Bottom-Up Shape Analysis

Bhargav S. Gulavani[1], Supratik Chakraborty[1], Ganesan Ramalingam[2], and Aditya V. Nori[2]

[1] IIT Bombay
[2] Microsoft Research Bangalore

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

Specifically, we present a logic of iterated separation formula (LISF) which uses the iterated separating conjunct of Reynolds [17] 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 decision procedures for LISF.

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

1 Introduction

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

Shape analysis is intrinsically challenging. Bottom-up shape analysis is particularly challenging because it requires analyzing complex pointer manipulations when nothing is known about the initial state. Hence, traditional shape analyses are based on an iterative top-down (forward) analysis, where the statements are analyzed in the context of a particular (abstract) state. Though challenging, a 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.* [8]. The algorithm we present is based on ideas introduced by Calcagno *et al.* [8].

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 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` will terminate safely (with no memory errors) in a state satisfying $\widehat{\varphi}$.

```
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 the list segment

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 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) above, which yield a *functional specification* for the analyzed procedure. One of the key challenges in shape analysis is relating the value of the final data-structure to the value of the initial data-structure. We utilize an extension of separation logic, described later, to achieve this.

Composition via Strong Bi-Abduction. We now informally describe how summaries $[\varphi_1]$ `S1` $[\widehat{\varphi}_1]$ and $[\varphi_2]$ `S2` $[\widehat{\varphi}_2]$, in separation logic, can be composed to obtain summaries for `S1;S2`. The intuition behind the composition rule, which is similar to the composition rule in [8], 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 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 [15], where $*$ is the separating conjunction of Separation Logic [17] (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. Refer Section 2 for details.

Iterative Composition. A primary contribution of this paper is to extend the above intuition to obtain loop summaries. Suppose we have a summary $[\varphi]$ `S` $[\widehat{\varphi}]$, where `S` is the body of a loop (including the loop condition). We can apply strong bi-abduction to compose this summary with itself: for simplicity, suppose we identify 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 $[\varphi * \varphi^k_{pre}]$ $\mathtt{S}^\mathtt{k}$ $[\varphi^k_{post} * \widehat{\varphi}]$ 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 Formulas. In this paper, we introduce LISF, an extension of separation logic that enables us to meet our goal, and present sound procedures for strong bi-abduction and satisfiability in LISF. 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 of computing functional specifications that can relate the value of the final data-structure to the value of the initial data-structure. LISF gives us a generic ability to define recursive predicates useful for describing recursive data-structures. The use of 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 LISF for a bottom-up analysis, it can also be used to represent program states in top down interprocedural analysis.

Empirical Evaluation. We have implemented our inference rules in a prototype bottom-up analyzer and evaluated it on several shape analysis benchmarks. 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. This is due to the incompleteness of our strong bi-abduction algorithm.

Related Work. Our work is most closely related to the recent compositional shape analysis algorithm presented by Calcagno *et al.* [8], which derives from the earlier work in [9]. The algorithm described by Calcagno *et al.* 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 precondition, if the precondition is found to potentially lead to a memory error, or find a corresponding sound postcondition. The key idea in the Calcagno *et al.* 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 the Calcagno *et al.* work in the following 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 present LISF, which serves as the basis for our algorithm, while their 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 the initial data-structure.

Several recent papers [16,2,13] describe techniques to obtain preconditions by going backwards starting from some bad states. Unlike our approach, these techniques are not compositional or bottom-up. The work on regular model-checking [1,6,5,7] represents input-output relations by a transducer, which can be looked upon as a functional specification. But these works do not provide compositional techniques to compute the transducer for a loop.

Extrapolation techniques proposed in [18,4] compute sound overapproximations by identifying the growth in successive applications of transducers and iterating that growth. Similarly, [12] proposes a technique to guess the recursive predicates characterizing a data structure by identifying the growth in successive iterations of the loop and 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.

Program Syntax
```
e ::= v | null
C ::= v = e | v != e
S ::= v.f := e | v := u.f | v := new | dispose v | S; S
      | assert(C) | v := e | if(C, S, S) | while(C) S
```

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

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

Fig. 2. Program syntax and assertion logic syntax

Contributions. (i) We present the logic of iterated separation formulas LISF (Section 3) and give sound algorithms for satisfiability checking and strong bi-abduction in this logic (Section 5). (ii) We present inference rules to compute Hoare triples in a compositional bottom-up manner (Section 4). (iii) We have a prototype implementation of our technique. We discuss its performance on several challenging programs (Section 6).

2 Composition via Strong Bi-abduction

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

2.1 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 a constant **null**. This language does not support 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, mapping program variables to their values (Locs $\cup$ $\{\mathbf{null}\}$), and a heap, mapping a finite set of non-null locations to their values, which is a n-tuple of (primitive) values.

Assertion Logic. We illustrate some of the key ideas using standard Separation Logic, using the syntax shown in Figure 2. The '. . . ' in Figure 2 refer to constructs and extensions we will introduce in Section 3. We assume the reader is familiar with the basic ideas in Separation Logic. An expressions e evaluates to a location. Given a stack s, a variable v evaluates to a location. 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. **emp** denotes that the heap has no allocated cells. $x \mapsto (f : l)$ denotes a heap consisting of a single allocated cell pointed to by x, and the f field of this cell has value l. The $*$ operator is called the separating conjunct; $s_1 * s_2$ 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. The meaning of pure assertions depends only on the stack, and the meaning of spatial assertions depends on both the stack and the heap.

