12-2019

Compositional verification of heap-manipulating programs through property-guided learning

Long H. PHAM
Singapore University of Technology and Design

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Quang LOC LE
Teesside University

Compositional Verification of Heap-Manipulating Programs Through Property-Guided Learning

Long H. Pham[1(✉)], Jun Sun[2], and Quang Loc Le[3]

[1] Singapore University of Technology and Design, Singapore, Singapore
longph1989@gmail.com
[2] Singapore Management University, Singapore, Singapore
[3] School of Computing & Digital Technologies, Teesside University, Middlesbrough, UK

Abstract. Analyzing and verifying heap-manipulating programs automatically is challenging. A key for fighting the complexity is to develop compositional methods. For instance, many existing verifiers for heap-manipulating programs require user-provided specification for each function in the program in order to decompose the verification problem. The requirement, however, often hinders the users from applying such tools. To overcome the issue, we propose to automatically learn heap-related program invariants in a property-guided way for each function call. The invariants are learned based on the memory graphs observed during test execution and improved through memory graph mutation. We implemented a prototype of our approach and integrated it with two existing program verifiers. The experimental results show that our approach enhances existing verifiers effectively in automatically verifying complex heap-manipulating programs with multiple function calls.

1 Introduction

Analyzing and verifying heap-manipulating programs (hereafter heap programs) automatically is challenging [45]. Given the complexity, the key is to develop compositional methods which allow us to decompose a complex problem into smaller manageable ones. One successful example is the Infer static analyzer [1], which applies techniques like bi-abduction for local reasoning [36] to infer a specification for each function in a program to be analyzed.

While Infer generates function specifications for identifying certain classes of program errors, we aim to develop compositional methods for the more challenging task of verifying heap programs with data structures. In recent years, there have been multiple tools developed to verify heap programs in a compositional way, including Dafny [31], GRASShopper [43,44] and HIP [10]. These tools are, however, far from being applicable to real-world complex programs. One reason is that substantial user effort is needed. In particular, besides providing a specification to verify against, users must provide auxiliary specification to decompose the verification problem. For instance, Dafny, GRASShopper and HIP all require users to provide a specification for each function used in the program. Writing the function specification is highly non-trivial. It is thus desirable to develop approaches for verifying heap programs in a compositional way which requires minimum user effort.

A. W. Lin (Ed.): APLAS 2019, LNCS 11893, pp. 405–424, 2019.
https://doi.org/10.1007/978-3-030-34175-6_21

In this work, we propose to automatically generate function specifications for compositional verification of heap programs. Our approach differs from existing approaches like Infer in three ways. Firstly, because our goal is to verify the correctness of heap programs with *data structures*, our approach generates more expressive function specifications than those generated by Infer.

Secondly, we learn a specification of each function call (rather than each function) in a property-guided way. For instance, assume that we have the following verification problem (expressed in the form of a Hoare triple) $\{pre\}func(); func(); \{post\}$ where pre is a precondition, $post$ is a postcondition and $func(); func()$ are two consecutive calls of the same function. We automatically generate a program invariant inv after the first function call and before the second function call. As a result, we generate the specification $\{pre\}func()\{inv\}$ for the first function call and the specification $\{inv\}func()\{post\}$ for the second function call. The (smaller) problems of verifying these two Hoare triples thus replace the problem of verifying the original Hoare triple.

Thirdly, our invariant generation method is based on a novel technique, namely, a combination of classification and memory-graph mutation. We start with generating multiple random test cases (based on existing methods [37]). We then instrument the program and execute the test cases to obtain values of multiple features which are related to the memory graphs before and after each function call in the program. The obtained feature vectors are labeled according to the testing results (i.e., whether the postcondition is satisfied or not). Then we apply a classification algorithm [8] to find an invariant that separates the feature vectors with different labels. The invariant is an arbitrary boolean formula of the features, which is then used to decompose the verification problem.

There are two technical challenges which we must solve in order to make the above approach work. First, what features of the memory graphs shall we use? In this work, we adopt an expressive specification language for heap programs which combines separation logic, user-defined inductive predicates and arithmetic [10,23,27,45]. We then define a set of features based on the specification language. In addition, our approach allows users to define their own features. Secondly, how do we solve the problem of the lack of labeled samples, i.e., the test cases which we learn from may be limited. To overcome the problem, we mutate the memory graphs according to the learned invariant to validate whether the learned invariant is correct. We refine the invariant based on the validation result (if necessary) and repeat the process until the invariant is validated.

We implement our idea in a prototype, called SLearner, which takes a program to be verified as input, generates multiple invariants and outputs a set of decomposed verification tasks. We integrate SLearner with two existing state-of-the-art verifiers for heap programs, i.e., GRASShopper and HIP. Experiments are then performed on 110 programs manipulating 10 challenging data structures. The experimental results show that, enhanced with our approach, both GRASShopper and HIP are able to successfully verify programs with multiple function calls without user-provided function specifications.

The novelty of our work is in learning heap-related specification in a property-guided way and applying graph mutation to improve the learning process. The rest of the paper is organized as follows. Section 2 presents an illustrative example. Section 3 presents the details of our approach. Section 4 evaluates our approach. Section 5 reviews related work. Finally, Sect. 6 concludes.

```
1  public void main(int m, int n) {
2    //precondition : m ≤ n
3    Node x = createSLL(m);
4    Node y = createSLL(n);
5    getSum(x, y);
6    //postcondition : sll(x, _)*sll(y, _)
7  }
8  private Node createSLL(int n) {
9    if (n <= 0) return null;
10   else {
11     Node x = new Node(n, null);
12     x.next = createSLL(n − 1);
13     return x;
14   }
15 }
16 private int getSum(Node x, Node y) {
17   int sum = 0;
18   if (x != null) {
19     sum += x.data + y.data;
20     sum += getSum(x.next, y.next);
21   }
22   return sum;
23 }
```

Fig. 1. An illustrative example

2 An Illustrative Example

In this section, we illustrate our approach with an example. The program is shown as function `main` in Fig. 1, where function `createSLL`(n) returns a singly-linked list with length n and function `getSum`(x, y) returns the sum of the `data` in two disjoint singly-linked list objects (pointed to by the two pointers x and y). Note that both functions are recursively defined. The precondition and postcondition are shown at line 2 and 6 respectively. They are specified in an assertion language based on separation logic (refer to details in Sect. 3). The precondition is self-explanatory. The postcondition `sll`$(x, \_)$*`sll`$(y, \_)$ intuitively means that x and y are two disjoint singly-linked list objects, i.e., `sll`(x, n) is an inductive predicate denoting that x is a singly-linked list object with n nodes, and $*$ is the separating conjunction predicate specifying the disjointness in separation logic. Besides the postcondition, we assume that memory safety is always implicitly asserted and thus must be verified. For instance, we aim to verify that `x.data` at line 19 would not result in null-pointer de-referencing.

Our experiment shows that state-of-the-art verifiers like GRASShopper and HIP cannot verify this program. Only after specifications for both functions `createSLL` and `getSum` are provided manually, the program is verified. On one hand, providing a specification for every function called by the given program is highly nontrivial. On the other hand, part of the function specification may be irrelevant to verifying the given program. For an extreme example, if we change the postcondition of the program shown in Fig. 1 to `true`, a complete specification for singly-linked lists would not be necessary to verify the program.

