > l + u$
  6:     EquateHigher($e, e', F$)
  7: for all predicates $rev(e, e')$ in $\varphi$ do
  8:     $v \leftarrow \langle arr(e), idim(e) \rangle$
  9:     $v' \leftarrow \langle arr(e'), idim(e') \rangle$
 10:     $F \leftarrow F \wedge (v = v')$
 11:     EquateHigher($e, e', F$)
 12: for all $(\mathbf{X}, i, l) \in$ LB($varphi$) do
 13:     $F \leftarrow F \wedge (l + 1 \leq \langle \mathbf{X}, i \rangle)$
 14: for all $(\mathbf{X}, i, u) \in$ UB($varphi$) do
 15:     $F \leftarrow F \wedge (u + 1 \leq \langle \mathbf{X}, i \rangle)$
 16: for all $(\langle \mathbf{X}, i \rangle, \langle \mathbf{Y}, j \rangle) \in$ IterConstr($\varphi$) do
 17:     $F \leftarrow F \wedge (\langle \mathbf{X}, i \rangle = \langle \mathbf{Y}, j \rangle)$
 18: if sat_dc($F$) then
 19:     return SolveDiff($F$)
 20: else
 21:     raise unsat
```

Fig. 19. Satisfiability procedure: satL(φ).

Let $arr(e)$ give the array name used to build the array expression e and $idim(e)$ give the dimension number corresponding to first iterated index in e. The predicate $sub(e, l, u, e')$ requires that the length, len, of dimension $idim(e')$ of $arr(e')$ be equal to length of dimension $idim(e)$ of $arr(e)$ - $(l + u)$ (as defined in Eq. (7.6)). Lines 2-5 add such constraints to F. The predicate $rev(e, e')$ requires that the length of dimension $idim(e)$ of $arr(e)$ be same as the length of dimension $idim(e')$ of $arr(e')$ (as defined in Eq. (7.7)). Lines 7-10 of GetLengthsL add these constraints to F. Suppose for a predicate $sub(e, l, u, e')$ (or $rev(e, e')$), the number of dimensions of $arr(e)$ and $arr(e')$ are k and k', respectively. The definition of sub (respectively, rev) requires that for every $0 \leq j \leq k - idim(e)$, the length of dimension $idim(e) + j$ of $arr(e)$ is same as the length of dimension $idim(e') + j$ of $arr(e')$. The function EquateHigher(e, e', F) adds such constraints to F (lines 6 and 11). Lines 12-17 add constraints imposed on array lengths by RS and RP predicates and expressions with fixed indices. In contrast to constraints obtained in GetLengths, constraints in GetLengthsL may have difference constraints. This is due to the constraints imposed by the predicate $sub(e, l, u, e')$ in line 5. Hence, the formula F may be unsatisfiable. The function sat_dc(F) at line 18 checks whether F is satisfiable. If F is satisfiable, GetLengthsL returns the model constructed by SolveDiff(F) (line 19); otherwise, it raises an an error indicating unsatisfiability of φ (line 21). Any structure having array sizes confirming to $lentbl$ returned by GetLengthsL(φ) is a well-formed structure for φ.

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

LEMMA 8.2. *Given a $\mathcal{LISF}$ formula φ with sub and rev predicates, if* satL(φ) *returns true then φ is satisfiable.*

The satisfiability procedures presented in the previous subsections are sound but incomplete. This is because GetLengths(φ) and GetLengthsL(φ) return only one of the many (possibly infinite) mappings from array dimensions to their lengths. The formula φ may be satisfiable, but not for the array length mappings returned by the function GetLengths or GetLengthsL. In Gulavani et al. [2009], we show that satisfiability checking of a subclass of $\mathcal{LISF}$ having only single dimensional arrays is decidable.

Any formula φ belonging to this subclass is satisfiable iff it is satisfiable for some array length mapping in the finite set M_φ of array length mappings. This means that, if φ is satisfiable, then there exists a model of bounded size. Hence satisfiability checking is decidable for this subclass of $\mathcal{LISF}$. Unfortunately, the size of the finite set is doubly exponential in the size of φ 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 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.[2] 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, that is, 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 Figure 20. Programs in Figure 20(a) are adopted from [Calcagno et al. 2007]. Program `delete` is the same as the motivating example in Section 1. Programs in Figure 20(b) are adopted from Abdulla et al. [2008] and Møller and Schwartzbach [2001]. 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 Figure 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

[2]acronym for **Sp**efication **In**ference **E**ngine.

