Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 207 (2008) 137–151
www.elsevier.com/locate/entcs

Shape Analysis by Refining on Abstract Evaluation Path [1]

Xiaodong Ma Ji Wang Wei Dong

{xd.ma, wj}@nudt.edu.cn dong.wei@263.net
National Laboratory for Parallel and Distributed Processing, P. R. China

Abstract

This paper presents a novel method for shape analysis, which can deal with complex expressions in C language. It supports taking addresses of fields and stack variables. The concept of abstract evaluation path (AEP) is proposed, which is generated from the expression in the language. AEP is used to refine the abstract shape graph (ASG) to get a set of more precise ASGs, on which the semantics of the statement can be defined easily. The results can be used to determine "shape invariants" and detect memory leak conservatively. A prototype has been implemented and the results of the experiment are shown.

Keywords: shape analysis, memory leak, AEP

1 Introduction

Shape analysis algorithms statically analyze a program to determine information about the heap-allocated data structures that the program manipulates. It is an important methods used to understand or verify programs [7,8,9,10,11,12,13,14]. Abstraction is needed if we want to do shape analysis efficiently. There are many abstraction methods, such as k-limited heap abstraction [8], shape graph and reference counts [11], 3-valued logic abstraction [12].

Shape analysis can also be used to detect Memory leak. Memory leak is one of the most common errors in programs written in languages with pointers. Occurring in large, memory intensive, long-time running programs, it can exhaust the available memory and cause the programs to fail. Many models and methods are proposed to detect memory errors. In [1,2,3,4,5] an ownership model is used to detect memory errors. Hackett et al [6] uses region based method to detect memory errors. Escape

[1] This work is supported by National Natural Science Foundation of China(60673118 and 90612009), National 863 project of China(2006AA01Z429), National Basic Research Program of China(973) under Grant 2005CB321802 and Program for New Century Excellent Talents in University under grant No. NCET-04-0996

doi:10.1016/j.entcs.2008.03.090

analysis is used in [16] to detect memory leaks. Dor et al [15] defines cleanness conditions to check memory leaks by pointer shape analysis.

Complex expressions–where all the operators, including &, * and →, may occur–make shape analysis more difficult. Based on the work of [12], we present a shape analysis algorithm which can deal with many kinds of complex expressions. We have implemented the algorithm and used it to detect memory leak errors in some typical but complex, pointer intensive programs.

The difficulty in dealing with complex expressions lies in the fact that the value of a complex expression cannot be easily defined in the abstract shape graph (ASG). We can get the value of a basic type datum, such as an integer, by accessing the corresponding memory. However, if we want to know the value of a pointer expression, we must access several memories along a path. In this paper, we proposed abstract evaluation path (AEP) to deal with this problem. AEP is an abstract access path of the expression. It denotes how the l-(r-)value of an expression can be defined. In other words, it shows all the possible access paths along which the value of an expression can be defined in an ASG. we call these paths **evaluation paths**. AEP can be used to refine the ASG to make the defining of the value of an expression easily.

Using AEP, we can deal with most expressions in programming languages like C. We take a more aggressive materialization strategy in doing shape analysis. By supporting taking addresses of fields and stack variables, our method allows pointers into the middle of structures, not just the beginning of structures. Our method is also conservative, which means it may generate more shape graphs than the program really generates.

Outline. The rest of the paper is organized as follows. Section 2 gives the definition of concrete shape graph(CSG). Section 3 defines AEP and shows our algorithms of refining ASGs based on AEP. In Section 4, we give the abstract semantics of the statements on ASGs. Section 5 shows the results of our experiments. Section 6 lists some related work and gives the conclusion of our work.

2 Concrete Shape Graph

A shape graph is a directed graph which consists of a set of nodes and edges. It is used to represent memory structures and the connectivity between them. It is different from ordinary directed graph since it may have many types of nodes, which are used to represent different data structures.

We classify the data types in C into two main kinds: the basic type and the composite type. The basic type includes all the basic data types in C programs, e.g. integer, float, pointer; the composite type consists of structures which may have multiple fields. A node in a CSG may have several cells, each cell's name is in the form of *NName.FName*, where *NName* is the name of the node and *FName* is the name of the field. The node of a basic type datum has just one cell, whose name is *NName.bas*, where *bas* denotes the basic and only field of the node. The node of a composite type datum has several cells, one for each field, plus an additional one