Table 1. 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)` $[v \neq e]$

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, called 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 1. These are similar to the small axioms of [15].

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 post conditions 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. We use $free(\varphi)$ to refer to the set of free variables in φ. Similarly, mod(S) denotes the set of logical variables corresponding to program variables modified by S. 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 use φ^s and φ^p to refer to the pure and spatial parts, respectively, of φ. The notation $\exists X.\varphi * \exists Y.\psi$ is used to denote $\exists X, Y.\ \varphi^p \wedge \psi^p \wedge \varphi^s * \psi^s$, when φ and ψ are quantifier free and do not have free Y and X variables, respectively.

2.2 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 $\widehat{\varphi}_1 * \varphi_{pre} \Leftrightarrow \varphi_{post} * \varphi_2$, then by application of Frame rule we can infer the summary $[\varphi_1 * \varphi_{pre}]$ S1; S2 $[\varphi_{post} * \widehat{\varphi}_2]$. We can compose the two given summaries even under the slightly modified condition $\widehat{\varphi}_1 * \varphi_{pre} \Leftrightarrow \exists Z.\ (\varphi_{post} * \varphi_2)$, if $Z \subseteq$ Aux. The summary inferred in this case is $[\varphi_1 * \varphi_{pre}]$ S1; S2 $[\exists Z.\ (\varphi_{post} * \widehat{\varphi}_2)]$.

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

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

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

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

Fig. 3. Inference rules for sequential composition, loops, and branch statements

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 $\widehat{\varphi}_1 * \varphi_{pre} \Leftrightarrow \exists Z.\ (\varphi_{post} * \varphi_2)$ as *strong bi-abduction.* The concept of strong bi-abduction is similar to that of bi-abduction presented in [8] (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 $\widehat{\varphi}_2 * \varphi_{pre} \Rightarrow \varphi_{post} * \varphi_2$, whereas we seek equivalence instead of implication while allowing some auxiliary variables to be existentially quantified in the right hand side of the equivalence. While the above composition rule is sound even if we use bi-abduction, bi-abduction may not yield good post-conditions. Specifically, 'total' and 'strong' properties of specifications are preserved under composition using strong bi-abduction. The 'strong' property is not preserved under composition using bi-abduction, although composition is sound. A drawback of using strong bi-abduction, though, is that there exist Hoare triples which 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 specification for most of the benchmark programs using strong bi-abduction in a bottom-up analysis.

Example 1. In this and subsequent examples, we use $v \mapsto w$ as a short-hand for $v \mapsto (next : w)$. Let us compose two summaries, $[v = \_a]\ \texttt{v := new}\ [\exists \_b.\ v \mapsto \_b]$ and $[v = \_c \wedge \_c \mapsto \_d]\ \texttt{v := v.next}\ [v = \_d \wedge \_c \mapsto \_d]$. 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 $(\exists \_b.\ v \mapsto \_b) * \mathbf{emp} \Leftrightarrow \exists \_c \_d.\ \mathbf{emp} * (v = \_c \wedge \_c \mapsto \_d)$ we can compose the two summaries and deduce $[v = \_a]\ \texttt{v := new; v := v.next}\ [\exists \_c \_d.\ v = \_d \wedge \_c \mapsto \_d]$.

We now present a set of Hoare inference rules in Separation Logic for our programming language. The rules are formally presented in Figure 3. 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 triples of the form $[\varphi]\ \texttt{S}^{+}\ [\widehat{\varphi}]$. The triple $[\varphi]\ \texttt{S}^{+}\ [\widehat{\varphi}]$ means that for every initial state satisfying φ, there exists a $k \geq 1$ such that the state resulting after k executions of S satisfies $\widehat{\varphi}$. In next two sections we present a technique for computing triples of this form.

3 Logic of Iterated Separation Formulae: LISF

Let S be the loop: `while (v!=null) v := v.next`. Let us use $\odot_{i=0}^{k} \psi^i$ informally to represent the iterated separating conjunction $\psi^0 * \cdots * \psi^k$ [17]. We would like to infer the following summary for S: $[v = \_x_0 \wedge \_x_k = \mathbf{null} \wedge \odot_{i=0}^{k-1} \_x_i \mapsto \_x_{i+1}]$ S $[v = \_x_k \wedge \_x_k = \mathbf{null} \wedge \odot_{i=0}^{k-1} \_x_i \mapsto \_x_{i+1}]$. In this section, we present a formal extension of Separation Logic that lets us express such triples involving iterated separating conjunction, in a restricted form. We first motivate this restricted form of iteration by informally explaining how we plan to infer summaries such as the one above.

Assume that we have a Hoare triple $[\varphi]$ S $[\widehat{\varphi}]$ where φ and $\widehat{\varphi}$ are quantifier free formulas. We can compute a Hoare triple for k executions of S by repeated applications of the COMPOSE rule as follows. Let φ^i (resp. $\widehat{\varphi}^i$) denote φ (resp. $\widehat{\varphi}$) with every variable $x \in \mathsf{Aux}$ replaced by an indexed variable x_i. Consider the following valid Hoare triples with variables renamed, $[\varphi^i]$ S $[\widehat{\varphi}^i]$ and $[\varphi^{i+1}]$ S $[\widehat{\varphi}^{i+1}]$. Let φ^i_{pre} and φ^i_{post} be such that, $free(\varphi^i_{pre}) \cap mod(\mathtt{S}) = free(\varphi^i_{post}) \cap mod(\mathtt{S}) = \emptyset$, and $\widehat{\varphi}^i * \varphi^i_{pre} \Leftrightarrow \varphi^i_{post} * \varphi^{i+1}$. Note that unlike φ^i or $\widehat{\varphi}^i$, φ^i_{pre} and φ^i_{post} may have free variables with index i as well as $i+1$. We can now inductively apply the compose rule and conclude the following Hoare triple.