Progs	size	time(s)	IV	V
init	16	0.007	2	Yes
del-all	21	0.006	2	Yes
del-circ	23	0.007	2	Yes
delete	42	0.058	* 19	No
append	23	0.010	3	Yes
ap-disp	52	0.036	6	Yes
copy	33	0.324	3	Yes
find	28	0.017	4	Yes
insert	53	0.735	6	Yes
merge	60	0.511	12	No
reverse	20	0.012	* 3	No

(a)

Progs	size	time(s)	IV	V
dll-reverse	23	0.084	3	No
fumble	20	0.010	2	Yes
zip	37	0.374	4	No

(b)

Progs	size	time(s)	IV	V
BusReset	145	0.043	* 3	Yes
CancelIrp	87	0.743	* 32	Yes
SetAddress	96	0.122	* 6	Yes
GetAddress	94	0.122	* 6	Yes
PnpRemove	460	37.600	34	No

(c)

Progs	size	time(s)	IV	V
nested	24	0.028	5	Yes
rev-rev	30	0.150	3	No
off-trav	31	0.122	0	No
dll-trav-2	24	0.126	2	No

(d)

Fig. 20. Experimental results on (a) list manipulating examples from Calcagno et al. [2007], (b) examples from Abdulla et al. [2008] and 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, 1GB RAM.

Progs	size	time(s)	IV	V
delete	42	0.082	* 21	No
rev-rev	30	0.025	4	No
off-trav	31	0.016	1	Yes
dll-trav-2	24	0.014	3	Yes
PnpRemove	460	23.800	* 32	Yes

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

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 susmmaries output by SpInE.[3] 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 Figure 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 Figure 21.

[3]Available to the interested readers at http://www.cfdvs.iitb.ac.in/~bhargav/spine.html.

Complete specification can now be generated for programs `off-trav` and `PnpRemove`. The use of *rev* (respectively, *sub*) predicate was instrumental for generating richer specifications for `rev-rev` and `dll-trav-2` (respectively, `off-trav`). The use of JOIN rule was instrumental for generating complete specification for `PnpRemove`. `PnpRemove` has several nested branching constructs of the form `if (v !=` **null**`) delete(v)` inside while loops. The use of JOIN rule enabled 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 `PnpRemove`. With the options `-lemma` and `-join`, 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 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 $\mathcal{LISF}$ to express the Hoare triples. $\mathcal{LISF}$ 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 $\mathcal{LISF}$ 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 $\mathcal{LISF}$ formulas. Although neither of these procedures are complete, we identify a fragment of $\mathcal{LISF}$ that has a small model property. Hence, checking satisfiability of this fragment is decidable. But, its worst, case complexity is doubly exponential. Second, we do not yet know whether the satisfiability checking of entire $\mathcal{LISF}$ is decidable. Hence, we use the sound procedure sat in our implementation for checking satisfiability of $\mathcal{LISF}$ formulas.

One possible direction for future work is to enhance the strong bi-abduction procedure to make it complete for an expressive fragment of $\mathcal{LISF}$. 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 the future, we would like to extend our technique to generate expressive specifications for programs having recursive procedures and those manipulating tree-like data-structures.

APPENDIX

A. COMPOSITION OF STRONG HOARE TRIPLES USING STRONG BI-ABDUCTION

Let $\mathsf{Post}(\mathtt{S}, (s, h))$ 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 $(s', h') \in \mathsf{Post}(\mathtt{S}, (s, h))$, we have $dom(h') \supseteq dom(h)$. A program statement `S` satisfies *minimal resource* property if $(s', h') \in \mathsf{Post}(\mathtt{S}, (s, h))$ implies that for all h_0 disjoint from h and h', $(s', h' \sqcup h_0) \in \mathsf{Post}(\mathtt{S}, (s, h \sqcup h_0))$. 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.

