A Shape System and Loop Invariant Inference

Hongjin Liang\textsuperscript{*†} Yu Zhang\textsuperscript{*†}
\textsuperscript{*}School of Computer Science and Technology
University of Science and Technology of China
Hefei, China
\{ljh1018, zpli\}@mail.ustc.edu.cn

Yiyun Chen\textsuperscript{*†} Zhaopeng Li\textsuperscript{*†}
\textsuperscript{†}Software Security Lab
Suzhou Institute of Advanced Study, USTC
Suzhou, China
\{yiyun, yuzhang\}@ustc.edu.cn

Abstract—Pointer programs remain a major challenge for program analysis and verification. Shape analysis can discover the shape invariants of data structures in the heap and detect errors about manipulating pointers in a program. This paper presents a shape analysis for linked list programs based on a new shape graph representation. Our shape graphs could describe unbounded data structures without loss of pointer information. A novel shape system is designed to help the shape analysis. The shape system contains a set of shape inference rules to deduce the shapes of the heap contents at each program point and a set of shape checking rules to find shape errors in pointer programs. In this paper, programmers are expected to declare the shapes of the data structures constructed by recursive data types and to annotate each pointer variable with the shape of the objects which it should point to, so that compilers or other tools can check whether the programs have shape errors and generate loop invariants and even pre/post conditions for program verification.

Keywords—Shape Graph, Loop Invariant Inference, Shape Analysis, Program Analysis, Pointer Logic

I. INTRODUCTION

Software systems have become a vital part of the modern world. It is crucial for safety-critical infrastructures and security-critical applications to ensure correctness, safety and security of software systems. Shape analysis is a static analysis technique that can be used at compile time to find software bugs or to verify high-level correctness properties of programs.

Shape analysis is a form of pointer analysis which goes beyond the tabulation of shallow alias information to deeper properties of the heap such as the shapes of the heap contents at each program point. It can in principle prove that programs do not commit pointer-safety errors (dereferencing a null or dangling pointer, or leaking memory). Moreover, shape analysis attempts to discover invariants that describe the data structures in a program (e.g., [4–9]), which provides considerably useful information for loop invariant inference. In program verification, the highest expectation of shape analysis is to make the user avoid having to write loop invariants or even pre/post specification for procedures; these are inferred during analysis [10]. To do this, the analysis typically has to accurately identify (or distinguish) cyclic and acyclic linked structures, nested lists, and so on. Unfortunately, limited success has only be made on acyclic and cyclic singly-linked lists and binary trees.

Much work on shape analysis has been done by associating each program point with shape graphs. To guarantee that the analysis terminates for programs containing loops, sets of concrete heap cells are summarized by a single node (e.g., [5], [6], [11], [12]) or an edge (e.g., [16], [17]) in the graph. The central challenge of summarization is to retain enough precision so that summarized graphs can be materialized at certain key points of the analysis. This paper presents a new design of shape graphs. We abstract several concrete nodes which are linked as a chain by a compressed node. Our shape graphs do not lose pointer information such as whether a pointer is null, dangling or valid (a valid pointer points to an object in the heap) and the equality between valid pointers. We could precisely judge whether two pointer access paths are aliases by a shape graph at the program point.

Shape graphs can be used to detect pointer-safety errors such as memory leaks, dereferences of dangling pointers; but deeper errors are not easily discovered. In this paper, a novel shape system is designed to find out shape errors, e.g., a pointer which should point to an acyclic list actually points to a cyclic list at some program point. The shape system can be processed along with the shape analysis based on our shape graphs.

In the shape system, programmers are expected to declare the shapes of the data structures constructed by recursive data types and to annotate each pointer variable with the shape of the objects which it should point to, so that compilers or other tools can check whether the programs have shape errors. We take inspiration from the development of programming languages. Assembly languages require programmers to add symbol definitions and directives in order to avoid writing binary code. Typed languages require programmers to declare types of variables and functions in order to make static type checking by compilers. Object-oriented programming languages require programmers to describe inheritance relations between classes in order to guide compliers to reuse the code.

The shape system consists of several shapes (including singly-linked list, circular singly-linked list, doubly-linked list, circular doubly-linked list, and singly-linked list seg-
ment and doubly-linked list segment). We use shape graphs to intuitively illustrate the data structures of these shapes in the heap. The major part of the shape system contains a set of shape inference rules and a set of shape checking rules. By using the shape inference rules, shapes at each program point can be automatically deduced from the shape declarations provided by programmers. The shape declarations can also be viewed as shape constraints imposed on programs, so we can check whether the programs have shape errors according to the shape checking rules.

For pointer programs, the shape system can be used to:

- discover hidden errors of pointer operations, e.g., the actual shape of a data structure constructed in a program is inconsistent with the shape declaration, or a procedure to reverse a linked list introduces cycles into the list.
- carry out program analysis (especially shape analysis), as we could figure out the shape of the heap contents by using the shape inference rules.
- help loop invariant inference and program verification. The shape invariants could be automatically discovered in the programs satisfying shape constraints. As a result, programmers may not need to write loop invariants or even pre/post specifications about pointers.

