Li ZP, Zhang Y, Chen YY. A shape graph logic and a shape system. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 28(6): 1??–??? Nov. 2013. DOI 10.1007/s11390-013-1398-1

A Shape Graph Logic and A Shape System

Zhao-Peng Li (李兆鹏), *Member*, *CCF*, Yu Zhang* (张　昱), and Yi-Yun Chen (陈意云), *Member*, *CCF*

School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China

Software Security Laboratory, Suzhou Institute for Advanced Study, University of Science and Technology of China Suzhou 215123, China

E-mail: {zpli, yzhang, yiyun}@ustc.edu.cn

Received November 21, 2012; revised September 20, 2013.

Abstract Analysis and verification of pointer programs are still difficult problems so far. This paper uses a shape graph logic and a shape system to solve these problems in two stages. First, shape graphs at every program point are constructed using an analysis tool. Then, they are used to support the verification of other properties (e.g., orderedness). Our prototype supports automatic verification of programs manipulating complex data structures such as splay trees, treaps, AVL trees and AA trees, etc. The proposed shape graph logic, as an extension to Hoare logic, uses shape graphs directly as assertions. It can be used in the analysis and verification of programs manipulating mutable data structures. The benefit using shape graphs as assertions is that it is convenient for acquiring the relations between pointers in the verification stage. The proposed shape system requires programmers to provide lightweight shape declarations in recursive structure type declarations. It can help rule out programs that construct shapes deviating from what programmers expect (reflected in shape declarations) in the analysis stage. As a benefit, programmers need not provide specifications (e.g., pre-/post-conditions, loop invariants) about pointers. Moreover, we present a method doing verification in the second stage using traditional Hoare logic rules directly by eliminating aliasing with the aid of shape graphs. Thus, verification conditions could be discharged by general theorem provers.

Keywords shape graph logic, program verification, shape analysis, automated theorem proving, loop invariant inference

1 Introduction

Formal verification is a major method to improve the dependability of computer software. There are two main approaches to doing formal verification on software. The first one is model checking, which consists of a systematically exhaustive exploration of the mathematical model. This approach has been progressively used in the industry. The second one is logical inference. It consists of using a formal version of mathematical reasoning about software systems, usually using theorem proving software such as Isabelle/HOL[1] or Coq①. Most research (such as Ynot[2], Spec#[3] and ESC/Java[4]) using the second approach revolve around the design of logic systems to reason about programs and generate verification conditions (VC for short), and then use certain theorem prover to prove them. Researches such as Smallfoot[5] and jStar[6] use symbolic execution and entailment proof to avoid generating large VCs. Although these tools have been developed in labs, there is no product that can be applied to real-world software yet. The root of this problem lies in the difficulty in automated theorem proving, since many problems are influenced by the power of underlying automated theorem proving techniques. These problems include aliasing analysis, loop invariant inference, the expressivity of assertion languages, the design of domain-specific logics, etc.

While trying to make breakthroughs in automated theorem proving techniques, we should consider lowering requirements on the capability of theorem provers. For instance, we can design new mechanisms for programming languages to raise the threshold of being a legal program and reject certain programs in which there exist errors logically. Moreover, program analysis can be used in collecting program information to support program verification. All these approaches could somewhat alleviate the burdens of automated theorem

Regular Paper

This research was supported by the National Natural Science Foundation of China under Grant Nos. 61003043, 61170018, the National High Technology Research and Development 863 Program of China under Grant No. 2012AA010901-2, and the Postdoctoral Science Foundation of China under Grant No. 2012M521250.

*Corresponding Author

①The Coq development team. The Coq proof assistant reference manual (Version 8.2), http://coq.inria.fr, Sept. 2013.

©2013 Springer Science + Business Media, LLC & Science Press, China

proving. This paper introduces our research result on the automatic verification of pointer programs in two stages: shape analysis and program verification.

First, we propose a method using shape graphs as assertions of pointer equality and validity. Based on this approach, we design a shape graph logic (SGL for short) as an extension of Hoare logic. We mainly add some inference rules for statements dealing with pointers (pointer statements for short). The resulting logic can be used in the analysis and verification of pointer programs manipulating mutable data structures.

Under restrictions on pointer arithmetic and uses of the address-of operator (&), our shape graphs can precisely describe the equalities between pointers (including statically declared pointer variables and pointer-typed fields of dynamically allocated structure variables) and validity of pointers (valid pointers point to program objects, i.e., structure variable allocated on the heap) at program points. Given shape graphs, we can acquire whether two access paths are aliases or not at corresponding program point. Shape graphs are bridges between shape analysis and program verification. In the shape analysis stage, we use inference rules to construct shape graphs at each program point. In the following stage, these shape graphs provide information of pointers and support the verification of other properties (e.g., orderedness of binary trees). As it were, certification about pointer-related properties is peeled off from program verification and done in the shape analysis stage. We use SGL in both stages in favor of soundness proof of the logic.

Using shape graphs as assertions makes work in both stages easy. In the analysis stage, it is simple to calculate shape graphs of the post-condition for statement-based on shape graphs of the pre-condition. In the verification stage, shape graphs provide needed information about pointers. For example, we can easily know whether more than one pointers point to a particular node in the graph (to avoid memory leaks). In addition, we can judge whether two access paths are aliases or not using shape graphs (when eliminating aliasing of access paths).

Second, we propose a shape system. Roughly speaking, it is similar to type systems. We design and implement a shape system for the PointerC programming language[7]. It contains shapes and their definitions, rules for shape inference and shape checking. The shape system requires programmers to provide shape declarations in recursive structure type declarations. Compilers or other tools can use shape inference rules to infer shapes constructed by programs via dynamically allocated structures, and then do shape checking to judge whether shapes confirm to what the programmer has declared, according to the rules of shape checking. In addition, the shape system can help rule out programs that construct (or operate on) shapes deviating from what programmers expect in shape declarations in the analysis phase. In this way, we report errors earlier and decrease the cases needed to be considered in both analysis and verification.

With a shape system, programmers need not to provide shape graphs in the pre-/post-conditions of functions and loop invariants for program verification. Hence, it brings no trouble to programmers using shape graphs as the graphical expression of assertions.

Third, we propose a method doing verification in the second stage using traditional Hoare logic rules directly by eliminating aliasing with the aid of shape graphs. One important limitation using Hoare logic is that different names (including access paths) must represent different program objects (i.e., no aliasing is allowed). Hoare logic is sound when we apply rules to a program where there is no aliasing in both statements and its specification (assertions). Shape graphs provide simple ways of judging and eliminating aliasing. We eliminate aliasing problems in access paths via alias substitution before applying corresponding Hoare logic rules.

Using this method, VCs could be discharged by general theorem provers without the need of designing specialized provers (e.g., separation logic's special prover Smallfoot[5]).

Finally, we implement a prototype for automatic verification of pointer programs. Our prototype supports programs manipulating complex data structures such as sorted circular doubly-linked lists, binary search trees, splay trees, treaps, AVL trees and AA trees.

The rest of the paper is organized as follows. Section 2 introduces shape graphs as assertions. Section 3 presents our SGL. Section 4 presents the shape system. Section 5 introduces our program verification prototype for PointerC. Section 6 compares our work with related work, and Section 7 concludes the paper.

2 Shape Graph as Assertion

Shape graphs are directed graphs. They can describe point-to relations of statically declared pointer variables (pointer for short) and pointer fields of dynamically allocated structure variables (field pointer for short). Shape graphs precisely express the equalities between pointers. They can be used to judge whether two access paths are aliases or not.

In this section, shape graphs are defined and the semantic is given. We present our approach using shape graphs as assertions on pointer relations. In Fig.1, some quick examples of shape graph assertions and their co-

unterparts in the separation logic are shown. Assertions on the first line are describing a singly-linked list of length n. Assertions on the second line are used to describes doubly-linked lists. Assertion Fig.1(e) is semantically the same as the assertion Fig.1(f). The difference is that assertion Fig.1(f) uses quantifiers to describe that the fields l and r are dangling pointers.

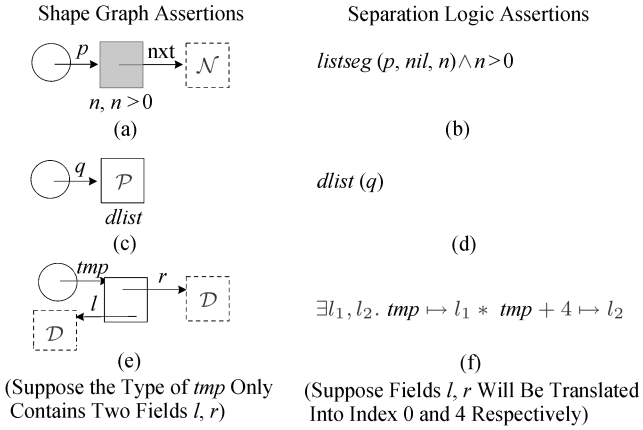

Fig.1. Quick example: shape graph and separation logic assertions.

2.1 Syntax of Shape Graph

Vertices. There are six kinds of vertex (see Fig.2). They are used to denote stack slots or heap blocks manipulated by programs. A vertex is also called a node. The following are their names and syntax.

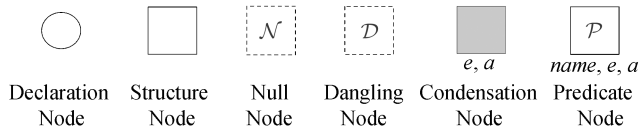

Fig.2. Six kinds of nodes in shape graphs.

1) Declaration node: it is a circle and represents a stack slot (a declared pointer variable).

2) Structure node: it is a rectangle and represents a heap block (a structure variable).

3) Null node: it is a dashed rectangle with a symbol $\mathcal{N}$ inside. Pointers pointing to null nodes are null pointers.

4) Dangling node: it is a dashed rectangle with a symbol $\mathcal{D}$ inside. Pointers pointing to dangling nodes are dangling pointers.

5) Condensation node: it is a gray rectangle. There is information below the node, including an expression e and an assertion a constraining e. Expression e is a linear integer expression using integer constants and declared variables. Assertion a is logical conjunctions of relations between such expressions. The gray rectangle without expression e and assertion a is called an unconstrained condensation node. Such node represents arbitrarily positive number (including zero) of structure nodes.

6) Predicate node: it is a rectangle with a symbol $\mathcal{P}$ inside. There is also information below the node, including the name of the predicate, an expression e, and an assertion a. e and a have exactly the same constraints as these of a condensation node.

Directed Edges and Their Labels. Each directed edge has an identifier as its label (labels are above the edges in our figures). Directed edges and related nodes satisfy the following syntactic constraints.

1) Declaration node: it has only one out-edge and no in-edge.

2) Structure and condensation node: they have in-/out-edges and the labels of their outgoing edges are different from each other.

3) Null node, dangling node and predicate node: they have no outgoing edges and there is no restriction on their incoming edges.

Definition 1 (Shape Graph).

1) *Shape graphs, whose nodes and edges satisfy aforementioned syntactic constraints, are connected graphs if directions of edges are ignored. The labels of edges from declaration nodes in a shape graph should be different from each other.*

2) *Suppose G_1, G_2 are shape graphs. If the label sets of edges from declaration nodes in G_1, G_2 are disjoint, $G_1 \wedge G_2$ is also a shape graph ($\wedge$ is logical conjunction).*

3) *Suppose G_1, G_2 are shape graphs. If the label sets of edges from declaration nodes in G_1, G_2 are equal to each other, $G_1 \vee G_2$ is also a shape graph ($\vee$ is logical disjunction).*

In this paper, we use symbol G to denote shape graphs without connective $\vee$. Shape graph G without logical conjunction connectives is called shape sub-graph. Obviously, shape sub-graphs in one shape graph are not connected to each other. If $G_1 \wedge G_2$ is a shape graph, G_1 and G_2 assert on two different (separated) parts of one program state; while G_1, G_2 in $G_1 \vee G_2$ denote two possible program states at one program point.

Shape graphs in this paper are subsets of shape graphs defined in Definition 1 due to the constraints of the type system. For instance, the type system will guarantee that two structure nodes of different types will not be adjacent to each other in shape graphs.

Fig.3 presents two shape graphs for singly-linked lists. The loop invariant shape graph of the following program fragment is shown as Fig.3(a) (suppose the singly-linked list pointed by the variable *hd* has at least one node). The condensation nodes pointed by *hd* and *ptr* denote two list segments whose lengths are m and

n respectively (m, $n \geqslant 0$). It is easier to understand after we introduce the semantics of shape graphs.

```
ptr1=hd; ptr=hd->nxt;
while(ptr!=NULL){
   ptr1=ptr; ptr=ptr->nxt;}
```

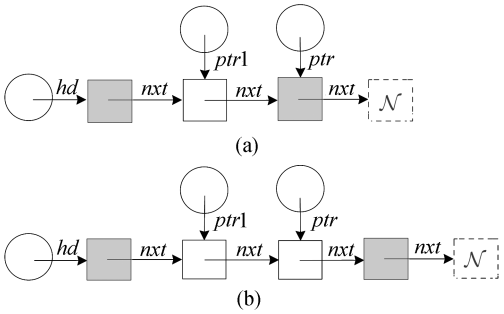

Fig.3. Two instances of shape graphs.

