Applying Traversal-Pattern-Sensitive Pointer Analysis to Dependence Analysis *

Yuan-Shin Hwang   Joel Saltz

Department of Computer Science
University of Maryland
College Park, MD 20742
{shin, saltz}@cs.umd.edu

November 12, 1997

Abstract

This paper presents a technique for dependence analysis on programs with pointers or dynamic recursive data structures. It differs from previously proposed approaches in analyzing structure access conflicts between traversal patterns before gathering alias and connection information. Conflict analysis is conducted under the assumption that each unique path leads to a distinct storage location, and hence traversal patterns can be analytically compared to identify possible conflicts. The rationale of this assumption is that if statements are deemed to be dependent by this approach, they are inherently sequential regardless of the shapes of the data structures they traverse. Consequently, there is no need to perform alias/connection analysis on the statements that construct such data structures. Furthermore, the information of traversal patterns gathered in conflict analysis phase can direct alias/connection analysis algorithm to focus on statements that are crucial to optimizations or parallelization. A such *traversal-pattern-sensitive* pointer analysis algorithm will also be presented.

*This work was sponsored in part by NSF (ASC-9213821 and CDA9401151).

1 Introduction

Dependence analysis is the key technique behind parallelization and optimizations on programs with pointers or recursive data structures. There have been several algorithms proposed by researchers [7, 8, 9, 10, 14]. It is a considerably complicated problem, since it involves other pointer analysis techniques as well, such as alias analysis [2, 4, 5, 12, 20], side effect analysis [2, 13], and even shape analysis [1, 6, 16, 18]. These proposed dependence analysis techniques first identify aliases of pointer variables and connections of recursive data structures by examining all pointer assignment statements, and then apply the information to dependence test after read and write sets are gathered. In other words, analysis process of these techniques can be divided into two phases: an alias/connection analysis phase followed by an interference/conflict analysis phase.

None of these techniques performs the alias/connection analysis with any knowledge derived from the second phase. One implication of this approach is that all alias/connection information must be gathered, since patterns of traversal are unknown in this phase. The drawback is that information of aliases and connections might be inappropriate for interference/conflict analysis [11]. Consider programs that construct cyclic recursive data structures but have acyclic traversal patterns, e.g. graph algorithms that traverse cyclic graphs following acyclic spanning trees. The results of alias/connection analysis on this type of programs will show that constructed data structures are cyclic and hence provide no useful information for parallelization or optimizations [6]. Furthermore, if programs are inherently sequential regardless of connections of recursive data structures, prior knowledge of traversal patterns can avoid unnecessary alias/connection analysis.

Another implication is that access conflicts can not be judged by the sets of reference patterns alone. Every reference pattern in the read and write sets will have to be mapped onto all possible aliases and connections before interference/conflict analysis can proceed. This process of mapping all reference patterns to all possible aliases and connections makes interference/conflict analysis complicated.

This paper presents a different approach — interference/conflict analysis process is performed before alias/connection analysis. Interference/conflict analysis is conducted on the assumption that each unique path leads to a distinct storage location. Under this assumption, any programs fragments which are deemed to be dependent in this phase are inherently sequential regardless of the shapes of actual data structures. Only the reference patterns of those program fragments with parallelism will be mapped onto possible aliases and connections by alias/connection analysis phase to confirm the results of the first phase. The DEF/USE information of pointer statements will be used to connect these two analysis phases [11]. The special feature of this approach is its ability to identify traversal patterns and estimate possible shapes of the structures specified by the traversal patterns.

The first advantage of this approach is that it simplifies the interference/conflict analysis process. Each reference pattern in the read and write sets can be analytically represented by a single symbolic path expression, instead of a set of reachable locations. Access conflicts will be identified by comparing the path expressions of reference patterns. Furthermore, path expressions symbolize the patterns of traversal on recursive data structures. This paper will introduce an algorithm to determine access conflicts among iterations of the same loops and dependence within sequences of statements.

The second advantage is that alias/connection analysis can focus on the statements which will contribute to construction of recursive structures that are specified by traversal patterns identified in the interference/conflict analysis phase. This approach can be called *traversal-pattern-sensitive* pointer analysis. This paper will present an algorithm to estimate possible shapes of dynamic recursive data structures specified by traversal patterns. The result of shape estimation can be used to confirm the results of interference/conflict analysis phase.

The rest of this paper is organized as follows. Section 2 describes the programming model and outline of the dependence analysis algorithm. Section 3 presents the algorithm to conduct dependence test for loop iterations and sequences of statements. Section 4 describes an algorithm to perform traversal-pattern-sensitive shape estimation. Related work is compared in Section 5 and summary is presented in Section 6.

2 Background

2.1 Programming Model

The algorithms presented in this paper are designed to analyze programs with dynamic recursive data structures that are connected through pointers defined in the languages like Pascal and Fortran 90. Pointers are specified by declared pointer variables, and are simply references to nodes with a fixed number of fields, some of which are pointers. Memory allocations are done by the function *new()*. Pointer arithmetic and casting in languages such as C are not allowed. Although multi-level pointers are not considered in this paper, they can be handled by converting them into levels of records, each of which contains only one field that carries the node location of the next level. Consequently, pointer dereferences of multi-level pointers can be treated as traversal of multi-level records.

Programs will be normalized such that each statement contains only simple binary access paths, each of which has the form $v.n$ where v is a pointer variable and n is a field name. Therefore, excluding regular assignment statements, the three possible forms of pointer assignment statements are

1. $p = q$ (*aliasing statements*)
2. $p = q.n$ (*link traversing statements*)
3. $p.n = q$ (*link defining statements*)

The first two forms of statements will induce aliases without changing any connections of recursive data structures, whereas the execution of each statement of the last form will remove one (maybe null) link from existing dynamic data structures and then introduce a new link. Note that although the other possibility $p.m = q.n$ is also valid, it is represented by two consecutive statements, $t = q.n$ and $p.m = t$, for the reason of simplicity.