In this paper, we consider a C-like programming language PointerC [13] in which pointer arithmetic and the address-of operator (&) are forbidden. We make the following contributions:

- A novel shape system is proposed which could solve a variety of problems in pointer programs.
- An intuitive shape graph representation is designed to implement the shape analysis of linked list programs. Since the difficulty of shape analysis lies in loops, we discuss an iterative algorithm of loop invariant inference based on our shape graphs.

The rest of the paper is organized as follows. Section II introduces the shape graphs and illustrates the algorithm of loop invariant inference based on the shape graphs. The shape system is described in section III. Section IV presents the related work and section V concludes our work.

II. SHAPE GRAPHS AND LOOP INARIANT INFERENCE

Shape analysis usually operates on shape graphs. Shape graphs describe active cells in the heap and their connectivity by pointers of the heap cells. In this paper, we design a new shape graph representation which could preserve precise pointer information. As usual, a shape graph could contain nodes for list elements and for predefined heap predicates. The novel aspect of our design is, we summarize several concrete nodes which are linked as a chain without any other pointer pointing to some node thereof by a compressed node. The length of that chain (i.e., the number of the concrete nodes which are compressed) is recorded. Since the recorded length of a compressed node could be infinite, we are able to represent unbounded data structures by our shape graphs. For loop invariant inference, we can construct a numeric lattice on the recorded lengths of compressed nodes, and solve the problem of abstracting heap cells and their points-to relations by abstraction in the numeric lattice.

The design of the compressed nodes is guided by the observation that objects constructed by loops always have repetitive structures. These repetitive structures can be specified by a regular pattern and a number of repetitions. For lists, we can use only one node (i.e., a compressed node) and some pointers (around the compressed node) to express the pattern.

A. Shape Graphs and the Pointer Logic

Data structures should be formally defined. We give their definitions using shape graphs, in which heap cells are represented by graph nodes. There are six kinds of nodes in our shape graphs:

- Normal nodes. We use a normal square node without any marker to represent a node for a list element, i.e., a dynamically allocated cell in the heap.
- Declared nodes. To distinguish the cells in the heap and those in the store, we use a circular node to represent a node for a statically declared variable.
- Predicate nodes. A square node with a $P$ in it is a node for a predefined heap predicate of data structure, and the heap predicate is written below the square node. Sometimes the heap predicate has a length parameter which is written after the heap predicate, and if there is an assertion about the variables in the length expression then it is written after the length expression. For example, in Figure 1 the pointer $s$ points to a data structure satisfying the heap predicate $dlist$ and the length parameter $n$ is associated with an assertion $n > 0$.
- Compressed nodes. We use a gray square node to represent a compressed node. The recorded length of a compressed node is written below the compressed node, and like predicate nodes, the assertion about the variables in the length expression is written after the length.
- $N$ node. All null pointers point to a special square node with a $N$ in it.
- $D$ node. Similarly, it is a square node with a $D$ in it for dangling pointers.

A directed edge between graph nodes denotes a points-to relationship. The graph edge for a pointer type field is

![Figure 1. An example for predicate nodes: $dlist(q, n) \land n > 0$](image)
labeled by the field name, while the graph edge for a pointer type declared variable is labeled by the variable name.

Figure 2 shows the definitions of the heap predicates of data structures discussed in this paper. We only show the definitions without length parameters, in which the type of the list elements is:

\[ \text{struct list} \{\text{struct list }^*\text{next};\} \quad \text{(for list and c_list),} \]

or

\[ \text{struct dist} \{\text{struct dist }^*\text{r, }^*\text{l};\} \quad \text{(for dist and c_dist).} \]

Every list can be empty (i.e., the case that s points to N node), because one can construct a non-empty list from an empty one. Here all the data structures are defined directly by enumerating all the nodes in the lists. For example, the non-empty case of the circular doubly-linked list can be redrawn in Figure 3, since a compressed node is compressed by several concrete nodes which are linked as a chain.

Our shape graphs can be formalized using the pointer logic \[13\]. In the pointer logic, pointers are partitioned into several access path sets. An access path is a name (i.e., a syntactic form) of a pointer, starting with a declared pointer variable (e.g., \( s \rightarrow \text{next} \) is an access path starting with s). Since a pointer could have several names, we call the access paths which represent the same pointer are aliases. The main idea of the pointer logic is that, all the valid pointers which are equal, are contained in the same access path set. In other words, the valid pointers are not equal to each other if they are not in the same access path set. But only concise pointer information is preserved, i.e., access paths in a set are not aliases.