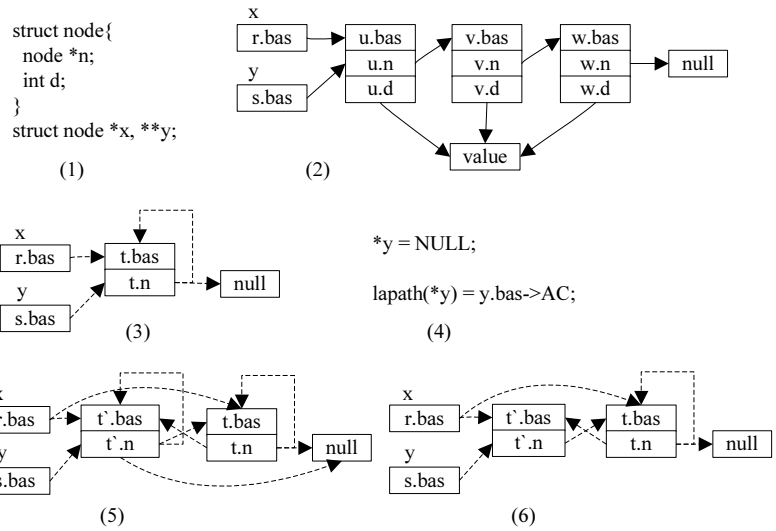

Fig. 1. (1): data structure definition. (2): a concrete shape graph. (3): the abstract shape graph of (2) where dashed lines denote the value of the corresponding *edge* predicate is 1/2. (4): assignment statement and the *lapath* of it's left hand side. (5): the shape graph of (3) after `split` of node t on $lapath(*y)$. (6): the shape graph of (5) after `cut` operation.

whose name is *NName.bas*, which is not a field of the datum but denotes the basic address of the structure. We call the node memory location and the field memory cell. All the memory cells within the same node have the same *NName*. A cell expression is an expression whose value is a memory cell. Figure 1.(2) is a simple CSG whose data structures are defined in Fig.1.(1). x is a pointer of structure node and y points to pointer. In Fig.1.(2) y points into a field of structure node. Others subgraphs of Fig.1 will be explained later.

A CSG consists of a set of nodes and a set of predicates. The predicates used in this paper are listed in the following.

$x_i(v)$: x_i is the name of some variable, $x_i(v) = 1$ if and only if node v denotes the memory location which stores the value of x_i, i.e. v is the l-value of x_i.

$r_{cell}(v)$: *cell* is the basic cell of a pointer variable or the pointer field cell of a structure variable. It denotes the reachability of memory location v from *cell*. If $r_{cell}(v) = 1$, then there is a path from *cell* to any memory cell of node v, which may have multiple memory cells.

$type_i(v)$: we use an integer to denote a structure type. Different memory locations may have different structures. $type_i(v) = 1$ if and only if memory location v belongs to type i. All the basic data types are classified into one type and different structures are classified into different types.

$shared(v)$: it denotes whether the basic cell of the memory location v is the target of two or more edges.

$unique(v)$: it is only used in the definition of ASG. In order to make the nodes in the shape graph finite, some node in the ASG may denote a set of concrete nodes. For concrete shape graphs, for all addresses, v, $unique(v) = 1$.

$edge(cell_0, cell_1)$: it is true if and only if there is an edge from $cell_0$ to $cell_1$.

In Fig.1.(2), if we encode `int` type as 0 and `node` type as 1, then we can get: $y(s) = 1$, $r_{s.bas}(u) = 1$, $type_1(u) = 1$, $shared(u) = 0$, $unique(u) = 1$ and $edge(u.n, v.bas) = 1$.

Definition 2.1 [CSG] A CSG is a four tuple: $G_C = (V_C, edge, Type, Map)$ where:

(i) V_C is the set of nodes in the graph and there are two special nodes: *null* and *value*. *null* is the null memory in C language and *value* denotes all the possible values of the non-pointer basic type data in C language.

(ii) For every $cell_0$, there is at most one memory $cell_1$ which satisfies $edge(cell_0, cell_1) = 1$.

(iii) $Type$ is a set of predicates $type_i$. For each node v, there is one and only one predicate $type_i$ which satisfies $type_i(v) = 1$.

(iv) Map consists of all the predicates x_i. For each predicate x_i, there is one and only one node v which satisfies $x_i(v) = 1$.

Using the node *value*, all the variables can be viewed as pointers. Figure 1.(2) is an example.

CSG can faithfully represent the memory state of C programs without unions. Every expression has l-value and right value except that it is null dereferenced or uninitialized. The semantics on CSGs is also obvious.

3 Refining ASG Based On AEP

Abstract interpretation [20] is a good method to deal with infinite systems; it can also be used to generate finite number of ASGs from any possible infinite number of CSGs. There are two reasons which cause the program states to be infinite: first, some variables may have infinite values, e.g. integer variables, float variables; second, the memories dynamically allocated may be infinite. Since we have introduced the abstract memory location *value*, the infinite values are reduced to one abstract *value*. A set of predicates are used to classify the possible infinite memory locations into different classes. All the memory locations in the same class should have the same value for any predicate in the set. Since the number of predicates is finite, the number of classes is also finite.

3.1 Abstract Shape Graph

The predicates used to classify the memory locations, called core predicates, are among the ones listed in Section 2. The predicates used in ASG are 3-valued [19]. We also define a partial order $\sqsubseteq$ on truth values to reflect information content: $l_0 \sqsubseteq l_1$ denotes that l_0 has more definite information than l_1. $l_0 \sqsubseteq l_1$ if $l_0 = l_1$ or $l_1 = 1/2$. Symbol $\sqcup$ denotes the least upper bound operation with respect to $\sqsubseteq$, such as $1 \sqcup 0 = 1/2$, $0 \sqcup 1/2 = 1/2$.

Definition 3.1 [ASG] An ASG is a triple: $G_A = (V_A, pcore, pinst)$ where:

(i) V_A is a set of abstract nodes, including *null* and *value*.