2.2 Intermediate Program Representation and Path Expressions

Programs will be transformed into an SSA (Static Single Assignment) intermediate representation [3]. Although SSA form is designed specially for programs with fixed-location variables only, e.g. Fortran-77 programs, same transformation can be applied to pointer variables since contents (i.e. location addresses) of pointer variables can be treated as values in regular variables.

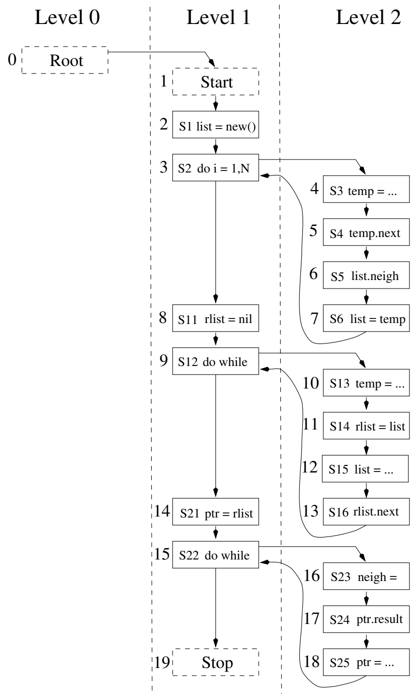
Once programs are transformed into SSA form, each pointer instance will have a unique name. Furthermore, instances of pointer variables are defined by alias statements and link traversing statements, since link defining statements do not define new instances. In other words, the associations of pointer variables with storage locations are defined by alias statements and link traversing statements. Therefore, every pointer instance can be represented by a *path expression*, which is denoted by a tuple

$$< p,\ e,\ f,\ r >$$

where p is the the pointer variable instance, e is the entry point of the referenced data structure, and f is the path that induction pointer advances forward at each iteration, and r is the relative path from the induction pointer to the instance. It is equivalent to the regular expression $p:\ e\,(.f)^{\star}.r$.

C	Create a doubly-linked list
S1	$list_1$ = new()
S2	do i = 1, N
S2′	$list_2 = \phi\ (list_1, list_3)$
S3	node = new()
S4	node.next = $list_2$
S5	$list_2$.neigh = node
S6	$list_3$ = node
S7	end do
C	Reverse the doubly-linked list
S11	$rlist_1$ = nil
S12	do while ($list_3$)
S12′	$list_3 = \phi\ (list_2, list_4)$
S12″	$rlist_2 = \phi\ (rlist_1, rlist_3)$
S13	temp = $rlist_2$
S14	$rlist_3$ = $list_3$
S15	$list_4$ = $list_3$.next
S16	$rlist_3$.next = temp
S17	end do
C	Traverse the reversed doubly-linked list
S21	ptr_1 = $rlist_2$
S22	do while (ptr_2)
S22′	$ptr_2 = \phi\ (ptr_1, ptr_3)$
S23	neigh = ptr_2.neigh
S24	ptr_2.result += neigh.value
S25	ptr_3 = ptr_2.next
S26	end do

(a) Create, Reverse and Traverse a Doubly-Linked List



(b) Interval Flow Graph

Figure 1: Example Program and Its Interval Flow Graph

2.3 The Interval Flow Graph

The dependence analysis algorithm in this paper performs an interval analysis since it considers loop nesting hierarchies of programs. The control flow information is represented by *interval flow graphs* [19]. The advantage of interval flow graphs is that interval analysis can be performed without explicitly constructing a sequence of graphs in which intervals are recursive collapsed into single nodes. An interval flow graph is a directed graph $G = (N, E)$, where N is the set of nodes and E is the set of edges. For each node $n \in N$, $Level(n)$ is the loop nesting level of n. Figure 1(b) depicts the interval flow graph of a simple program which creates a doubly-linked list, reverses it, and then traverses the reversed linked list, as shown in Figure 1(a). Note that ϕ-function statements at the headers of loops are part of loop header nodes.

2.4 Algorithm Outline

The algorithm to perform dependence analysis can be broken into three steps.

- *DEF/USE Information Construction*
  This step identifies aliases of pointer variables induced by aliasing statements, and constructs DEF/USE chains between link defining statements and link traversing statements [11].

- *Dependence Analysis*
  Dependence test is performed on references based on the assumption that each unique path expression leads to a distinct location. Any programs which are deemed to be dependent in this process are inherently sequential regardless of the shapes of actual data structures. The algorithm will be presented in Section 3.

- *Traversal-Pattern-Sensitive Shape Analysis*
  Shapes and connections of recursive data structures specified traversal patterns obtained from the previous step are analyzed to confirm the results of dependence analysis. The relationships between traversal patterns and the statements that build the structures specified by traversal patterns are established by the DEF/USE information. The algorithm of traversal-pattern-sensitive shape analysis will be introduced in Section 4.

The example program in Figure 1 will be used to demonstrate that this approach will be able to identify parallelism on programs even with cyclic data structures. The first loop creates a doubly-linked list since it contains cycles, and the second loop reverses the list to show that the algorithms presented in this paper can handle destructive updating.

3 Dependence Analysis

3.1 Access Conflicts

Access conflicts occur when two statements potentially access the same storage locations during the program execution. Let $Read(S)$ denote the set of locations read by statement S, and let $Write(S)$ denote the set of locations written by statement S. Statements S_1 and S_2 have access conflicts if

$$(Write(S_1) \cap (Read(S_2) \cup Write(S_2))) \cup$$
$$(Write(S_2) \cap (Read(S_1) \cup Write(S_1))) \neq \emptyset$$

In contrast to other techniques, this approach does not build graphs or matrices to represent connection information [8, 14]. It assumes that each unique path expression leads to a distinct location. Consequently, elements of $Read$ and $Write$ sets can be represented by path expressions. To determine if two path expressions can reach same locations, comparison operations similar to the $Match$ operation defined in Deutsch [4] can be applied.