A normal node in a shape graph can be mapped to an access path set in the corresponding pointer logic representation. For example, the shape graph in Figure 3 represents a circular doubly-linked list with \( n \) nodes and \( 2n+1 \) pointers \((n > 0)\). The access paths representing those \( 2n+1 \) pointers could be partitioned into \( n \) valid access path sets, which are just corresponding to the \( n \) nodes:

\[ \{s\rightarrow r, s\rightarrow r\rightarrow r\rightarrow l\} \land \ldots \land \{s(\rightarrow r)^{n-1}, s(\rightarrow r)^n\rightarrow l\} \land \{s, s(\rightarrow r)^n, s\rightarrow r\rightarrow l\}. \]

In the representation, \( s(\rightarrow r)^n \) is an access path with superscript which is shorthand for \( s\rightarrow r\rightarrow r\ldots\rightarrow r \) (in which \( \rightarrow r \) appears \( n \) times). The meaning of \( s(\rightarrow r)^{n-1} \) is similar. The conjunction between two access path sets indicates a disjointness property between the two heap cells pointed to by those pointers, which seems similar to the separation conjunction in separation logic \[2\].

The access path sets above can be rewritten using a universal quantifier bounded pointer set:

\[ \forall k:1..n-1. \{s(\rightarrow r)^k, s(\rightarrow r)^{k+1}\rightarrow l\} \land \{s, s(\rightarrow r)^n, s\rightarrow r\rightarrow l\}. \]

As the \( n-1 \) valid pointer sets are abbreviated by a universal quantifier bounded pointer set in the pointer logic, the corresponding \( n-1 \) normal nodes can be summarized by a compressed node. Thus the shape graph in Figure 3 can be summarized into the non-empty case of the definition of c_dist in Figure 2 as we have mentioned. In other words, a compressed node in a shape graph is mapped to a universal quantifier bounded pointer set in the pointer logic, and the recorded length of the compressed node is expressed by the range of the universal quantifier.

In a shape graph, null pointers and dangling pointers point to \( N \) node and \( D \) node respectively. Correspondingly in the pointer logic, we have two special access path sets with subscripts \( N \) and \( D \) for null pointers and dangling pointers respectively.

Predicate nodes are mapped to the corresponding heap predicates in the pointer logic, and the assertion about the length parameter of the predicate can be in the pointer logic representation as well. For example, the shape graph in Figure 1 can be represented as dist \((q, n) \land n > 0\).

The definitions of lists in Figure 2 can be rewritten using access path sets in the pointer logic. For example, the corresponding pointer logic representation of the definition of list is:

\[ \text{list}(s) \triangleq \exists n : N^+. \{ \{s\} \land \forall i:1..n-1. \{s(\rightarrow \text{next})^i\} \land \{s(\rightarrow \text{next})^n\}, \forall (s)X \} \lor \{s\}X. \]

Access path sets can make up an assertion of disjunctive normal form (DNF), such as the definition of list(s) above. In a DNF assertion, we use a \( \Psi \) to denote the access path sets of a clause. Intuitively, a \( \Psi \) corresponds to a shape graph. So an assertion (in DNF) at a program point corresponds to a disjunction of several shape graphs. Each shape graph might contain several disjoint subgraphs (e.g., two separate singly-linked lists). No matter how complicated a shape graph is, every valid pointer set in a \( \Psi \) can map to a node (which is pointed to by pointers in the set) in the shape graph.
However, a shape graph may correspond to several $\Psi$'s which are equivalent, because any access path in a pointer set can be replaced with its alias.

Shape graphs describe memory states at a program point. To reflect memory updates in the graph, we modify the appropriate edges and nodes. For example, for the pointer assignment $p = q$, we move the pointer $p$ to the node pointed to by $q$ and remain the other parts of the original shape graph unchanged, and then we obtain the result shape graph after the assignment. Since a shape graph can be mapped to a $\Psi$ in the pointer logic, the modifications after a statement can also be performed according to the inference rules of the pointer logic [13]. On the other hand, the pointer logic can be implemented using our shape graphs without complicated alias calculus and alias substitution which are required in the inference rules of the pointer logic.

When an assignment moves a pointer to a compressed node, we need to unfold the compressed node. If the compressed node represents $n$ concrete nodes and $n \geq 0$, we can divide it into zero case and non-zero case. In the non-zero case, the compressed node is split into a normal node and a compressed node. A simple example is shown in Figure 4.

Sometimes we also need to fold several nodes into a compressed node to simplify the shape graph. For example, in Figure 5 we fold the first compressed node and its next node in every case of the left side, and then get the shape graphs of the right side. The recorded lengths of the result compressed nodes are computed.

For the loop while($b$) c,
1) Compute the precondition at the loop entry point $A_0$. Let $i = 0$.
2) Compute $A_{i+1}$ such that $\{A_i \land b\} \Rightarrow \{A_{i+1}\}$.
3) Perform abstraction to compute $A_{i+1}$, such that $A_{i+1} \Rightarrow A_i$.
4) If $A_{i+1} \Rightarrow A_0 \lor \ldots \lor A_i$, then $A_0 \lor \ldots \lor A_i$ is a loop invariant. Otherwise, $i = i + 1$, and return to step 2).