$$[\varphi^0 * (\odot_{i=0}^{k-1}\ \varphi^i_{pre})]\mathtt{S}^{k+1}[(\odot_{i=0}^{k-1}\ \varphi^i_{post}) * \widehat{\varphi}^k] \qquad (3.1)$$

Example 2. Let S be the statement: $\mathtt{assert}(\mathtt{v}! = \mathbf{null}); \mathtt{v} := \mathtt{v.next}$. Let us 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 $\varphi_{pre} \equiv \_x_1 = \_y_0 \wedge \_x_1 \mapsto \_y_1$ and $\varphi_{post} \equiv \_x_1 = \_y_0 \wedge \_x_0 \mapsto \_y_0$. Application of 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 $[(\_x_1 = \_y_0 \wedge \_x_0 \mapsto \_y_0) * (v = \_y_1 \wedge \_x_1 \mapsto \_y_1)]$. Iterative application of compose gives the summary: $[v = \_x_0 \wedge \_x_0 \mapsto \_y_0 * \odot_{i=0}^{k-1}(\_x_{i+1} = \_y_i \wedge \_x_{i+1} \mapsto \_y_{i+1})]$ $\mathtt{S}^+$ $[\odot_{i=0}^{k-1} (\_x_{i+1} = \_y_i \wedge \_x_i \mapsto \_y_i) * (v = \_y_k \wedge \_x_k \mapsto \_y_k)]$.

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

Fig. 4. LISF assertion syntax

LISF *Syntax and Informal Semantics:* We now formally introduce a restricted form of the iterated separating conjunct illustrated above. Fig. 4 presents the syntax of LISF, where "…" represents standard constructs of Separation Logic from Figure 2.We first illustrate the syntax with an example relating the informal notation introduced earlier to the formal syntax. The informal notation $v = \_x_0 \wedge \_x_k = \mathbf{null} \wedge \odot_{i=0}^{k-1} \_x_i \mapsto \_x_{i+1}$ is represented in LISF as $v = \mathbf{A}[0] \wedge \mathbf{A}[\$0] = \mathbf{null} \wedge \mathsf{RS}(\mathbf{A}[\cdot] \mapsto \mathbf{A}[\cdot + 1], 0, 0)$. This represents an acyclic singly linked list pointed to by v.

$$
\begin{array}{ll}
m \models e_1 \sim e_2 & \text{iff } \mathcal{E}(e_1, L, s, \mathcal{V}) \sim \mathcal{E}(e_2, L, s, \mathcal{V}) \\
m \models \mathsf{RP}(P, l, u) & \text{iff } \exists k.\ k+1 = len(\mathcal{V}, L, P) \wedge \forall l \leq i \leq k-1-u.\ (s, h, \mathcal{V}, i :: L) \models P \\
m \models e_1 \mapsto (f : e_2) & \text{iff } h(\mathcal{E}(e_1, L, s, \mathcal{V})) = (f : \mathcal{E}(e_2, L, s, \mathcal{V})) \ \wedge dom(h) = \{\mathcal{E}(e_1, L, s, \mathcal{V})\} \\
m \models \mathsf{RS}(S, l, u) & \text{iff } \exists k, u', h_l, \ldots, h_{u'}.\ k+1 = len(\mathcal{V}, L, S) \wedge u' = k-1-u \wedge h = \bigsqcup_{i=l}^{u'} h_i \ \wedge \\
& \quad \forall l \leq i, j \leq u'.\ i \neq j \Rightarrow h_i \# h_j \wedge \forall l \leq i \leq u'.\ (s, h_i, \mathcal{V}, i :: L) \models S
\end{array}
$$

Fig. 5. Subset of semantics of LISF. m is $(s, h, \mathcal{V}, L)$, len is as explained in text.

A LISF formula may reference a new type of logical variable such as $\mathbf{A}$, which we call an *array variable*. We will denote array variable names with bold-faced upper case letters, as a convention. As we will see later, the semantics of LISF will utilize a mapping from such array variables to a *sequence* of values $(v_0, \cdots, v_k)$. LISF also utilizes multi-dimensional arrays to handle nested recursive data-structures. In such cases, the values v_i may themselves be sequences.

Expressions are extended in LISF to permit indexed variable references, which consist of an array variable name followed by a sequence of one or more indices. An index can take one of the following four forms: (i) $arr[c]$, (ii) $arr[\$c]$, (iii) $arr[\cdot]$, or (iv) $arr[\cdot + 1]$. *Fixed indices* $arr[c]$ and $arr[\$c]$ refer to the element at an offset c from the beginning or end of the sequence that arr denotes, respectively. E.g., if $\mathbf{A}$ is bound to $(v_0, \cdots, v_k)$, then $\mathbf{A}[0]$ and $\mathbf{A}[\$0]$ evaluate to v_0 and v_k respectively. *Iterated indices* $arr[\cdot]$ and $arr[\cdot + 1]$ will be explained soon.