Since programs will be normalized, each reference pattern in $Read$ or $Write$ sets will be either of two forms – a pointer variable p or a simple binary access path $p.n$. The former form will not cause any structure access conflicts and the dependence between pointer variables can be traced following the edges of SSA representation. On the other hand, each of the reference patterns with the latter form reads or writes the field n of structure node specified by the pointer p, which in turn can be represented by a single path expression. Therefore, access conflicts can be determined by comparing the path expressions.

Access conflict analysis provides essential information for recognizing dependence in programs. The dependence analysis algorithm presented in this section will determine dependence among a sequence of n statements, which correspond to the statements on the same levels of the interval flow graph, and dependence among loop iterations, which can be viewed as inter-level dependence.

3.2 Inter-Level Dependence

Iterations of loops can be executed in parallel if they carry no dependence. However, since elements of recursive data structures must be traversed through links sequentially, dependence always exists between iterations for loops that traverse recursive data structures. On the other hand, if sequential traversal is the only dependence between iterations of a loop, the computation of loop body will generate the same results regardless of order of traversal. That is, the computations of the iterations of such loop carry no dependence. Therefore, the algorithm in this section will identify the loops with this property.

3.2.1 Alias Information and Path Expressions

Aliases of pointer variables are gathered during the DEF/USE construction phase [11]. The process is performed iteratively until a fixed point is achieved.

Pointer variables in a loop can be classified into three groups: global pointers, local pointers, and iteration pointers. *Global pointers* are the pointers that are defined before entering the loop and not redefined during the execution of the loop, while *local pointers* are defined at every iteration and are only referenced at the iterations they are defined. In contrast to local pointers, contents of *iteration pointers* might be passed between iterations. Iteration pointers can be further divided into two classes, induction pointers and non-induction pointers. Each *induction pointer* points to a certain location of a recursive data structures before entering the loop and then advances a fixed path at every iteration, whereas *non-induction pointers* do not have this property. The DEF/USE construction algorithm will differentiate induction pointers from non-induction pointers.

Pointer types can be easily identified from SSA representation. Each definition of a ϕ-function at the header of a loop corresponds to an iteration pointer, while every definition in loop body creates a local pointer. Therefore, the fixed-point solutions are only required for iteration pointers, since every local pointer can be represented by a path expression with an entry starting from an iteration pointer or a global pointer. As a result, the aliases of iteration pointers at each iteration represent the aliases caused by the previous iterations of loop execution. For instance, the iteration pointer ptr_2 of S22$'$ in Figure 1 might be an alias to ptr_1 of S21 before the loop or ptr_3 of S25 of previous iteration, whereas the local pointer $neigh$ can be represented by the path expression $ptr_2.neigh$. Another example is that the local pointer $temp$ of S13 is a must-alias to the iteration pointer $rlist_2$ of S12$''$, regardless of the aliases of $rlist_2$.

3.2.2 Dependence Test

The iterations of a loop can be executed in parallel if there are no structure access conflicts and data dependence between iterations. Therefore, if any iteration pointers of a loop are non-induction pointers and are referenced by link defining statements, i.e. they are involved in structure construction process, then the iterations of the loop can not be executed in parallel. The result of parallel execution of such loop will be nondeterministic. Consider the program in Figure 1(a). The iteration pointers $list_2$ of loop S2 and $rlist_2$ of loop S12 are referenced by link defining statements that create and then reverse a doubly-linked list. Consequently, loops S2 and S12 have to be executed sequentially.

If iteration pointers of a loop are induction pointers, they will start from certain locations and advance fixed paths after each iteration. Furthermore, if operations on these induction pointers and local pointers do not cause any access conflicts and data dependence, the iterations of this loop are independent. Specifically, let S be the statements of the loop body and let $S^{(i)}$ specify the ith iteration, then the conditions for inter-level dependence test of a loop are as follows:

- Iteration pointers are induction pointers.
- $\bigcup_{i \neq j} \left[\begin{array}{l} Write(S^{(i)}) \cap (Read(S^{(j)}) \cup Write(S^{(j)})) \cup \\ Write(S^{(j)}) \cap (Read(S^{(i)}) \cup Write(S^{(i)})) \end{array} \right] = \emptyset$

Take loop S22 of the program shown in Figure 1(a). The iteration pointer ptr_2 is an induction pointer with the path expression <ptr_2, ptr_1, $next$, $->$>. That is, at ith iteration, the induction pointer ptr_2 will point to location $ptr_1.next^{i-1}$. The element of $Write$ set that might contribute to access conflicts is <$S24$, $ptr_2.result$>, and those of $Read$ set are <$S24$, $ptr_2.result$> and <$S24$, $neigh.value$>. Since the same reference pattern <$S24$, $ptr_2.result$> in $Read$ and $Write$ will not cause any access conflicts, unless the path specified by induction pointer ptr_2 is cyclic, the iterations of this loop can be declared as independent in this phase. This situation will be confirmed by traversal-pattern-sensitive shape analysis in the next phase.

The structures specified by induction pointers are called *main traversal structures* and the links of main traversal structures can be called *main traversal links*, while the rest links can be called *secondary traversal links*. The main traversal structure of this example is specified by induction pointer ptr_2 and the main traversal links are traversed by the statement S25. The DEF/USE information reveals that the statement that defines these main traversal links is S16, which in turn reverses a list defined by statement S4. The links traversed by S23 are secondary traversal links and are constructed by S5. Shape analysis phase will use above information to estimate the main traversal structure and confirm the result of dependence test.

Note that the reference pattern <$S24$, $neigh.value$> does not contribute to structure access conflicts since it references a different field than that of the other patterns. Consequently, the connections caused by second traversal link $ptr_2.neigh$ need not to be examined by shape estimation phase. However, if the statement S24 is replaced by

S24 ptr$_2$.result += neigh.result

the shape analysis phase will have to determine if the second traversal links cause access conflicts, even though this phase will consider this loop has independent iterations.