Our approach is to automatically learn a just-enough invariant before and after each function call so that we can verify the program in a compositional way. For this example, we learn two invariants: $\texttt{inv}_1$ right after the first function call at line 3 and $\texttt{inv}_2$ right after the second function call at line 4. Next, we verify the program by verifying the following three Hoare triples: $\{m \leq n\}$`createSLL(m)`$\{\texttt{inv}_1[\texttt{res}/x]\}$; $\{\texttt{inv}_1\}$`createSLL(n)`$\{\texttt{inv}_2[\texttt{res}/y]\}$; and $\{\texttt{inv}_2\}$`getSum`$\{$`sll`$(x, \_)*$`sll`$(y, \_)\}$ with `res` is a special variable for the return value of a function and $\texttt{inv}_1[\texttt{res}/x]$ is a substitution of all variable x in $\texttt{inv}_1$ by variable `res`. As the program in each Hoare triple involves only one function, existing verifiers like GRASShopper and HIP can automatically verify the Hoare triples.

Table 1. Collected feature vectors and labels

	$\texttt{is\_sll}(x)$	$\texttt{is\_sll}(y)$	$\texttt{is\_sll}(x) \wedge \texttt{is\_sll}(y) \wedge \texttt{sep}(x, y)$	$\texttt{len\_sll}(x) \leq \texttt{len\_sll}(y)$	label
m=1, n=0	true	true	true	false	negative
m=0, n=1	true	true	true	true	positive

To learn $\texttt{inv}_1$ and $\texttt{inv}_2$, we instrument the program to collect a set of features at the learning points and collect their values during test executions. For instance, Table 1 shows a few of the features and their values for the above program after line 4 for learning $\texttt{inv}_2$. The first row shows the features and the second and third rows show the values of the features given two test cases $\{m{=}1, n{=}0\}$ and $\{m{=}0, n{=}1\}$ respectively. The features are designed based on our assertion language. In particular, feature $\texttt{len\_sll}(x)$ is a numeric value denoting the length of a singly-linked list x which is extracted based on the user-defined predicate `sll`; feature $\texttt{is\_sll}(x)$ denotes whether x points to a singly-linked list, and feature $\texttt{sep}(x, y)$ denotes whether x and y are disjoint in the heap. We label each feature vector with either *negative* or *positive*, where *negative* means that a memory error is generated, the postcondition is violated, or the test case likely runs into infinite loop (i.e., it does not stop after certain time units); and *positive* means otherwise.

Next, we apply a classification algorithm [8] to generate a predicate which separates the *positive* and *negative* feature vectors. The predicate takes the form of an arbitrary boolean formula of the features. Given the feature vectors in Table 1, the generated predicate is: $\texttt{len\_sll}(x) \leq \texttt{len\_sll}(y)$. Although this predicate is an invariant after line 4, it is not strong enough to verify the postcondition. This is in general a problem due to having a limited number of test cases. To solve the problem, we systematically mutate the memory graphs obtained during the test executions to obtain more labeled feature vectors with the aim to improve the predicate (see details in Sect. 3.5). In our example, with the additional feature vectors, the classification algorithm generates the following predicate for $\texttt{inv}_2$.

$$(\texttt{is\_sll}(x) \wedge \texttt{is\_sll}(y) \wedge \texttt{sep}(x, y) \wedge x{=}\texttt{null}) \vee \\ (\texttt{is\_sll}(x) \wedge \texttt{is\_sll}(y) \wedge \texttt{sep}(x, y) \wedge \texttt{len\_sll}(x) \leq \texttt{len\_sll}(y))$$

We obtain $x{=}\texttt{null} \vee (\texttt{is\_sll}(x) \wedge \texttt{len\_sll}(x) \leq n)$ similarly for $\texttt{inv}_1$ after line 3.

Afterwards, $\texttt{inv}_1$ and $\texttt{inv}_2$ are translated into the formulas in our assertion language. Note that the translation is straightforward since the features are designed based on the assertion language. The last step is to verify three verification problems. This is done using state-of-the-art verifiers for heap programs. For instance, HIP solves the three verification problems automatically, which verifies the program.

For efficiency, in the verification step we perform the following two simplifications. First, for dead code detection, we invoke a separation logic solver (e.g., the one presented in [27,29]) to check the satisfiability of inferred invariant. Secondly, we identify and eliminate the frame of a Hoare triple before sending them to the verifiers. For example, for the Hoare triple $\{\texttt{inv}_1\}\texttt{createSLL(n)}\{\texttt{inv}_2[\texttt{res}/y]\}$, we find that x has not been accessed by the code, the occurrences of the singly-linked list x in both the precondition and postcondition of the triple are eliminated before sending it to the verifiers.

$$
\begin{array}{l}
\Phi ::= \Delta \mid \Phi_1 \vee \Phi_2 \\
\Delta ::= \exists \bar{v} \cdot (\kappa \wedge \pi) \\
\kappa ::= \texttt{emp} \mid r{\mapsto}c(\bar{t}) \mid \texttt{P}(\bar{t}) \mid \kappa_1 * \kappa_2 \\
\pi ::= \texttt{true} \mid \phi \mid \neg\pi \mid \pi_1 \wedge \pi_2
\end{array}
\qquad
\begin{array}{l}
\phi ::= i \mid v{=}\texttt{null} \\
i ::= a_1{=}a_2 \mid a_1{\leq}a_2 \\
a ::= k \mid v \mid k{\times}a \mid a_1{+}a_2 \mid -a
\end{array}
$$

Fig. 2. Syntax: where c is a data type; k is an integer value; t_i, v, r are variables; and $\bar{t}$ is a sequence of variables

3 Our Approach

3.1 Problem Definition

Our input is a Hoare triple $\{pre\}prog\{post\}$, where pre is a precondition, $post$ is postcondition and $prog$ is a heap program which may invoke other functions. One example is the function `main` shown in Fig. 1. The precondition and postcondition are in an expressive specification language previously developed in [10,23,27,45]. The language supports separation logic, inductive predicates and Presburger arithmetic [19], which is shown to be expressive to capture many properties of heap programs.

The syntax of the language is presented in Fig. 2. In general, a predicate Φ in this language is a disjunction of multiple symbolic heaps. A symbolic heap Δ is an existentially quantified conjunction of a heap formula κ (i.e., a predicate constraining the memory structure) and a pure formula π (i.e., a predicate constraining numeric variables). A heap formula κ is an empty heap predicate `emp`, a points-to predicate $r{\mapsto}c(\bar{t})$ (where r is its root variable), a user-defined predicate $\texttt{P}(\bar{t})$, or a spatial conjunction of two heap formulas $\kappa_1 * \kappa_2$. User-defined predicates are defined in the same language. A pure formula π can be `true`, an (in)equality on variables, a Presburger arithmetic formula, negation of a formula, or their conjunction. We refer the readers to [19] for details on Presburger arithmetic. We note that $v_1{\neq}v_2$ (resp. $v{\neq}\texttt{null}$) is used to denote $\neg(v_1{=}v_2)$ (resp. $\neg(v{=}\texttt{null})$) and we may use _ to indicate "don't care" values.

For instance, the following predicate $\texttt{sll}(x,n)$ defines a singly-linked list (with a root-pointer x and size n), which is used in the illustrative example.

$$
\begin{array}{l}
\texttt{sll}(x,n) \equiv (\texttt{emp} \wedge x{=}\texttt{null} \wedge n{=}0) \\
\qquad\qquad \vee (\exists\, q, n_1 \cdot x{\mapsto}Node(\_, q) * \texttt{sll}(q, n_1) \wedge n{=}n_1{+}1)
\end{array}
$$

Our problem is to automatically verify the Hoare triple. Different from existing approaches, we aim to do that in a compositional way without user-provided function specifications.