(ii) $pcore = \{x_i,\ r_{cell},\ type_i\}$, is a set of unary predicates. There are some constraints: first, the truth value of predicate x_i must be definite and there is one and only one memory location v which satisfies $x_i(v) = 1$; second, for any node v, the truth value of $type_i(v)$ is definite. This is the set of core predicates.

(iii) $pinst = \{unique, shared, edge\}$. $unique(v)$ can be 1 or 1/2, but never 0.

Figure 1.(3) is an ASG of a linked list. The set of *pcore* predicates are $\{x, y, r_{r.bas}, r_{s.bas}, type_0, type_1\}$

An ASG $G_A = (V_A, pcore, pinst)$ is minimal if and only if for any two nodes in V_A, there is a predicate in *pcore* which can distinguish them. It can be described formally as: $\forall\ v_1, v_2 \in V_A\ \exists p \in pcore : (p(v_1) \neq p(v_2))$. Since there are finite predicates and truth values, all the possible minimal ASGs of a given program are also finite. It can be easily seen that the predicates r_{cell}, *shared* can also be evaluated in a CSG. For any node v in a CSG, $unique(v) = 1$.

Definition 3.2 A CSG $G_C = (V_C, edge_C, Type_C, Map_C)$ is embedded in an ASG $G_A = (V_A, pcore, pinst)$ if there is surjective function $f : V_C \to V_A$, and the following conditions are satisfied, where *ACells(G)* denotes all the memory cells in graph G:

(i) For any unary predicate p except *unique* predicate, $\forall v \in V_C : p(v) \sqsubseteq p(f(v))$.

(ii) If formula $\exists v_0, v_1 \in V_C : f(v_0) = f(v_1) = v' \wedge v_0 \neq v_1$ evaluates to false, then $unique(v') = 1$, else $unique(v') = 1/2$.

(iii) There is a function $g : ACells(G_C) \to ACells(G_A)$ derived from f, which is: $g(v.f_x) = f(v).f_x$, where $v \in V_C$ and f_x denotes any field of v. The only binary predicate edge satisfies: $edge(c_1, c_2) \sqsubseteq edge(g(c_1), g(c_2))$, where $c_1, c_2 \in ACells(G_C)$.

The value of expression $v_0 = v_1$ is also 3-valued. It evaluates to true if and only if v_0 and v_1 are the same unique node. In other words, it is equivalent to $v_0 = v_1 \wedge unique(v_0) \wedge unique(v_1)$.

A minimal ASG G_A can be generated from any shape G (whether it is a CSG or an ASG) through the following steps.

(i) The core predicates are used to classify the nodes in G. The nodes which cannot be distinguished by the predicates in *pcore* are classified into one class.

(ii) For each class, there is a corresponding node in G_A and G_A contains no more nodes. We use a function $f : V \to V_A$ to define the corresponding relation between nodes. As described in Definition 3, a function g can also be derived. The truth value of any unary predicate p except *unique* in G_A is defined as: $p(v) = \sqcup\{p(u_i) | f(u_i) = v\}$; $unique(v) = 1$ only if formula $\exists v_0, v_1 : f(v_0) = f(v_1) = v \wedge v_0 \neq v_1$ evaluates to false, else $unique(v) = 1/2$. $edge(c'_0, c'_1) = \sqcup\{edge(c_0, c_1) | g(c_i) = c'_i, i = 0, 1\}$.

Figure 1.(3) is a minimal ASG and Fig.1.(2) can be embeded into it.

3.2 Algorithms of Refinement

In order to make the shape graphs finite, abstraction must be used. However, some constraints in CSGs are not applicable to ASGs, e.g. one memory cell in an ASG may have several output edges (the truth value of the corresponding predicate *edge* is 1/2), which makes it difficult to define the semantics of the statements on ASGs. AEP is introduced to solve this problem. As described in Section 1, AEP denotes all the evaluation paths in an ASG. Based on AEP, we can refine an ASG into a set of ASGs in which an expression's l-(r-)value can be defined as in CSGs.

An expression may have several l-(r-)values in an ASG which makes it difficult to define the semantics of the statements. Our method is called *refinement by AEP*. Starting from a memory cell of some variable, we can go along some evaluation paths in the ASG to get an expression's l-(r-)value. We refine the ASG at each step along the path and the l-(r-)value of an expression is a unique node in every ASG in the result ASG set after refining. No matter how complex an expression is, an AEP can be generated from it and the semantics can be defined easily on the result ASGs after refining.

An AEP is an evaluation path where every node is an abstract node named "AL" instead of a concrete node name except the head of the path where the node name is the name of a variable, or in some cases, there is no occurrence of nodes at all, just a symbol "AC" to denote an abstract cell.

Definition 3.3 [AEP]

The left AEP is defined in the following.