3.3 Intra-Level Dependence

The statements on the same level, either simple basic statements or loops, will be examined if access conflicts might occur to prohibit parallel execution. Since in most cases it is not practical to expect that n sequential statements on the same level can be transformed into a single parallel statement, the technique presented in this section will identify the dependence among the n statements. The results of this analysis can be used either to parallelize the statements or to remove redundant synchronizations.

Unlike inter-level dependence test, which needs to construct explicit $Read$ and $Write$ sets since it determines dependence between iterations of the same sets of statements, access conflicts among a sequence of statements on the same levels can be inferred from the edges of SSA form and DEF/USE chains. A *dependence graph* $DG = (V, E)$ will be constructed for each sequence of statements on the same level, where V is the set of nodes and E is the set of edges. Each node of a dependence graph represents either a statement, a ϕ-statement at the header of a loop, or a definition to a global variable. A directed edge $<i, j>$ means node j is dependent on node i.

Since programs are normalized, the rules to build a dependence graph for a sequence of n statements can be summarized as follows.

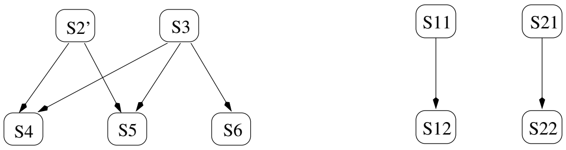
1. *Reference to* p
   An edge will be added from the node that represents the statement defining p to the current node.

2. *Definition to* p
   No new edges will be created because of the SSA renaming property.

3. *Reference* $p.n$
   In addition to the edge caused by the reference to p as rule 1, edges will be added to represent dependence of $p.n$, which can obtained from the DEF/USE information.

4. *Definition to* $p.n$
   Since the last statements that reference the reaching definitions of $p.n$ before this definition must be executed before the current statement, edges from these statements will be built. Dependence of p is handled by rule 1.

Note that these rules basically copy the edges of SSA representation and DEF/USE chains to build a dependence graph. If current statement is the header node of a loop, any edges of SSA form or DEF/USE chains into or from the loop body can be copied to the dependence graph. In other words, the current node can be treated as the result of collapsing the whole loop body into a single node.



(a) Loop S2 in Figure 1

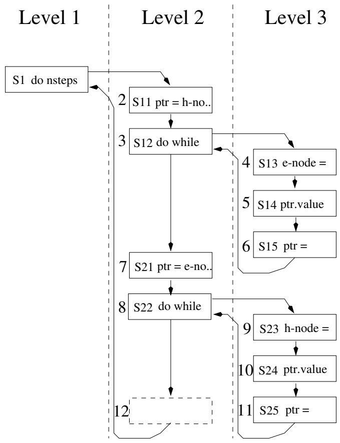
(b) EM3D Loop

Figure 2: Dependence Graphs

```
S1     do nsteps = 1, N
S11      ptr1 = h-nodes
S12      do while (ptr2)
S12'     ptr2 = φ (ptr1, ptr3)
S13        e-node = ptr2.neigh
S14        ptr2.result += e-node.value
S15        ptr3 = ptr2.next
S16      end do

S21      ptr4 = e-nodes
S22      do while (ptr5)
S22'     ptr5 = φ (ptr4, ptr6)
S23        h-node = ptr5.neigh
S24        ptr5.result += h-node.value
S25        ptr6 = ptr5.next
S26      end do
S27    end do
```

(a) EM3D Loop



(b) Interval Flow Graph

Figure 3: Simplified EM3D Loop and Its Interval Flow Graph

Figure 2(a) shows the dependence graph for the sequence of statements (S3–S6) in the loop body of loop S2 in Figure 1(a). It demonstrates that not much parallelism is available among these statements. On the other hand, Figure 2(b) depicts the dependence graph of a simplified version of EM3D loop [15], which is displayed in Figure 3. This graph shows that the two sequences of statements S11–S12 and S21–S22 can be executed independently, i.e. the loops that traverse lists of E nodes and H nodes respectively can be executed independently. Furthermore, if the inter-level dependence test is applied to the EM3D loop, it will reveal that both loops have independent iterations. Combining the results of inter- and intra-level dependence test, iterations of both loops can be executed independently and synchronization between these two loops can be removed. Note that although the overall graphs referenced by EM3D loops are bipartite, the traversal-pattern-sensitive shape analysis algorithm in the next section will recognize that both loops traverse the graphs via singly-linked lists specified by induction pointers ptr_2 and ptr_5, respectively.

4 Traversal-Pattern-Sensitive Pointer Analysis

This section presents an algorithm to perform traversal-pattern-sensitive shape analysis. This algorithm is adapted from the shape analysis algorithm proposed by Sagiv et al [18]. The main difference is its ability to represent the links of main traversal structures and to maintain the connection information of secondary traversal links on the same shape graphs. It explicitly represents the main traversal links by edges on shape graphs and performs shape estimation on these links, whereas the information of secondary traversal links is stored implicitly in the nodes of shape graphs. The shape estimation of main traversal structures will be especially useful for parallelization since it can aid dependence test to determine any dependence among instances of iterative or recursive traversal.

4.1 Shape Graphs

Shape analysis is performed on finite directed graphs, called *shape graphs*, which represent unbounded recursive data structures. The shape graphs presented in this paper are closely related to the Storage Shape Graph (SSG) proposed by Chase et al. [1], the Abstract Storage Graph (ASG) by Plevyak et al. [16], and Shape-Graphs by Sagiv et al. [18]

Shape graphs have two types of nodes: *pointer stances* and *storage nodes*, which can be further divided into *simple nodes* that represent allocated allocations and *summary nodes* each of which represents a set of allocated locations. New storage nodes can be created by calling the storage allocation function $new()$ or extracted from summary nodes by link traversing statements (it is called *materialization* [18]). On the other hand, storage nodes will be removed when they are no longer reachable, or be absorbed by summary nodes when they are not directly connected to any pointer instances (i.e. *summarization*). In order to uniquely name each storage node, every storage node will be annotated by a distinct tag. The purpose of assigning tags to storage nodes is to specify secondary traversal links without creating edges on shape graphs such that shape analysis on main traversal structures will be simplified.