A condensation node represents several adjacent nodes and the expression e below it represents the number of these adjacent nodes. Pointers pointing to null nodes are null pointers. Fig.3(b) is the shape graph at the program point before the first statement of the loop body.

2.2 Shape Graph for Data Structure

Fig.4 shows shape graphs for (circular) singly-linked lists, (circular) doubly-linked lists and binary trees. The symbols, $dlist(s, e, a)$, etc., on the left-hand side of the definitions are shorthand of shape graphs for easy citing below in this paper. Labels of edges in given shape graphs are only placeholders. Proper names should be used to instantiate them with respect to corresponding programs.

As can be seen, if we remove e and a in the right-hand side shape graphs of the two definitions of $dlist(s, e, a)$ and connect them with $\vee$, we can obtain the definition of $dlist(s)$. Due to limited space, we do not include the definitions of shape graphs for $list(s)$, $c\_list(s)$, and $c\_dlist(s)$ in Fig.4.

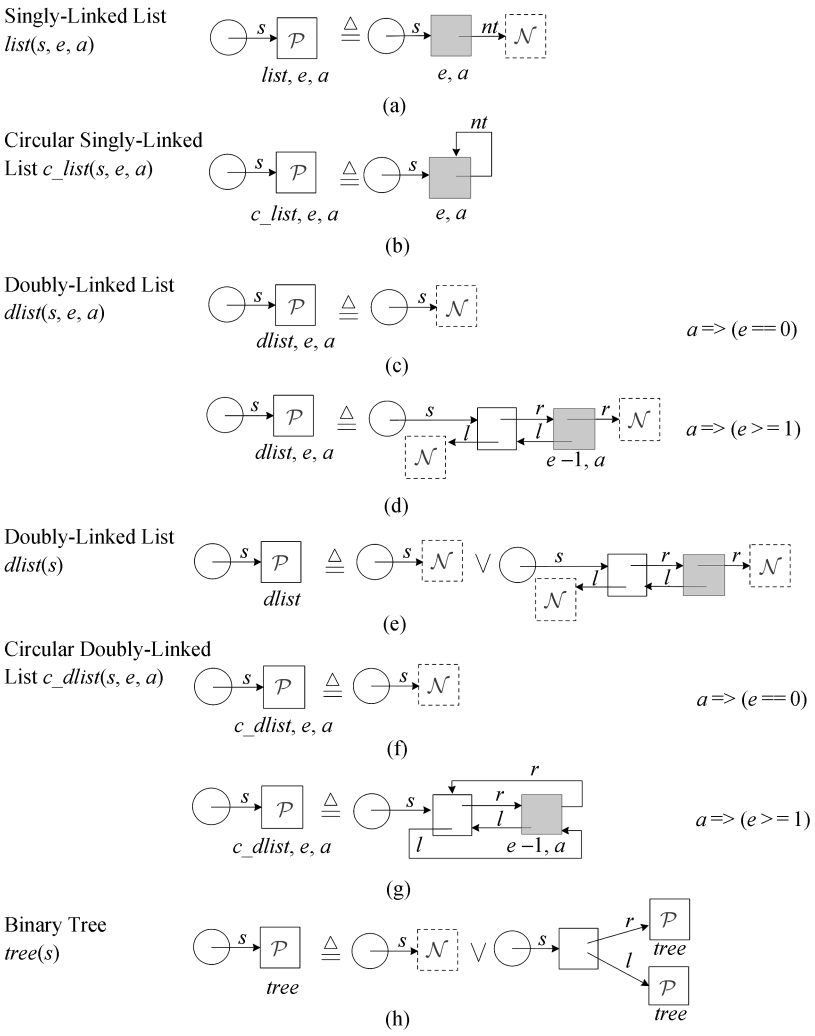

Fig.4. Part of the definitions of shape predicates.

Definition 2 (Minimal Shape Graph for Data Structure). *Shape sub-graphs on the left-/right-hand sides of definitions shown in Fig.*4 (*including definitions not shown here due to limited space*) *are all called minimal shape graphs for data structure.*

2.3 Transformation Rules of Shape Graph

Before giving the rules, we define the concept of a window, which describes the concerned part of one shape sub-graph in terms of local reasoning.

Definition 3 (Window). *A window is nodes and edges in a shape sub-graph encircled by a frame in dot and dash lines. The other part of the shape graph is called the context of this window.*

The followings should be satisfied.

1) *Nodes of shape graphs are in a window or in the context of the window* (*nodes cannot span the window and the context*).

2) *Edges between nodes in a window are all in the window. Edges across the frame, which are used to connect nodes inside and outside the window, belong to the window. There is a copy of these edges across the frame in its context of this window.*

We use W and $X[]$ to denote a window and a context. A window W and a matching context $X[]$ can form a shape graph $X[W]$. Edges across the frame of W and edges in $X[]$ as a copy are checked whether they are coincident. If coincident, the context is a matching one for the window.

We use the following notations in rules using windows. If there exists at least one edge (e.g., p_1 in Fig.5) pointing to certain node in the window, all the other possible edges (may be zero) pointing to this node are represented using one edge labeled p_k.

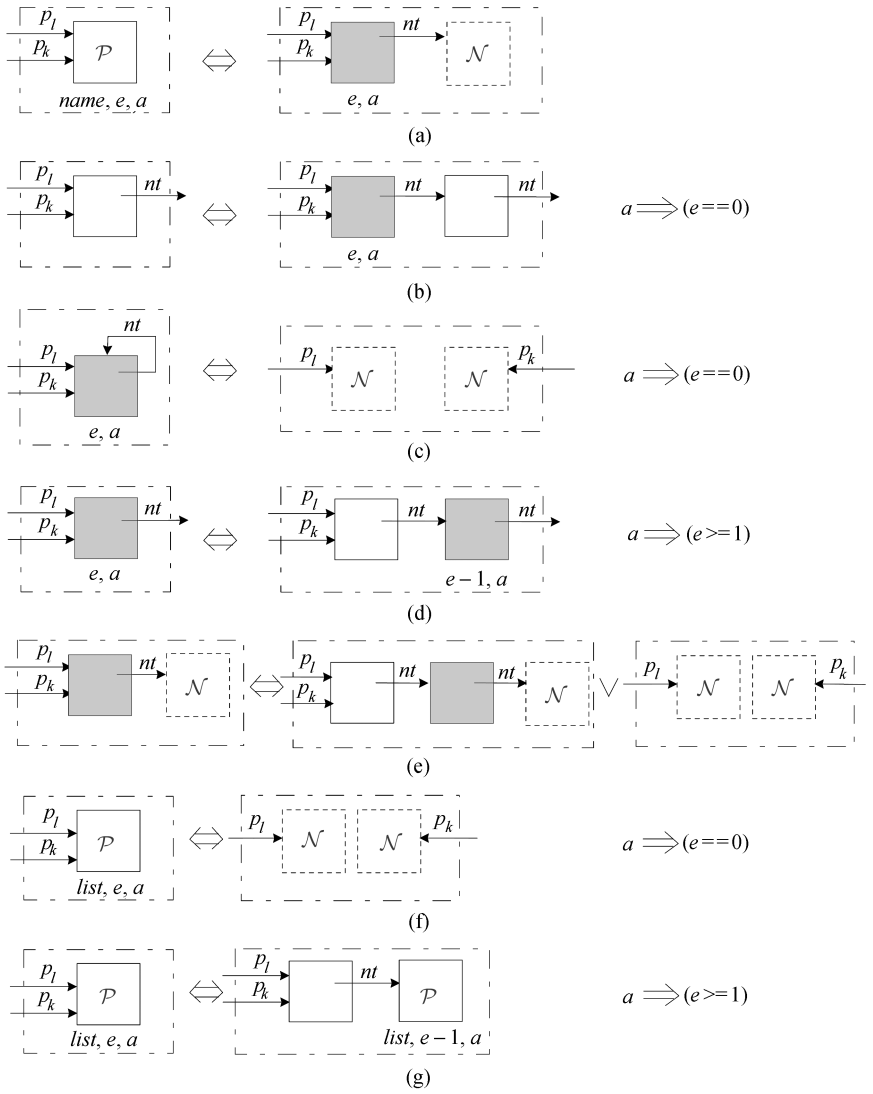

Fig.5. Part of the equivalent transformation rules for (circular) singly-linked list.

Next we introduce the equivalent transformation rules of shape graphs. They are used to represent transformations between shape graphs preserving semantic equivalence. We give the proof after introducing semantics of shape graphs.

2.3.1 *Equivalent Transformation Rules for (Circular) Singly-Linked List*

Basic Rules.

• Rules based on predicate definitions of (circular) singly-linked list. For example, rule in Fig.5(a) is a rule based on definition of $list(s, e, a)$ in Fig.4.

• Rules adding or removing a condensation node whose expression e equals zero. Rules in Figs. 5(b), 5(c) are this kind of rules. For rule in Fig.5(b), a similar rule is not included in Fig.5, in which the condensation node interchanges its position with the structure node in the right-hand side window. Moreover, if we change the structure node into predicate node, null node or dangling node (remove its out-edge), the result is also a basic rule of this kind.

• Rules folding and unfolding a condensation node whose expression e is greater than zero. Rule in Fig.5(d) is one of this kind of rules. Similarly, we can give a rule by interchanging the positions of the structure node and the condensation node in the right-hand side window.

• Rules folding and unfolding an unconstrained condensation node. Rule in Fig.5(e) is one of this kind of rules. Rules similar to rule in Fig.5(e) can be given if the nt edge sourced from the condensation node points to other kinds of nodes.

• Rules folding and unfolding predicate nodes. Rules in Figs. 5(f) and 5(g) are two of this kind of rules.

Derived Rules.

• One derived rule states an equivalent transformation between a condensation node with $e_1 + e_2$, $a_1 \wedge a_2$ below it and two adjacent condensation nodes (with e_1, a_1 and e_2, a_2 below respectively). Similarly, it is an equivalent transformation, from two adjacent unconstrained condensation nodes to one unconstrained condensation node.

• Another derived rule is for an equivalent transformation between one structure node and one condensation node with 1, true below it.

• Another kind of derived rule is related to an alternative inductive definition of $list(s, e, a)$. We can use the disjunction of right-hand sides without edges labeled with p_k in rules in Figs. 5(f) and 5(g) as the right-hand side of the definition. Similar rules can be given for $list(s)$.

2.3.2 *Equivalent Transformation Rules for (Circular) Doubly-Linked List*

Basic Rules.

• Rules based on predicate definitions of (circular) doubly-linked lists.

• Rules adding or removing a condensation node whose expression e equals zero (e.g., rule in Fig.6(a)).

• Rules folding and unfolding a condensation node whose expression e is greater than zero (e.g., rule in Fig.6(b)). Due to the symmetric status of labels l and r, rules in Figs. 6(a) and 6(b) can also be applied to cases where the condensation node is pointed by edge r from the structure node.

• Rules folding and unfolding a condensation node, which is a marginal node (head node or tail node) in a doubly-linked list. They are slightly different from rules in Figs. 6(a) and 6(b).

• Rules folding and unfolding an unconstrained condensation node (e.g., rule in Fig.6(c)). Similar rules can be given if edge r sourced from the condensation node points to other kinds of nodes, or the condensation node itself is a marginal node.

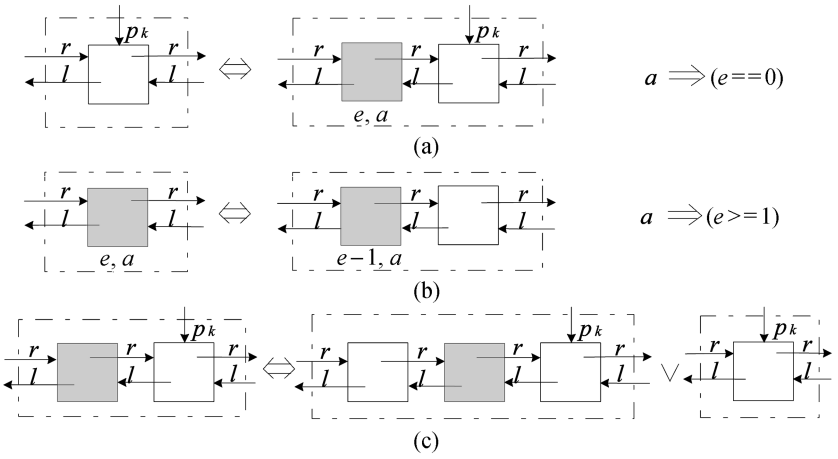

Fig.6. Part of the equivalent transformation rules for (circular) doubly-linked list.