(1) $lapath(const) = \perp$

(2) $lapath(null) = \perp$

(3) $lapath(x) = x.bas$ $\quad x$ is a variable

(4) $lapath(\&e) = \perp$

(5) $lapath(*e) = \pi \rightarrow AC$ $\quad$ iff $lapath(e) = \pi$

(6) $lapath(e{\rightarrow}f) = \pi \rightarrow (AL.bas, AL.f)$ $\quad$ iff $lapath(e) = \pi$

(7) $lapath(e.f) = \begin{cases} (x.bas, x.f) & \text{if } lapath(e) = x.bas \\ \pi \rightarrow (AL.bas, AL.f) & \text{else, where } lapath(e) = \pi \rightarrow AC \end{cases}$

The right AEP:

(1) $rapath(const) = value$

(2) $rapath(null) = null$

(3) $rapath(x) = x.bas \rightarrow AC$

(4) $rapath(\&e) = lapath(e)$

(5) $rapath(*e) = \pi \rightarrow AC$ $\quad$ iff $lapath(*e) = \pi$

(6) $rapath(e{\rightarrow}f) = \pi \rightarrow AC$ iff $lapath(e{\rightarrow}f) = \pi$

(7) $rapath(e.f) = \pi \rightarrow AC$ iff $lapath(e.f) = \pi$

An AEP is independent of any graph. For example,

$$lapath((*x) \rightarrow n)) = x.bas \rightarrow AC \rightarrow (AL.bas, AL.n)$$

$$rapath((*x) \rightarrow n)) = x.bas \rightarrow AC \rightarrow (AL.bas, AL.n) \rightarrow AC$$

We define the semantics of statements on ASGs by transfer functions whose input is an ASG and output is a set of ASGs. In the definition of the statement's semantics on CSGs, we can evaluate the l-(r-)value of an expression and change the current CSG to get the result CSG. This method cannot be used directly on ASGs, because a cell expression may have more than one l-(r-)value along different evaluation path. How can we deal with this problem? First, we compute the left AEP of the left expression and use it to refine the ASG and get a set of ASGs, then we use the right AEP to refine the result set of ASGs. During the refinement of an ASG, we may change the value of some $edge(cell_x, cell_y)$ from 1/2 to 1 on the path or create some nodes to make the l-(r-)value of a left(right) expression the cell of a unique node (whose truth value of predicate *unique* is 1) and all the nodes along the path unique nodes. After the two stages of the refinement, we can get a set of more precise ASGs where them both the left and the right expressions of the statement can be evaluated to unique cells. Thus, we can define the semantics of the statements on ASGs in the way as on CSGs.

The algorithm is described in Algorithm 1. There are some new functions. Every graph has a *current* attribute which identifies the current memory cell. Function *node* maps the memory cell to its node. `split` is used to generate a new node from a non-unique node and `cut` is used to delete the inconsistent edges and graphs.

Algorithm 1

```
01. function refine(G_A, π)
02. input G_A: An ASG; π: An AEP
03. output AnswerSet: a set of refined shape graphs
04. begin
05.  WorkSet = φ
06.  AnswerSet = {G_A}
07.  while π ≠ φ do
08.    let π = w → π';   π = π'
09.    WorkSet = AnswerSet;    AnswerSet = φ
10.    while WorkSet ≠ φ do
11.      let WorkSet = G_1 ∪ WorkSet'
12.      switch w
13.      case ⊥:
14.        AnswerSet = φ
15.        return AnswerSet
16.      case null, value:
17.        current(G_1) = w
18.        AnswerSet = {G_1}
19.        return AnswerSet
20.      case x.bas, (x.bas, x.f): /*where x represents a variable*/
21.        current(G_1) = v.bas where x(v) = 1
22.        insert G_1 into AnswerSet
23.      case AC, (AL.bas, AL.f):
24.        let cell_0 = current(G_1)
25.        if ∃cell_1 : edge(G_1)(cell_0, cell_1) > 0
26.          current(G_1) = cell_1
27.          if unique(node(cell_1)) == 1
28.            edge(G_1)(cell_0, cell_1) = 1
29.            AnswerSet = AnswerSet∪cut(G_1)
30.          elseif unique(node(cell_1)) == 1/2
```

31. $G'_1 = G_1[unique(node(cell_1))/1]$
32. $edge(G'_1)(cell_0, cell_1) = 1$
33. $AnswerSet = AnswerSet \cup \texttt{cut}(G'_1)$
34. $AnswerSet = AnswerSet \cup \texttt{cut}(\texttt{split}(G_1, cell_0, cell_1))$
35. **endif**
36. **endif**
37. **endswitch**
38. $WorkSet = WorkSet'$
39. **endwhile**
40. **if** $w = (AL.bas, AL.f)$ or $w = (x.bas, x.f)$
41. **foreach** G_2 in $AnswerSet$
42. $current(G_2) = node(current(G_2)).f$
43. **endif**
44. **endwhile**
45. **end**

In the `refine` algorithm, we start from the variable node in an ASG and go along the AEP π. It is required that all the nodes we reach must be unique. If it does(line 29), we just step further; else we first consider it as unique(G' in line 33), and then split a new unique node from it(line 36). Every cell of a unique node has at most one output edge. We guarantee this property by generating more graphs, which is done in the `cut` algorithm.

The idea behind the algorithm `split` is simple: we just split a new unique node from a non-unique node and the new node inherits most properties of the old one. The current memory cell is transferred from the cell of the old node to the corresponding cell of the new one. Algorithm 2 shows the algorithm `split`. After splitting, there may be many redundant edges. We use the `cut` operation to delete the impossible edges and graphs.