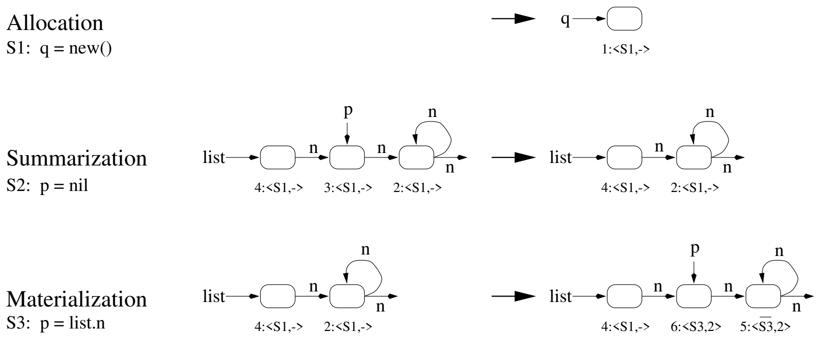


Figure 4: Storage Nodes and Tags

Each tag is denoted by the form *c:<s, l>*, where c is a unique number generated from a global counter, s is the statement that generates the corresponding node, and l is the link to the tag of a summary node if the node is extracted (materialized) from the summary node and is nil otherwise. Figure 4 depicts the tag creation processes and corresponding nodes operations. A new storage node with tag *1:<S1, –>* is created when *S1: q = new()* is executed. On the other hand, the tag of the summary node will remain the same when summarization is performed after *S2: p = nil* is executed, since both nodes are allocated at $S1$. However, the tag of the summary node is modified after materialization by the statement *S3: p = list.n* to reflex the fact that the summary node before execution of S3 is different from that after execution. The new summary node has the tag *5:<$\overline{S3}$, 2>* and the materialized node is denoted by tag *6:<S3, 2>*.

Since each summary node represents a set of nodes, extra tags (called *sharing tags*) are required to specify the connection relationships among these nodes. The tag has the form $\{f_1, f_2, \cdots, f_n\}$, where f_1, f_2, $\cdots$, f_n are the field names of self-cyclic edges of the summary node. This tag means all nodes along any paths specified by the regular expression $(f_1|f_2|\cdots|f_n)^\star$ will be distinct. For example, the summary node with sharing tag $\{n\}$ in Figure 5(a) represents an unshared list specified by the path $(n)^\star$. Similarly, the summary node with tag $\{l, r\}$ of Figure 5(b) is a tree-like structure accessible via the path $(l|r)^\star$. On the other hand, if a summary carries more than one sharing tag, it means nodes on the path designated by the field names of the same tag will be distinct, but the same assertion might not hold when field names are not from the same tag. Figure 5(c) symbolizes that $(n)^\star$ and $(p)^\star$ each references a list of unshared nodes, but $(n|p)^\star$ or $(n)^\star(p)^\star$ might reference cycles. Similarly, Figure 5(d) means multiple paths of the form $(l|r)^\star$ might reach the same node.

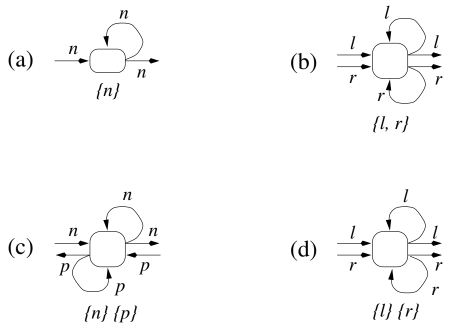


Figure 5: Sharing Information of Summary Nodes

Tail nodes of recursive data structures usually are not summarized into summary nodes to distinguish cyclic structures from acyclic structures. Figure 6 depicts the shape graphs that characterize different types of data structures. The difference between the tail nodes of Figure 6(a) and Figure 6(d) distinguishes a singly-linked list from a circular list. However, the tail nodes will be summarized when it is not possible to locate them, such as the arbitrary graphs shown in Figure 6(f).

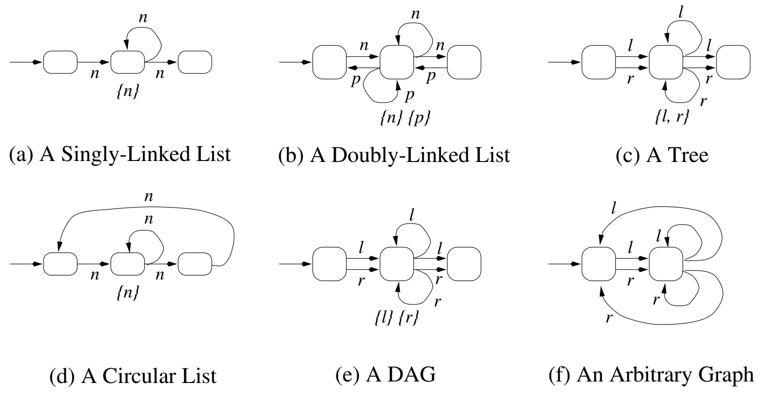


Figure 6: Data Structures Represented by Shape Graphs

The edges presented on shape graphs are used to symbolize the links of main traversal structures. The secondary traversal links are not represented on shape graphs, but the connections caused by these links are described by node tags. Figure 7 depicts some examples of cyclic recursive data structures with acyclic traversal patterns, where the dashed edges represent the secondary traversal links and their destinations are specified by tags of storage nodes. When cyclic structures with acyclic traversal patterns are modeled by the shape graphs, their acyclic structures specified by the traversal patterns can be easily characterized. As a result, this representation can facilitate pointer analysis on programs even with cyclic data structures.