Note that although the program fragment $\mathtt{S} : \mathtt{x} := \mathtt{new};\ \mathtt{dispose\ x}$ satisfies the domain expansion property, it does not satisfy the minimal resource property. This can be shown as follows. Consider $(s', h') \in \mathsf{Post}(\mathtt{S}, (s, h))$, where $s'(x) = s'(y) = l'$, $s(x) = l, s(y) = l'$, and $dom(h') = dom(h) = \emptyset$. Let $dom(h_0) = \{l'\}$. Starting from a state $(s, h \sqcup h_0)$, execution of S cannot result in a state $(s', h' \sqcup h_0)$ because the statement `x := new` cannot allocate a new object at an already allocated location $l' \in dom(h \sqcup h_0)$. Hence, $(s', h' \sqcup h_0) \notin \mathsf{Post}(\mathtt{S}, (s, h \sqcup h_0))$, although h_0 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-abudction yields strong Hoare triples.

LEMMA A.1. *If statements* $\mathtt{S}_1$ *and* $\mathtt{S}_2$ *satisfy domain expansion and minimal resource properties, then their composition* $\mathtt{S}_1; \mathtt{S}_2$ *also does.*

PROOF. Consider $(s'', h'') \in \mathsf{Post}(\mathtt{S}_1; \mathtt{S}_2, (s, h))$. Let (s', h') be an intermediate state such that $(s', h') \in \mathsf{Post}(\mathtt{S}_1, (s, h))$ and $(s'', h'') \in \mathsf{Post}(\mathtt{S}_2, (s', h'))$. Since $\mathtt{S}_1$ and $\mathtt{S}_2$ both satisfy domain expansion property, it follows that $dom(h'') \supseteq dom(h') \supseteq dom(h)$. Hence, $\mathtt{S}_1; \mathtt{S}_2$ satisfies the domain expansion property.

Consider a trace starting from (s, h) such that $(s', h') \in \mathsf{Post}(\mathtt{S}_1, (s, h))$ and $(s'', h'') \in \mathsf{Post}(\mathtt{S}_2, (s', h'))$. By the domain expansion property, we have $dom(h'') \supseteq dom(h') \supseteq dom(h)$. Hence, for all h_0 such that $h_0\#h''$, we have $h_0\#h'$ and $h_0\#h$. Combining these with the fact that both $\mathtt{S}_1$ and $\mathtt{S}_2$ satisfy minimal resource property, we obtain that for all h_0 such that $h_0\#h''$, $(s'', h_0 \sqcup h'') \in \mathsf{Post}(\mathtt{S}_2, (s', h_0 \sqcup h'))$ and $(s', h_0 \sqcup h') \in \mathsf{Post}(\mathtt{S}_1, (s, h_0 \sqcup h))$. Hence, $\mathtt{S}_1; \mathtt{S}_2$ 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* $\mathtt{S}_1$ *satisfies domain expansion and minimal resource properties,* $[\varphi_1]\ \mathtt{S}_1\ [\widehat{\varphi}_1]$ *is a strong Hoare triple, and* $\varphi_{pre} \cap mod(\mathtt{S}_1) = \emptyset$ *then* $[\varphi_1 * \varphi_{pre}]\ \mathtt{S}_1\ [\widehat{\varphi}_1 * \varphi_{pre}]$ *is a strong Hoare triple.*

PROOF. By frame rule, it is evident that $[\varphi_1 * \varphi_{pre}]\ \mathtt{S}_1\ [\widehat{\varphi}_1 * \varphi_{pre}]$ is a valid Hoare triple.