3.2 Test Generation and Code Instrumentation

Given $\{pre\}prog\{post\}$, we first automatically generate a test suite S using existing test case generation methods like [37]. Note that we do not require the test cases to satisfy the precondition because negative feature vectors from invalid test cases will be filtered out by our learning process. Based on the testing results, we divide S into two disjoint sets. One set includes passed test cases that terminate normally without any

memory error or violation of the postcondition, denoted as S^+. The other set contains the remaining ones, denoted as S^-. Note that we heuristically consider that a test case does not terminate after waiting for a threshold number of time units. Afterwards, we identify all function calls in $prog$ and add learning points before and after each call. At each learning point l, we identify a set of relevant variables, denoted as V_l. We apply static program slicing to remove the variables which are visible at l but irrelevant to the postcondition or memory safety. In the example shown in Fig. 1, the sets of relevant variables at learning point 1 and 2 are $\{x, n\}$ and $\{x, y\}$ respectively. For each learning point, we instrument the program to extract a vector of features from each test.

3.3 Feature Extraction

Central to our approach is the answer to the question: what features to extract? In this work, we view a program state as a memory graph and systematically extract two groups of features based on the memory graph. One group contains generic features of the memory graph and the other contains features which are specific to the verification task. Formally, a memory graph G is a tuple $(M, init, E, Ty, L)$ such that

- M is a set of heap nodes including a special node `null`;
- $init \in M$ is a special initial node;
- E is a set of labeled and directed edges such that $(s, n, s') \in E$ means that we can access heap node s' via a pointer named n from s. An edge starting from $init$ is always labeled with one of the variables in the program.
- Ty is a total labeling function which labels each heap node in M by a type;
- and L is a labeling function which labels a heap node of primitive type by a value.

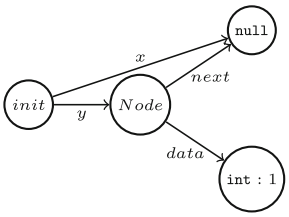

Fig. 3. A memory graph

Given a test case and a learning point, we represent the program state at the learning point during the test execution in the form of a memory graph $(M, init, E, Ty, L)$. Figure 3 shows the memory graph for our example at the learning point 2 with test input $m = 0$ and $n = 1$. Note that any rooted path of a memory graph represents a variable, e.g., the path with the sequence of labels $\langle y, next \rangle$ in the above memory graph is a variable $y.next$ at learning point 2. For complicated programs, the memory graph might contain many paths and thus many variables from which we can extract features. We thus set a bound on the number of de-referencing to limit the number of variables. For example, if we set the bound to be 2, we focus on variables $\{x, x.data, x.next, y, y.data, y.next\}$ at learning point 2 and similarly variables $\{x, x.data, x.next, n\}$ at learning point 1. With length bounded to 1, we focus only on $\{x, y\}$ at learning point 2 and $\{x, n\}$ at learning point 1.

We extract two groups of boolean features based on the memory graph. The first group contains generic heap-related features, which include the following.

- For each reference type variable x, we extract two features which represent if it is `null` or not, i.e, whether its corresponding path leads to the special node `null`.

Table 2. Features

#	feature	#	feature	#	feature
1	$x = \texttt{null}$	10	$x \mapsto Node() \wedge \texttt{is\_sll}(y) \wedge \texttt{sep}(x, y)$	19	$\texttt{len\_sll}(x) + \texttt{len\_sll}(y) > 0$
2	$y = \texttt{null}$	11	$\texttt{is\_sll}(x) \wedge y \mapsto Node() \wedge \texttt{sep}(x, y)$	20	$\texttt{len\_sll}(x) - \texttt{len\_sll}(y) > 0$
3	$x \mapsto Node()$ (a.k.a. $x \neq \texttt{null}$)	12	$\texttt{is\_sll}(x) \wedge \texttt{is\_sll}(y) \wedge \texttt{sep}(x, y)$	21	$-\texttt{len\_sll}(x) + \texttt{len\_sll}(y) > 0$
4	$y \mapsto Node()$ (a.k.a. $y \neq \texttt{null}$)	13	$\texttt{len\_sll}(x) > 0$	22	$-\texttt{len\_sll}(x) - \texttt{len\_sll}(y) > 0$
5	$x = y$	14	$\texttt{len\_sll}(y) > 0$	23	$\texttt{len\_sll}(x) + \texttt{len\_sll}(y) = 0$
6	$x \neq y$	15	$\texttt{len\_sll}(x) < 0$	24	$\texttt{len\_sll}(x) - \texttt{len\_sll}(y) = 0$
7	$\texttt{is\_sll}(x)$	16	$\texttt{len\_sll}(y) < 0$	25	$-\texttt{len\_sll}(x) + \texttt{len\_sll}(y) = 0$
8	$\texttt{is\_sll}(y)$	17	$\texttt{len\_sll}(x) = 0$	26	$-\texttt{len\_sll}(x) - \texttt{len\_sll}(y) = 0$
9	$x \mapsto Node() \wedge y \mapsto Node() \wedge \texttt{sep}(x, y)$	18	$\texttt{len\_sll}(y) = 0$		

- For each pair of reference type variables, we extract two features which represent if the two variables are aliasing or not, i.e., whether their corresponding paths lead to the same non-null node.
- For each pair of reference type variables, we extract a feature which represents whether two variables are separated in the memory. Assume that variables x and y lead to nodes n_x and n_y, x and y are separated, denoted as $\texttt{sep}(x, y)$, if and only if all reachable nodes except `null` from n_x (including n_x) are not reachable from n_y and vice versa.
- For each pair of the numeric variables, we extract boolean features in difference logic and the octagon abstract domain [34], e.g., $\pm$x $\pm$y >c, $\pm$x $\pm$ y = c, $\pm x > c$ or $x = c$ where c is a constant. We apply a heuristic to collect constants in conditional expressions in the given program as candidate values for c. The value 0 is chosen by default.

While general heap-related features are often useful, some programs can only be proven with features which are specific to the verification problem. Thus, we extract a second group of features based on user-defined predicates used to assert the correctness of the given program, which include the following.

- For every permutation of n variables, we extract a feature which represents whether the variables satisfy the predicate. For instance, given the user-defined predicate `sll` which has one reference type parameter, we extract a feature which represents whether x satisfies the predicate, for each reference variable x.
- For a pair of two sequences of variables X and Y which satisfy some user-defined predicates, we extract a feature which represents whether the variables are separated in the memory, i.e., all nodes reachable from any variable in X (except `null`) are not reachable from any node in Y and vice versa. This feature is inspired by the separation conjunction operator $*$ in our assertion language. For instance, given x and y which both satisfy `is_sll`, this feature value is true if and only if all objects in the singly-linked list x and singly-linked list y are disjoint in memory. Note that this feature subsumes the feature $\texttt{sep}(x, y)$ explained above.
- For each numeric parameter of the user-defined predicate, we use a variable to represent its value for each sequence of variables which satisfy the predicate. For instance, as `sll` has a numeric parameter, if variable x satisfies `sll`, we use a fresh variable

(denoted as `len_sll` for readability) to represent the value of the numeric parameter. Boolean features of these numeric variables, together with existing numeric variables, are then extracted in the chosen abstract domains.

In general, user-defined predicates can be complicated. Existing heap program verifiers like GRASShopper and HIP maintain a library of commonly used predicates. We adopt the predicates in their library and define the corresponding functions to extract the above-mentioned features in the form of an extensible library for our approach. Note that this is a one-time effort. For instance, Table 2 shows the list of 26 features which we extract at learning point 2 for the program shown in Fig. 1.