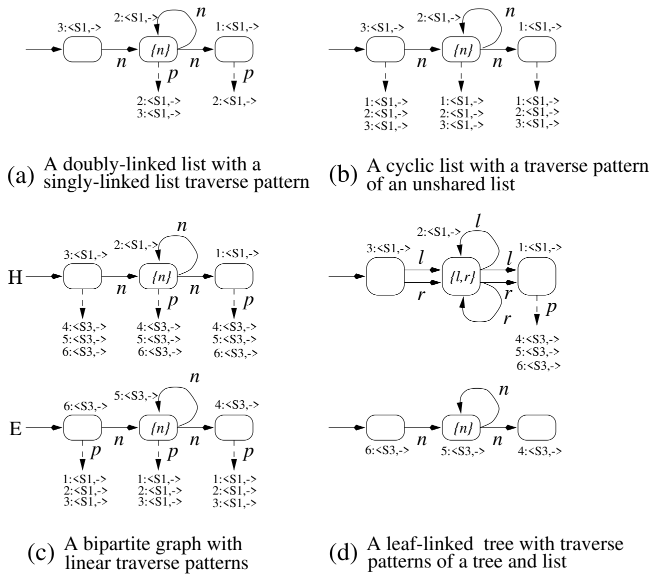


Figure 7: Cyclic Structures with Acyclic Traversal Patterns

In summary, a shape graph is a finite directed graph $G = (V, E)$ where V is the set of nodes and E is the set of edges.

- V consists of two types of nodes: *pointer stances* and *storage nodes*, which can be further divided into *simple nodes* that represent allocated allocations and *summary nodes* each of which represents a set of allocated allocations.

- The set of edges is comprised of two kinds of edges: *pointer edges* each of which has the form $[v, n]$ where v is a pointer instance and n is a storage node, and *field edges*, each of which is denoted by a tuple $<s, f, t>$ where s and t are storage nodes and f is a field name. Consequently, $E = <E_p, E_f>$.

- A unique tag is associated with each storage node.

- Sharing tags are annotated on each summary node to characterize the sharing information along the path specified by the field names of self-cyclic edges.

- Edges which are not part of main traversal structures are not represented as edges on shape graphs. They are described by connections to node tags.

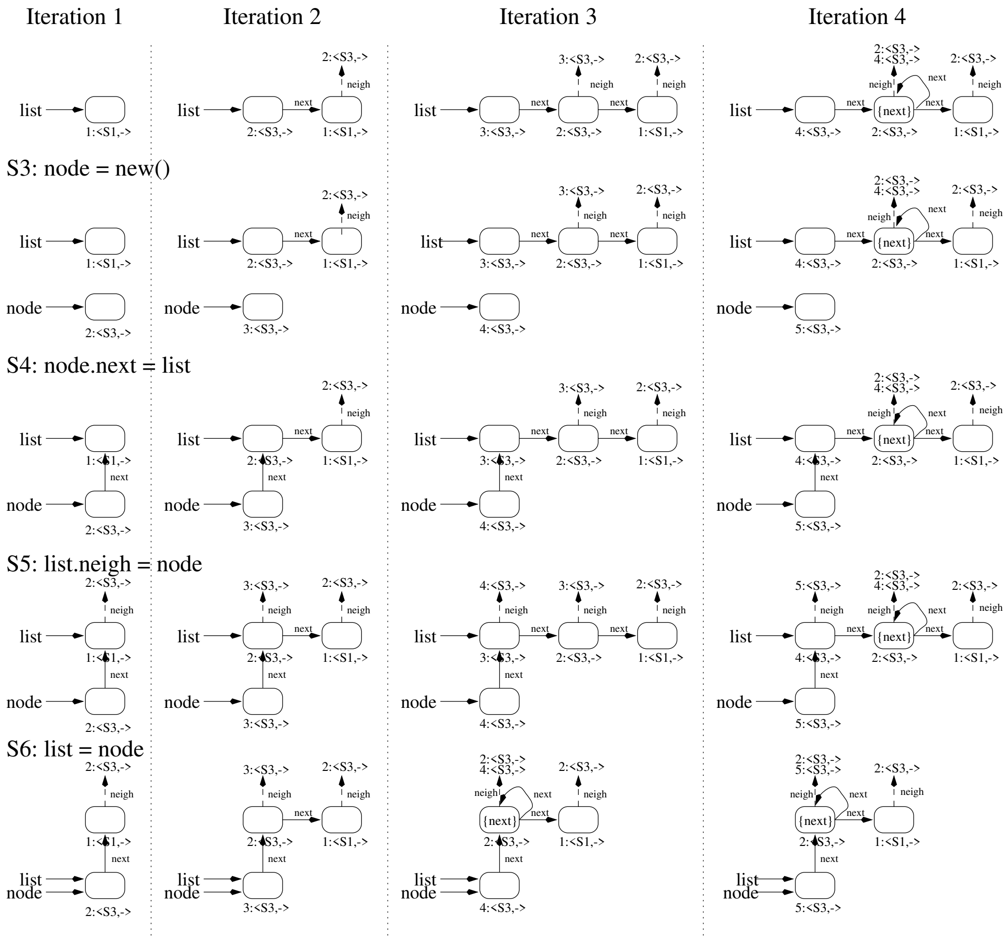


Figure 8: Shape Graphs Generated by Doubly-Linked List Building Program

4.2 Algorithm

This algorithm performs traversal-pattern-sensitive shape analysis, which estimates the possible shapes of dynamic recursive data structures specified by traversal patterns. It works as follows:

- The set of pointer assignment statements to be examined is determined by results of dependence analysis phase and DEF/USE information.
- Perform the iterative algorithm to compute a shape graph for every pointer statement. The transformations of shape graphs are defined by the types of pointer statements. Only those sets that will be modified are presented.
  - $S : p_i = new()$
    $V = V_{in} \cup \{C :< S, - >\}$
    $E_p = E_{p_{in}} \cup \{[p_i, C :< S, - >]\}$
  - $S : p_i = q_j$
    $E_p = E_{p_{in}} \cup \{[p_i, s] | [q_j, s] \in E_{p_{in}}\}$

– $S : p_i = q_j.n$
$E_p = E_{p_{in}} \cup \{[p_i, t] | [q_j, s] \in E_{p_{in}} \wedge [s, n, t] \in E_{f_{in}}\}$
Materialization will occur if $[p_i, t] \in E_p$ and t is a summary node.