We extend the pure and spatial formulas with predicates $\mathsf{RP}(P, l, u)$ and $\mathsf{RS}(S, l, u)$, respectively, to capture repeated structures. Loosely speaking, $\mathsf{RS}(S, l, u)$ corresponds to our informal notation $\odot_{i=l}^{k-1-u} S$, except that there is no explicit representation of the index variable i or the bound k. The values of i and k are actually provided by the evaluation context in the semantics. The dot in the *iterated indices* $arr[\cdot]$ and $arr[\cdot + 1]$ is used to refer to the implicit index variable i. Thus, $arr[\cdot]$ refers to the element at offset i, and $arr[\cdot + 1]$ refers to the element at offset $i+1$. Expressions with iterated indices are used within RP or RS predicates. For example, consider the predicate $\mathsf{RS}(\mathbf{A}[\cdot] \mapsto \mathbf{A}[\cdot + 1], 0, 0)$, where $\mathbf{A}$ is bound to a sequence of length $k+1$. This predicate asserts that for all $i \in [0, k-1]$, the i^{th} element of $\mathbf{A}$ is the location of a cell in the heap whose next field has the same value as the $i+1^{th}$ element of $\mathbf{A}$. Further, the predicate also asserts that the elements $\mathbf{A}[0]$ to $\mathbf{A}[k-1]$ are distinct. For notational convenience we denote the formulas $\mathsf{RP}(P, l, u)$ and $\mathsf{RS}(S, l, u)$ by $\mathsf{RP}(P)$ and $\mathsf{RS}(S)$, respectively, when both l and u are 0.

LISF *Semantics* Expression evaluation semantics is extended in a straightforward fashion to evaluate indexed variable references. Expressions involving array variable with multiple indices require the value of that array and a list of indices, one for every iterated index, for their evaluation. The semantics of an expression e, which evaluates to a location, is given by the function $\mathcal{E}(e, L, s, \mathcal{V})$ where L is the index list (provided by the evaluation context), s is the stack, and $\mathcal{V}$ is the mapping of array names to their values (uni or multi-dimensional sequences of locations). Definition of $\mathcal{E}$ and detailed semantics are provided in the extended version [11].

The structures modeling LISF formulas are $(s, h, \mathcal{V})$ where s is the stack, h is the heap, and $\mathcal{V}$ is the mapping of array names to their values. The semantics of assertions is given by the satisfaction relation ($\models$) between a structure extended with a list of indices L, and an assertion φ. The list of indices L facilitates evaluation of expressions by the function $\mathcal{E}$. The structure $(s, h, \mathcal{V})$ models φ iff $(s, h, \mathcal{V}, []) \models \varphi$. Semantics of constructs novel to LISF are given in Figure 5. We assume that φ is a well formed formula (*wff*) and $(s, h, \mathcal{V})$ is a well formed structure for φ (wfs_φ). Intuitively, *wff* and wfs_φ avoid indexing error in the evaluation of φ. We write $h_1 \# h_2$ to indicate that h_1 and h_2 have disjoint domains, and $h_1 \sqcup h_2$ to indicate the disjoint union of such heaps.

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 array accessed by the n^{th} iterated index of every 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 array accessed by the n^{th} iterated index of any expression in P (or S). The semantics of $\mathsf{RP}(P, l, u)$ 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_is are pair-wise disjoint.

4 Inductive Composition

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. φ^1 : φ 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

$[\varphi\alpha * \mathsf{Iter}(\varphi^0_{pre})]$ S^+ $[\exists X.\ \mathsf{Iter}(\varphi^0_{post}) * \widehat{\varphi}\beta]$

Fig. 6. Inference rule for acceleration

The rules introduced in Figure 3 are valid even with LISF extension of Separation Logic. The set of auxiliary variables, Aux, includes the array variables in this extension. For clarity, we adopt the following convention in the remainder of the paper: (i) unless explicitly stated, all formulas in LISF are quantifier free, (ii) Hoare triples are always expressed as $[\varphi]$ S $[\exists X.\ \widehat{\varphi}]$, (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 $\exists X.\ \widehat{\varphi}$.

Let $[\varphi]$ S $[\exists X.\ \widehat{\varphi}]$ be a Hoare triple. We wish to compute a strong summary for S^+. Figure 6 presents the rule INDUCT to compute such a summary. As in the previous Section, we use φ^i (resp. $\widehat{\varphi}^i$) to denote φ (resp. $\widehat{\varphi}$) with every free auxiliary variable $w \in W$ replaced by an indexed variable w_i. Let φ^0_{pre}, φ^0_{post} be formulas such

[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 defn. 124 in [10]).

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 (3.1).

We use α, β, and Iter to express φ^0, $\widehat{\varphi}^k$ and the iterated separating conjunct of accelerated summary (3.1) in 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 $\mathbf{x}[\$0]$.

Inductq

Given
1. $[\varphi]\ \mathsf{S}\ [\exists X.\ \widehat{\varphi}]$
2. $\widehat{\varphi}^0$: $\widehat{\varphi}$ with every $w \in W$ and $x \in X$ replaced by w_0 and x_1, resp.
3. φ^1 : φ 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. $(\exists X.\ \widehat{\varphi}^0) * \varphi^0_{pre} \Leftrightarrow \exists Z_1.\ (\varphi^0_{post} * \varphi^1)$
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. α, β, 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. 7. Inference rule Inductq

The function Iter in premise 9 takes an LISF formula ψ, computes an intermediate formula ψ_{ren}, and returns $\mathsf{RP}(\psi^p_{ren}) \wedge \mathsf{RS}(\psi^s_{ren})$. 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 w_0 (resp. w_1) by a fresh array with iterated index $\mathbf{w}[\cdot]$ (resp. $\mathbf{w}[\cdot + 1]$). If ψ^p_{ren} and ψ^s_{ren} do not have any common array variable, it performs a second pass 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 RP and RS predicates in $\mathsf{Iter}(\psi)$.