Derived Rules.

Rules can be given in a way similar to derived rules of (circular) singly-linked lists (except rules related to alternative inductive definitions).

No other edges (except the two edges labeled with l and r) point to the condensation node is an important principle in given definitions and rules for (circular) doubly-linked lists. The reason is that if we allow other edges to point to condensation nodes, we do not know which nodes they should point to in the unfolded shape graph.

2.3.3 Equivalent Transformation Rules for Binary Tree

Besides the rules based on the predicate $tree(s)$, there are rules for condensation nodes similar to those of singly-linked lists. The difference is that there is one additional edge from each structure node or condensation node to a tree predicate node.

2.3.4 Equivalent Transformation Rules for Substitutions in e and a of Condensation Node

Rules in Figs. 7(a) and 7(b) are rules for condensation nodes of (circular) singly-linked lists and (circular) doubly-linked lists respectively. Similar rules can be given for condensation nodes of binary trees or for condensation nodes as the marginal nodes in doubly-linked lists.

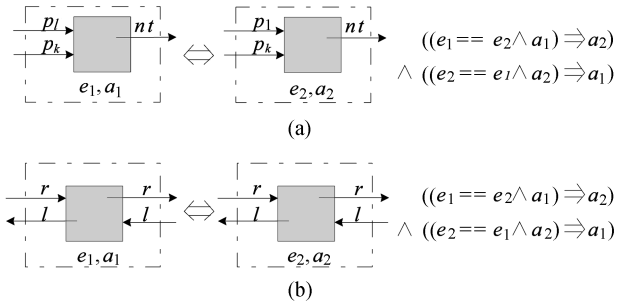

Fig.7. Part of the equivalent transformation rules for substitutions in condensation node.

We can do transformations on shape graphs using these equivalent transformation rules. For example, for a proper context $X[]$ and rule $W_1 \Leftrightarrow W_2$, we can apply this rule to shape graph $X[W_1]$ to get $X[W_2]$ and vice versa. That is, $X[W_1] \Leftrightarrow X[W_2]$. Similarly, we can get $X[W] \Leftrightarrow X[W_1] \vee X[W_2]$ from corresponding rule $W \Leftrightarrow W_1 \vee W_2$.

Next, implication transformation rules are introduced. They are transformation rules for shape graphs preserving implication in semantics. Corresponding proof will be given in Subsection 2.4.

1) From equivalent transformation rules in the form of $W \Leftrightarrow W_1 \vee W_2$, we can obtain two implication rules $W_1 \Rightarrow W$ and $W_2 \Rightarrow W$.

2) From equivalent transformation rules in the form of $W_1 \Leftrightarrow W_2$ with side condition $((e_1 == e_2 \wedge a_1) \Rightarrow a_2) \wedge ((e_2 == e_1 \wedge a_2) \Rightarrow a_1)$, we can obtain two implication rules $W_1 \Rightarrow W_2$ and $W_2 \Rightarrow W_1$ with side condition $(e_1 == e_2 \wedge a_1) \Rightarrow a_2$ and $(e_2 == e_1 \wedge a_2) \Rightarrow a_1$ respectively.

3) There are implication transformation rules for shape graphs changing condensation nodes with e, a to unconstrained condensation nodes.

• For equivalent transformation rules in the form of $W_1 \Leftrightarrow W_2$, if there are condensation nodes with e_1, a_1 and e_2, a_2 in W_1 and W_2 respectively and e_1 is less than e_2 under constraint $a_1 \wedge a_2$, we can obtain an implication rule $W_1' \Rightarrow W_2'$. $W_1'(W_2')$ differs from $W_1(W_2)$ in that the former has only replaced the condensation node with an unconstrained condensation node and other nodes and edges are the same with the latter.

• For equivalent transformation rules in the form of $W_1 \Leftrightarrow W_2$, if there is a condensation node with e, a in W_2 and no condensation node in W_1, we can obtain an implication rule $W_1 \Rightarrow W_2'$. W_2' differs from W_2 in the above mentioned way.

2.4 Semantics of Shape Graph

For a programming language with dynamic memory allocation such as C, nodes denote stack slots or heap blocks (except for the null and dangling node), and edges represent values of corresponding pointers. A shape graph without condensation nodes and predicate nodes is a graphical representation of machine state. A general shape graph is a graphical representation of machine state set.

First, we define the semantics of nodes and edges. In a machine with a stack and a heap, denotations (or meanings) of nodes and edges in shape graphs are given as follows.

1) A declaration node represents a declared pointer whose name is the label of the out-edge of this declaration node. Different declaration nodes denote distinct stack slots.

2) A structure node represents a structure variable created by calling *malloc*. The number of out-edges is the same with the number of the field pointers in the corresponding structure variable. Labels of these edges are corresponding names of field pointers. A structure node denotes a heap block whose cell number is its out-degree and abstract addresses of cells are corresponding labels.

Cells of other types have little relationship with our discussion on shapes, so they are not considered here. Unconstrained condensation nodes are regarded as condensation nodes with expression n and assertion $n \geqslant 0$ in the following part of this subsection.

3) A condensation node with e, a below represents n structure variables and denotes n separated heap blocks (assume the value of e is n).

4) Null nodes and dangling nodes do not represent any program object. So they do not denote anything of the machine.

5) A predicate node represents several structure variables allocated dynamically. It denotes a number of separated heap blocks. Based on the rules of predicate unfolding in Subsection 2.3, the relations among these blocks are determined according to semantics of edges.

Edges do not represent any program object. So they do not denote any locations in the machine memory. They are used to represent the values of corresponding pointers.

1) An edge pointing to a structure node represents the address of the heap block denoted by the node.

2) Edges pointing to null nodes are used to express that the values of corresponding pointers are null. Edges pointing to dangling nodes are used to express that corresponding pointers are dangling pointers. Hence, it does not affect the meaning of the shape graph, whether multiple edges point to one null (dangling) node, or they point to different null (dangling) nodes.

3) For an edge pointing to a condensation node with e and a, it represents the address of the heap block denoted by the first structure node (ordered by the direction of edges) of the unfolded condensation node if $e > 0$. Otherwise, if $e = 0$, it should be determined after removing this node using corresponding transformation rules.

4) The denotation of edges pointing predicate nodes can be determined after unfolding corresponding predicate nodes using transformation rules.

Next we present the semantics of shape graphs focusing on pointers by omitting variables of other types and their corresponding memory cells. For non-null and non-dangling pointers, their values indicate the abstract addresses of the heap blocks pointed by them. The addresses of heap blocks dynamically allocated are different abstract values. In our abstract machine, stack slots and heap cells are accessed via names of declared pointers and names of field pointers respectively. Thus, the abstract state of the machine (the machine state for short) can be represented by two functions: $s_d : DecVar \rightarrow AbsValue \cup \{\mathcal{N}, \mathcal{D}\}$ and $s_f : AbsValue \times FieldVar \rightarrow AbsValue \cup \{\mathcal{N}, \mathcal{D}\}$. The domain of s_d is the set of declared pointers. s_d maps declared pointers to their abstract values. s_f maps the abstract address of a heap block and a field pointer name to the abstract value of the field pointer ($\mathcal{N}, \mathcal{D}$ are two special abstract values for null and dangling pointers). Next s or $\langle s_d, s_f \rangle$ is used to represent the machine state.

Under such a model, a shape graph without predicate nodes and condensation nodes is a graphic representation of the machine state based on the semantics of the nodes and edges. A generic shape graph is a graphic representation of a certain set of machine state.

Definition 4. *The set of machine states $\mathcal{S}[\![G]\!]$ represented by shape graph G can be defined using the following rules*:

1) *If there is no predicate node or condensation node in G, then there is only one state in $\mathcal{S}[\![G]\!]$. G represents this state directly where all labels of out-edges of declaration nodes and the values of corresponding out-edges form function s_d and structure nodes plus the labels of their out-edges form function s_f.*

2) *If there is a condensation node with e and a below it and e has k values (implied by a), under these k cases the condensation node can be unfolded completely into shape graphs $G_1, \ldots, G_k$. Then, $\mathcal{S}[\![G]\!] = \mathcal{S}[\![G_1]\!] \cup \cdots \cup \mathcal{S}[\![G_k]\!]$. If a implies that e could equal $0, 1, \ldots$, and the condensation node can be unfolded completely into the shape graph G_n with n structure nodes ($n \geqslant 0$), then $\mathcal{S}[\![G]\!] = \mathcal{S}[\![G_0]\!] \cup \mathcal{S}[\![G_1]\!] \cup \cdots \cup \mathcal{S}[\![G_n]\!] \cup \cdots$.*

3) *If there is a predicate node in G and its corresponding definition body is G' or $G_1 \vee G_2$, then $\mathcal{S}[\![G]\!] = \mathcal{S}[\![G']\!]$ or $\mathcal{S}[\![G]\!] = \mathcal{S}[\![G_1]\!] \cup \mathcal{S}[\![G_2]\!]$.*

Similarly, we can define the semantics of $G_1 \vee G_2$.

The syntax for the access path in shape graphs is the same as that in PointerC in favor of discussing the relations between them. Clearly, only access paths from a declaration node to the other node are considered. We will introduce in the following two cases.

• Full representation of access paths. In this case, the access path does not cross any condensation node, and the labels of the edges on the path should be listed in turn to represent such an access path. For example, if the labels are *p*, *left*, *right* in turn, the access path is written as *p->left->right*.

• Condensed representation of access paths. In this case, the access path crosses one or more condensation nodes. For instance, if the path includes an out-edge labeled *left* of a condensation node which stands for n number of nodes, then $(\text{->}left)^n$ should appear in the access path, such as $hd(\text{->}nxt)^m$ and $hd(\text{->}nxt)^m\text{->}nxt$ in Fig.3(b).

A path in shape graphs represents a pointer related to the last edge, which is identical to the access path for this pointer in programs. They are collectively called access paths later.

Shape graphs are graphic representation of equalities and validity of pointers at given program points.

Definition 5. *The set of assertions $\mathcal{A}[\![G]\!]$ represented by shape graph G includes the following assertions about pointers*: 1) *Pointers pointing to structure nodes are valid*; *pointers pointing to the null/dangling nodes are not valid.* 2) *Pointers pointing to the same structure or predicate node are equal*; *pointers pointing to the same structure node of an unfolded condensation node are equal.* 3) *The value of pointers pointing to the null node is* NULL; *pointers pointing to dangling nodes are dangling pointers.* 4) *Pointers pointing to predicate nodes make corresponding predicates hold.*

For instance, $ptr == hd(->nxt)^m->nxt$ appears in the assertion set represented by the shape graph of Fig.3(b). One can infer from the set of assertions that whether two access paths of pointer types are equal or not and whether two access paths are aliases or not.

Theorem 1. *For any shape graph G, any state in $\mathcal{S}[\![G]\!]$ satisfies all assertions in $\mathcal{A}[\![G]\!]$. That is, $\forall s : \mathcal{S}[\![G]\!]$. $s \vDash \mathcal{A}[\![G]\!]$. And for any $s' \notin \mathcal{S}[\![G]\!]$, if $dom(s'_d) = dom(s_d)$ $(s \in \mathcal{S}[\![G]\!])$ and the size of AbsValue and FieldVar in $s'_f : AbsValue \times FieldVar \to AbsValue \cup \{\mathcal{N}, \mathcal{D}\}$ is the same as those in s_f, then $s' \nvDash \mathcal{A}[\![G]\!]$.*

Proof. *AccessPath* is used to represent the set of access paths of shape graph G. *State* is used to stand for $\mathcal{S}[\![G]\!]$. Then $GetAbsValue{:}AccessPath \times State \to AbsValue \cup \{\mathcal{N}, \mathcal{D}\}$, which maps access paths to their abstract values, can be defined as follows.

$$GetAbsValue[\![u]\!]\langle s_d, s_f\rangle =$$
$$\begin{cases} s_d(u), & \text{if } u \text{ is a declared variable;} \\ s_f(GetAbsValue[\![v]\!]\langle s_d, s_f\rangle, next), & \\ \quad \text{if } u \text{ is in the form of } v->next. & \end{cases}$$

Obviously, the abstract value of an access path u $(u \in AccessPath)$ pointing to a node is the address of the heap block represented by the node or in the set $\{\mathcal{N}, \mathcal{D}\}$. If access paths u and v $(u, v \in AccessPath)$ point to the same node in a shape graph, they have the same abstract value. Hence, if there is neither condensation node nor predicate node in G, Theorem 1 holds. Theorem 1 can be generalized to generic G which can be proved by induction using similar steps as shown in Definition 4.