Similarly, we fold and unfold heap predicates (except list segments) by using the definitions shown in Figure 2. For example, we can unfold list(s) as shown in Figure 6 and fold a non-empty doubly-linked list into $\text{dlist}(q, n) \land n > 0$ as shown in Figure 7.

B. An Algorithm for Loop Invariant Inference

We use our shape graphs to implement the shape analysis. Since the difficulty of shape analysis lies in loops, we only discuss loop invariant inference based on our shape graphs.

An algorithm for loop invariant inference under the abstract interpretation framework [1] is presented in Figure 8. It operates on our shape graphs and computes the fixed-point iteratively. In order to ensure termination, abstraction is performed utilizing compressed nodes. We can construct a numeric lattice (e.g., arithmetic intervals [1]) on the recorded lengths of compressed nodes, and convert the problem of abstracting heap cells and their points-to relations to the abstraction in the numeric lattice.

The design is guided by capturing the characteristics of loops manipulating lists. In general, there are two kinds of loops:

1) The loop constantly inserts a node into (or deletes a node from) a list at a certain position, and may manipulate several lists at the same time.
2) The loop attempts to find a proper position in a list, in order to perform other operations such as insertions and deletions. The loop control condition is mainly:
   a) judging whether a certain pointer has reached the tail of the list, which is usually irrelevant to integer variables;
   b) or concerning with some integer variable which is recursively assigned in the loop.

We observe that some part of the list will become longer (or shorter) constantly during the iterations of a loop. For example, if the loop body is to insert a node into the head of a list, then the length of the list always increases one after a iteration. As a result, we can utilize the iteration history to
void list_traversal(struct list *s) {
    struct list *p; p = s;

    while (p != NULL) {
        p = p->next;
    }
}

Figure 9. List traversal

guide the abstraction process. Specifically, we compare the result shape graphs of the current iteration with those of the previous iteration, and then try to perform abstraction on the differences of the two results. Abstraction is done by simply transforming some normal nodes into a compressed node. The amount of these normal nodes determines the recorded length of the compressed node, which is equipped with an abstract operator of the numeric lattice. Thus we apply the numeric widening operator on the recorded lengths of the compressed nodes in the end.

Most operations are simplified on shape graphs rather than directly on $\Psi_s$, because one has to consider aliases of access paths for the latter. On shape graphs, to test whether $\Psi_1 \Rightarrow \Psi_2$ is to check whether the two shape graphs are the same. And to test whether $\Psi_1 \equiv \Psi_2$ is to check whether the shape graph of $\Psi_2$ is an instance of that of $\Psi_1$.

C. Case Study

We have implemented the loop invariant inference algorithm in a demo tool [15] and inferred a variety of loop invariants of linked list programs, including simple operations such as list reversal, and more complicated procedures such as list partition (discussed in [3]). In this subsection, we illustrate how to use the algorithm of loop invariant inference by two simple examples in Figure 9 and Figure 10. In these examples, we show the abstract memory states at three main program points using shape graphs. There are two memory states shown at each program point: one for the first iteration (left) and the other for the fixed point (right).

The first example is list traversal (in Figure 9), in which a pointer traverses from the head to the tail of a list. Although it is simple, it reveals a conventional pattern of loops dealing with linked lists. The precondition of the function is $\{\text{list(s)}\}$, which indicates that the pointer $s$ points to a singly-linked list before the program. It can be obtained from the user-supplied shape declarations.

1) Compute the precondition $A_0$ at the loop entry point (program point 1), in which $\text{list(s)}$ has been unfolded.

2) The first iteration:
   - The left shape graph at program point 2 shows the memory state before executing the loop body in the first iteration. Since $p != \text{NULL}$ contradicts the second case of $A_0$, the shape graph at program point 2 only keeps the first case.
   - The left shape graph at program point 3 shows the memory state after executing the loop body of the first iteration, which represent $A_1'$. The assignment $p = p->next$ moves the pointer $p$ to a compressed node. So we unfold the compressed node.

Then we perform abstraction. Comparing the shape graphs at program point 1 and 3 (i.e., $A_0$ and $A_1'$), we observe that the segment from the node pointed to by $s$ to the node pointed to by $p$ has become longer. Thus we transform the node pointed to by $s$ in $A_1'$ to a compressed node. The recorded length of the compressed node is 0 (in $A_0$) or 1 (in $A_1'$). After widening, the recorded length is $[0..n]$ ($n$ denotes the length of the list). After all, we obtain $A_1$ which is shown at the right side of program point 1. Actually, it is the loop invariant. The corresponding pointer logic representation is:

$$\{\exists n: n \in \mathbb{N}. \forall i: 0 \leq n. \{s(i->next) \wedge (s(i->next)' \wedge p) \wedge (\forall i: 0 \leq n. \{s(i->next) \wedge (s(i->next)' \wedge p) \} \wedge (\forall i: j \leq n. \{s(i->next) \wedge (s(i->next)' \wedge p) \}) \wedge (\exists n: n \in \mathbb{N}. \forall i: 0 \leq n. \{s(i->next) \wedge (s(i->next)' \wedge p) \})$$

3) We perform the second iteration, and obtain $A_2$ which is the same with $A_1$. We can find $A_2 \Rightarrow A_0 \lor A_1$, so the algorithm terminates and the loop invariant is $A_0 \lor A_1$, which can be reduced to $A_1$.