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. The modified rule Inductq is presented in Figure 7. We use a refined notation in Inductq where φ^i (resp. $\widehat{\varphi}^i$) 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 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 $Z_1 \cup \ldots \cup Z_k$ 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. As a technical subtlety, the variables in Z_0 need not be quantified. This is taken care of by adding extra equalities in $\mathsf{Iter}(\varphi^0_{post})$ (see [11] for details).

Soundness of INDUCT and INDUCTQ can be proved by appealing to the soundness of the COMPOSE rule, and by using structural induction. Note that if any Hoare triple in the premise of an inference rule in Figure 3, 6, or 7 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 1. *Inference rules* INDUCT *and* INDUCTQ *are sound.*

Example 3. Recall Example 2 where two instances of the summary $[v = \_x \wedge \_x \mapsto \_y]\ \mathsf{S}\ [v = \_y \wedge \_x \mapsto \_y]$ are composed using $\varphi^0_{pre} : (\_x_1 = \_y_0 \wedge \_x_1 \mapsto \_y_1)$ and $\varphi^0_{post} : (\_x_1 = \_y_0 \wedge \_x_0 \mapsto \_y_0)$. For this example, $\mathsf{Iter}(\varphi^0_{pre})$ generates the 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: $[v = \mathbf{X}[0] \wedge \mathsf{RP}(\mathbf{X}[\cdot + 1] = \mathbf{Y}[\cdot]) \wedge \mathbf{X}[0] \mapsto \mathbf{Y}[0] * \mathsf{RS}(\mathbf{X}[\cdot + 1] \mapsto \mathbf{Y}[\cdot + 1])]\ \mathsf{S}^+$ $[v = \mathbf{Y}[\$0] \wedge \mathsf{RP}(\mathbf{X}[\cdot + 1] = \mathbf{Y}[\cdot]) \wedge \mathsf{RS}(\mathbf{X}[\cdot] \mapsto \mathbf{Y}[\cdot]) * \mathbf{X}[\$0] \mapsto \mathbf{Y}[\$0]]$.

Discussion. In the above example, the equality $\_x_1 = \_y_0$ in φ^0_{pre} and φ^0_{post} identifies *folding points* of the repeated sub-heaps. Hence we can rewrite the pre and postcondition as $v = \_x_0 \wedge \odot_{i=0}^{k-1} \_x_i \mapsto \_x_{i+1}$ and $v = \_x_k \wedge \odot_{i=0}^{k-1} \_x_i \mapsto \_x_{i+1}$, respectively, using the equality $\_y_0 = \_x_1$ to eliminate $\_y_0$ in both the formulas. The corresponding summary in LISF is: $[v = \mathbf{X}[0] \wedge \mathsf{RS}(\mathbf{X}[\cdot] \mapsto \mathbf{X}[\cdot + 1])]\ \mathsf{S}^+\ [v = \mathbf{X}[\$0] \wedge \mathsf{RS}(\mathbf{X}[\cdot] \mapsto \mathbf{X}[\cdot + 1])]$.

Instead of translating a recurrence into a LISF formula, one could also translate it into a recursive predicate. For example, the recurrence $\odot_{i=0}^{k-1} \_x_i \mapsto \_x_{i+1}$ obtained above can be translated into a recursive predicate $\mathsf{Rec}(\_x_0, \_x_k)$, where $\mathsf{Rec}(\_x_0, \_x_k) \equiv \_x_0 \mapsto \_x_k \vee \exists \_x_1 .\ \_x_0 \mapsto \_x_1 * \mathsf{Rec}(\_x_1, \_x_k)$. We choose to represent the values of variables in successive instances of a repeated formula by an array rather than 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. For example, this enables our analysis to establish the fact that traversing a list using `next` field does not modify contents of the cells or the relative links between them.

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 and INDUCTQ 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. In general, determining the sequence of application of the rules COMPOSE, INDUCTQ, EXIT and WHILE to obtain useful loop summaries is an important but orthogonal issue and needs to be guided by heuristics. Heuristics used for acceleration in [3] can be adapted to guide the application of these rules for synthesizing useful loop summaries. Given procedure summaries, non-recursive procedure calls can be analyzed by the COMPOSE rule, as in [8]. The INDUCTQ rule can also be used to compute accelerated summaries of tail recursive procedures having at most one self-recursive call.