We now show that $[\varphi_1 * \varphi_{pre}]\ \mathtt{S}_1\ [\widehat{\varphi}_1 * \varphi_{pre}]$ is strong. Consider $(s, h) \models \widehat{\varphi}_1 * \varphi_{pre}$. Let $h = h_1\#h_2$ such that $(s, h_1) \models \widehat{\varphi}_1$ and $(s, h_2) \models \varphi_{pre}$. Since $[\varphi_1]\ \mathtt{S}_1\ [\widehat{\varphi}_1]$ is a strong Hoare triple, there exists $(s', h'_1) \models \varphi_1$ such that $(s, h_1) \in \mathsf{Post}(\mathtt{S}_1, (s', h'_1))$. Since s and s' map variables other than $mod(\mathtt{S}_1)$ to same values, and since φ_{pre} is independent of $mod(\mathtt{S}_1)$, it follows that $(s', h_2) \models \varphi_{pre}$. Moreover, since $\mathtt{S}_1$ satisfies domain expansion property, $dom(h'_1) \subseteq dom(h_1)$ and hence $h'_1\#h_2$. Consequently, $(s', h'_1 \sqcup h_2) \models \varphi_1 * \varphi_{pre}$. Furthermore, since $\mathtt{S}_1$ satisfies minimal resource property, $(s, h_1 \sqcup h_2) \in \mathsf{Post}(\mathtt{S}_1, (s', h'_1 \sqcup h_2))$. Thus, for every $(s, h) \models \widehat{\varphi}_1 * \varphi_{pre}$, there exists $(s', h') \models \varphi_1 * \varphi_{pre}$ such that $(s, h) \in \mathsf{Post}(\mathtt{S}_1, (s', h'))$. Hence, $[\varphi_1 * \varphi_{pre}]\ \mathtt{S}_1\ [\widehat{\varphi}_1 * \varphi_{pre}]$ is a strong Hoare triple. □

LEMMA A.4. *If statements* $\mathtt{S}_1$ *and* $\mathtt{S}_2$ *satisfy domain expansion and minimal resource properties,* $[\varphi_1]\ \mathtt{S}_1\ [\widehat{\varphi}_1]$ *and* $[\varphi_2]\ \mathtt{S}_2\ [\widehat{\varphi}_2]$ *are strong Hoare triples,* $\widehat{\varphi}_1 * \varphi_{pre} \Leftrightarrow \exists Z.\ (\varphi_{post} * \varphi_2)$, *and* $\varphi_{pre} \cap mod(\mathtt{S}_1) = \varphi_{post} \cap mod(\mathtt{S}_2) = \emptyset$ *then* $[\varphi_1 * \varphi_{pre}]\ \mathtt{S}_1; \mathtt{S}_2\ [\exists Z.\ (\varphi_{post} * \widehat{\varphi}_2)]$ *is a strong Hoare triple.*

PROOF. Given the assumptions and using the frame rule, it is straightforward to show that $[\varphi_1 * \varphi_{pre}]\ \mathtt{S}_1; \mathtt{S}_2 [\exists Z.\ (\varphi_{post} * \widehat{\varphi}_2)]$ is a valid Hoare triple.

From Lemma A.3, it follows that $[\varphi_1 * \varphi_{pre}]\ \mathtt{S}_1\ [\widehat{\varphi}_1 * \varphi_{pre}]$ and $[\varphi_{post} * \varphi_2]\ \mathtt{S}_2\ [\varphi_{post} * \widehat{\varphi}_2]$ are strong Hoare triples. Hence, $[\exists Z.\ (\varphi_{post} * \varphi_2)]\ \mathtt{S}_2\ [\exists Z.\ (\varphi_{post} * \widehat{\varphi}_2)]$ is a strong Hoare triple.

Since $\widehat{\varphi}_1 * \varphi_{pre} \Leftrightarrow \exists Z.\ (\varphi_{post} * \varphi_2)$, it follows that $[\varphi_1 * \varphi_{pre}]\ \mathtt{S}_1; \mathtt{S}_2\ [\exists Z.\ (\varphi_{post} * \widehat{\varphi}_2)]$ is a strong Hoare triple. □

ACKNOWLEDGMENTS

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.

REFERENCES

Abdulla, P., Bouajjani, A., Cederberg, J., Haziza, F., and Rezine, A. 2008. Monotonic abstraction for programs with dynamic memory heaps. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 341–354.

Abdulla, P. A., Jonsson, B., Nilsson, M., and Saksena, M. 2004. A survey of regular model checking. In *Proceedings of the International Conference on Concurrency Theory (CONCUR)*. Springer, 35–48.

Bardin, S., Finkel, A., Leroux, J., and Schnoebelen, Ph. 2005. Flat acceleration in symbolic model checking. In *Proceedings of the International Symposium on Automated Technology for Verification and Analysis (ATVA)*. 474–488.

Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P. W., Wies, T., and Yang, H. 2007. Shape analysis for composite data structures. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 178–192.

Biering, B., Birkedal, L., and Torp-Smith, N. 2005. Bi-hyperdoctrines and higher-order separation logic. In *Proceedings of the European Symposium on Programming Languages and Systems (ESOP)*. 233–247.

Boigelot, B., Legay, A., and Wolper, P. 2003. Iterating transducers in the large. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*. Springer, 223–235.