Algorithm 2
01. function `split`(G_A, $cell_0$, $cell_1$)
02. **input** G_A: An ASG; $cell_0, cell_1$: cells of the ASG
03. **output** A result shape graph
04. **begin**
05. Create a unique node v_1 in G_A with the same $type_i$, $shared$, x_i and reachable property
06. as $node(cell_1)$
07. $current(G_A) = v_1.f_x$ iff $cell_1 = node(cell_1).f_x$
08. $edge(G_A)(cell_0, current(G_A)) = 1$
09. $edge(G_A)(cell_0, cell_1) = 0$
10. **foreach** edge in G_A where $edge(cell_2, cell_1) > 0$
11. $edge(G_A)(cell_2, v_1.f_x) = edge(G_A)(cell_2, cell_1)$
12. **foreach** edge in G_A where $edge(cell_1, cell_3) > 0$
13. $edge(G_A)(v_1.f_x, cell_3) = edge(G_A)(cell_1, cell_3)$
14. **return** G_A
15. **end**

Figure 1.(5) shows the result shape graph of Fig.1.(3) after splitting a unique node t' from node t when dealing with statement in Fig.1.(4). t' inherits all the in/out edges of t.

Due to the limit of the space, we just list the main steps of the `cut` algorithm.

Algorithm 3
01. function `cut`(G_A)
02. **input** G_A: An ASG
03. **output** $AnswerSet$: A set of ASGs
04. **begin**
05. $AnswerSet = \phi$
06. if a cell of a unique node has a definite output edge, then delete all other
07. output edges whose truth value is 1/2
08. if an unshared basic cell has a definite input edge, then delete all other input edges
09. whose truth value is 1/2
10. $AnswerSet$ = duplicate G_A so every unique cell has at most one output edge
11. **foreach** G in $AnswerSet$
12. check the consistency of the reachable predicates and the $shared$ predicates
13. of G and delete it from $AnswerSet$ if there is any conflict.
14. **return** $AnswerSet$

15. **end**

If a unique cell has more than one output edge in graph G, then we use a graph set S to replace G. For each graph G' in S, every unique cell has at most one output edge. The total number of the graphs in S is the multiplication of the number of output edges of each unique cell. The reachable predicates are conflict in the following two conditions: (1) $r_{cell}(v) = 1$, but there are no paths from the *cell* to v; (2) there is a definite path (all the edges along the path have truth value of 1) from a variable cell to node v, but the corresponding reachable predicate for v has truth value of 0. The share predicates are conflict in two conditions: (1) $shared(v) = 1$, but there are no input edges to $v.bas$ or just an input edge from a unique cell; (2) $shared(v) = 0$, but there are more than one definite input edges to $v.bas$.
Figure 1.(6) shows the result shape graph of Fig.1.(5) after the `cut` operation. We explain the disappearance of edge $(r.bas, t.bas)$. It is clear that variable x has at most one output edge. If $(r.bas, t.bas)$ exists, then $(r.bas, t'.bas)$ can't exists–which conflicts with the fact that t' is reachable from x.
Due to the fact that our `cut` operation is not very powerful, there is a redundant edge $(t.n, t'.bas)$ in Fig.1.(6). It can be deleted by a more powerful `cut` algorithm. Since t' is reachable from r.bas, then $edge(r.bas, t'.bas)$ must be definite true. We also know that t' is not a shared node, then we can say that $(t.n, t'.bas)$ does not exist. We plan to improve our `cut` algorithm in the future.

3.3 Abstract Semantics on ASGs

Given a statement and an ASG G_A, the steps needed to define the semantics are listed in the following.
step 1: use the left AEP to refine G_A;
step 2: use the right AEP to refine the results of step 1 individually;
step 3: if the results of step 2 are empty, then there may be a null dereference, else the l-(r-)value of the left(right) expression is a unique memory cell and we can define the transfer function as in the concrete semantics;
step 4: for each result graph of step 3, we recompute the truth value of the predicates which have been possibly changed;
step 5: `collapse` the result graphs of step 4.

step 5 needs more explanations. Given two ASGs $G = (V, pcore, pinst)$ and $G' = (V', pcore', pinst')$, they are said to be isomorphic if the following conditions are satisfied:

(i) There is a bijection f from V to V' and for each node v in V, the type of v is the same as that of $f(v)$, which also means that there is a bijection g from the cells in V to the cells in V'.

(ii) The truth value of each predicate on nodes or cells in G is the same as that of the predicate on the corresponding parameters generated by f or g in G'.

The `collapse` operation reduces each ASG into a minimal ASG and collapses the isomorphic graphs.