5 A Strong Bi-abduction Algorithm for LISF

```
BiAbduct($\varphi$, $\psi$, $mod_1$, $mod_2$)
 1: $res \leftarrow \{\}$
 2: for all $(M, C, L_1, L_2) \in$ Match$(\varphi^s, \psi^s)$ do
 3:     $\Delta \leftarrow (\varphi^p \wedge L_1) * (M \wedge C) * (\psi^p \wedge L_2)$
 4:     if sat$(\Delta)$ then
 5:         $\delta_1 \leftarrow$ RemoveVar$(M \wedge \psi^p \wedge L_2, \varphi, mod_1, V \cup W)$
 6:         $\delta_2 \leftarrow$ RemoveVar$(M \wedge \varphi^p \wedge L_1, \psi, mod_2, V \cup Y)$
 7:         $\gamma \leftarrow$ ComputeRenaming$(\delta_1, mod_1, Y)$
 8:         $\kappa_1 \leftarrow \delta_1\gamma$
 9:         $\hat{Z} \leftarrow$ Range$(\gamma)$
10:         if IsIndep$(\kappa_1, mod_1)$ and IsIndep$(\delta_2, mod_2)$
            then
11:             $\theta \leftarrow$ ComputeRenaming$(\kappa_1, X \cup Y, X)$
12:             $\widetilde{Z} \leftarrow$ Domain$(\theta)$
13:             if IsIndep$(\kappa_1\theta, X)$ then
14:                 $res \leftarrow res \cup (\kappa_1\theta, \delta_2\bar{\theta}, \hat{Z} \cup \widetilde{Z})$
15: return $res$
```

Fig. 8. Algorithm BiAbduct

We now present a sound algorithm for computing $\varphi_{pre}, \varphi_{post}$ 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 LISF formulas $\exists X.\ \varphi(V, W, X)$ and $\psi(V, Y)$ where V, W, X, Y are disjoint sets of variables, we wish to compute $\varphi_{pre}, \varphi_{post}$, 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 8. We illustrate the algorithm through the following example: $\varphi \equiv v \mapsto \_x_0$, $\psi \equiv v = \_y_0 \wedge \_y_0 \mapsto \_y_1$, $V = \{v\}, W = \{\}, X = \{\_x_0\}, Y = \{\_y_0, \_y_1\}$ and $mod_1 = mod_2 = \{v\}$.

The key step in bi-abduction is the Match procedure used in line 2. Match takes as input two spatial formulas φ^s and ψ^s 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 non-overlapping parts defined by L_1 and L_2 respectively.

We present procedure Match as a set of inference rules in Figure 9. In these inference rules we use a set of spatial facts and a star conjunction of spatial facts interchangeably. The function $\mathsf{unroll_f}\ \mathsf{RS}(S, l, u)$ required by rule MIII *unrolls* RS once from the beginning. This is done by instantiating that iterated index $[\cdot]$ (resp. $[\cdot + 1]$) of every array expression in S corresponding to the nesting depth of S with fixed index $[l]$ (resp. $[l+1]$). Similarly $\mathsf{unroll_b}\ \mathsf{RS}(S, l, u)$ unrolls RS once from the end. These rules can be easily implemented as a recursive algorithm. Note that in rules MIII and MIV, 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 non-termination of the recursion. In practice we circumvent this problem by limiting the number of applications of these rules.

Lemma 2. *Every* (M, C, L_1, L_2) *computed in line 2 of* BiAbduct *satisfies* (*i*) $M \wedge \varphi^s \Leftrightarrow (M \wedge C) * L_1$, *and* (*ii*) $M \wedge \psi^s \Leftrightarrow (M \wedge C) * L_2$.

$$\text{M0}\ \frac{}{(\mathbf{true}, \mathbf{emp}, S_1, S_2) \in \mathsf{Match}(S_1, S_2)}$$

$$\text{MI}\ \frac{\begin{array}{c} k_1 \in S_1,\ k_2 \in S_2,\ S_1' = S_1 \setminus k_1,\ S_2' = S_2 \setminus k_2 \\ (M, C, L_1, L_2) \in \mathsf{Match}(k_1, k_2) \\ (N, C', L_1', L_2') \in \mathsf{Match}(S_1' \cup L_1, S_2' \cup L_2) \end{array}}{(M \wedge N, C * C', L_1', L_2') \in \mathsf{Match}(S_1, S_2)}$$

$$\text{MII}\ \frac{\begin{array}{c} k_1 \equiv x \mapsto (f^i : x^i),\ \ k_2 \equiv y \mapsto (f^i : y^i) \\ M \equiv x = y \wedge \bigwedge\{x^i = y^i\} \end{array}}{(M, x \mapsto (f^i : y^i), \{\}, \{\}) \in \mathsf{Match}(k_1, k_2)}$$

$$\text{MIII}\ \frac{\begin{array}{c} k_1 :\ \mathsf{RS}(S, l, u),\ \ k_2 :\ x \mapsto (f : y), \\ S_1 :\ \mathsf{unroll_f}\ \mathsf{RS}(S, l, u) \\ (M, C, L_1, L_2) \in \mathsf{Match}(S_1, k_2) \end{array}}{(M, C, L_1 * \mathsf{RS}(S, l+1, u), L_2) \in \mathsf{Match}(k_1, k_2)}$$

$$\text{MIV}\ \frac{\begin{array}{c} k_1 :\ \mathsf{RS}(S, l, u),\ \ k_2 :\ x \mapsto (f : y), \\ S_1 :\ \mathsf{unroll_b}\ \mathsf{RS}(S, l, u) \\ (M, C, L_1, L_2) \in \mathsf{Match}(S_1, k_2) \end{array}}{(M, C, L_1 * \mathsf{RS}(S, l, u+1), L_2) \in \mathsf{Match}(k_1, k_2)}$$

$$\text{MV}\ \frac{\begin{array}{c} k_1 :\ \mathsf{RS}(S_1, l, u),\ \ k_2 :\ \mathsf{RS}(S_2, l, u), \\ (M, C, \{\}, \{\}) \in \mathsf{Match}(S_1, S_2) \end{array}}{(\mathsf{RP}(M, l, u), \mathsf{RS}(C, l, u), \{\}, \{\}) \in \mathsf{Match}(k_1, k_2)}$$

Fig. 9. Rules for procedure Match

Given a possible decomposition (M, C, L_1, L_2) of φ^s and ψ^s, line 4 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 from $M \wedge \psi^p \wedge L_2$ and $M \wedge \varphi^p \wedge L_1$, respectively, using function RemoveVar (lines 5, 6). The function RemoveVar(ϕ_1, ϕ_2, A, B) replaces every free variable $v \in A$ in ϕ_1 by e if ϕ_2 implies $v = e$ and $free(e) \in B \setminus A$. For our running example $\delta_1 \equiv v = \_y_0 \wedge \_x_0 = \_y_1$ and $\delta_2 \equiv \_x_0 = \_y_1$ is one such pair.