Theorem 2. *Using any transform rule of shape graphs, $G \Leftrightarrow G'$ or $G \Leftrightarrow G_1 \vee G_2$ can be inferred, $\mathcal{S}[\![G]\!]$=$\mathcal{S}[\![G']\!]$ or $\mathcal{S}[\![G]\!]$=$\mathcal{S}[\![G_1]\!] \cup \mathcal{S}[\![G_2]\!]$ holds. With respect to $G \Leftrightarrow G'$, for any state s: $\mathcal{S}[\![G]\!]$, s⊨$\mathcal{A}[\![G]\!]$ if and only if $s \vDash \mathcal{A}[\![G']\!]$. With respect to $G \Leftrightarrow G_1 \vee G_2$, for any state s: $\mathcal{S}[\![G]\!]$, $s \vDash \mathcal{A}[\![G]\!]$ if and only if $s \vDash \mathcal{A}[\![G_1]\!]$ or $s \vDash \mathcal{A}[\![G_2]\!]$. For rules of implication, this theorem still holds if the equality between state sets is changed into $\subseteq$ and if and only if is changed to the logical implication.*

For each transform rule, Theorem 2 can be proved based on the construction rules of state sets in Definition 4. For instance, from the rule in Fig.5(e) we can get $G \Leftrightarrow G_1 \vee G_2$. Hence using rule 3 in Definition 4, $\mathcal{S}[\![G]\!]$ is $\mathcal{S}[\![G_1]\!] \cup \mathcal{S}[\![G_2]\!]$, where the states in $\mathcal{S}[\![G_1]\!]$ and $\mathcal{S}[\![G_2]\!]$ satisfy $\mathcal{A}[\![G_1]\!]$ and $\mathcal{A}[\![G_2]\!]$ respectively.

From Theorem 2, if shape graphs are regarded as assertion sets, the transform rules of shape graphs are equivalence or implication rules of the assertion calculus. These rules are valid with respect to our machine model.

Shape graphs, as assertions about pointers in data structures, are part of program assertions. We use $G_1 \vee G_2 \vee \cdots \vee G_k$ in disjunction normal form (DNF) to express different cases of data structure at a program point. Symbols such as $\Leftrightarrow$ and $\Rightarrow$ can be regarded as logical equivalence and implication connectives respectively.

2.5 Relations Between Shape Graph and Symbolic Assertion

As assertion, a shape graph can be calculated with symbolic assertions. When using rules extended from Hoare logic, Boolean expressions, such as u==NULL, u!=NULL, u==v and u!=v (where u and v are access paths of pointer types), will be used in conjunctions with shape graphs. So special rules are needed to deal with this kind of conjunctions. Such symbolic assertion will be removed if it does not conflict with the shape graph and otherwise the whole assertion implies false. Fig.8(a) is one of this kind of rules.

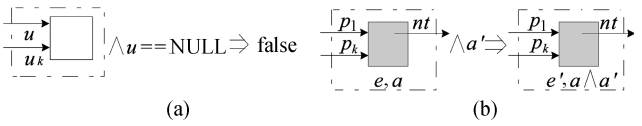

Fig.8. Part of the rules for conjunctions of shape graphs and symbolic assertions.

If an assertion a' appears in the symbolic assertion and it affects e, a' should be added to corresponding node constraint. One of this kind of rules is shown in Fig.8(b). If $a \wedge a'$ is false, the whole assertion is also false.

Note that names u, u_k under the arrows represent access paths of corresponding edges (names above the edges are labels) in Fig.8(a). In the following the paper confirms to these conventions. Different requirements should be satisfied when applying rules with ac-

cess paths below edges other than with labels above edges. We name the window W in this rule. If we want to apply this rule to $X[W]$, there must exist an access path u in $X[W]$ to the node of W.

These rules can be easily proven to be valid with respect to our machine model.

2.6 Relations Between Shape Graphs and Access Path Sets

Prior to the design of shape graph logic, we have designed a pointer logic[8] using access path set (e.g., $\{p$, s-> r-> $l\}$ in which p and s->r->l are access paths in a C-like program language) as assertions. Pointer logic can be regarded as a symbolic version of shape graph logic. So in this subsection, we compare shape graphs with access path sets in the pointer logic.

We show the differences and relations using a quick example (see Fig.9). This example is a code fragment from a function inserting a node into a circular doubly-linked list. The type of the nodes is *typedef struct node*{*Node* l*; *Node* r*;} *Node*. In Fig.9, p points to a node of the circular doubly-linked list, and s points to the node which should be inserted into the left side of the node pointed by q. Shape graphs and corresponding access path sets are given as assertions at some program points. In the assertions, n ($n \geqslant 0$) is used to denote the length of the list. And i ($0 < i < n$) is the number of nodes from the node pointed by p to the node pointed by q following the r link (the node pointed by q is not included).

Relations between shape graphs (nodes and directed edges) with access path sets include the following aspects:

1) Every structure node *nd* corresponds to exact one access path set *st*. Directed edges pointing to the structure node *nd* have one-to-one correspondences with

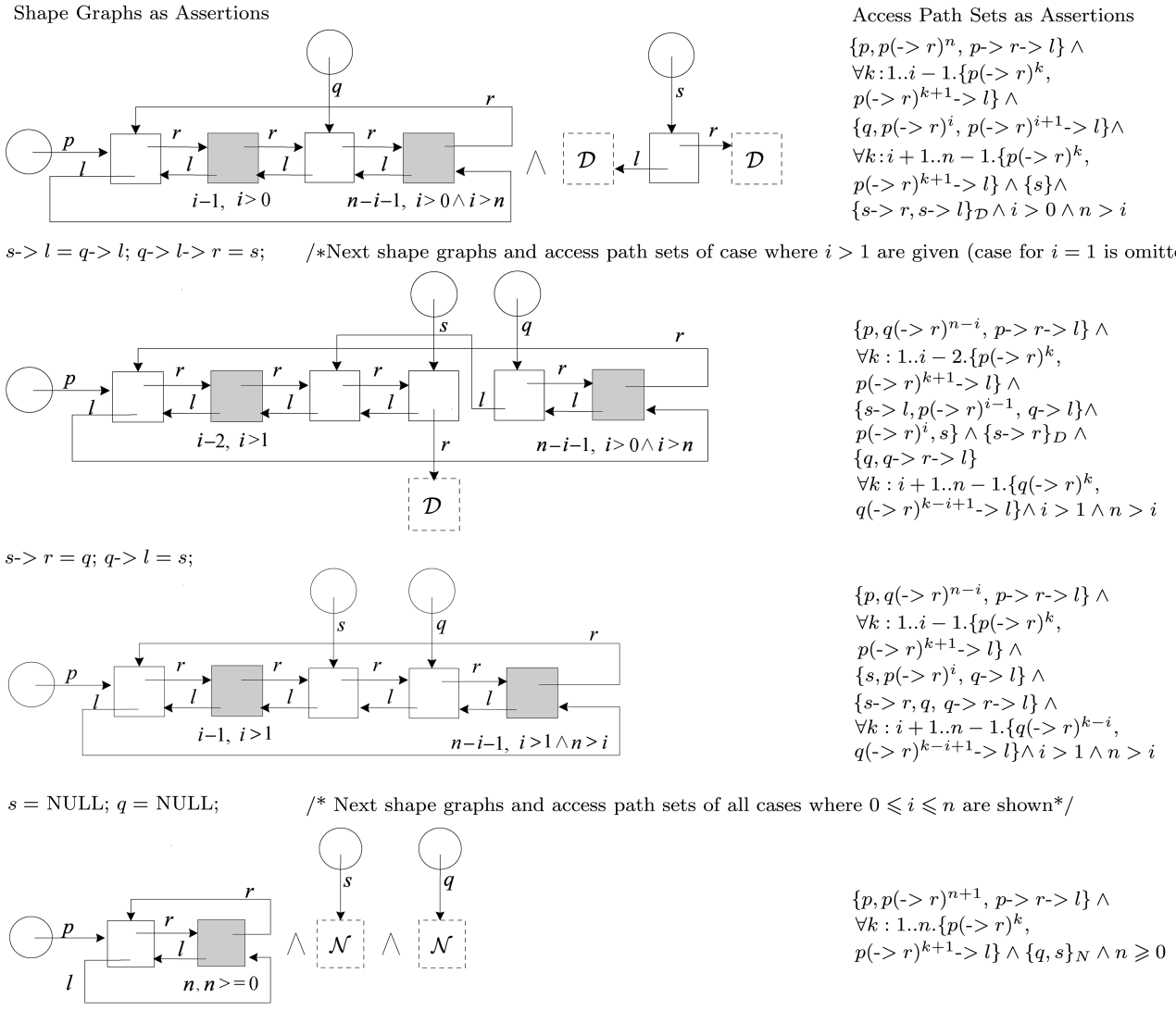

Fig.9. Code fragment with assertions for function inserting a node into a circular doubly-linked list.

access paths in the corresponding set *st*. In Fig.9, access paths with superscript are only used in assertions. For example, $p(\text{-}> r)^2\text{-}>l$ denotes $p\text{-}>r\text{-}>r\text{-}>l$.

2) A condensation node corresponds to several access path sets (decided by e and a below the condensation node). These access paths sets can be universal quantified. But when the condensation node in a singly-linked list is pointed by multiple edges (pointers), access path set of this condensation node cannot be quantified anymore.

3) All null nodes correspond to one access path set with the subscript $\mathcal{N}$. All dangling nodes correspond to one access path set with the subscript $\mathcal{D}$. All directed edges pointing to null (dangling) nodes have one-to-one correspondences with access paths in the access path set with the subscript $\mathcal{N}$ ($\mathcal{D}$, respectively).

4) Predicate nodes correspond to no access path set, but shape predicates correspond to predicates defined using access path sets. For example, the binary tree in Fig.4 is corresponding to the following inductive definition using access path sets:

$$tree(s) \triangleq \{s\}_{\mathcal{N}} \vee (\{s\} \wedge tree(s\text{-}> l) \wedge tree(s\text{-}> r)).$$

5) Declaration nodes correspond to no access path set, since there is no pointer pointing to declaration nodes.

Obviously, pointers represented by directed edges pointing to the same node (excluding dangling node) correspond to pointers represented by access paths in one access path set (excluding set with subscript $\mathcal{D}$). Access path sets are sets of pointers equal to each other. When one pointer can be represented by multiple access paths, we only choose one of them as an element in some set in our pointer logic assertion. Thus, access path sets can be regarded as assertions on pointer equality. So access paths can be connected using logical conjunction and disjunction, such as we connect shape graphs using these two logical connectives.

Transformation rules for shape graphs in Subsection 2.3 correspond to a set of assertion calculation rules in the pointer logic. In the pointer logic, the rule corresponding to the rule in Fig.8(a) is $\mathcal{S}_1 \wedge \cdots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^u \wedge \mathcal{N} \wedge \mathcal{D} \wedge (u == \text{NULL}) \Rightarrow \text{false}$, where $\mathcal{S}_i$ $(1 \leqslant i \leqslant n-1)$ and $\mathcal{S}^u$ are access path sets corresponding to structure nodes, $\mathcal{N}$ and $\mathcal{D}$ are sets of NULL pointers and dangling pointers respectively. $\mathcal{S}^u$ is an access path set including pointer u, which behaves similarly to the window (context) in the rule of Fig.8(a).

3 Shape Graph Logic

Assertions at each program point include shape graphs and possible symbolic assertions (Q). The assertion at each program point preserves the DNF $(G_1 \wedge Q_1) \vee (G_2 \wedge Q_2) \vee \cdots \vee (G_n \wedge Q_n)$ in the verification stage. We only need to discuss one case $G \wedge Q$ (G and Q both are conjunctions). The general form of program specifications is $\{G \wedge Q\}\ C\ \{G' \wedge Q'\}$.

For pointer statements (e.g., assignment statements of pointers, allocation statements, deallocation statements), G' is different from G. And Q' may be different from Q slightly caused by alias substitutions. For non-pointer statements, G' differs from G at most in expression e and assertion a below nodes.

With respect to the specification in form of $\{G\}C\{G'\}$, Subsection 3.1 will only give the rules representing changes of shape graphs for pointer statements. These rules are used in the shape analysis stage. Complete rules concerning both shape graphs and symbolic assertions will be introduced in Subsection 3.2.

3.1 Inference Rules Focusing on Changes of Shape Graphs

When using the following rules, symbolic assertions of non-/equation between pointers, such as $u ==$ NULL, have already been removed using assertion calculus rules; access paths in statement C must appear in G at the program point before C, and they can reach condensation nodes and will not cross them. Otherwise, corresponding condensation nodes should be unfolded, or errors will be reported if unfolding operation cannot be done).

Pointer Assignment Statement $u = v$.

1) Pointer u points to a null/dangling node, and pointer v points to a structure/predicate node, or a condensation node of a (circular) singly-linked list.

In the new shape graph at the program point after the assignment, pointer u will point to the node originally pointed by pointer v. Fig.10 shows one of the rules. There are windows (i.e., local nodes and edges which we focus on currently), not full shape graphs, in braces of the rule. This rule means that if we name the four sub-graphs as W_{11}, W_{12}, W_{21} and W_{22}, respectively, for any proper shape context $X[]$ (i.e., $X[W_{11}, W_{12}]$ is a shape graph), we can get

$$\{X[W_{11}, W_{12}]\}u = v\{X[W_{21}, W_{22}]\},$$

where windows W_{11}, W_{12} are not overlapped with each other.