Bouajjani, A., Habermehl, P., Moro, P., and Vojnar, T. 2005. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 13–29.

Bouajjani, A., Habermehl, P., and Rogalewicz, A. 2006. Abstract regular tree model checking of complex dynamic data struct ures. In *Proceedings of the International Symposium on Static Analysis (SAS)*. Springer, 52–70.

Bouajjani, A., Habermehl, P., and Tomas, V. 2004. Abstract regular model checking. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*. Springer, 372–386.

Calcagno, C., Distefano, D., O'Hearn, P., and Yang, H. 2009. Compositional shape analysis by means of bi-abduction. In *Proceedings of the Annual Symposium on Principles of Programming Languages (POPL)*.

Calcagno, C., Distefano, D., O'Hearn, P. W., and Yang, H. 2007. Footprint analysis: A shape analysis that discovers preconditions. In *Proceedings of the International Symposium on Static Analysis (SAS)*. 402–418.

Cousot, P. 1990. Methods and logics for proving programs. In *Formal Models and Semantics*, J. van Leeuwen, Ed., Handbook of Theoretical Computer Science, vol. B. Elsevier Science Publishers B.V., Chapter 15, 843–993.

Distefano, D., O'Hearn, P. W., and Yang, H. 2006. A local shape analysis based on separation logic. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 287–302.

Gulavani, B. S., Chakraborty, S., Ramalingam, G., and Nori, A. V. 2009. Bottom-up shape analysis using lisf. Tech. rep. TR-09-31, CFDVS, IIT Bombay. www.cfdvs.iitb.ac.in/~bhargav/spine.html.

Guo, B., Vachharajani, N., and August, D. I. 2007. Shape analysis with inductive recursion synthesis. In *Proceedings of the Conference on Programming Languages Design and Implementation (PLDI)*. 256–265.

Jeannet, B., Loginov, A., Reps, T. W., and Sagiv, S. 2004. A relational approach to interprocedural shape analysis. In *Proceedings of the International Symposium on Static Analysis (SAS)*. 246–264.

Lev-Ami, T., Sagiv, M., Reps, T., and Gulwani, S. 2007. Backward analysis for inferring quantified preconditions. Tech. rep. TR-2007-12-01, Tel-Aviv University.

Møller, A. and Schwartzbach, M. I. 2001. The pointer assertion logic engine. In *Proceedings of the Conference on Programming Languages Design and Implementation (PLDI)*. (Also in *SIGPLAN Notices 36*, 5).

O'Hearn, P. W., Reynolds, J. C., and Yang, H. 2001. Local reasoning about programs that alter data structures. In *Proceedings of the Symposium on Computer Science Logic (CSL)*. Lecture Notes in Computer Science, vol. 2142, Springer 1–19.

Podelski, A., Rybalchenko, A., and Wies, T. 2008. Heap assumptions on demand. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 314–327.

Reynolds, J. C. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)*. 55–74.

Rinetzky, N., Bauer, J., Reps, T. W., Sagiv, S., and Wilhelm, R. 2005a. A semantics for procedure local heaps and its abstractions. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. 296–309.

Rinetzky, N., Sagiv, M., and Yahav, E. 2005b. Interprocedural shape analysis for cutpoint-free programs. In *Proceedings of the International Symposium on Static Analysis (SAS)*. 284–302.

Rinetzky, N. and Sagiv, S. 2001. Interprocedural shape analysis for recursive programs. In *Proceedings of the Conference on Computer Construction (CC)*. Lecture Notes in Computer Science, vol. 2027. Springer, 133–149.

Sagiv, M., Reps, T., and Wilhelm, R. 1999. Parametric shape analysis via 3-valued logic. *Trans. Prog. Lang. Syst. 24*, 2002.

Touili, T. 2001. Regular model checking using widening techniques. In *Proceedings of the Conference on Verification of Parameterized Systems (VEPAS'01)*. 342–356.

Yorsh, G., Rabinovich, A. M., Sagiv, M., Meyer, A., and Bouajjani, A. 2006. A logic of reachable patterns in linked data-structures. In *Proceedings of the Foundations of Software Science and Computation Structures (FoSSaCS)*. 94–110.

Received December 2009; revised July 2010, April 2011; accepted July 2011