3.4 Learning for Compositional Verification

In the following, we present our approach on learning an invariant based on the extracted feature vectors. Recall that we systematically instrument the program at every learning point, then extract a value for every feature we discussed above. In our implementation, each feature is extracted using a function which returns a boolean value. Afterwards, each test case is executed so that we collect a vector of boolean values (a.k.a. a feature vector) which represents an abstraction of the memory graph according to the chosen features. If the test case finishes successfully, the feature vector is labeled $positive$; otherwise, it is labeled $negative$. The labeled feature vectors can be organized into a matrix M whose rows are feature vectors and whose columns are the feature values in all test cases. To ensure all feature vectors have the same dimension, if a feature does not apply (e.g., a variable is not accessible in the test case), we set the corresponding feature value to a special default value. For instance, Table 3 shows the matrix where the features are sequenced in the same order of Table 2.

The first step in our learning process is normalising the matrix M. If there are two rows with the same feature values and same labels, one of them is redundant and removed. Next, we apply the algorithm in [8] to learn a boolean combination of features to separate positive and negative vectors. Informally, the algorithm considers each feature vector as a point in space and every positive point is connected to every negative point by a line. A feature 'cuts' a line if the corresponding positive point and negative point have different values for the feature. The goal is to find a list of features that can cut all the lines, i.e., separate all positive and negative points. The features are chosen using a greedy algorithm. At each step, the feature which cuts the most number of uncut lines is selected. After all lines are cut, the selected features partition the space into multiple regions, each of which contains either positive points only or negative points only. Each region can be characterised by a conjunction of the features and the disjunction of all the formulas characterising the positive regions is a boolean formula which separates all the positive and negative feature vectors.

The details are shown in Algorithms 1 and 2. In Algorithm 1, the input is a normalised matrix M and the output is the list of features K which can classify all positive and negative rows in M. K is initialised as an empty list (line 1). A list L is initialized to contain all pairs of rows (i, j) such that i is the index of a positive row and j is that of a negative row (line 1). During each iteration, the feature k that 'cuts' the most number of pairs in L is identified (line 3). Note that we do not consider the case

Table 3. Matrix of feature vectors

Vectors obtained from test cases

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	Label
1	**0**	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**positive**
1	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**positive**
0	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**positive**
0	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**negative**

Vectors obtained from memory graph mutation

0	0	1	1	0	1	1	1	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	1	1	0	positive
0	0	1	1	0	1	0	1	1	1	0	0	N	1	N	0	N	0	N	N	N	N	N	N	N	N	negative
0	1	1	0	0	1	0	1	0	1	0	0	N	0	N	0	N	1	N	N	N	N	N	N	N	N	negative
0	0	1	1	0	1	1	1	1	1	1	1	1	1	0	0	0	0	1	1	0	0	0	0	0	0	negative
1	0	0	1	0	1	1	0	0	0	1	0	0	N	0	N	1	N	N	N	N	N	N	N	N	N	negative
0	0	1	1	0	1	1	0	1	0	1	0	1	N	0	N	0	N	N	N	N	N	N	N	N	N	negative
0	0	1	1	1	0	1	1	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	1	1	0	negative

Algorithm 1: Choose the list of features `choose`(M)

1 $K = \{\}$; $L = \{(i, j) \mid$ row i is positive and row j is negative$\}$;
2 **while** L *is not empty* **do**
3 Find k s.t. $\{(i, j) \in L \mid M_{ik} = 1 \wedge M_{jk} = 0\}$ is the largest;
4 **if** *the number of pairs* (i, j) *that* k *can classify is* 0 **then**
5 Stop and ask for user input for a new feature;
6 **else**
7 Remove (i, j) s.t. $M_{ik} = 1 \wedge M_{jk} = 0$ from L;
8 Add k to K;
9 Return K;

$M_{ik} = 0 \wedge M_{jk} = 1$ because it will create the negations of features, which may not be easily transformed into separation logic. We then remove from L the pairs that are classified correctly by k (line 7) and add the new feature k into K (line 8). The loop stops when L is empty (line 2) or the best feature at the current iteration cannot classify more pairs (line 4). In the former case, we return the list of features K (line 9). In the latter case, it means the features are not sufficient to distinguish all positive and negative rows. We thus stop and may ask users to provide a new feature (line 5).

Algorithm 2 then shows how a boolean formula that classifies all positive and negative rows in M is constructed from the chosen features. The input is a normalised matrix M and a list of features K chosen using Algorithm 1 and the output is a boolean combination of these features. Initially, the list of regions R is empty; PP and NP are the set of indexes of positive and negative rows respectively (line 1). Recall that each row can be seen as a point in space. All points in PP are marked as uncovered at line 2. Favoring simple hypothesis (which is a heuristics often applied in machine learning),

Algorithm 2: Combine the features `combine`(M, K)

```
1  R = {}; PP = {p | row p is positive}; NP = {n | row n is negative};
2  Mark all p ∈ PP as uncovered;
3  for i = 1 to |K| do
4      Create all combinations C with i elements from the list of features K;
5      for each combination c ∈ C do
6          if ∀n ∈ NP ∃k ∈ c : M_nk = 0 then
7              CP = {p | p ∈ PP and ∀k ∈ c : M_pk = 1};
8              if CP contains at least one uncovered index then
9                  Remove from R the combinations that have the covered indexes are
                   proper subsets of CP;
10                 Add c to R; Mark all p ∈ CP as covered;
11                 if all p ∈ PP are covered then
12                     Return R;
```

we try the combination from 1 feature to $|K|$ (which is the number of features in K) features (line 3). At line 4, all the combinations of i features are created. For each combination (line 5), we check if the created region contains no negative points (line 6). If it is the case, we find a list of positive points that are covered by the region (line 7). If this region contains at least one uncovered point (line 8), we add this combination into R and mark the positive points in the region as covered (line 10). Line 9 simplifies the results by removing the chosen regions that only cover a proper subset of positive points in the new region. When all positive points are covered, we return the set of combinations R (lines 11 and 12). Each combination is a conjunction of features and the set of combinations is the disjunction of these conjunctions.

For our example, at the learning point 2, after removing redundant rows, we have a matrix with 4 rows and 26 columns, i.e., the bolded rows in Table 3. Rows 1, 2 and 3 are positive, whereas row 4 is negative. To separate these rows, two columns 1 and 4 are chosen. From this, we can form two regions, in particular, the first one with only column 1, the second one with only column 4. These two columns represent feature $x = \texttt{null}$ (column 1) and $y \neq \texttt{null}$ (column 4). As a result, we learn the predicate $x = \texttt{null} \vee y \neq \texttt{null}$. Note that this predicate is incorrect and it is to be improved later.

It can be shown that Algorithms 1 and 2 always terminate. The worst-case complexity of Algorithm 1 is $\mathcal{O}(Row^4 * Col)$ where Row and Col are the number of rows and columns in the input matrix respectively. For Algorithm 2, the worst-case complexity is $\mathcal{O}(2^{|K|} * (Row * Col + Row^3))$. While the worst-case complexity is high, these algorithms are often reasonably efficient (as we show in our empirical study). The main reason is that the number of features K (which dominates the overall complexity) is often small (average 1.05 in our experiments).

3.5 Automatic Memory Graph Mutation

Recall that we only need a correct predicate, which is an invariant at the learning point and sufficient to prove the postcondition. A fundamental limitation of using classification techniques is that the learned predicate is likely incorrect if the feature vectors (i.e., test cases) are insufficient. One way to solve this problem is to use a program verifier to check whether the predicate is correct. If it is not correct, the verifier would generate a counterexample and the learning process can continue with a new feature vector obtained from the counterexample. This approach is not ideal for two reasons. One is that verifying heap programs is often costly and thus we would like to avoid it as long as possible. The other is that it is highly nontrivial to construct counterexamples when verifying heap programs [5].