Lemma 3. *Every* (δ_1,δ_2) *pair computed in lines 5 and 6 of* BiAbduct *satisfies* $\varphi * \delta_1 \Leftrightarrow \delta_2 * \psi$.

Next, we process the formula δ_1 so as to make it independent of mod_1. In line 7, we compute a renaming $\gamma : \langle mod_1 \hookrightarrow Y \rangle$ such that if κ_1 represents $\delta_1\gamma$ and $\hat{Z}$ equals the range of γ, then $\varphi * \kappa_1 \Leftrightarrow \exists \hat{Z}.\ (\delta_2\ *\ \psi)$. 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 $a = b$. 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 10). 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, $\hat{Z} = \{\_y_0\}$ and $\gamma : \langle \_y_0 \rightarrow v \rangle$ gives a valid renaming, since $\delta_1\gamma \equiv \_x_0 = \_y_1$ is independent of v.

Lemma 4. *Every* κ_1 *and* $\hat{Z}$ *computed in lines 8 and 9 of* BiAbduct *satisfy* $\varphi * \kappa_1 \Leftrightarrow \exists \hat{Z}.\ (\delta_2\ *\ \psi)$.

For every κ_1 at line 11 we compute a renaming $\theta\ :\ \langle \widetilde{Z}\ \hookrightarrow\ X \rangle$, where $\widetilde{Z} \subseteq X \cup Y$, so as to render $\kappa_1\theta$ independent of X (lines 11, 12, 13). The function ComputeRenaming$(\kappa_1, X \cup Y, X)$ computes the renaming θ. Let $\bar{\theta}\ :\ \langle X\ \hookrightarrow\ \widetilde{Z} \rangle$ be a renaming such that $\bar{\theta}(x) = z$ only if $\theta(z) = x$. If $\kappa_1\theta$ is independent of X, then algorithm BiAbduct returns $(\kappa_1\theta, \delta_2\bar{\theta}, \widetilde{Z} \cup \hat{Z})$ as one of the solutions of strong bi-abduction.

The invocations of ComputeRenaming in lines 7 and 11 have one important difference: in line 7 only non-array variables in mod_1 are renamed, whereas in line 11 array variables in $X \cup Y$ may be renamed. The function ComputeRenaming(ϕ, A, B) renames array variables as follows. An array variable

Table 2. Experimental results on (a) list manipulating example, (b) functions from Firewire Windows Device Drivers, and (c) examples from [2,14]. Experiments performed on Pentium 4 CPU, 2.66GHz, 1 GB RAM. All programs are available at [11].

Progs	LOC	Time (s)	# triples discovered	Complete?
init	16	0.010	2	Yes
del-all	21	0.009	2	Yes
del-circ	23	0.013	2	Yes
delete	42	0.090	* 19	No
append	23	0.013	3	Yes
ap-disp	52	0.047	6	Yes
copy	33	0.532	3	Yes
find	28	0.023	4	Yes
insert	53	1.270	6	Yes
merge	60	0.880	12	No
reverse	20	0.015	* 3	No

(a)

Progs	LOC	Time (s)	# triples discovered	Complete?
BusReset	145	0.080	* 3	Yes
CancelIrp	87	1.060	* 32	Yes
SetAddress	96	0.185	* 6	Yes
GetAddress	94	0.185	* 6	Yes
PnpRemove	460	75.321	34	No

(b)

Progs	LOC	Time (s)	# triples discovered	Complete?
dll-reverse	23	0.130	3	No
fumble	20	0.017	2	Yes
zip	37	0.650	4	No
nested	20	0.130	10	Yes

(c)

$a \in A$ is renamed to another array variable $b \in B$ if ϕ^p implies one of the following facts: (i) $\mathsf{RP}(a[\cdot] = b[\cdot]) \wedge a[\$0] = b[\$0]$, or (ii) $\mathsf{RP}(a[\cdot + 1] = b[\cdot + 1]) \wedge a[0] = b[0]$, or (iii) $\mathsf{RP}(a[\cdot] = b[\cdot] \wedge a[\cdot + 1] = b[\cdot + 1])$. Higher dimensional arrays can be renamed by performing similar checks for each dimension. For our running example, we have $X = \{\_x_0\}, \widetilde{Z} = \{\_y_1\}$ and $\theta : \langle \_y_1 \rightarrow \_x_0 \rangle$. It is evident that $(\exists \_x_0.\ v \mapsto \_x_0) * (\mathbf{true}) \Leftrightarrow \exists \_y_0, \_y_1.\ (\mathbf{true}) * (v = \_y_0 \wedge \_y_0 \mapsto \_y_1)$. Thus $\varphi_{pre} \equiv \kappa_1\theta \equiv \mathbf{true}$, $\varphi_{post} \equiv \delta_2\bar{\theta} \equiv \mathbf{true}$, and and $Z = \{\_y_0, \_y_1\}$ is a solution of strong bi-abduction between $\exists \_x_0.\ \varphi \equiv \exists \_x_0.\ v \mapsto \_x_0$ and $\psi \equiv v = \_y_0 \wedge \_y_0 \mapsto \_y_1$.