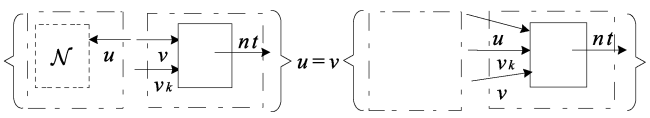

Fig.10. One case of the rules for assignment statement.

So strictly speaking, the rule in Fig.10 is a rule for windows. We also use inference rules to name this kind of rules for simplicity.

2) Pointer v points to a null/dangling node (v can be the constant NULL), and pointer u points to a structure/predicate node, or a condensation node of a (circular) singly-linked list.

One rule is shown in Fig.11. If only u points to the structure node, memory leak will be reported in the analysis stage. Similar rules can be given if the structure node has two out-edges. We use a simple example to explain the necessity of the side condition in this rule. G is a shape graph formed by an edge q connecting a declaration node with a one-node circular singly-linked list. Applying this rule to statement $q = \text{NULL}$ with respect to G, memory leak should be reported. However, it cannot report memory leak without this side condition.

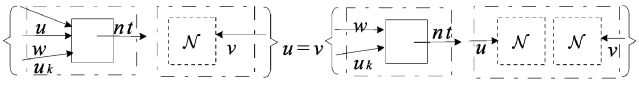

Fig.11. Another case of the rules for assignment statement.

We use Fig.12 to explain the side condition for the rule in Fig.11. Suppose we use this rule to reason the assignment statement $p = \text{NULL}$ whose precondition is the shape graph in Fig.12. If there is no such a side condition, we can apply the rule, but memory leak will be caused although there are two pointers (edges, in access paths p and p->nxt->nxt) point to the same node pointed by edge p. The side condition forbids the application of this rule under such circumstance, because access path p->nxt->nxt has p as its prefix.

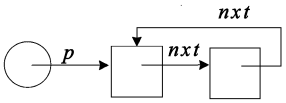

Fig.12. Example shape graph of circular singly-linked list.

3) Pointers u and v point to different structure, condensation or predicate nodes. In such a case, this statement will be reasoned as a sequence $dummy = v$; $u = \text{NULL}$; $u = dummy$; $dummy = \text{NULL}$ where *dummy* is a virtual pointer for every function (different from any other local variables). Using this method, we avoid designing one complex rule for this case of assignment. For example, with respect to the shape graph in Fig.12 as the precondition, assignment statement $p = p$->nxt will not cause memory leak. If we design a specific rule reporting all memory leaks in the analysis stage without false positives, the resulting rule will be very complex.

Allocation Statement $u = malloc(t)$.

1) Pointer u points to a null/dangling node. The shape graph for the post-condition will be given by: adding a new structure node; making u point to this node; adding edges (with labels) pointing to a dangling node for each field pointer of the structure node according to type t. Fig.13 shows one case of such rules, supposing the type t has one field pointer f.

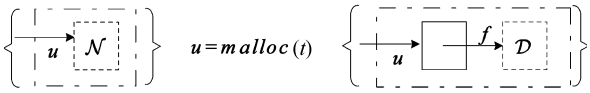

Fig.13. One rule for allocation statement.

2) Pointer u points to a structure, condensation or predicate node. This statement will be treated as a sequence $u = \text{NULL}$; $u = malloc(t)$. Corresponding rules will be used to get the resulting shape graph.

Statement free(u). According to the type of the node pointed by pointer u, we can know all the access paths for the field pointers such as u->$r_1, \ldots, u$->r_n. Then, this statement can be treated as a sequence u->$r_1 = \text{NULL}$; ...; u-> $r_n = \text{NULL}$; *free*(u) instead. Corresponding rules will be used to get the resulting shape graph. The last statement *free*(u) will delete the node pointed by u, including its out-edges and the nodes pointed by these out-edges, and make all the in-edges of this node point to a dangling node.

Statements Related to the Function Construction. Due to limited space, next rules for functions with one pointer parameter (namely *arg*) and return value of pointer types will be given. Other rules can be figured out similarly. Suppose that these functions have two virtual pointers *res* (representing the return value) and *dummy* (mentioned above).

1) Declaration statements of local pointers. First, one shape sub-graph is the constructed where a declaration node connects a dangling node by an edge with one of the following names as the label: $q_1, \ldots, q_k$ (representing local pointers), *res* and *dummy*. The resulting shape graph is the conjunctions of these shape sub-graphs and the shape graph of the function's precondition.

2) Statement *return exp*. This statement will be treated as a sequence *res*=*exp*; $q_1 = \text{NULL}$; ..., $q_k = \text{NULL}$; $arg = \text{NULL}$; *return*. Corresponding rules will be used to get the resulting shape graph. The last *return* statement will delete all of the shape sub-graphs which $q_1, \ldots, q_k$, *dummy*, and *arg* belong to. Note that, if local variables or parameters appear in e, a of condensation nodes, corresponding condensation node should be replaced with an unconstrained condensation node.

3) Function call statement $ret = f(act)$. Only one case will be considered, where *ret* equals NULL and *ret* is not the same pointer as *act* (shown in Fig.14, S is the body of function f). Note that a declaration node arg', which points to the same node pointed by the actual argument, is added to the shape graph of the function pre-condition. It points to the same node as the declaration node *arg*. If the node pointed by the actual argument is modified in the function, it will be fed back to the shape graph of the caller. If we constrain the syntax of function call statement as $act{=}f(act)$, there is no need to add such a declaration node arg'.

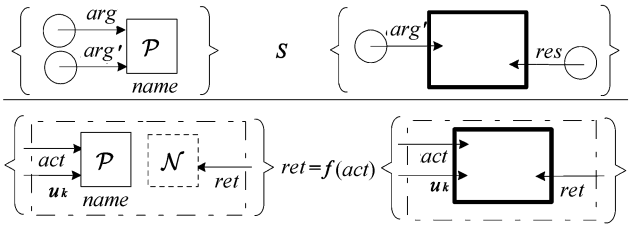

Fig.14. One of the rules for function call statement.

The squares in bold solid line (black box) represent that only the edges cross an edge of the square and nodes outside the squares are concerned (graphs inside the squares are not changed), in contrast with the rectangles in dot and dash lines. In this rule, the effect of the function f is reflected in modifying G_{caller} using the shape graph G_{callee} of the post-condition of the function f to get the shape graphs after the function call statement. That is, the predicate node and null node are replaced using graphs in the black box of G_{callee}. Then we use *act*, u_k to replace arg' across the black box and use *ret* to replace *res*. The resulting shape graph is the shape graph after the function call statement.

Rules for Integer Assignment Statements. If modified variables by such assignment statements appear in expression e and assertion a of condensation nodes, e and a in corresponding G' should be changed with respect to assignment axiom.

Rules for Composition/Conditional/Loop Statements. Composition rule, conditional rule, loop rule and consequence rule are identical to corresponding rules in Hoare logic.

Case Analysis Rule. If the pre-condition of C is $G_1 \vee G_2$, then its post-condition can be obtained by applying the following rule.

$$\frac{\{G_1\}C\{G'_1 \vee \cdots \vee G'_m\}\ \{G_2\}C\{G''_1 \vee \cdots \vee G''_n\}}{\{G_1 \vee G_2\}C\{G'_1 \vee \cdots \vee G'_m \vee G''_1 \vee \cdots \vee G''_n\}}$$
$$m, n \geqslant 1.$$

Frame Rule. We use local reasoning ideas in aforementioned rules. That is, when pointer statement C changes the program state, the changes are reflected by changing shape graphs (point-to relations of edges and nodes) in the window W, not the context $X[]$). The following rule is one of our frame rules for contexts and windows.

$$\frac{\{W_1\}C\{W_2\}}{\{X[W_1]\}C\{X[W_2]\}}.$$

3.2 Inference Rules Dealing with Shape Graphs and Symbolic Assertions

In this subsection, we give rules for non-pointer statements (statements not dealing with pointers) and complete the rules for pointer statements to include symbolic assertions based on the rules presented in Subsection 3.1. These rules are used in the verification stage. When we reason statement C under the pre-condition $G \wedge Q$ using the corresponding rule to get the post-condition $G' \wedge Q'$, the following should be guaranteed:

• The legality of access paths in Q and C (with respect to G), and no aliasing in Q and C.

• There is no assertion in Q duplicated or contradict with G, such as $p{=}{=}$NULL, $p{=}{=}q$.

• More guarantee should be fulfilled if predicates and quantifiers appear in Q (See Subsection 5.1).

If any alias exists, we can eliminate aliasing using information provided by the shape graph G (using one access path to replace its aliases), then the following rules can be used. We define a function eliminate_aliases(G, C, Q) which returns (C', Q') by eliminating the aliases in C and Q with respect to G.

For example, suppose G is as shown in Fig.15, C is $p{\text{->}}data{=}10$ and Q is $p{\text{->}}data{=}{=}5 \wedge q{\text{->}}data{=}{=}5$. From G we can easily infer that $p{\text{->}}data$ and $q{\text{->}}data$ are aliases to each other. If we apply the assignment axiom for non-pointer statements in Subsection 3.2, we can get a wrong post-condition for statement C: $x{=}{=}5 \wedge q{\text{->}}d\text{ata}{=}{=}5 \wedge p{\text{->}}d\text{ata}{=}{=}10$ where x is fresh variable. So we must do alias substitution before we apply corresponding Hoare rules. In this example, eliminate_aliases(G, C, Q) returns C' (same as C) and Q'. Q' is $p{\text{->}}data{=}{=}5 \wedge p{\text{->}}data{=}{=}5$ by replacing q by p in Q since p are equal to q.

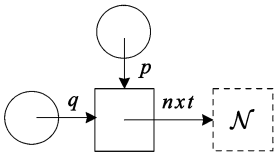

Fig.15. Example for eliminating aliases.

All of our inference rules must be used when there is no aliasing in both assertions and statements. So the following rule (eliminating alias rule) is added in order

to reason about programs containing aliases of access paths.

$$\frac{\{G \wedge Q''\}C'\{G' \wedge Q'\}}{\{G \wedge Q\}C\{G' \wedge Q'\}} \quad (C', Q'')$$
$$= \mathsf{eliminate\_aliases}(G, C, Q).$$

Non-Pointer Statements. For assignment statements for non-pointers, the assignment axiom of Hoare logic can be used if x does not appear in expressions or assertions below nodes in G.

$$\frac{x' \text{ is fresh}}{\{G \wedge Q\}x = e\{G \wedge Q[x'/x] \wedge x == E[x'/x]\}}.$$

Other rules of Hoare logic for other non-pointer statements (e.g., function call statement which does not modify any pointer) can be used directly.

Pointer Statements. For pointer statements, complete rules are given to adjust Q based on the rules of Subsection 3.1.

1) Pointer assignment statement u=v. The rule is

$$\frac{\{G\}u = v\{G'\}}{\{G \wedge Q\}u = v\{G' \wedge Q[u'/u]\}}$$
$$(u' \text{ is equal but not alias to } u),$$

where the premise can be acquired in the analysis stage using the corresponding rule of Subsection 3.1.

2) Other statements assigning values to pointers. Symbolic assertion Q in an allocation statement $u = malloc(t)$ and function call statement $ret = f(act)$ is dealt with in the way similarly as in pointer assignment statements.

3) Free statement $free(u)$. After the heap block pointed by u is released, the assertions in Q about pointer u or its alias should not exist anymore. So the rule is:

$$\frac{\{G\}free(u)\{G'\}}{\{G \wedge Q\}free(u)\{G' \wedge Q\}}(u \text{ does not appear in } Q).$$

3.3 Soundness of SGL

The inference rules in Subsection 3.1 can be regarded as graphic representations of the operational semantics for corresponding statements. Changes in shape graphs before and after the executions of statements are reflected in the inference rules. And shape graphs are the graphic representation of the program states. So naturally, changes of the shape graphs are in accord with changes of the program states caused by executions of the statements.

Theorem 3. *For any* $\{G\}$ C $\{G'\}$ *inferred using rules in Subsection* 3.1, *if* $s_1 \in \mathcal{S}[\![G]\!]$ *and the operational semantic of the statement* C *is* $\langle C, s_1\rangle \rightarrow s_2$, *then* $s_2 \in \mathcal{S}[\![G']\!]$.