The second example is the function of creating a doubly-linked list (in Figure 10). In these examples, we will not discuss the details.

The result loop invariant is:

$$\{\forall i: 0 \leq j \leq n. \{q(i->r) \wedge (q(i->r)' \wedge l) \wedge (q(i->r)' \wedge l) \wedge 1 \leq j \wedge j \leq n \wedge n > 0 \}$$

After the return statement, the declared local pointer variable $p$ should be removed since it can not be accessed. Then we can fold the non-empty doubly-linked list and get $\{\text{dist}(q, n) \wedge n > 0\}$ which is the postcondition of the function.

III. A SHAPE SYSTEM

In order to detect hidden errors in pointer programs, we design a shape system. Programmers are expected to annotate each pointer variable with the shape of the objects which it should point to. The shape system finds out the actual shape of the object which a pointer points to at a program point. By comparing the declared shape and the
actual shape, the shape system checks whether a program has shape errors at each program point. For example, if a procedure to reverse a linked list introduces cycles into the list, the shape system can precisely locate the statement in the procedure which causes shape errors.

To design the shape system, we should first consider what shapes can be formed during the construction, destruction, insertion or deletion of each data structure. And then we should determine the specific shapes allowed at each program point. The key technical problem is to effectively identify the shapes of the heap contents at a program point.

The shape system focuses on the shapes. It detects shape errors, and identifies the new shapes when the shape graphs are changed by a statement. We can attach the shape system to an existing program analysis or verification system. In this paper, we process the shape system along with the shape analysis based on our shape graphs.

In this section, we introduce the basic concepts of the shape system, and discuss the shape inference rules which are used to deduce the shapes at each program point, and the shape checking rules which are used to find out shape errors. We will not discuss the shape declarations since the way to declare them is not important.

A. Shapes and Objects

In this paper, we only discuss the shapes of linked lists: list, c_list, dlist and c_dlist. We have formally defined the data structures of these shapes in Figure 2. When manipulating these data structures, such as inserting and deleting nodes, their shapes will be temporarily changed. Two kinds of list segments are used to describe the shape characteristics when the data structures are modified. The shape lsseg (singly-linked list segment) is used for singly-linked lists and circular singly-linked lists when we do not care about the last pointer of such a list, i.e., the last pointer of a singly-linked list segment might be null or dangling or valid. Similarly, the shape dlseg (doubly-linked list segment) is used during constructing or modifying doubly-linked lists and circular doubly-linked lists. Besides, we introduce two special shapes N and D to deal with null pointers and dangling pointers respectively, and the shape of an empty list is N.

An object is an instance of a certain shape. We use shape graphs to intuitively illustrate the shapes of the objects in the heap. As we have mentioned, a shape graph may contain several subgraphs. Each subgraph may contain several objects which are connected together (e.g., the shape graph in Figure 11(a) is constructed by a circular doubly-linked list and a doubly-linked list segment), so two nodes in a shape graph might belong to different objects. Suppose in a shape graph, a node \( n_1 \) belongs to an object and a field of \( n_1 \) points to a node \( n_2 \), then the possible belong-to relation between \( n_2 \) and \( n_1 \) could be:

- \( n_2 \) and \( n_1 \) belong to different objects. For example, in Figure 11(a), the node \( n_2 \) belongs to a circular doubly-linked list, while the node \( n_1 \) belongs to a doubly-linked list segment.
- \( n_2 \) and \( n_1 \) belong to the same object via the field. For example, in Figure 11(b), the nodes \( n_2 \) and \( n_1 \) belong to the same doubly-linked list segment and \( q \rightarrow l \) is a pointer linking the two nodes and deciding the belong-to relation (i.e., if \( q \rightarrow l \) does not points to \( n_2 \), then \( n_2 \) and \( n_1 \) would belong to two different doubly-linked list segment respectively).
- \( n_2 \) and \( n_1 \) belong to the same object but not via the field. For example, in Figure 11(c), the nodes \( n_2 \) and \( n_1 \) belong to the same doubly-linked list segment. The pointer \( p \rightarrow l \) links the two nodes but does not decide the belong-to relation, because even if \( p \rightarrow l \) does not point to \( n_2 \) (e.g., it is a dangling pointer), \( n_1 \) and \( n_2 \) would still belong to the same doubly-linked list segment.