Because of that, in this work, to improve the learned predicate, we apply an idea similar in spirit to [11] to automatically mutate the memory graphs obtained from the test cases and generate more program states. For each learned predicate Φ, we systematically apply a set of mutation operators based on Φ. For each variable x in Φ, if it is a reference type, the following mutation operators are applied.

1. Point x to a freshly constructed object of the right type.
2. Point x to a heap node of the right type in the memory graph (including `null`).
3. Swap x with another reference type variable.

If x is a primitive type, we follow the idea in [42] and mutate it by setting it to a constant, increasing/decreasing its value with a pre-defined offset, or swapping it with another primitive variable. The number of mutants we generate depends on the current learned predicate.

These mutation operators are designed to create states which potentially invalidate the learned predicate. For instance, if the current predicate is $\texttt{is\_sll}(x) \wedge \texttt{is\_sll}(y) \wedge \texttt{sep}(x, y)$, where x and y are two reference variables, applying the mutation operators allows us to obtain memory graphs which invalidate $\texttt{is\_sll}(x)$, $\texttt{is\_sll}(y)$ and/or $\texttt{sep}(x, y)$. The expectation is that such a mutated program state would lead to violation of the postcondition and thus be labeled with $negative$. If our expectation is met, the predicate is now more likely to be correct; otherwise, the predicate is incorrect and is refined with the new feature vector.

In the extreme cases when all feature vectors are labeled $positive$ or $negative$, the learned predicates are `true` or `false` respectively. We then apply all mutation operators to all variables at the learning point. In our implementation, the mutation is done automatically by instrumenting statements which mutate the according variables at the learning point. We then run the test suite with the mutated program, collect new feature vectors and new test results. These new feature vectors are added into the matrix to learn new predicates.

The mutation at a learning point in the middle of the program may result in program states which may not be reachable. As a result, the final learned predicate, which is expected to be an invariant, may be weaker than the actual one (if the mutated program state is labeled as $positive$). However, a weaker invariant may still serve our goal of verifying the program. To give an example, in the extreme case, if the postcondition is

true (and there is no risk of memory error), it is sufficient to learn the invariant true. We repeat this process of mutation and learning until the learned invariant converges.

For our example, at the learning point 2, after obtaining the first predicate $x = \texttt{null} \vee y \neq \texttt{null}$, we apply mutation and obtain more feature vectors. The new feature vectors are shown in Table 3 where N is a special value denoting that the feature is not applicable. Next, applying Algorithm 1, the chosen features this time are $x = \texttt{null}$, $\texttt{is\_sll}(x) \wedge \texttt{is\_sll}(y) \wedge \texttt{sep}(x, y)$, $\texttt{len\_sll}(x) < \texttt{len\_sll}(y)$, and $\texttt{len\_sll}(x) = \texttt{len\_sll}(y)$ (column 1, 12, 21 and 24). From these 4 columns, we form 3 regions: $\{12, 1\}$, $\{12, 21\}$ and $\{12, 24\}$, which are transformed into the invariant $\texttt{inv}_2$ we show in Sect. 2. Similarly, with the help of state mutation, we improve the learned invariant at l_1 from $x = \texttt{null} \vee n > 0$ to $x = \texttt{null} \vee (\texttt{is\_sll}(x) \wedge \texttt{len\_sll}(x) \leq n)$.

The process of mutation and learning always terminates. As we only have a finite set of variables and features, the set of feature vectors is finite and thus the process of mutation converges eventually. Furthermore, matrix normalisation guarantees we do not have redundant rows in the matrix and, hence, the matrix is finite and the learning process always terminates.

3.6 Compositional Verification

Lastly, we show how we use the learned invariants to verify heap programs in a compositional way. Firstly, we transform each loop in the program into a fresh tail recursive function. Then the loop is replaced with a call to the corresponding function. Note that in the case of nested loops, we create multiple functions in which the function according to the outer loop will call the function according to the inner loop. This is a standard strategy adopted from existing program verifiers for heap programs [10]. We then treat loops in the same way as (recursive) function calls.

Secondly, we identify the learning points, i.e., before and after each function call statements and learn invariants at these points. Note that we do not learn before/after recursive function calls. This is because program verifiers for heap programs like GRASS-hopper and HIP support inductive reasoning and thus one specification for each recursive function is sufficient. Assume that the invariant learned before function call C_i is I_i and the one learned after C_i is I_{i+1}.

Thirdly, for each function call C_i, we generate a proof obligation in the form of a Hoare triple $\{I_i\}C_i\{I_{i+1}\}$, to prove that calling function C_i with I_i being satisfied results in a state satisfying I_{i+1}. Each proof obligation is submitted to a program verifier. Once the proof obligation is discharged, we replace the function call C_i with its now-established specification, i.e., two statements $\texttt{assert I}_\texttt{i}\texttt{;}$ $\texttt{assume I}_{\texttt{i+1}}$. That is, we instrument the learned invariants into the program such that the invariant learned before/after C_i becomes an assert/assume-statement respectively.

Finally, we use an existing program verifier to verify the transformed program. Note that the program does not contain any function call (other than possibly a recursive call of itself) now. It is straightforward to see that the program satisfies the postcondition and is memory-safe with the precondition if all proof obligations are discharged and the transformed program is verified. If any part is not proved and a counterexample is constructed by the verifier, we use the counterexample to learn new invariants and then try to prove new Hoare triples.

Table 4. Results on GRASShopper (Gh)

				Gh		Gh+SLearner			Gh+SLearner-Mutation		
Data structure	Functions	#Calls	#Progs	#V	Time(s)	#V	Time(s)	L Time(s)	#V	Time(s)	L Time(s)
Singly-linked list	Traverse, Dispose, Insert, Remove, Concat	1	5	5	1.50	5	1.50	0	5	1.50	0
		2	12	0	-	12	4.97	202	0	-	32
		3	18	0	-	18	10.74	610	0	-	99
Sorted list	Traverse, Dispose, Insert	1	3	3	1.40	3	1.40	0	3	1.40	0
		2	6	0	-	6	4.94	152	4	2.71	12
		3	6	0	-	6	6.96	368	2	2.32	32
Binary tree	Traverse, Dispose, Insert	1	3	3	43.63	3	43.63	0	3	43.63	0
		2	6	0	-	4	90.23	134	4	90.23	12
		3	6	0	-	2	89.26	313	2	89.26	30

4 Implementation and Evaluation

Our approach has been implemented as a prototype, called SLearner, with 3070 lines of Java code. In the following, we evaluate SLearner to answer multiple research questions (RQ). All experiments are conducted on a laptop with one 2.20 GHz CPU and 16 GB RAM. To reduce the effect of randomness, we run each experiment 20 times with 10 random test cases each time.

RQ1: Can our approach enhance state-of-the-art verifiers for heap programs? We integrate SLearner into two state-of-the-art verifiers for heap programs: GRASShopper and HIP. Although GRASShopper and HIP target the same class of programs, their approaches differ in multiple ways, e.g., they provide a different library of user-defined predicates and they have different verification strategies. They thus allow us to check whether SLearner is general enough to support different program verifiers. We remark that alternative program verifiers like CPAChecker [6] and SeaHorn [20] target different classes of programs or program properties and hence are not applicable. The only other tool which is capable of verifying heap programs with heap-related specification is jStar [12], which is, however, no longer maintained.

We conduct two sets of experiments based on these two verifiers. Our first experiment is with GRASShopper. Although GRASShopper supports inductive predicates for describing data structures, unlike HIP, it does not support reasoning about separation logic directly. The inductive predicates in GRASShopper are defined based on first-order logic with some built-in predefined predicates. Due to GRASShopper's limitation, we conduct an experiment based on a set of benchmark programs in its distribution. All programs and experimental results are available at [2] and the tool is available at [3].