This theorem can be proved by induction on the structure of statement C. We take pointer assignment statement $u = v$ for example to show how to prove it. The operational semantics are: if u is a declared pointer,

$$\langle u = v, \langle s_d, s_f\rangle\rangle \rightarrow \langle s_d[u \mapsto GetAbsValue[\![v]\!]\langle s_d, s_f\rangle], s_f\rangle;$$

and if u is in the form of w->$next$, and $GetAbsValue[\![w]\!]\langle s_d, s_f\rangle = x$,

$$\langle u = v, \langle s_d, s_f\rangle\rangle \rightarrow$$
$$\langle s_d, s_f[\langle x, next\rangle \mapsto GetAbsValue[\![v]\!]\langle s_d, s_f\rangle]\rangle.$$

From the semantics, we know that the execution of this statement depends on u to modify a stack slot or a heap cell of the state s_1 with other parts unchanged. Only one edge is modified in shape sub-graphs shown in the rules in Fig.10 and Fig.11. It is consistent with the operational semantics.

The changes of shape graphs in other rules are also consistent with the operational semantics of corresponding statements.

Theorem 4. *The inference rules in Subsection* 3.1 *are sound with respect to the operational semantics.*

For any inference rule in the form of $\{W_1\}C\{W_2\}$, we need to prove that for any shape context $X[]$ (if all declared pointers can be covered exactly by both $X[W_1]$ and $X[W_2]$), if $s_1 \vDash \mathcal{A}[\![X[W_1]]\!]$ and $\langle C, s_1\rangle \rightarrow s_2$, then $s_2 \vDash \mathcal{A}[\![X[W_2]]\!]$.

Using the rule $\{W_1\}C\{W_2\}$, we can get $\{X[W_1]\}C\{X[W_2]\}$. From Theorem 1, we can infer that if $s_1 \vDash \mathcal{A}[\![X[W_1]]\!]$, then $s_1 \in \mathcal{S}[\![X[W_1]]\!]$. Then we can get $s_2 \in \mathcal{S}[\![X[W_2]]\!]$ using Theorem 3. Hence we can get $s_2 \vDash \mathcal{A}[\![X[W_2]]\!]$ from Theorem 1.

For inference rules with premises, the soundness with respect to semantics is not difficult to prove.

Based on the soundness of Hoare logic, the soundness proof of SGL needs to be supplemented in proving the following aspects (detailed proof omitted):

1) For PointerC, the assignment axiom of Hoare logic is sound when used with no aliasing in assertions and statements.

2) C' has the same semantics as C, and Q' is equivalent to Q in semantics where $(C', Q') = \mathsf{eliminate\_aliases}(G, C, Q)$.

3) The rule for eliminating aliasing is sound.

3.4 Relations Between Rules of SGL and Pointer Logic

There are correspondences between rules of SGL and the pointer logic. For example, with respect to the rule

in Fig.13, corresponding rule in the pointer logic is: $\{\mathcal{S}_1 \wedge \cdots \wedge \mathcal{S}_n \wedge \mathcal{N}^p \wedge \mathcal{D}\}p = malloc(t)\{\mathcal{S}_1 \wedge \cdots \wedge \mathcal{S}_n \wedge \{p\} \wedge (\mathcal{N}^p - p) \wedge (\mathcal{D} + p\text{-}> f)\}$.

Another example, corresponding rule for the rule in Fig.11 is: $\{\mathcal{S}_1 \wedge \cdots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^u \wedge \mathcal{N}^v \wedge \mathcal{D}\}\ u = v\{\mathcal{S}_1/u \wedge \cdots \wedge \mathcal{S}_{n-1}/u \wedge (\mathcal{S}^u/u - u) \wedge (\mathcal{N}^v/u + u) \wedge \mathcal{D}/u)\}\ (\neg leak(\mathcal{S}^u, u))$.

$\mathcal{S}/p$ is prefix substitution which is strictly defined in [8]: for every access path q in access path set $\mathcal{S}$ which has access path p or alias of p as its prefix (e.g., q is in the form of p->nxt->nxt), q is substituted by its alias which has no prefix like p or alias of p; other access paths will not change. Prefix substitution is a very complex operation. Similarly, the predicate $leak(\mathcal{S}^u, u)$ which is used to judge whether memory leak will appear when we try to assign a new value to pointer u. So, the rules in pointer logic are very complex due to such operations (e.g., prefix substitution for access paths, alias judgement, memory leak judgement) on access paths. Since we do not use access paths in the shape graph logic, such problems will not exist. For example, the first two assignment statements in Fig.9 will break the circular doubly-linked list pointed by pointer p. Access paths for some pointers will change from using p as prefix to using q as prefix. So it is more convenient using shape graphs as assertions (in this case, we only unfold the condensation node and adjust the targets of some edges).

With respect to the shape graphs and access path sets of line 1 in Fig.9, we can judge more easily from the shape graphs whether pointer p points to a circular doubly-linked list. In the implementation of the pointer logic, under many circumstances, we have already transformed the access path sets into some kind of graphs in order to reason properties or do operations much easier.

It is hard to design structural rules for our pointer logic using full access paths to represent pointers.

In the view of these reasons, we design SGL after the pointer logic.

4 Shape System

We have designed a shape system besides SGL in order to raise the threshold of being a legal program. The shape system is used to exclude programs that do not construct or operate on shapes expected by programmers. In this way, we can knock down difficulties of shape analysis and program verification.

4.1 Design of Shape System

The shape system includes: shapes allowed to be declared and their definitions, shape inference rules and shape checking rules. The design and implementation of our shape system are based on SGL. Currently, shapes allowed are those defined in Fig.4 (including definitions not shown).

Shape inference is to judge the shape of the shape sub-graph including a given declared pointer p. We could do shape inference as the followings.

1) get the shape sub-graph G_1 containing p from given shape graphs;

2) make only one declared pointer (maybe not p) left in G_1 using the inference rule for statement $q =$NULL; let the resulting shape graph be G_2.

3) get a shape graph G_3 which cannot be folded anymore using transformation rules of shape graphs on G_2.

4) if G_3 is a minimal shape sub-graph for data structures, and the shape is identical to the declared shape in the structure definition related to p, then G_1 is a shape graph for the corresponding declared shape.

For example, shape graph in Fig.16(a) is the original shape graphs. Suppose we need to infer the shape of the shape sub-graph including a declared pointer *hd*. After step 1, we can get shape graph in Fig.16(b). After step 2, we get shape graph in Fig.16(c). And after step 3 we can get shape graph in Fig.16(d). So we can infer from shape graph in Fig.16(d) that the shape is the singly-linked list.

If only pointers (not p) pointing to the same node as p can be deleted in step 2, the inference is called strict

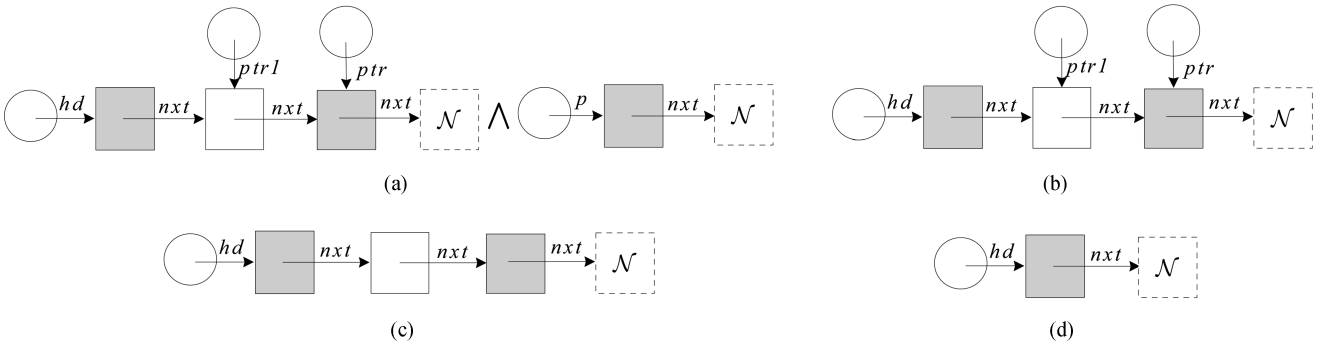

Fig.16. Example for shape inference.

shape inference. Otherwise, it is called relaxed shape inference. That is, strict shape inference does not allow other declared pointers to point to other nodes (excluding the node pointed by p) in the shape sub-graph.

Shape checking judges, whether the result of shape inference on the shape sub-graph containing a declared pointer p is identical to the corresponding declared shape (the shape of the data structure pointed by p). So shape inference could be done only at the program points where shape checking is needed.

At which program points to do shape checking on which pointers using relaxed or strict shape checking is decided by shape checking rules. It is not easy to make such a strategic decision since shapes may be destructed temporarily when inserting or deleting certain nodes. So it is allowed that pointers point to shapes not identical to the corresponding declared ones at some program points. Currently, checking points include the following three cases.

• *At the Function Call and Return Sites.* The shape of the actual pointer argument (by strict shape inference) must be identical to its declared shape. Under this constraint, we avoid difficulties in shape analysis of functions if other pointers are allowed to point to the inner nodes of shape graphs of the pre-condition.

At a return site, we do shape checking on two kinds of pointers, namely return value and actual arguments of pointer types. If they have the same type and point to a same shape sub-graph, they could possibly point to different nodes in it. So we do relaxed shape checking at return sites. If necessary, programmers can add a command requesting the shape system to do a strict shape checking just following the function call statement in the caller. Note that there is one special case, the shape system will not report shape errors if the node pointed by an actual argument is freed in a function (this actual argument turns into a dangling pointer).

• *At the Entry Points of Loops.* Each declared pointer variable should point to its declared shape at the entry point of a loop using relaxed shape inference, if it is modified in the loop body, but does not point to null/dangling nodes at the end of the loop body. We use relaxed shape inference here since there may be multiple declared pointers pointing to different nodes of a data structure when this data structure is manipulated in a loop (e.g., loop invariant shape graph in Fig.3(a)).

• *At the End Points of Loops.* In most cases shape checking can be performed at the end of the loop body so that checks will be done after each iteration. However, not all algorithms ensure that shapes are preserved in each iteration. A typical example is a program for doubly-linked list reversal (details see Example 1 in Section 5). In this example, a loop is used to reverse the two pointers on each node by iteration. At the end point of the loop body, both the traversed part and the untouched part look like doubly-linked lists. Even so, they do not form a doubly-linked list in a whole until pointers of all nodes are reversed (i.e., after the loop).

Shape checking at the end of loop bodies contributes earlier reports on errors of programs that construct shapes deviating from what the programmers expect. It also benefits the termination of inferences of loop invariant shape graphs. In essence, shape checking constrains the behaviors of loop bodies in manipulating data structures. For example, a data structure with two field pointers l and r is a doubly-linked list at a loop entry by relaxed shape inference. This list is traversed along the field pointer r and the field pointer l of each node is made to point to null (dangling) nodes. This program will be rejected if we set a shape checking point at the end of the loop body since it is not a doubly-linked list any more. Otherwise, the inference process of loop invariant shape graphs cannot terminate since there is no folding rule to be used.

Due to these considerations, we do more relaxed checks at the end of loop bodies. That is, the shape system reports no errors if folding operations can be done on the traversed nodes in the loops.

Although both shape system and type system analyze programs statically, they are different essentially. Firstly, common type systems make contextual constraints on program syntax, and they do not relate to language semantics; our shape system limits the shapes constructed by programs dynamically, which depends on language semantics. Secondly, the typing rules are structured. That is, the type of each construction of the language depends on the type of sub-construction(s). But our shape inference and checking judge what shape is constructed and whether the shape is legal or not based on the point-to relations between various nodes.

Our shape system differs from shape types[9] although the goal is the same. In shape types, shapes are defined in terms of context-free graph grammars. We directly use graphs rather than symbolic representations of graphs to define shapes. Moreover, they use abstract graph transformers to describe operations (e.g., insertion and deletion) on data structure and present an algorithm for the static shape checking of graph transformers. We do shape checking directly on PointerC programs manipulating data structures.

4.2 Benefits of Shape System for Verification

There are three following benefits using the shape system.

1) Programmers need not provide shape graphs in pre-/post-conditions. Shape constraints at function call

sites make the interface of functions simple and make the inference of shape graphs in the post-condition possible. For recursive functions dealing with pointers, inference of shape graphs in the post-condition, please refer to [7].

2) Programmers need not provide loop invariant shape graphs. We make the inference of loop invariant shape graphs (details please refer to [7]) simple and automatic by excluding programs which cannot pass our shape checking.

3) Shape system helps programmers find errors earlier. Besides the shape checking points mentioned above, we allow programmers to insert checks at desired program points. These inserted checks are done statically, so they do not affect executions at runtime.

5 Implementation and Prototype

Based on SGL, we have implemented a program verifier② for PointerC. It can verify programs dealing with shapes defined in Fig.4 (including some shapes not listed). We can also verify properties of data on the nodes. Programmers need not to provide shape related assertions in the pre-/post-conditions for functions and loop invariants. So graphical assertions bring no trouble to programmers.

5.1 Overview

Our prototype does analysis and verification in the following steps.

Preprocessing. The tool generates abstract syntax tree (AST) and does traditional static checks (e.g., type checking).