Table 1
Graph-Update that defining the abstract semantics of $e \leftarrow \text{malloc}(i)$

element	value	condition
V'_A	$V_A \cup \{v\}$	v is fresh in V_A
$x'_i(u)$	$x_i(u)$	$u \neq v$
	0	$u = v$
$type'_m(u)$	1	$u = v$ and $m = i$
	0	$u = v$ and $m \neq i$
	$type_m(u)$	otherwise
$shared'(u)$	0	$u = v$
	$shared(u) \wedge \exists cell_0, cell_1 :$ $edge'(cell_0, u.bas) \wedge edge'(cell_1, u.bas)$ $\wedge (cell_0 \neq cell_1)$	$edge(cell_l, u.bas) \neq 0$
	$shared(u)$	otherwise
$r'_{cell}(u)$	$r_{cell}(node(cell_l))$	$u = v$
	$\exists cell_0, \cdots, cell_n : edge'(cell, cell_0)$ $\wedge \cdots \wedge edge'(cell_{n-1}, cell_n)$ $\wedge cell_n \in cells(u)$	u is reachable from $cell_l$ in G_A and $r_{cell}(u) \neq 0$ and $r_{cell}(node(cell_l)) \neq 0$ and $cell_l \notin cells(u)$
	$r_{cell}(u)$	otherwise
$unique'(u)$	1	$u = v$
	$unique(u)$	otherwise
$edge'(cell_0, cell_1)$	1	if $cell_0 = cell_l$ and $cell_1 = v.bas$
	0	if $cell_0 = cell_l$ and $cell_1 \neq v.bas$
	$edge(cell_0, cell_1)$	$cell_0 \notin cells(v)$ and $cell_1 \notin cells(v)$
	0	otherwise

Our method is conservative, that is, all the possible shape graphs the program may generate are preserved in the results of the analysis.

The abstract semantics of the statement is also a transfer function. $[\text{statement}]_A$: $ASGs \rightarrow ASGs$. Supposing $G_A = (V_A, pcore, pinst)$ is the input graph and $G_A = (V'_A, pcore', pinst')$ is the output graph. After refinement, the values of all the expressions in the statement are unique cells in G_A. The abstract semantics is defined as follows.

(1) The abstract semantics of $e \leftarrow \text{malloc}(i)$.
Assume: the unique memory cell of the l-value of expression e in G_A is $cell_l$;
Result: $[e \leftarrow \text{malloc}(i)]_A(G_A) = G'_A$. (See Table 1).

(2) The abstract semantics of free(e).
Assume: the unique memory cell of the right value of expression e is $v.bas$;
Result: $[\text{free}(e)]_A(G_A) = G'_A$. (See Table 2).

(3) The abstract semantics of $e_0 \leftarrow e_1$.
Assume: the unique memory cell of the l-value of expression e_0 is $cell_l$ and that of the right value of expression e_1 is $cell_r$. This statement is divided into two steps: $e_0 \leftarrow null$; $e_0 \leftarrow e_1$. We introduce an intermediated graph G''_A and the transfer operations can be written as $[e_0 \leftarrow null]_A(G_A) = G''_A$. (See Table 3). $[e_0 \leftarrow e_1]_A(G''_A) = G'_A$. (See Table 4).

Note that the logic in the abstract semantics is 3-valued, so the logic operator is

Table 2
Graph-Update that defining the abstract semantics of free(e)

element	value	condition
V'_A	$V_A - \{v\}$	
$x'_i(u)$	$x_i(u)$	
$type'_m(u)$	$type_m(u)$	
$shared'(u)$	$shared(u) \wedge \exists cell_0, cell_1 :$ $edge'(cell_0, u.bas) \wedge edge'(cell_1, u.bas)$ $\wedge (cell_0 \neq cell_1)$	$\exists cell_2 \in cells(v) :$ $edge(cell_2, u.bas) \neq 0$
	$shared(u)$	otherwise
$r'_{cell}(u)$	$\exists cell_0, \cdots, cell_n : edge'(cell, cell_0) \wedge \cdots$ $\wedge edge'(cell_{n-1}, cell_n) \wedge cell_n \in cells(u)$	u is reachable from cells in $cells(v)$ in G_A and $r_{cell}(u) \neq 0$ and $r_{cell}(v) \neq 0$
	$r_{cell}(u)$	otherwise
$unique'(u)$	$unique(u)$	
$edge'(cell_0, cell_1)$	$edge(cell_0, cell_1)$	

Table 3
Graph-Update that defining the abstract semantics of $e_0 \leftarrow null$

element	value	condition
V''_A	V_A	
$x''_i(u)$	$x_i(u)$	
$type''_m(u)$	$type_m(u)$	
$shared''(u)$	$shared(u) \wedge \exists cell_0, cell_1 :$ $edge''(cell_0, u.bas) \wedge edge''(cell_1, u.bas)$ $\wedge (cell_0 \neq cell_1)$	$edge(cell_l, u.bas) \neq 0$
	$shared(u)$	otherwise
$r''_{cell}(u)$	$\exists cell_0, \cdots, cell_n : edge''(cell, cell_0)$ $\wedge \cdots \wedge edge''(cell_{n-1}, cell_n)$ $\wedge cell_n \in cells(u)$	u is reachable from $cell_l$ in G_A and $r_{cell}(u) \neq 0$ and $r_{cell}(node(cell_l)) \neq 0$ and $cell_l \notin cells(u)$
	$r_{cell}(u)$	otherwise
$unique''(u)$	$unique(u)$	
$edge''(cell_0, cell_1)$	1	if $cell_0 = cell_l$ and $cell_1 = null$
	0	if $cell_0 = cell_l$ and $cell_1 \neq null$
	$edge(cell_0, cell_1)$	otherwise

also 3-valued. The equivalence between two memory cells is defined similar to that between two memory nodes in Section 3.1.