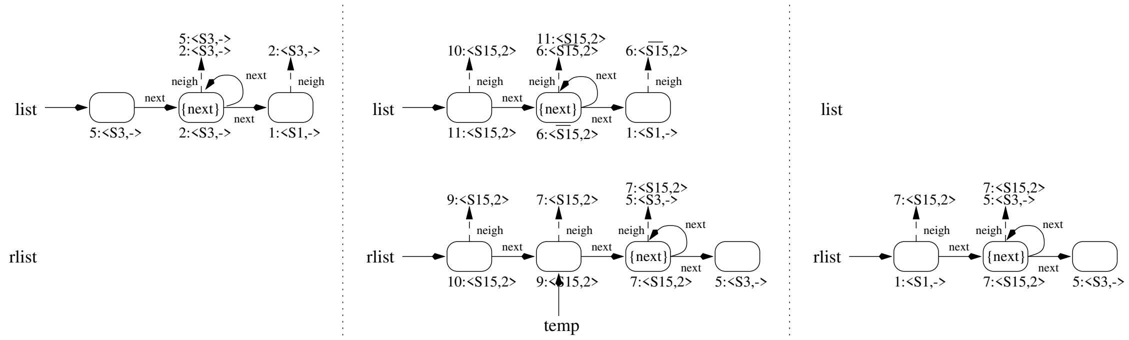
– $S : p_i.n = q_j$
$E_f = E_{f_{in}} \cup \{[s, n, t] | [p_i, s] \in E_{p_{in}} \wedge [q_j, t] \in E_{p_{in}}\}$

– $S : p_i = \phi(p_j, p_k)$
$V = V_{in\,j} \cup V_{in\,k}$
$E_i = E_{in\,j} \cup E_{in\,k}$
$E_p = E_i \cup \{[p_i, s] | [p_j, s] \in E_i\} \cup \{[p_i, s] | [p_k, s] \in E_i\} - \{[p_j, \star]\} - \{[p_k, \star]\}$

4.3 Examples and Features

Take the example shown in Figure 1(a) which builds a doubly-linked list, reverses the list, and then traverses the reversed list through the $next$ link. The main traversal structure is determined by statement S25, and the statement that constructs this structure is S16, which in turn reverses a list defined by statement S4. Consequently, only the edges constructed by statements S4 and S16 will be shown on shape graphs explicitly and hence only these edges will participate in the process of deciding possible shapes of the constructed data structure. The shape graphs which will be generated when this algorithm is applied to the first loop of this example are listed in Figure 8.

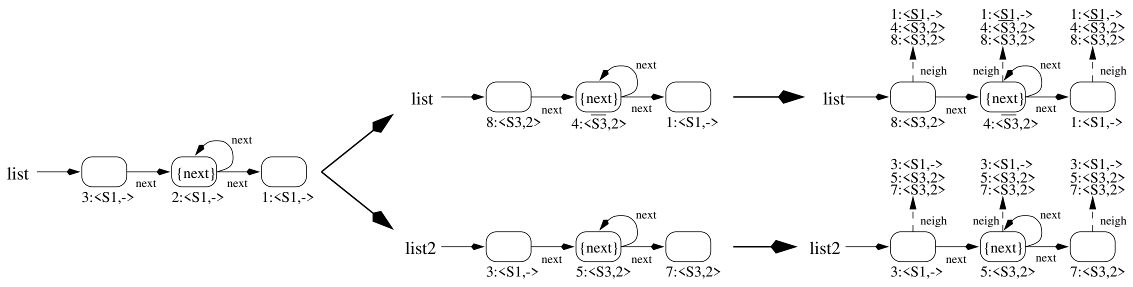
The same process will be applied to the second loop of the program, and the shape graphs are displayed in Figure 9. This process recognizes the linear traversal structure of the reversed list, and validates the results of dependence analysis phase on the program shown in Figure 1. Similarly, shape analysis can build shape graphs for the EM3D loop and the final graph will resemble the shape graph shown in Figure 7(c).



(a) Before S12 (b) After S16 (c) After S17

Figure 9: Shape Graphs Generated by Cyclic List Reverse Program

A similar example can be presented and compared with this cyclic list reverse loop to demonstrate the special ability of this approach. If a program splits a singly-linked (unshared) list into two lists and then converts both singly-linked lists into cyclic lists, this approach will be able to specify that the set of destinations specified by $neigh$ links of one list will be distinct from the set of the other list (see Figure 10).



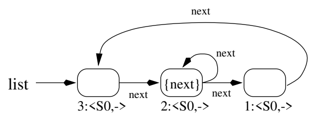
(a) Original List (b) Split into Two Lists (c) Convert to Cyclic Lists

Figure 10: Shape Graphs Generated by Program that Splits a List and Constructs Cyclic Lists

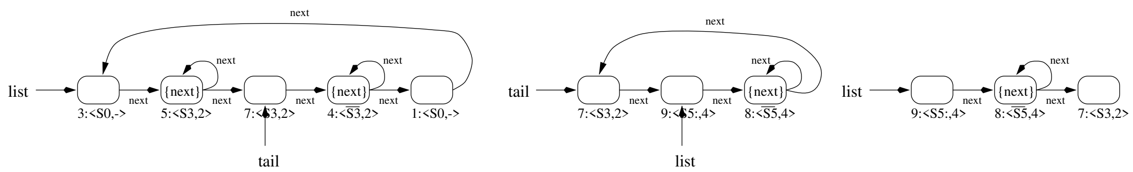
Another interesting feature of this shape analysis algorithm is its ability to detect that a circular list is broken into an unshared list, as the outcome of execution of the program shown in Figure 11(a). This program traverses the circular list and stops at a node when the condition is met. It then removes the $next$ link of the node pointed by $tail$ after forwarding $list$ to the next node. The result of this program execution is to turn a circular list into an unshared list. The shape graphs for this program, shown in Figure 11, demonstrate this ability.

```
{list is a circular list}
S1: tail = list
S2: do while (tail.vaule = ...)
S3:    tail = tail.next
S4: end do
S5: list = tail.next
S6: tail.next = nil
```