The GRASShopper distribution contains many functions for different types of data-structures. We focus on those non-trivial recursive functions with precondition and post-condition. To check how GRASShopper performs with and without SLearner, we generate a set of composite programs which randomly invoke one or more of these functions. The function call sequence is formed such that the postcondition of a previous function is identical (via syntactical checking) to the precondition of the subsequent function. The precondition of the composite program is composed from preconditions of invoked

functions and the postcondition of the last function in the call sequence is the postcondition of the composite program. In total, we generate 65 composite programs containing 1, 2 and 3 function calls.

Table 5. Results on HIP

		HIP			HIP+SLearner				HIP+SLearner-Mutation			
Data structure	Program	Result	#Succ	Time(s)	Result	#Succ	Time(s)	L Time(s)	Result	#Succ	Time(s)	L Time(s)
Singly-linked list	Clean	Fail	0	-	Succ	20	0.37	17	Fail	0	-	3
	Clone	Fail	0	-	Succ	20	0.45	17	Fail	0	-	3
	Min	Fail	0	-	Fail	0	-	17	Fail	0	-	2
	Reverse	Fail	0	-	Fail	0	-	17	Fail	0	-	3
	Sort	Fail	0	-	Fail	0	-	17	Fail	0	-	3
	Insert	Fail	0	-	Succ	20	0.42	38	Fail	0	-	3
	Delete	Fail	0	-	Succ	20	0.42	37	Fail	0	-	2
	Append	Fail	0	-	Succ	20	0.45	90	Fail	0	-	6
	GetLast	Fail	0	-	Succ	20	0.42	17	Fail	0	-	3
	GetSum	Fail	0	-	Succ	**15**	1.02	77	Fail	0	-	6
	ToDll	Fail	0	-	Succ	20	0.30	17	Fail	0	-	3
Doubly-linked list	Clean	Fail	0	-	Succ	20	0.43	17	Succ	20	0.43	3
	Clone	Fail	0	-	Succ	20	0.67	17	Succ	20	0.67	3
	Min	Fail	0	-	Fail	0	-	17	Fail	0	-	3
	Reverse	Fail	0	-	Fail	0	-	17	Fail	0	-	3
	Sort	Fail	0	-	Fail	0	-	17	Fail	0	-	3
	Insert	Fail	0	-	Succ	20	0.58	18	Succ	19	0.58	3
	Delete	Fail	0	-	Succ	20	0.65	17	Succ	20	0.65	3
	Append	Fail	0	-	Succ	20	0.40	92	Fail	5	-	6
Sorted list	Clean	Fail	0	-	Succ	20	0.35	17	Succ	20	0.37	3
	Clone	Fail	0	-	Succ	20	0.37	17	Succ	18	0.35	3
	Min	Fail	0	-	Succ	20	0.37	17	Succ	19	0.37	3
	Travel	Fail	0	-	Succ	20	0.54	17	Succ	18	0.54	2
	Insert	Fail	0	-	Fail	0	-	16	Fail	0	-	3
	Delete	Fail	0	-	Fail	0	-	18	Fail	0	-	3
Cycle list	Clean	Fail	0	-	Fail	0	-	17	Fail	0	-	3
	Min	Fail	0	-	Fail	0	-	17	Fail	0	-	3
	Travel	Fail	0	-	Succ	20	0.30	17	Fail	0	-	3
	ToSll	Fail	0	-	Fail	0	-	17	Fail	0	-	3
Binary tree	InOrder	Fail	0	-	Succ	20	0.43	16	Succ	20	0.43	2
	PreOrder	Fail	0	-	Succ	20	0.46	17	Succ	20	0.46	3
	PostOrder	Fail	0	-	Succ	20	0.45	17	Succ	20	0.45	3
	Min	Fail	0	-	Succ	20	0.51	17	Succ	20	0.51	3
	Max	Fail	0	-	Succ	20	0.51	17	Succ	20	0.51	3
	Prec	Fail	0	-	Succ	20	0.57	17	Succ	20	0.57	3
	Succ	Fail	0	-	Succ	20	0.57	17	Succ	20	0.57	3
	Insert	Fail	0	-	Succ	20	0.67	17	Succ	20	0.67	3
	Delete	Fail	0	-	Fail	0	-	22	Fail	0	-	3
AVL tree	Insert	Fail	0	-	Fail	0	-	17	Fail	0	-	3
	Delete	Fail	0	-	Fail	0	-	24	Fail	0	-	3
Red-black tree	Insert	Fail	0	-	Fail	0	-	22	Fail	0	-	3
	Delete	Fail	0	-	Fail	0	-	38	Fail	0	-	3
MCF	Travel	Fail	0	-	Fail	0	-	17	Fail	0	-	3
Rose tree	Travel	Fail	0	-	Fail	0	-	17	Fail	0	-	3
Tll	SetRight	Fail	0	-	Succ	20	2.40	16	Succ	19	2.40	2

Table 4 shows the results, where the first four columns show the type of data structure, the involved functions, the number of function calls and the number of programs in the category. The next column shows the result of GRASShopper without the help of SLearner, i.e., the program is verified using GRASShopper without the specification of each invoked function in the program. We measure the number of verified programs (column *#V*) and the time taken. The next column shows the results of GRASShopper enhanced with SLearner. No additional user-defined predicates besides those provided in GRASShopper are used in our experiments. Note that we extract features automatically based on the user-defined predicates in GRASShopper in the experiment.

Without SLearner, GRASShopper only verifies 11 (out of 65) programs with 1 function call. For the remaining 54 programs which have 2 or 3 function calls, GRASShopper fails to verify any of them. This is expected as GRASShopper is unable to derive the necessary function specification automatically. Enhanced with SLearner, GRASShopper verifies 59 (out of 65) programs. For all these programs, we learn the correct invariants in every one of the 20 runs.

The second experiment is with HIP. We generate 45 programs based on common operations for 10 different data structures. Each program consists of multiple function calls. Each program starts with a call of a constructor which creates an object of the target data structure (e.g., a singly-linked list), or a function which reads the data structure (e.g., checking whether the root node is `null`, or traveling through the data structure). Lastly, a function supported by HIP for this data structure is called which may modify the data structure. The postcondition of the program is the postcondition of the last function. The precondition is manually written and checked to guarantee that the program terminates and satisfies the postcondition without any memory error.

Table 5 shows the results, where column *Program* shows the last function called in the program. Column *HIP+SLearner* shows the results using HIP enhanced with SLearner. Note that we may not be able to learn the same invariants every time due to randomness in generating the initial set of test cases. Thus, we add a column *#Succ* to show how many times, out of 20, we are able to learn the invariant and verify the program. No additional user-defined predicates besides those defined in HIP are used in our experiments. Column *HIP* shows that without SLearner, none of these programs is verified. With SLearner, HIP successfully verifies 27 programs. In all but 1 case (highlighted with bold) we are able to learn the same invariant consistently.

RQ2: Which features are useful in verifying heap programs? We learn invariants based on two groups of features, i.e., general heap-related features and those specific to user-defined predicates. The question is whether these two groups of features are useful and whether there are other features which we could learn based on.