We lift the operations on graphs to the operations on sets of graphs in the natural way.

$$\overline{op}(XS) = \bigcup_{s \in XS} op(s) \tag{1}$$

The collecting semantics is expressed as the least fix-point of equation (2) over the variables $GraphSet(n)$. Let FG be the control flow graph; let n be a vertex of the FG, which could be a statement or two special vertexes: *start* and *exit*; $E(FG)$ denotes the set of edges in the control flow graph; $As(FG)$ denotes the set of assignment statements, including malloc and free statements, and there are no

Table 4
Graph-Update that defining the abstract semantics of $e_0 \leftarrow e_1$ (after $e_0 \leftarrow null$)

element	value	condition
V'_A	V''_A	
$x'_i(u)$	$x''_i(u)$	
$type'_m(u)$	$type''_m(u)$	
$shared'(u)$	$shared''(u) \vee \exists cell_0, cell_1 :$ $edge'(cell_0, u.bas) \wedge edge'(cell_1, u.bas)$ $\wedge (cell_0 \neq cell_1)$	$cell_r = u.bas$
	$shared(u)''$	otherwise
$r'_{cell}(u)$	$\exists cell_0, \cdots, cell_n : edge'(cell, cell_0)$ $\wedge \cdots \wedge edge'(cell_{n-1}, cell_n)$ $\wedge cell_n \in cells(u)$	u is reachable from $cell_l$ in G'_A and $r''_{cell}(u) \neq 1$ and $r''_{cell}(node(cell_l)) \neq 0$ and $cell_l \notin cells(u)$
	$r''_{cell}(u)$	otherwise
$unique'(u)$	$unique''(u)$	
$edge'(cell_0, cell_1)$	1	if $cell_0 = cell_l$ and $cell_1 = cell_r$
	0	if $cell_0 = cell_l$ and $cell_1 \neq cell_r$
	$edge''(cell_0, cell_1)$	otherwise

left expression in the free statement and no right expression in the malloc statement; $Tb(FG)$ and $Fb(FG)$ denote the subset of edges in FG that represent the true and false branches of some conditional statements respectively; $cond(m)$ denotes the truth value of the condition expression at statement m.

$$
GraphSet(n) = \begin{cases} \phi \quad : \quad \text{if } n = start \\ \bigcup_{m \to n \in E(FG), m \in As(FG)} \{\texttt{collapse}(\overline{[st(w)]}_A(\overline{\texttt{refine}}(\texttt{refine}(G, \pi_l), \pi_r)))| \\ G \in GraphSet(m)\} \bigcup_{m \to n \in Tb(FG)} \{\texttt{collapse}(G) | G \in GraphSet(m) \text{ and} \\ (cond(m) = 1 \text{ or } 1/2)\} \bigcup_{m \to n \in Fb(FG)} \{\texttt{collapse}(G) | G \in GraphSet(m) \\ \text{and } (cond(m) = 0 \text{ or } 1/2)\} \quad : \quad \text{otherwise} \end{cases} \tag{2}
$$

Since we generate all the possible graphs, we can detect the memory leak errors conservatively. There are two cases in which memory leak may occur.

case 1: there is an ASG in the fix-point of vertex n which has such a node v: the truth value is false or unknown for any reachable predicate.

case 2: it is similar to case 1, the only difference is that the truth value of every reachable predicate is false.

In fact, our method can be used to determine the truth value of some conditional

Table 5
The results of the experiments

program	case 1 error	case 2 error	real error	total graph
elem_delete	2	0	0	78
elem_reverse	8	0	0	288
elem_merge	14	0	0	381
reverse.appl	7	7	7	227
singleRotateWithLeft	3	3	0	195
bintree_rotate_left	7	0	0	191

statements in some cases, such as whether an expression points to null. We can also assume all conditional expressions' truth values are 1/2 for simplicity.

4 Experiments

We have implemented a prototype of the presented method and performed some experiments. The results are shown in Table 5. The first column lists the programs in our experiments. The first four programs are from [21] which operate on linked lists. The last two programs operate on trees. The fifth is from [22] and the sixth is from [23]. There are some complex expressions in the last program, such as $(*t)\rightarrow right$ and $m\rightarrow left\rightarrow right$. The second and third columns show the memory leak error alarms corresponding to the conditions in case 1 and case 2 respectively. The fourth column shows the number of the real errors. The last column is the number of the total shape graphs in the fix-point of all the vertexes in the control flow graph.

Our experiments lead to several interesting observations. First, minimization is not always efficient. Minimization is used to make the computation of the program feasible, but it also introduces imprecision. We use `collapse` operation in (2) to get the minimal ASGs at each vertex in the control flow graph. For the `bintree_rotate_left` program, since it doesn't contain `while` statement, we can get the fix-point without the `collapse` operation and the number of ASGs is less than that with the `collapse` operation. The number of total graphs shown in Table 5 of the `bintree_rotate_left` program is computed without the `collapse` operation.

Second, different programs may need different core predicates. The core predicates can be different from the definition in this paper. Actually, we add *shared* into the core predicate set in dealing with the `elem_merge` program and the total number of shape graphs is greatly reduced. But if we also add *shared* into the core predicate set in dealing with the last two programs, the total number of shape graphs will increase.