(a) Program



(b) Circular List Before S1



(c) After S4 (d) After S5 (e) Unshared List After S6

Figure 11: Shape Graphs Generated by Program that Breaks a Circular List

5 Related Work

Various dependence analysis techniques have been proposed for programs with pointers or dynamic recursive data structures [7, 8, 9, 10, 14]. Horwitz et al. developed an algorithm to determine dependence by detecting interferences in reaching stores [9], while Larus and Hilfinger proposed to identify access conflicts on alias graphs [14]. These methods build either stores or alias graphs to represent associations of pointers and storage locations, and the former imposes the k-limited rule to limit store sizes while the alias graphs of the latter method are not ideal for analyzing loop iterations. Furthermore, this store-based approach might miss the dependence between pointer variables. In contrast to these approaches, Hendren and Nicolau used path matrices to record connection information among pointers and presented a technique to recognize interferences between computations for programs with acyclic structures [8].

The approach proposed by Guarna, Jr. is similar to this work since it determines dependence between patterns of traversal using syntax tree matching [7]. However, multiple syntax trees would be generated for each pointer if aliases exist, and syntax trees might be complicated caused by structure traversal by sequences of statements. Unlike the above techniques, the method proposed by Hummel et al. relies on alias information provided by users [10]. Commutativity analysis proposed by Rinard and Diniz is another approach for parallelization [17]. It is designed for objected-based programs and can discover when operations of objects commute.

The distinct feature of this work is that conflict analysis is performed before alias/connection analysis phase. Each reference pattern can be represented by a path expression under the assumption that each unique path leads to a distinct location, and hence conflict analysis can be done by comparing path expressions. The outcome of conflict analysis will be forwarded to guide traversal-pattern-sensitive pointer analysis. This dependence analysis approach was briefly outlined in [11] and a simple shape analysis was also presented, which could only handle programs without destructive updating. This paper develops an algorithm to perform such dependence analysis. Furthermore, the shape analysis algorithm presented in this paper uses a different approach and can handle destructive updating such as list reverse operations.

The shape analysis algorithm is adapted from the approach proposed by Sagiv et al [18]. The main difference is its ability to represent the links of main traversal structures and to maintain the connection information of secondary traversal edges on the same shape graphs, and hence it is an ideal technique for traversal-pattern-sensitive shape analysis. Other proposed shape analysis methods can not handle destructive updating [1, 6, 16].

Symbolic path expressions have been proposed by other researchers [4, 14]. Larus and Hilfinger used path expressions to specify nodes in alias graphs [14], whereas Deutsch paired symbolic access paths to represent alias information between recursive data structures [4]. On the contrary, path expressions are used in this paper to specify traversal patterns. In other words, the difference is that their symbolic path expressions are decided by the statements that construct recursive data structures, whereas path expressions in this paper are defined by statements that perform structure traversal.

6 Summary

This paper presents a dependence analysis algorithm that first analyzes structure access conflicts between statements and then applies the traversal-pattern-sensitive shape analysis to confirm the results. The DEF/USE information is used to connect these two phases. This approach can identify parallelism from programs even with cyclic data structures.

References

[1] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *SIGPLAN Notices*, 25(6):296–310, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*.

[2] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, January 1993.

[3] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[4] Alain Deutsch. Interprocedural May-Alias analysis for pointers: Beyond k-limiting. *SIGPLAN Notices*, 29(6):230–241, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[5] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural Points-to analysis in the presence of function pointers. *SIGPLAN Notices*, 29(6):242–256, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[6] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Conference Record of POPL '96: 23nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15, St. Petersburg Beach, Florida, January 1996.

[7] Vincent A. Guarna, Jr. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proceedings of 1988 International Conference on Parallel Processing, Volume 2*, pages 212–220, August 1988.

[8] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[9] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. *SIGPLAN Notices*, 24(7):28–40, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.

[10] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. *SIGPLAN Notices*, 29(6):218–229, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[11] Yuan-Shin Hwang and Joel Saltz. Identifying DEF/USE information of statements that construct and traverse dynamic recursive data structures. In *Proceedings of 10th International Workshop on Languages and Compilers for Parallel Computing*, University of Minnesota, August 1997. Available via anonymous ftp directory hyena.cs.umd.edu:/pub/shin/papers/lcpc97.ps.gz.

[12] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Notices*, 27(7):235–248, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*.

[13] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural side effect analysis with pointer aliasing. *SIGPLAN Notices*, 28(6):56–67, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*.

[14] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. *SIGPLAN Notices*, 23(7):21–34, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[15] N.K. Madsen. Divergence preserving discrete surface integral methods for maxwel l's curl equations using non-orthogonal grids. Technical Report 92.04, RIACS, February 1992.

[16] J. Plevyak, A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 37–56, Portland, Oregon, August 1993. Lecture Notes in Computer Science, Vol. 768, Springer Verlag.

[17] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. *SIGPLAN Notices*, 31(5):54–67, May 1996. *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*.

[18] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of POPL '96: 23nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31, St. Petersburg Beach, Florida, January 1996.

[19] Reinhard von Hanxleden and Ken Kennedy. Give-N-Take – A balanced code placement framework. *SIGPLAN Notices*, 29(6):107–120, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[20] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. *SIGPLAN Notices*, 30(6):1–12, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*.