In total, SLearner learned 104 invariants (74 with GRASShopper and 30 with HIP) to help solving the verification tasks. Among them, 93 invariants (66 with GRASShopper and 27 with HIP) contain only features extracted based on the user-defined predicates (e.g., $ds(x)$ or $ds(x)*ds(y)$ with ds being a user-defined predicate). The remaining 11 invariants are additionally constituted with generic features (e.g., $x = \texttt{null}$ or $x \neq \texttt{null}$). None of the invariants is constituted with general heap-related features only. The results show that the user-defined predicates are important and invariants specific to a verification problem are needed for proving the program. Generic heap-related features are also necessary sometimes (in 11% of the cases).

A total of 24 programs (6 with GRASShopper and 18 with HIP) are not verified. There are two main reasons why they cannot be proved even with the help of SLearner. Firstly, some programs can only be verified with complex function specifications which require features that are not supported in SLearner. For example, to prove the remaining 6 programs in the experiment with GRASShopper, we need a feature characterizing the paths in the tree, which cannot be derived from user-defined predicates. This is similarly the case for experiments with HIP. One remedy is to extend our implementation with additional features through automatic lemma learning [28]. Secondly, there are programs that have a hierarchy of function calls, e.g., function calls within recursive functions. Some of the function calls occur under strict condition which is never satisfied by the test cases and thus we are unable to learn the specification of those function calls. This is a fundamental limitation of dynamic analysis approaches, which could be overcome with a comprehensive test suite from a systematic test case generation approach [39–41].

RQ3: Is memory graph mutation helpful? We compare the performance of the enhanced GRASShopper and HIP with and without memory graph mutation. The results are shown in the last columns of Tables 4 and 5. It can be observed that without memory graph mutation, the number of verified programs by GRASShopper is reduced from 59 to 23, and the number of verified programs by HIP is reduced from 27 to 17. It thus clearly shows that memory graph mutation helps to improve the correctness of the learned invariants. Furthermore, we observe that without memory graph mutation, it is more likely that different invariants are learned in different runs of the same experiments (refer to column *#Succ*). This is expected as without memory graph mutation, we cannot discard invariants which are the result of limited test cases.

RQ4: What is the overhead of invariant generation? We measure the time taken to learn the invariants. Columns *L Time* in Tables 4 and 5 show the results. In general, the learning time depends on the number of learning points, the complexity of the program and the initial test suite. Overall, the time required for learning is reasonable, ranging from seconds to minutes. In the most time consuming case, we spent 92 s to learn two invariants for program "doubly-linked list append". For most of the cases, the learning time is about 20 s for each learning point.

RQ5: Does our invariant generation approach complement existing ones? The most noticeable invariant generation tool for heap program is Infer [1]. However, Infer is not designed to support verification task. Instead, it generates generic specifications to capture the footprints of the pointers used in the functions based on bi-abduction. We apply Infer to generate specifications (e.g., pre/postconditions) for every function experimented above and notice that they are too weak for program verification.

Threats to Validity. Firstly, the set of programs used in our experiments are limited compared to real-world data-structure libraries. This is because state-of-the-art verifiers for heap programs are still limited to relatively simple programs due to the great difficulty in verifying heap properties. As our experiments show, SLearner successfully enhances

the capability of state-of-the-art heap program verifiers so that programs with multiple functions can be automatically verified. Secondly, SLearner only works when we have the right features in the learning process. We expect that applying lemma synthesis could help us obtain more features and overcome this limitation.

5 Related Work

The closest to our work is approach for invariant inference using dynamic analysis with separation logic abstraction [30]. Similar to our work, it generates invariant based on user-defined predicates (i.e., features in our work). In contrast to ours, it made use of positive features only and did not support mutation. Close to our work are proposals for automatic program verification using black-box techniques adopted from the machine learning community. In particular, the method presented in [47] is based on user-supplied templates. It is designed to learn specification for heap programs which ensures no memory errors. The approach in [32] proposes to learn features from graph-structured inputs based on neural networks. The authors showed an application on verifying memory safety using the learning results. In contrast to [32], our goal is to learn invariants to compositionally verify the program against a given specification as well as ensure no memory errors. In [25], the authors presented a method to learn shared module codes and reuse them during an analysis. The work in [16] builds polynomial time active learning algorithms for automaton model of array and list structures. Our proposal also relies on a learning algorithm and actively improves the learned invariants. In [35], the authors proposed a learning method targeted lists only. This method learns the sequence of actions (remove or insert) from a program and infers the data structures manipulated by the program. However, it is hard to extend the method to support arbitrary heap programs. Similarly to ours, [7] guesses invariants from concrete program states and checks them by a theorem prover. However, their work only focuses on list-based programs. The ICE method proposed in [17,18] supports inductive properties of loop invariant learning. Besides using the positive and negative points, ICE proposes additional implication points to encode the inductive checking for learning invariant. It is our future work to integrate the idea of ICE learning with our graph-based learning. The work in [38] presents an approach for precondition inference. The main contribution is feature learning for functional programs. It is interesting to apply the feature learning techniques in our future work.

Our work is also related to automatic and static analyzers for the shape analysis problems, e.g., TVLA [46] and separation logic [9,10,13,22,26], and for the verification problem of programs that requires both heap and data reasoning, e.g., PDR [24], interpolation [4] and template-based invariant generation [33]. To infer shape-based specification, while tools [9,13,26] are based on the bi-abduction technique, we use machine learning to obtain a generalized invariant from a set of concrete executions. In our implementation, we use GRASSHopper and HIP as external verification engines. As our approach is independent from the program verifiers, we plan to build a general framework so that different verifiers can be used. Lastly, this work is related to previous works on invariant generation, e.g., Daikon [14], or Houdini [15]. However, those works do not focus on learning invariants related to data structures like this one.

6 Conclusion

We have presented a novel learning approach to the automated and compositional verification of heap programs. The essence of our approach is an algorithm to infer invariants based on a set of memory graphs representing the program states obtained from concrete executing traces. We further enhance the precision of learned invariant with memory graph mutation. We have implemented a prototype tool and evaluated it over a set of programs which manipulate complex data structures. The experimental results show that our tool enhances the capability of existing program verifiers to verify non-trivial heap programs. In the future, we might apply our tool to more verifiers and more test subjects as well as compare our tool with other tools, e.g., Predator [13], Forester [21,22], S2 [26], and SLING [30].

Acknowledgments. This research is supported by MOE research grant MOE2016-T2-2-123.

References

1. Facebook Infer. https://fbinfer.com
2. https://figshare.com/s/ba1c12ad90c138fbb240
3. https://github.com/sunjun-group/Ziyuan
4. Albarghouthi, A., Berdine, J., Cook, B., Kincaid, Z.: Spatial interpolants. In: Vitek, J. (ed.) ESOP 2015, pp. 634–660 (2015). https://doi.org/10.1007/978-3-662-46669-8_26
5. Berdine, J., Cox, A., Ishtiaq, S., Wintersteiger, C.M.: Diagnosing abstraction failure for separation logic-based analyses. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012, pp. 155–173 (2012). https://doi.org/10.1007/978-3-642-31424-7_16
6. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011, pp. 184–190 (2011). https://doi.org/10.1007/978-3-642-22110-1_16
7. Brockschmidt, M., Chen, Y., Kohli, P., Krishna, S., Tarlow, D.: Learning shape analysis. In: Ranzato, F. (ed.) SAS 2017, pp. 66–87 (2017). https://doi.org/10.1007/978-3-319-66706-5_4
8. Bshouty, N.H., Goldman, S.A., Mathias, H.D., Suri, S., Tamaki, H.: Noise-tolerant distribution-free learning of general geometric concepts. J. ACM **45**(5), 863–890 (1998). https://doi.org/10.1145/290179.290184
9. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional Shape Analysis by Means of Bi-Abduction. J. ACM **58**(6), 26:1–26:66 (2011). https://doi.org/10.1145/2049697.2049700
10. Chin, W., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program **77**(9), 1006–1036 (2012). https://doi.org/10.1016/j.scico.2010.07.004
11. Cleve, H., Zeller, A.: Locating causes of program failures. In: Roman, G., Griswold, W.G., Nuseibeh, B. (eds.) ICSE 2005, pp. 342–351 (2005). https://doi.org/10.1145/1062455.1062522
12. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. In: Harris, G.E. (ed.) OOPSLA 2008, pp. 213–226 (2008). https://doi.org/10.1145/1449764.1449782
13. Dudka, K., Peringer, P., Vojnar, T.: Predator: a practical tool for checking manipulation of dynamic data structures using separation logic. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011, pp. 372–378 (2011). https://doi.org/10.1007/978-3-642-22110-1_29

14. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program **69**(1–3), 35–45 (2007). https://doi.org/10.1016/j.scico.2007.01.015
15. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001, pp. 500–517 (2001). https://doi.org/10.1007/3-540-45251-6_29
16. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: Sharygina, N., Veith, H. (eds.) CAV 2013, pp. 813–829 (2013). https://doi.org/10.1007/978-3-642-39799-8_57
17. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: a robust framework for learning invariants. In: Biere, A., Bloem, R. (eds.) CAV 2014, pp. 69–87 (2014). https://doi.org/10.1007/978-3-319-08867-9_5
18. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Bodík, R., Majumdar, R. (eds.) POPL 2016, pp. 499–512 (2016). https://doi.org/10.1145/2837614.2837664
19. Ginsburg, S., Spanier, E.: Semigroups, presburger formulas, and languages. Pac. J. Math. **16**(2), 285–296 (1966)
20. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Pasareanu, C.S. (eds.) CAV 2015, pp. 343–361 (2015). https://doi.org/10.1007/978-3-319-21690-4_20
21. Holík, L., Hruska, M., Lengál, O., Rogalewicz, A., Simácek, J., Vojnar, T.: Forester: from heap shapes to automata predicates - (competition contribution). In: Legay, A., Margaria, T. (eds.) TACAS 2017, pp. 365–369 (2017). https://doi.org/10.1007/978-3-662-54580-5_24
22. Holík, L., Lengál, O., Rogalewicz, A., Simácek, J., Vojnar, T.: Fully automated shape analysis based on forest automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013, pp. 740–755 (2013). https://doi.org/10.1007/978-3-642-39799-8_52
23. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: Hankin, C., Schmidt, D. (eds.) POPL 2001, pp. 14–26 (2001)
24. Itzhaky, S., Bjørner, N., Reps, T.W., Sagiv, M., Thakur, A.V.: Property-directed shape analysis. In: Biere, A., Bloem, R. (eds.) CAV 2014, pp. 35–51 (2014). https://doi.org/10.1007/978-3-319-08867-9_3
25. Kulkarni, S., Mangal, R., Zhang, X., Naik, M.: Accelerating program analyses by cross-program training. In: Visser, E., Smaragdakis, Y. (eds.) OOPSLA 2016, pp. 359–377 (2016). https://doi.org/10.1145/2983990.2984023
26. Le, Q.L., Gherghina, C., Qin, S., Chin, W.: Shape analysis via second-order bi-abduction. In: Biere, A., Bloem, R. (eds.) CAV 2014, pp. 52–68 (2014). https://doi.org/10.1007/978-3-319-08867-9_4
27. Le, Q.L., Sun, J., Chin, W.: Satisfiability modulo heap-based programs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016, pp. 382–404 (2016). https://doi.org/10.1007/978-3-319-41528-4_21
28. Le, Q.L., Sun, J., Qin, S.: Frame inference for inductive entailment proofs in separation logic. In: Beyer, D., Huisman, M. (eds.) TACAS 2018, pp. 41–60 (2018). https://doi.org/10.1007/978-3-319-89960-2_3
29. Le, Q.L., Tatsuta, M., Sun, J., Chin, W.: A decidable fragment in separation logic with inductive predicates and arithmetic. In: Majumdar, R., Kuncak, V. (eds.) CAV 2017, pp. 495–517 (2017). https://doi.org/10.1007/978-3-319-63390-9_26
30. Le, T.C., Zheng, G., Nguyen, T.: SLING: using dynamic analysis to infer program invariants in separation logic. In: McKinley, K.S., Fisher, K. (eds.) PLDI 2019, pp. 788–801 (2019). https://doi.org/10.1145/3314221.3314634

31. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010, pp. 348–370 (2010).https://doi.org/10.1007/978-3-642-17511-4_20
32. Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.S.: Gated graph sequence neural networks. CoRR abs/1511.05493 (2015)
33. Malík, V., Hruska, M., Schrammel, P., Vojnar, T.: Template-based verification of heap-manipulating programs. In: Bjørner, N., Gurfinkel, A. (eds.) FMCAD 2018, pp. 1–9 (2018). https://doi.org/10.23919/FMCAD.2018.8603009
34. Miné, A.: The octagon abstract domain. High. Order. Symbolic Comput. **19**(1), 31–100 (2006). https://doi.org/10.1007/s10990-006-8609-1
35. Mühlberg, J.T., White, D.H., Dodds, M., Lüttgen, G., Piessens, F.: Learning assertions to verify linked-list programs. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015, pp. 37–52 (2015). https://doi.org/10.1007/978-3-319-22969-0_3
36. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001, pp. 1–19 (2001). https://doi.org/10.1007/3-540-44802-0_1
37. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ICSE, vol. 2007, pp. 75–84 (2007). https://doi.org/10.1109/ICSE.2007.37
38. Padhi, S., Sharma, R., Millstein, T.D.: Data-driven precondition inference with learned features. In: Krintz, C., Berger, E. (eds.) PLDI 2016, pp. 42–56 (2016). https://doi.org/10.1145/2908080.2908099
39. Pham, L.H., Le, Q.L., Phan, Q.S., Sun, J.: Concolic testing heap-manipulating programs. In: FM 2019. To appear
40. Pham, L.H., Le, Q.L., Phan, Q.S., Sun, J., Qin, S.: Enhancing symbolic execution of heap-based programs with separation logic for test input generation. In: ATVA 2019. To appear
41. Pham, L.H., Le, Q.L., Phan, Q.S., Sun, J., Qin, S.: Testing heap-based programs with Java StarFinder. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (eds.) ICSE 2018, pp. 268–269. ACM (2018). https://doi.org/10.1145/3183440.3194964
42. Pham, L.H., Thi, L.T., Sun, J.: Assertion generation through active learning. In: Duan, Z., Ong, L. (eds.) ICFEM 2017, pp. 174–191 (2017). https://doi.org/10.1007/978-3-319-68690-5_11
43. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic with trees and data. In: Biere, A., Bloem, R. (eds.) CAV 2014, pp. 711–728 (2014). https://doi.org/10.1007/978-3-319-08867-9_47
44. Piskac, R., Wies, T., Zufferey, D.: GRASShopper - complete heap verification with mixed specifications. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014, pp. 124–139 (2014). https://doi.org/10.1007/978-3-642-54862-8_9
45. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: LICS, vol. 2002, pp. 55–74 (2002). https://doi.org/10.1109/LICS.2002.1029817
46. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Appel, A.W., Aiken, A. (eds.) POPL 1999, pp. 105–118 (1999). https://doi.org/10.1145/292540.292552
47. Zhu, H., Petri, G., Jagannathan, S.: Automatically learning shape specifications. In: Krintz, C., Berger, E. (eds.) PLDI 2016, pp. 491–507 (2016). https://doi.org/10.1145/2908080.2908125