It can be proved that any real memory leak is alarmed either as case 1 error or case 2 error. From Table 5, we can see that case 2 error is more precise than case 1 error. There can also be memory leaks in case 1 but not in case 2.

5 Related Work

The shape-analysis problem was originally investigated by Reynolds for a Lisp-like language with no destructive updating [7]. They treated the problem as one of simplifying a collection of set equations. [8] uses a similar method, but for an imperative language supporting non-destructive manipulation of heap-allocated objects, which treated the problem as solving a collection of equations using regular tree grammars. In the work of [9,10,11,17,18], they all developed shape analysis methods that associate each program point with a single shape graph, which will be less precise but more compact representations. [11] uses storage shape graphs to do abstraction. It can distinguish between the heap cells directly pointed to by variables and "summary nodes" are used for the "deeper" heap cells. Heap reference counts are used for summaries. The work most similar to ours is in [12,13]. [12] is a parametric framework for shape analysis. It provides the basis for generating different shape analysis algorithms by varying the instrumentation predicates used. This paper provides a special method which can deal with arbitrary expressions and defines the conditions of memory leak. Our `split` method is similar to the *materialization* method in [13]. While their method can materialize a summary node, which can be seen as the non unique node in our method. Our `split` method only materializes a unique node, which makes the value of each expression a unique cell. [15] also uses a method similar to [12] and provides the "cleanness" conditions for absence of memory errors. [6] presents an inter-procedural shape analysis algorithm for languages with destructive updates and formulates it as dataflow analysis. It is a scalable demand-driven analysis that tracks heap objects and can precisely analyze some acyclic list implementations. [16] uses Boolean satisfiability to add path sensitivity to static memory leak detection. Their results demonstrate very good false-positive ratios. [1] also presents a sound method that can find memory leaks and double deletions of objects held in lists, stacks, maps and arrays. Their method is based on the concept of object ownership, which holds the exclusive right and obligation either to delete the object or to transfer the obligation.

6 Conclusion

In this paper, we present a shape analysis method which can deal with complex expressions. This method also allows pointers into the middle of structures. AEP is generated from arbitrary expression in C language and is used to refine the ASG. The semantics of a statement can be easily defined on the result shape graphs after the refinement. Memory error conditions can be checked on the ASGs. In the future, we plan to improve the efficiency of our method and provide more thorough experimental evaluations. We will also apply our shape analysis results to verify the correctness of programs in C language.

References

[1] David L. Heine and Monica S. Lam. Static Detection of Leaks in Polymorphic Containers. In *28th International Conference on Software Engineering (ICSE'06)*, May 2006.

[2] David L. Heine and Monica S. Lam. A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak detector. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.

[3] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of OOPSLA 2002: Object-Oriented Programming Systems, Languages and Applications*, pages 211-230, November 2002

[4] D. Clarke. An object calculus with ownership and containment. In *The 8th International Workshop on Foundations of Object-Oriented Languages*, January 2001.

[5] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of OOPSLA 2002: Object-Oriented Programming Systems, Languages and Applications*, pages 292-310, November 2002.

[6] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *The 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 310-323, January 2005.

[7] J. C. Reynolds. Automatic computation of data set definitions. In *Information Processing 68: Proceedings of the IFIP Congress*, pages 456-461, New York, NY, 1968.

[8] Neil D. Jones, Steven S. Muchnick: Flow Analysis and Optimization of Lisp-Like Structures. In S.S. Muchnick and N.D.Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102-131. Prentice-Hall, 1981.

[9] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *ACM Symposium on Principles of Programming Languages*, pages 66-74, 1982.

[10] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 21-34, 1988.

[11] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proc. of the ACM SIGPLAN'90 conf. on Programming Language Design and Implementation*, June 1990.

[12] Mooly Sagiv, Thomas Reps and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2002.

[13] Mooly Sagiv, Thomas Reps and Reinhard Wilhelm. Solving Shape-Analysis Problems in Languages with Destructive Updating. In *23rd Annual ACM SIGPLANSIGACT Symposium on Principles of Programming Languages*, January 1998.

[14] Radu Rugina. Quantitative Shape Analysis. In *Proceedings of the Static Analysis Symposium*, pages 228-245, August 2004

[15] N. Dor, M. Rodeh, and S. Sagiv. Checking cleanness in linked lists. In *Proceedings of the Static Analysis Symposium*, pages 115-134, July 2000.

[16] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of ESEC/FSE 2005*, September 2005.

[17] J. R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, University of California, 1989.

[18] J. Stransky. A lattice for abstract interpretation of dynamic (Lisp-like) structures. *Information and computation*, 101(1): 70-102, November 1992.

[19] S. Kleene. *Introduction to Metamathematics*, Second Ed. North-Holland, Amsterdam, 1987.

[20] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysisof programs by construction of approximation of fixed points. In *Symposium on Principles of Programming Languages, ACM*, New York, 238-252, 1977.

[21] Nurit Dor. *Detecting Memory Errors via Static Pointer Analysis*. M.Sc. Thesis, Tel-Aviv University, 1999.

[22] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C*. Second Ed., chapter 4, Addison-Wesley Longman Publishing Co., Inc., 1996.

[23] http://home.gna.org/gdsl/