There are parent-child relationships between shapes, as...
shown in Figure 12, in which the starting point of an arrow is a parent shape and the ending point is a child shape. Parent-child relationships are mainly used in the shape checking rules (discussed in subsection III-C). Pointers pointing to an object of a child shape are allowed to point to an object of its parent shape. For example, a pointer, which is declared to point to a doubly-linked list, can point to a doubly-linked list segment during the execution of a program. Moreover, an object of a parent shape is of its child shape if the pointers in the object satisfy some conditions to be the child shape. For example, if a pointer \( p \) points to a singly-linked list segment which satisfies the condition \( \exists p : (p \rightarrow next)^n = p \), then in fact \( p \) points to a circular singly-linked list. Those additional conditions are widely used in the shape inference rules.

B. Shape Inference Rules

Shape graphs are modified by pointer assignments, malloc statements, etc. In a shape graph, every node (except declared nodes) is labeled by the shape of the object the node belongs to. Labels on \( N \) node and \( D \) node are the special shapes \( N' \) and \( D' \) respectively. We design a set of shape inference rules to deduce the new shape label on every node in the modified shape graphs at a program point, from the original shape graphs with shape labels and the pointer assignment (or other statements) which causes the modifications.

A pointer assignment involves three nodes at most. Take the assignment \( p \rightarrow l = q \) which changes Figure 11(c) to Figure 11(d) for example. In Figure 11(c) before the assignment, the three nodes are, the node where \( p \rightarrow l \) is (i.e., the node pointed to by \( p \)), which is called ArrowFrom, the node pointed to by \( p \rightarrow l \) (called Old) and the node pointed to by \( q \) (called New). In Figure 11(d), the three nodes are marked with \( A \), \( O \) and \( N \) respectively. In general, the shapes of the three objects, which the nodes \( A \), \( O \) and \( N \) belong to respectively, are probably changed by the assignment (in particular, they may belong to the same object, e.g., \( A \), \( O \) and \( N \) are in the same doubly-linked list segment in Figure 11(c)).

Consider the case when \( A \), \( O \) and \( N \) all exist and belong to some objects, we have some conclusions about how the shapes are changed:

- If the pointer, which is broken up by the assignment, is in the object containing \( A \) (i.e., the pointer decides the shape), then the shape might be altered due to the pointer breaking. Otherwise, it will not be altered. For example, the pointer from \( n_2 \) to the node pointed to by \( p \) in Figure 11(a) is in the circular doubly-linked list (i.e., it decides the shape c_dist). Thus when it is broken up by the assignment \( p \rightarrow l \rightarrow r = q \), the shape of \( n_2 \) is changed to dseg in Figure 11(b).
  - The shape of the object containing \( N \) will not be altered due to the pointer breaking. But \( A \) and \( N \) might be in the same object after the assignment (since the assignment makes a pointer from \( A \) point to \( N \)), so that their shapes might be changed.
  - If the broken pointer decides the relation that \( A \) and \( O \) belong to the same object, then the shape of the object containing \( O \) might be altered due to the pointer breaking. Otherwise, it will not be altered, except when \( O \) and \( A \) are in the same object after the assignment, \( O \) changes its shape along with \( A \) (e.g., the assignment which changes Figure 11(c) to Figure 11(d)).

If the left side of a pointer assignment is a declared variable, then \( A \) is a declared node which does not belong to any object nor have a shape, thus the case is simpler. It is similar to malloc statement and other statements, so we will not discuss them in this paper.

The shape inference rules can be divided into two parts: the shape diagnosis rules to discover the new shapes of \( A \), \( O \) and \( N \), and the shape propagation rules to spread the new shapes to their surrounding nodes. According to the analysis above, we can diagnose the new shape of \( A \) based on the original shapes of \( N \) and \( A \) and some other conditions. And if the shape of \( N \) should be altered due to the assignment, it will be changed during the propagation of the new shape of \( A \). Finally, if the node \( O \) is not involved during the propagation of the new shape of \( A \), the new shape of \( O \) can be diagnosed based on its original shape and some other conditions.

The shape diagnosis rules for \( A \) are shown in Figure 14 by a table to save space. If the new shape of \( A \) can not be diagnosed by those rules, then it is lseg or dseg. The additional conditions in those rules illustrate how a list segment (i.e., its shape is lseg or dseg) can become an object of its child shape. For example, if a doubly-linked list segment satisfies the condition:

starting from some node pointed to by \( p \) in the list and traversing both forwards and backwards, the traversal process will not be suspended (i.e., for each node pointed to by some pointer \( s \), \( s = s \rightarrow r \rightarrow l \)) until reaching the end (i.e., its next node is \( N' \)),

then the doubly-linked list segment is actually a doubly-linked list. The additional conditions to be other child-shapes can be figured out as well. So in order to design the shape diagnosis rules for \( A \), at first we regard the result shape of \( A \) after the assignment as lseg or dseg, and then we check whether the result object containing \( A \) satisfies the additional
conditions to be an object of its child shape. But in fact, sometimes it is absolutely impossible for the result object to satisfy those conditions. For example, when \( p \) points to a doubly-linked list segment and \( q \) points to a doubly-linked list, after the assignment \( p \rightarrow r = q \), it is certain that \( p \) still points to a doubly-linked list segment since \( q \rightarrow l \) is not equal to \( p \). As a result, we design the rules in Figure 13 by highlighting the cases when the result object might satisfy the additional condition. Sometimes in practice, it is not necessary to check the additional condition because it is trivially satisfied. For example, if \( p \) points to a doubly-linked list originally, then after the assignment \( p \rightarrow r = NULL \), it is certain that the new shape of \( A \) is \( dlseg \).