Lemma 5. *Every θ and $\widetilde{Z}$ at line 14 of* BiAbduct *satisfy* $(\exists X.\ \varphi) * \kappa_1\theta \Leftrightarrow \exists \hat{Z}, \widetilde{Z}.\ (\delta_2\bar{\theta} * \psi)$.

Satisfiability checking. We provide a sound algorithm for checking satisfiability of LISF formulas. The basic idea is to convert a LISF formula to a formula in separation logic without iterated predicates. 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. See [11] for details.

6 Experimental Evaluation

We have implemented our inference rules to generate specifications of programs in a bottom-up and compositional manner. Our implementation takes as input a C program and outputs summaries for each procedure in the program. We currently do not handle pointer arithmetic.

The results of running our tool on a set of challenging programs are tabulated in Table 2. Programs in Table 2(a) are adopted from [9]. Program `delete` is the same as the motivating example in Section 1. All programs in Table 2 (c) except `nested` are adopted from [2,14]. These programs manipulate singly or doubly linked lists. Program `nested` traverses nested linked lists. In each of these tables, the fourth column indicates the number of summaries inferred by our tool. The last column indicates whether the inferred summaries provide a complete specification for the corresponding program. Our tool inferred richer summaries than those inferred by the tool in [9]. For example, for the programs `delete` and `reverse`, our tool infers preconditions with cyclic lists (indicated by * in fourth column). For the program `delete` some of the inferred preconditions even have a lasso structure.

The examples in Table 2(b) are program fragments modifying linked structures in the Firewire Windows Device Driver. We report only the summaries discovered for the main procedures in these programs. A complete set of summaries is discovered for all the other procedures in these programs. The original programs and data structures have been modified slightly so as to remove pointer arithmetic. These programs perform selective deletion or search through doubly linked lists. The program `PnpRemove` iterates over five different cyclic lists and deletes all of them; it has significant branching structure. All programs except `CancelIrp` refer to only the next field of list nodes. The program `CancelIrp` also refers 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. The summaries inferred for all programs except for `PnpRemove` have been manually checked and found to be 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.

7 Conclusion

We have presented inference rules for bottom-up and compositional shape analysis. Strong bi-abduction forms the basis of our inference rules. We have introduced a new logic, LISF, along with sound procedures for strong bi-abduction and satisfiability checking in LISF.

In future we would like to (i) enrich the Match procedure by additional lemmas so that our tool can generate more expressive summaries, (ii) extend strong bi-abduction procedure to operate over disjunctions of LISF formulas, and (iii) extend our technique to analyze programs manipulating tree-like structures.

Acknowledgment. We thank Hongseok Yang and Dino Distefano for introducing us to the idea of abduction and for providing us with benchmark programs. The first author was supported by Microsoft Corporation and Microsoft Research India under the Microsoft Research India PhD Fellowship Award.

References

1. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 35–48. Springer, Heidelberg (2004)
2. Abdulla, P.A., Bouajjani, A., Cederberg, J., Haziza, F., Rezine, A.: Monotonic abstraction for programs with dynamic memory heaps. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 341–354. Springer, Heidelberg (2008)
3. Bardin, S., Finkel, A., Leroux, J., Schnoebelen, P.: Flat acceleration in symbolic model checking. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 474–488. Springer, Heidelberg (2005)
4. Boigelot, B., Legay, A., Wolper, P.: Iterating transducers in the large. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 223–235. Springer, Heidelberg (2003)
5. Bouajjani, A., Habermehl, P., Moro, P., Vojnar, T.: Verifying programs with dynamic 1-selector-linked structures in reg ular model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 13–29. Springer, Heidelberg (2005)
6. Bouajjani, A., Habermehl, P., Rogalewicz, A.: Abstract regular tree model checking of complex dynamic data structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006)
7. Bouajjani, A., Habermehl, P., Tomas, V.: Abstract regular model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 372–386. Springer, Heidelberg (2004)
8. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Proc. of POPL (2009)
9. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Footprint analysis: A shape analysis that discovers preconditions. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 402–418. Springer, Heidelberg (2007)
10. Cousot, P.: Methods and logics for proving programs. In: van Leeuwen, J. (ed.) Formal Models and Semantics. Handbook of Theoretical Computer Science, vol. B, Ch. 15., pp. 843–993. Elsevier Science Publishers B.V., Amsterdam (1990)
11. Gulavani, B.S., Chakraborty, S., Ramalingam, G., Nori, A.V.: Bottom-up shape analysis. Technical Report TR-09-27, CFDVS, IIT Bombay (2009), `www.cfdvs.iitb.ac.in/~bhargav/shape-analysis.html`
12. Guo, B., Vachharajani, N., August, D.I.: Shape analysis with inductive recursion synthesis. In: Proc. of PLDI, pp. 256–265 (2007)
13. Lev-Ami, T., Sagiv, M., Reps, T., Gulwani, S.: Backward analysis for inferring quantified preconditions. Technical Report TR-2007-12-01, Tel Aviv University (2007)
14. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: Proc. of PLDI (June 2001); also in SIGPLAN Notices 36(5) (May 2001)
15. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
16. Podelski, A., Rybalchenko, A., Wies, T.: Heap assumptions on demand. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 314–327. Springer, Heidelberg (2008)
17. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. of LICS, pp. 55–74 (2002)
18. Touili, T.: Regular model checking using widening techniques. In: Proc. of VEPAS 2001 (2001)