Shape Analysis Stage. Traversing the AST, shape graphs at each program point will be produced based on SGL. Shape inference and shape checking will be done at necessary program points. When a loop statement is encountered, the loop body will be traversed several times in order to infer corresponding loop invariant shape graphs. The process is similar to the inference of shape graphs in post-conditions of recursive functions.

Program Verification Stage. This stage can be divided into two sub-stages: VC generation and automatic theorem proving.

According to non-pointer pre-/post-conditions for functions and loop invariants provided by programmers, VCs will be generated using strongest post-condition calculation based on SGL. The basic form of VCs is $G \rhd Q \Rightarrow Q'$, where G is the shape graph at the program point where this VC is generated. So G is the context for the proof $Q \Rightarrow Q'$.

Our assertion language supports predicates, quantifiers and common features in first-order logic. Access paths could be used in symbolic assertions. Examples of assertions, please refer to Subsection 5.2. We add the following restrictions on assertions.

• Pointer access paths in Q must be a legal path in G.

• Any node asserted (touched) by a predicate should not be pointed by foreign pointers except that they are equal to any argument of the predicate. We use foreign pointers to name pointers, which will not appear in the graph by unfolding the predicate.

• Any node asserted (touched) by a quantifier should confirm to the above restriction of predicates.

Besides checking the legality of assertions provided by programmers in the VC generation stage, the prototype preserves the legality of assertions by unfolding predicates or quantifiers.

Before submit VCs to provers, some needed information (e.g., equalities between pointers) in shape graph G should be converted into symbolic assertion P. There should be no aliasing in P, Q and Q'. P may include other assertions. For example, if there is a condensation node in G, assertion a below this node which constrains the expression e should be included in P. If built-in functions (e.g., *length*) appear in Q', assertions on length of the corresponding list (acquiring from G) should be included in P. We directly submit $\neg\ (P \wedge Q \Rightarrow Q')$ to the SMT (satisfiability modulo theories) solver Z3[10]. If it is not satisfied, then $P \wedge Q \Rightarrow Q'$ is valid.

A shape graph is a shared data structure between analysis and verification. It can be regarded as graphical presentation of assertions on equalities of pointers. Using shape graphs as assertions makes work in both stages easy.

We have verified some complex programs manipulating mutable data structures using our prototype automatically. Table 1 shows the statistical data about test cases running on a Windows 7 PC with an Intel® Core i5-2400 3.1 GHz CPU and 4 G memory. Note that in test cases of *insert* and *delete* functions for binary trees (AA, AVL, BST, splay and treap), listed data actually includes all auxiliary functions (e.g., function *balance* and function *splay*) which are called during these two operations. During the analysis and verification of programs dealing with splay trees, protrudent memory and time are used because the loop invariant shape graphs are disjunctions of prominently more shape graphs than those of other programs.

②The verifier prototype system of PointerC (based on the shape graph logic and the shape system), http://kyhcs.ustcsz.edu.cn/SGL, Oct. 2013.

Table 1. Statistical Data About Some Test Cases

Data Structure	Function	Time to Build SGs (ms)	Memory for SGs (KB)	Iterations/Time (ms) to Infer Loop Invariants	No. Predicate	No. Lemma	No. VC
Sorted singly-linked list	*Merge*	263	81.6	4/85.5	3	3	5
Sorted doubly-linked list	*Reverse*	213	50.0	4.5/23.5	0	0	0
Sorted circular doubly-linked list	*Insert*	92	30.4	3/16	0	0	0
AVL tree	*Insert, delete*	2 708	1 118.7	No Loop/-	5	4	32
AA tree	*Insert*	153	116.6	No Loop/-	6	13	8
Binary search tree (BST)	*Insert, delete*	137	40.4	3/15	7	10	6
Treap	*Insert, delete*	276	95.3	No Loop/-	5	6	18
Splay tree	*Insert*	43 927	2 687.9	9/40 388	5	8	9

Note: SG is shorthand for shape graph.

5.2 Examples

Programmers need to provide pre-/post-conditions for functions and loop invariants about data when programming. Moreover, they could define some inductive predicates to describe properties of inductive data structures in favor of easy expression of specifications. Besides defining predicates, lemmas about inductive properties between predicates must be given as hints to automatic theorem provers, which are not so powerful to discharge these kinds of VC.

In this subsection, two examples are introduced to help understand this paper.

Example 1. Function invert shown in Fig.17 inverts an input sorted doubly-linked list and returns a pointer to the resulting list. As stated in the given post-condition, the resulting list is sorted, which is expressed using universal quantifiers. Variables used in assertions but not appearing in programs are logical variables, such as n in the pre-condition. Assertion *length*(*hd, nxt*)==n is equivalent to $\exists$ n:*Z.length*(*hd, nxt*)==n.

For loops dealing with doubly-linked lists, the loop invariant shape graphs are more complex than those of loops dealing with singly-linked lists (the invariant for the first loop shown in Fig.18). hd' is an node (corresponding to the node for actual arguments) added by our prototype according to the function call rule.

```
typedef struct listnode { int d;
    ListNode *:DLIST nxt; ListNode *:DLIST pre;
    } ListNode;
ListNode* invert(ListNode *hd) {
```

assertion $length(hd,nxt)==n \wedge$
$\forall i{:}1..n-1.hd(->nxt)^{i-1}->d \leqslant hd(->nxt)^{i}->d;$

```
    ListNode *tl; ListNode *ptr;
    ListNode *tmp; int k;
    tl=hd; k=0;
    if(tl!=NULL){
        while(tl->nxt!=NULL)
```

loop_invariant $\forall i{:}1..k.hd(->nxt)^{i-1}->d \leqslant hd(->nxt)^{i}->d$
$\wedge \forall i{:}k+1..n-1.tl(->nxt)^{i-1}->d \leqslant tl(->nxt)^{i}->d;$

```
            { tl=tl->nxt; k=k+1; }
            ptr=tl; k=0; tmp=NULL;
            while(ptr!=NULL)
```

loop_invariant $\forall i{:}1..n-k-1.hd(->nxt)^{i-1}->d$
$\leqslant hd(->nxt)^{i}->d \wedge \forall i{:}1..k.tl(->nxt)^{\ i-1}->d \leqslant tl(->nxt)^{i}->d;$

```
            {  tmp=ptr->nxt; ptr->nxt=ptr->pre;
               ptr->pre=tmp; ptr=ptr->nxt; k=k+1;
            }
        }
        return tl;
```

assertion $length(tl,\ nxt) == n \wedge$
$\forall i{:}1..n-1.tl(->nxt)^{i-1}->d \geqslant tl(->nxt)^{i}->d;$

```
    }
```

Fig.17. Example 1: inverting a sorted doubly-linked list.

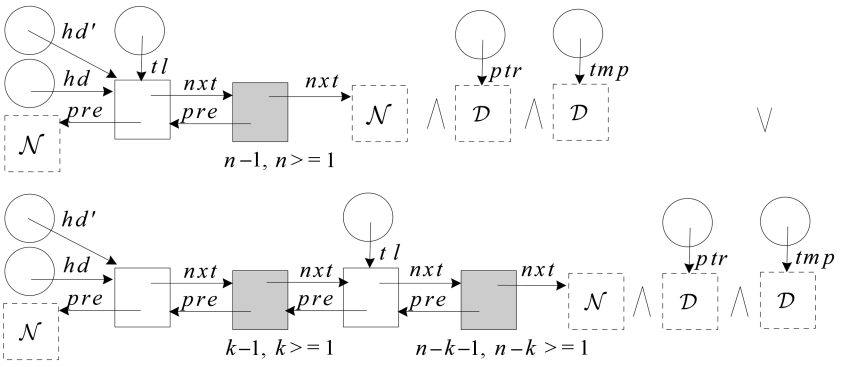

Fig.18. Loop invariant shape graph of the first loop in Example 1.

The doubly-linked list is temporarily not in shape in iterations of the second loop. Furthermore, pointers *hd* and *tl* point to two different nodes (the head node and tail node). These make the loop invariant shape graphs complicated (eight cases not listed here due to limited space).

In various program verifiers[11-15], specifications must be provided in order to verify shape and data properties of programs manipulating multiple complex data structures.

Example 2. In this example, we show a function inserting a node into a binary search tree and returning a pointer to the resulting sorted tree. Code and assertions, please refer to Fig.19. User-defined predicates and lemmas in Fig.20 are used to describe specifications on orderedness where p is a pointer variable and x, y are integer variables.

```
typedef struct node
{int data; Node *:TREE l; Node *:TREE r;} Node;
Node* insert(Node *p, int data) {
  assertion order(p)∧y>data∧gt(y,p)∧
  z<data∧lt(z,p);
    if(p==NULL) {
        p=malloc(Node);
        p->l=NULL;p->r=NULL; p->data=data;
    } else if(p->data>data) {
        p->l=insert(p->l, data);
    } else if(p->data<data){
        p->r=insert(p->r, data);
    }
    return p;
  assertion order(p)∧gt(y,p)∧lt(z,p); }
```

Fig.19. Inserting a node into a sorted binary search tree.

$order(p) \triangleq$ p==NULL$\vee order(p\text{->}l) \wedge order(p\text{->}r) \wedge gt(p\text{->}data,\ p\text{->}l) \wedge lt(p\text{->}data,\ p\text{->}r)$
$gt(x,\ p) \triangleq$ p==NULL$\vee\ x > p\text{->}data \wedge gt(x,\ p\text{->}l) \wedge gt(x, p\text{->}r)$
$lt(x,\ p) \triangleq$ p==NULL$\vee\ x < p\text{->}data \wedge lt(x,\ p\text{->}l) \wedge lt(x, p\text{->}r)$
$x < y \wedge lt(y,p) \Leftrightarrow lt(x,\ p)$
$x > y \wedge gt(y,p) \Leftrightarrow gt(x,\ p)$

Fig.20. User-defined predicates and lemmas for Example 2.

This is a recursive function. We can obtain the shape graph, in which the return value and actual argument point to a same tree predicate node, as the post-condition in the analysis stage.

6 Related Work

6.1 Comparison Between SGL and Separation Logic

Our SGL and separation logic[16] are both extensions to Hoare logic. They can be used to verify imperative programs manipulating mutable data structures. Separation logic can be used in the verification of arbitrary pointer programs, while SGL is designed to verify programs manipulating mutable data structures dedicatedly.

Shape graphs contain more information which can be used in program verification than assertions written in separation logic. We can make the following conclusion by comparing frame rules in separation logic and SGL.

$$\frac{\{P\}C\{Q\}}{\{P*R\}C\{Q*R\}} \quad \frac{\{W_1\}C\{W_2\}}{\{X[W_1]\}C\{X[W_2]\}}.$$

To our knowledge, the main difference between SGL and separation logic according to how to use the frame rule is whether the context should not be concerned completely or not. In separation logic, the context (R) is not concerned. That is, separation logic does not concern whether there is any pointer (i.e., objects asserted by R) pointing to objects asserted by P and Q. In SGL, when statement C changes the program state, the changes are reflected by changing point-to relations of edges and nodes in the window W, not the context $X[]$. Although $X[]$ is not changed, edges sourced from context to nodes in the window reflect our concern on the context.

In one word, separation logic emphasizes separation. It results in that when operating on one heap block, it is hard to concern the relation between the current heap block with other heap blocks. As a contrast, shape graphs give attention to both separation and integrity. On the one hand, heap blocks described by a context are separated with those described by the corresponding window. On the other hand, edges across windows are connections between contexts and windows. For example, we can guarantee that programs which can pass analysis and verification using SGL will be free of memory leaks. In separation logic, it is not intuitionistic to judge how many pointers point to a same heap block. This kind of information can be collected from assertions of heap blocks, but it is hard to give such a rule in separation logic. As a result, the rule for memory leaks is hard to figure out in separation logic. To solve this problem, O'Hearn *et al.* proposed the concept — precise assertion[17]. But it is defined based on the abstract machine other than the syntax of assertions.

Separation logic uses separation conjunction ($*$) and special designs in inference rules to avoid the effects on reasoning of aliasing. Alias can only appear in the assertion about one heap block, by using separated expression of assertions on various heap blocks and forbidding the use of access paths concerning multiple heap blocks in syntax (e.g., *p->next->next* in C). For example, if

$p==q$ (p, q are pointers) exists in the pure assertion, $p \mapsto 5$ and $q \mapsto 5$ can appear in the assertion about the heap block pointed by them ($p \mapsto$ and $q \mapsto$, i.e. $*p$ and $*q$ in the syntax of C, are aliases). However, when the heap cell pointed by them is accessed by some statement, there is no inference rule to be used directly to reason about the statement when these two assertions appear at the same time.