Figure 14 illustrates the shape diagnosis rules for \( O \) by a table as well (used only when the node \( N \) is not involved during the propagation of the new shape of \( A \)). If the shape of \( O \) does not change with that of \( A \) (i.e., \( O \) and \( A \) belong to the same object after the assignment), then as we have analyzed, the shape of \( O \) is changed only when the broken pointer \( p \rightarrow next \) or \( p \rightarrow r \) decides the relation that \( A \) and \( O \) belong to the same object. Thus the shape diagnosis rules for \( O \) are designed by checking whether \( A \) and \( O \) belong to the same object before the assignment. For example, if \( p \rightarrow r \) points to a doubly-linked list and satisfies \( p \rightarrow r \rightarrow l = \) \( p \) which indicates that \( p \rightarrow r \) is a pointer in the list (i.e., the node \( A \) pointed to by \( p \) in is the same object as the node \( O \) pointed to by \( p \rightarrow r \) originally), then after making \( p \rightarrow r \) point to another node, the shape of \( O \) is changed to \( dlseg \).

The shape propagation rules investigate every surrounding node of \( A \) (or \( O \)), and judge whether it should be changed to the same new shape of \( A \) (or \( O \)). The rules could be easily obtained based on the definitions of data structures. For example, to propagate the new shape \( c dlseg \) of \( A \), we need to change the shapes of the nodes which are in the circular doubly-linked list. So we go along the \( r \)-chain from \( A \) and change the shape of every node encountered which is not \( c dlseg \) until we go back to \( A \).

As an example, we explain how to deduce the new shapes in Figure 11(d) by using the shape inference rules. The shape graph before the assignment \( p \rightarrow l = q \) is shown in Figure 11(c), where all the shapes of the nodes are \( dlseg \). Then the assignment breaks the pointer \( p \rightarrow l \) and makes it point to the node \( N \). We can obtain the new shape of \( A \) is \( c dlseg \), because the original shapes of \( A \) and \( N \) are \( dlseg \), and the whole shape graph satisfies the additional condition in the corresponding shape diagnosis rule for \( A \). Afterwards, we propagate the new shape \( c dlseg \) to the surrounding nodes. Because all the nodes in the shape graph are in the list, their shapes are all updated to \( c dlseg \). And since the shape of \( O \) has been changed in the propagation of \( c dlseg \), we do not need to apply the shape diagnosis rule for \( O \). After all, we get the final result of the new shapes.

We have implemented the shape inference rules in our demo tool for loop invariant inference [15].

C. Shape Checking Rules

Program declarations restrict the shapes of the data structures constructed by recursive data types and the shapes which a pointer variable could point to. As a result, we can perform shape checking upon the basic type checking to discover hidden errors in pointer programs. We only enumerate part of the shape checking rules here:

- At the program point of a function call statement, the shape of the object pointed to by the pointer argument should be the same as the declared shape of the corresponding parameter of the callee.
- At the program point of a return statement, the shape of the object pointed to by the return variable should be the same as its declared shape.
- At any other program point, the shape of the object pointed to by a pointer variable should be either the same as or the parent of its declared shape.
- If the condition of a loop is \( p = = \) NULL or \( p != \) NULL, then \( p \) should point to a singly-linked list or a doubly-linked list or \( N \) every time at the loop entry point.

Note that for different applications, one might design different shape checking rules, since the restrictions imposed on
programs might be different.

With those shape checking rules above, we can turn back to simplify the shape diagnosis rules for $A$ (in Figure 13), because some cases are prohibited by the shape checking. For example, when the original shapes of $A$ and $N$ are $\text{c_list}$ and $\text{list}$ respectively, the case is illegal because $\text{list}$ is not a parent of $\text{c_list}$.

**D. The Shape System and Loop Invariant Inference**

The shape system helps a lot to implement the loop invariant inference algorithm (in subsection II-B).

Firstly, operations on data structures are disciplined by the shape constraints, such as cyclic lists cannot occur where an acyclic list is acquired. Sometimes it is difficult to compute or even express the loop invariant if the program violates the shape constraints. For example, consider a loop which attempts to dynamically find a position in a doubly-linked list according to a sequence of commands. Every command describes whether to go left or right or stop in the list. The loop should operate on a doubly-linked list, but unfortunately it operates on a binary tree in a program. It seems difficult to specify the actual loop invariant since every path in a binary tree leads to a different node. In addition, when a loop itself contains shape errors, the shape system can also find out the errors in time. As an example, still consider the loop which attempts to find a position in a doubly-linked list. The loop should remain the list itself unchanged, but actually pointers in the list are modified in the loop. The shape system could easily discover the mistakes in the loop, and then we can stop inferring the loop invariant. In a word, the shape system can rule out the programs which have shape errors, so that we do not need to perform further analysis or verification on those programs.