Our strategy in SGL is to guarantee no alias (by using information provided by constructed shape graphs as reasoning context) in the assertions and statements when using assignment axiom of Hoare logic. That is, we guarantee that corresponding rules of Hoare logic are used soundly. Similarly, when generating VCs, we guarantee that there is no alias in VCs. So our VCs use only first-order connectives other than special ones like separation conjunction. The proving of VCs will benefit from such considerations. Currently, in provers for separation logic[6,18], their inference rules are obtained from an existing proof system for separation logic with list segments[19].

6.2 Comparison with Other Shape Analysis

Shape analysis attempts to discover the shapes of data structures, which can be regarded as invariants in pointer programs. 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[20-21]. The approximation of memory states leads to loss of information about the shapes of recursive data structures. Precision can be improved by using associate grammars, which finitely summarize run-time heap structures, with the summary nodes of the shape graphs[22]. With the help of hints on shape declarations and under constraints on the operations of pointers, our shape analysis is a kind of precise pointer analysis. In our proposed shape graphs, by using the condensation node and information about the number of structure nodes represented by it, we can get accurate finite graphs of the potentially unbounded data structures. This kind of condensation node can be regarded as the graphic counterpart of segments[10], but the node number can be included to guarantee that no information is lost.

Chang *et al.* proposed a shape analysis[23-24] based on a shape graph representation which abstracts memory cells by edges. They described memory states in a manner based largely on separation logic. For comparison, their complete checker edge corresponds to our predicate node, and their partial checker edge is similar to our condensation node. Moreover, they only treated shape graphs as a representation easy to understand symbolic assertions. They still used symbolic assertions in their analysis. We use shape graphs directly in our assertion calculus and program reasoning. In addition, the shape analysis in [23-24] is guided by invariant checking code supplied by programmers, i.e., the definitions of shapes can come from checking code, which is translated to inductively defined predicates. It seems to be more flexible than our method, which provides only several fixed shapes. However, it is hard to define the inductive invariant checker for circular doubly-linked lists. Analysis on programs using inductive properties of data structure is difficult, such as programs deleting a node in an ordered binary search tree using the in-order successor (the leftmost child of its right sub-tree) or in-order predecessor (the rightmost child of its left subtree) of one node to replace itself. Moreover, there is no test case for circular lists or delete function of binary search trees in their analysis statistics table. Laviron *et al.*, using their shape graph, created an analysis that is capable of reasoning about low-level C features, such as unions, while not unnecessarily complicating the analysis of higher-level, Java-like code[25]. However, we have not tried our shape graph in such low-level C features.

6.3 Comparison with Other Pointer Program Verification

There are also some studies doing program verification on pointer programs during program analysis. Lev-Ami *et al.* proposed a method to verify programs dealing with lists via static analysis[26]. For example, they verified various sorting programs based on lists. However, it is hard to verify more properties such as list length in our first example. To verify such properties, core predicates, instrumentation predicates, and predicate-update formulae for them must be provided in their system. The research of Podelski and Wies showed that the techniques developed in CEGAR (counterexample-guided abstraction refinement) scheme[27] in software verification and the focus operator[21] in shape analysis can be fruitfully integrated to enhance one another for the inference of quantified invariants[28]. They applied the tool Bohne[28] to verify operations on a diverse set of data structure implementations, checking a variety of properties. Their experiments covered data structures such as (sorted) singly-linked lists, doubly-linked lists, two-level skip lists, trees, and trees with parent pointers, but not including circular singly/doubly-linked lists. We use two-staged method: program analysis and program verification, using shape graphs to express the most important property — point-to and equality relations among pointers. It makes the verification phase easy to take

advantage of the information provided by the analysis stage, and it is convenient to express the properties to verify. We use a shape system to standardize the behaviors of programs. It is useful in helping to infer the loop invariant shape graphs, and it frees programmers from providing pre-/post-conditions for functions and loop invariants.

Madhusudan *et al.* developed a new recursive extension of first-order logic[29]. The recursive logic over trees, DRYAD, is essentially a quantifier-free first-order logic over heaps augmented with recursive definitions of various types defined for location that have a tree under them. Based on this methodology, a sound and terminating procedure can prove a wide variety of algorithms on tree-structures written in an imperative language fully functionally correct. In this paper, footprint is made of a symbolic heap and DRYAD formulae. They are similar to our shape graphs and properties described using symbolic assertions respectively. There is no concept like our condensation node for loop programs in their symbolic heap. As a result, they only support recursive programs. Furthermore, no examples dealing with circular lists were presented.

7 Conclusions

In program verification, the highest expectation of shape analysis is to make users avoid having to write loop invariants or even pre/post specifications for procedures: these are inferred during analysis[30]. This paper proposed methods to ease burdens of programmers and provers when doing verification on pointer programs by using shape declarations, program analysis and verification based on SGL, and using a shape system to constrain program behaviors.

Our future work is to break the limitation of our method that only several shapes can be declared by programmers. We plan to classify the shapes, and use different methods to deal with different kinds of shapes. The shapes introduced in this paper are called basic shapes. Currently, we are trying to extend our work by supporting additional pointers on data structures except necessary pointers to maintain the basic shapes.

1) Additional pointers point to other shapes. An additional pointer is a pointer pointing to another independent nested shape. For example, on each node of a doubly-linked list, there is an additional pointer pointing to an independent singly-linked list. It is not hard to extend our prototype to support shape nesting.

2) Additional pointers point to nodes on the same shape. We solve these problems by dividing them into two cases.

• Targets of additional pointers are clear. Examples of such a case are binary trees with a pointer on each node pointing to its parent node, left-child right-sibling trees with two kinds of backward links, skip lists, queues, etc. We will provide certain description method to allow programmers to describe about these additional pointers using equations between pointers. We consider generating, from such a description, the corresponding code to check whether these pointers confirm to the description or not. At program points of shape checking, these codes are executed to do checking on additional pointers after original basic shape checking.

• Targets of additional pointers are not clear. One example of this case is a doubly-linked list as a request queue, in which some nodes are connected using additional pointers to form a singly-linked list as a ready queue. We consider using simple and feasible constraints in programming, in order to make sure that we can statically determine whether some nodes form a singly-linked list using additional pointers and additional pointers of other nodes are equal to null or not.

Acknowledgements We thank Master students Yang-Hui Song, Gang Liu, Zhi-Tian Zhang, Ya-Hui Han, Jian-Chao Meng for their hard work in the prototype implementation. We also thank the anonymous reviewers for their constructive suggestions and great enthusiasm.

References

[1] Paulson L C. Isabelle: A Generic Theorem Prover. Springer-Verlag Berlin, 1994.

[2] Nanevski A, Morrisett G, Shinnar A, Govereau P, Birkedal L. Ynot: Dependent types for imperative programs. In *Proc. the 13th ACM SIGPLAN Int. Conf. Functional Programming*, September 2008, pp.229-240.

[3] Barnett M, M. Leino K R, Schulte W. The spec# programming system: An overview. In *Proc. Int. Conf. Construction and Analysis of Safe, Secure and Interoperable Smart Devices* (*CASSIS*), March 2004, pp.49-69.

[4] Flanagan C, Rustan K, Lillibridge M *et al.* Extended static checking for Java. In *Proc. the ACM SIGPLAN 2002 Conf. Programming Language Design and Implementation*, June 2002, pp.234-245.

[5] Berdine J, Calcagno C, O'Hearn P W. Smallfoot: Modular automatic assertion checking with separation logic. In *Proc. the 4th Int. Symp. Formal Methods for Components and Objects*, Nov. 2005, pp.115-137.

[6] Distefano D, Parkinson M. jStar: Towards practical verification for Java. In *Proc. the 23rd ACM SIGPLAN Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2008, pp.213-226.

[7] Li Z P, Zhang Y, Chen Y Y, Meng J C. Automated theorem proving for theory of shape graphs and its application in program verification. Technical Report, USTC-Yale Joint Research Center for High-Confidence Software, http://kyhcs.ustcsz.edu.cn/SGL, Nov. 2012.

[8] Chen Y Y, Li Z P, Wang Z F, Hua B J. A pointer logic for pointer program verification. *Chinese Journal of Software*, 2010, 21(3): 124-137. (Chinese, English Version at http://kyhcs.ustcsz.edu.cn/∼zpli/pl.pdf)

[9] Fradet P, Metayer D L. Shape types. In *Proc. the ACM*

SIGPLAN-SIGACT Symp. Principles of Programming Languages, Jan. 1997, pp.27-39.
[10] de Moura L D, Bjørner N. Z3: An efficient SMT solver. In *Proc. the 14th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, Mar. 29-Apr. 6, 2008, pp.337-340.
[11] Lahiri S, Qadeer S. Verifying properties of well-founded linked lists. In *Proc. the 33rd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, Jan. 2006, pp.115-126.
[12] Chin W N, David C, Nguyen H H, Qin S. Automated verification of shape, size and bag properties. In *Proc. the 12th Int. Conf. Engineering of Complex Computer Systems*, July 2007, pp.307-320.
[13] Lahiri S, Qadeer S. Back to the future: Revisiting precise program verification using SMT solvers. In *Proc. the 35th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, Jan. 2008, pp.171-182.
[14] Bouajjani A, Dragoi C, Enea C, Sighireanu M. A logic-based framework for reasoning about composite data structures. In *Proc. the 20th Int. Conf. Concurrency Theory*, Sept. 2009, pp.178-195.
[15] Madhusudan P, Parlato G, Qiu X K. Decidable logics combining heap structures and data. In *Proc. the 38th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, Jan. 2011, pp.611-622.
[16] Reynolds J C. Separation logic: A logic for shared mutable data structures. In *Proc. the 17th IEEE Symp. Logic in Computer Science*, July 2002, pp.55-74.
[17] O'Hearn P W, Yang H, Reynolds J C. Separation and information hiding. In *Proc. the 31st ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, Jan. 2004, pp.268-280.
[18] Pérez J A N, Rybalchenko A. Separation logic + superposition calculus = Heap theorem prover. In *Proc. ACM SIGPLAn Conf. Programming Language Design and Implementation*, Jun. 2011, pp.556-566.
[19] Berdine J, Calcagno C, O'Hearn P W. A decidable fragment of separation logic. In *Proc. the 24th Int. Conf. Foundations of Software Technology and Theoretical Computer Science*, Dec. 2004, pp.97-109.
[20] Sagiv M, Reps T, Wilhelm R. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 1998, 20(1): 1-50.
[21] Sagiv M, Reps T, Wilhelm R. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 2002, 24(3): 217-298.
[22] Lee O, Yang H, Yi K. Automatic verification of pointer programs using grammar-based shape analysis. In *Proc. the 19th European Conf. Programming Languages and Systems*, Apr. 2005, pp.124-140.
[23] Chang B E, Rival X, Necula G C. Shape analysis with structural invariant checkers. In *Proc. the 14th Int. Static Analysis Symp.*, Aug. 2007, pp.384-401.
[24] Chang B E, Rival X. Relational inductive shape analysis. In *Proc. the 35th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, Jan. 2008, pp.247-260.
[25] Laviron V, Chang B E, Rival X. Separating shape graphs. In *Proc. the 19th European Conf. Programming Languages and Systems*, Mar. 2010, pp.387-406.
[26] Lev-Ami T, Reps T, Sagiv M, Wilhelm R. Putting static analysis to work for verification: A case study. In *Proc. the Int. Symp. Software Testing and Analysis*, Aug. 2000, pp.26-38.
[27] Clarke E M, Grumberg O, Jha S, Lu Y, Veith H. Counterexample-guided abstraction refinement. In *Proc. the 12th Int. Conf. Computer Aided Verification*, July 2000, pp.154-169.
[28] Podelski A, Wies T. Counterexample-guided focus. In *Proc. the 27th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, Jan. 2010, pp.249-260.
[29] Madhusudan P, Qiu X K, Stefanescu A. Recursive proofs for inductive tree data-structures. In *Proc. the 39th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, Jan. 2012, pp.123-135.
[30] Calcagno C, Distefano D, O'Hearn P W, Yang H. Compositional shape analysis by means of bi-abduction. In *Proc. the 36th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, Jan. 2009, pp.289-300.

Zhao-Peng Li is a postdoctoral researcher in School of Computer Science and Technology, University of Science and Technology of China, Hefei. He received his Ph.D. degree in computer science from University of Science and Technology of China in 2008. His research interests include program verification, certifying compiler and theorem proving. He is a member of China Computer Federation.

Yu Zhang received the M.E. degree in computer science from Hefei University of Technology in 1996, and the Ph.D. degree in computer science from University of Science and Technology of China (USTC) in 2004. She is an associate professor in School of Computer Science and Technology at USTC. Her research spans programming languages, runtime systems, and operating systems, with a particular focus on systems that transparently improve reliability, security, and performance.

Yi-Yun Chen is a professor in School of Computer Science and Technology, University of Science and Technology of China. He received his M. S. degree in computer science from East-China Institute of Computer Technology in 1982. His research interests include application of logic (including formal semantics and type theory), techniques for designing and implementing programming languages, program verification, and software safety and security. He is a member of China Computer Federation.