Secondly, the shape information gathered by the shape system is helpful to select an appropriate folding rule. For example, if a pointer $p$ points to an object of the shape $\text{dlist}$ at a program point, then we can fold the object into the heap predicate $\text{dlist}(p)$.

It seems that our shape system is similar to the type system. But actually, there are some notable differences between them. The type system restricts static program texts with context-sensitive constraints, while our shape system uses shape constraints for dynamic constructed data structures. In the type system, the typing rules are usually structural: the type of a language construct is determined by the types of its sub-constructs. On the contrary, the shape inference rules are formulated with respect to the shapes of the objects at a program point, and the shape of an object is determined by how the nodes in the object are linking together. Note that we utilize the shapes of the objects at the previous program point in order to reduce the complexity in identifying the new shapes. It is feasible because of the locality properties of the statements, e.g., an assignment involves three nodes at most. Thus their design and implementation are quite different.

**IV. RELATED WORK**

Shape analysis attempts to discover the shapes of data structures which can be regarded as invariants in pointer programs. In our shape system, programmers are expected to declare shapes of data structures, i.e., to specify the invariants of programs so that the shapes of data structures could only change in the prescribed domain. As a result, behaviors of programs are restricted. Then it is easy to discover more properties in programs satisfying those restrictions.

There has been a lot of work that builds shape analysis based on shape graphs, in which heap cells are represented by graph nodes. In particular, the elements of potentially unbounded data structures are grouped into a finite number of graph nodes by using summary nodes [5], [6], [11], [12]. The approximation of memory states leads to loss of information about the shapes of recursive data structures. In our shape graph, we introduce compressed nodes which do not lose information. To approximate unbounded structures, we perform abstraction on the recorded lengths of the compressed nodes. In other words, we convert the problem of approximating heap cells and their points-to relations to the abstraction of numerical relations. This method is similar to the widening of alias relation in Deutsch’s work [13]. In Deutsch’s algorithm, the widening of alias relation is accomplished by applying the numeric widening operator on the superscripts of access paths. Deutsch’s algorithm yields alias information but does not gather shape information, and the algorithm is not based on shape graphs.

Chang et al. propose a shape analysis [16], [17] based on a shape graph representation which abstracts memory cells by edges. They describe memory states in a manner based largely on separation logic [2]. For comparison, our shape graphs can be formalized using the pointer logic. In addition, the shape analysis in [16], [17] is guided by invariant checking code supplied by the user (i.e., the definitions of data structures can come from checking code), which seems more flexible than ours. The checking code is translated to inductively-defined predicates. But when using inductively-defined predicates, it is difficult to explicitly specify the relation between several pointer type fields in a data structure such as a doubly-linked list [17]. In our work, the predicates of data structures are defined by directly enumerating all the nodes in the lists, thus we can avoid some challenges about using inductively-defined predicates.

Magill et al. have developed a method [3] of inferring loop invariants for linked list programs in separation logic. They utilize the symbolic execution mechanism, and give a set of rewrite rules to iteratively compute fixed points. A similar method of shape analysis based on separation logic is proposed in [4], which could as well calculate loop invariants of acyclic and cyclic linked list programs.
For comparison, in this paper, the loop invariant inference algorithm operates on our shape graphs without loss of precision and could compute the lengths of lists at the same time. And since programmers provide more information such as shape declarations, loop invariant inference seems easier in our approach.

An interprocedural shape analysis based on separation logic is proposed in [9]. It performs inductive recursion synthesis to automatically infer any recursive shape invariants with a tree-like backbone, while our approach is proposed based on linked list programs. But it is possible to extend our shape graphs and shape system to other data structures.

V. CONCLUSION

This paper discusses several aspects of shape analysis, including the design and use of shape graphs, the identification of the shapes of heap contents and the discovery of hidden errors in pointer programs. We focus on an idea of adding shape declarations in programs to support static checking, program analysis and program verification.

For the future, we will extend our shape system to data structures with multiple possible traversals. A typical data structure with multiple possible traversals is, a doubly-linked list in which each node itself points to a singly-linked list and some nodes in the doubly-linked list links together to form another singly-linked list. It seems not difficult to extend the shape system to such complicated yet well-regulated data structures.

Besides, we are planning to extend Hoare logic based on our shape graphs. A shape graph can be viewed as an assertion describing the heap. It could express both the separation of nodes as in separation logic and the equality of pointers as in the pointer logic. Inference rules can be designed directly based on our shape graphs. Thus we can verify pointer programs in the shape graph logic and use program analysis techniques to help program verification.

REFERENCES


