# Applying DEF/USE Information of Pointer Statements to Traversal-Pattern-Aware Pointer Analysis [*]

Yuan-Shin Hwang        Joel Saltz

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

July 19, 1997

**Abstract**

Pointer analysis is essential for optimizing and parallelizing compilers. It examines pointer assignment statements and estimates pointer-induced aliases among pointer variables or possible shapes of dynamic recursive data structures. However, previously proposed techniques are not able to gather useful information or have to give up further optimizations when overall recursive data structures appear to be cyclic even though patterns of traversal are linear. The reason is that these proposed techniques perform pointer analysis without the knowledge of traversal patterns of dynamic recursive data structures to be constructed. This paper proposes an approach, *traversal-pattern-aware pointer analysis*, that has the ability to first identify the structures specified by traversal patterns of programs from cyclic data structures and then perform analysis on the specified structures. This paper presents an algorithm to perform shape analysis on the structures specified by traversal patterns. The advantage of this approach is that if the specified structures are recognized to be acyclic, parallelization or optimizations can be applied even when overall data structures might be cyclic. The DEF/USE information of pointer statements is used to relate the identified traversal patterns to the pointer statements which build recursive data structures.

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

## 1 Introduction

Pointer analysis is essential for optimizing and parallelizing compilers that support languages with pointers like C and Fortran 90. There has been a considerable number of techniques being proposed in this field. Researchers have developed algorithms to detect pointer-induced aliases [3, 5, 6, 9, 15], analyze side effects [3, 10], and identify conflicts/interferences among statements by static analysis [8, 11]. The results of these analysis procedures can be supplied to compilers for optimizations and parallelization on programs with dynamic recursive data structures. The characteristic of these pointer analysis approaches is that they estimate the possible locations referenced by each pointer.

Shape analysis is another form of pointer analysis. It differs from previous methods by estimating the possible shapes of recursive data structures accessible from pointers [2, 7, 13, 14]. The shape information can be exploited to parallelize or optimize programs by providing compilers more insight into the properties of data structures used by programs. However, there are cases that even precise shape estimation does not provide useful information for parallelization or optimizations, especially for programs with cyclic data structures but with linear traversal patterns [7]. The main reason is because these proposed techniques are performed without taking account of traversal patterns on dynamic recursive data structures by programs. They might gather information that is inappropriate for parallelization and optimizations.

Although many programs create recursive data structures which appear to be cyclic overall, they usually follow linear structures to reference all nodes on the data structures. For instance, graph algorithms frequently extract linear structures, such as spanning trees, from cyclic graphs and traverse the graphs following the links of the linear structures, while the rest of edges are merely used to reference values on neighboring nodes. Furthermore, recursive data structures can have unbounded numbers of nodes and are commonly traversed by either iterative or recursive program constructs, such as loops or recursive functions. The edges along which the iterative or recursive program constructs traverse a recursive data structure constitute the *main traversal structure*, which can be viewed as the skeleton of the recursive structure and can be served as the way to represent traversal patterns of the program constructs. Hence, these edges can be called as the *main traversal edges*. On the other hand, the remaining edges of the recursive data structures are generally used to reference values on other nodes, and they will be called as *reference edges* or *secondary traversal edges*. Accordingly, it will be beneficial if compilers have the knowledge of which pointer statements build the main traversal structures when pointer analysis is performed.

One example is the bipartite graph shown in Figure 1(a) constructed by the program EM3D [12], which models the propagation of electromagnetic waves. It updates the values of E nodes (electric field) by a weighted sum of neighboring H nodes (magnetic field), and then H nodes are similarly updated using the E nodes. As a result, it traverses the list of E nodes through the solid (main traversal) links and reads information from H nodes by dashed (secondary traversal) links, and then similarly traverses the H node list and references E nodes. Although the bipartite graph is cyclic, the main traversal structures (lists of E and H nodes) are not. Another example is leaf-connected tree created by the Barnes-Hut N-Body solver [1], as depicted in Figure 1(b). The structures traversed by the program are a list and a tree, respectively.

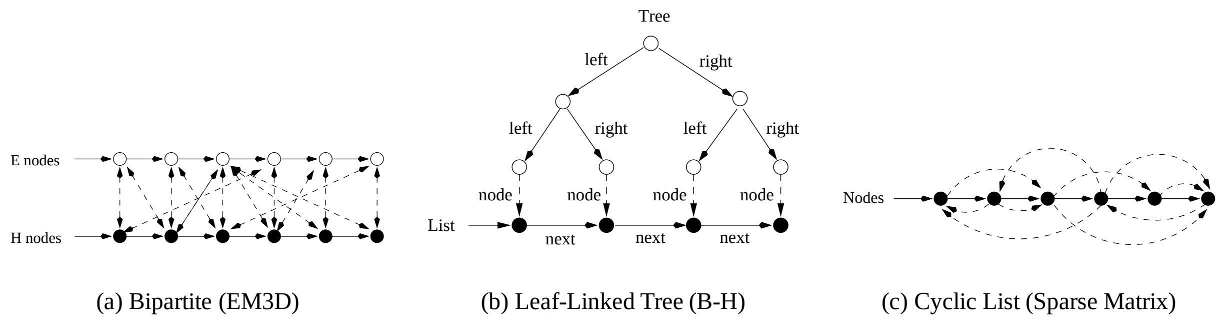


Figure 1: Cyclic Graphs with Acyclic Traversing Patterns

Figure 1(c) shows the cyclic structures used to represent sparse matrixes, which are common in programs that simulate interactions among entities. These type of cyclic data structures are common in computational scientific simulation programs. Most of these simulation programs exhibit high degree of parallelism since their main traversal structures are either lists or trees, even though the overall data structures appear to be cyclic.

These examples show that the knowledge of future traversal patterns can provide important hints for compilers to perform proper pointer analysis on programs even with cyclic recursive data structures. This paper proposes an approach to perform pointer analysis that utilizes the information of traversal patterns, called *traversal-pattern-aware* pointer analysis. The advantages of traversal-pattern-aware pointer analysis come from its ability to isolate the main traversal structures from overall (cyclic) graphs.

In order to perform pointer analysis with awareness of traversal patterns, the traversal patterns defined by iterative or recursive flow control constructs will be collected before pointer analysis is performed and the relationship between each statement that connects edges of recursive data structures and all corresponding traversal patterns must be identified. This paper proposes an algorithm to identify the traversal patterns of loops that reference recursive data structures and then propagate the information back to the statements that construct the structures. This process will establish the DEF/USE chains between statements that construct and traverse dynamic recursive data structures.

The traversal-pattern-aware pointer analysis algorithm presented in this paper is a shape analysis method, which estimates the possible shapes of the main traversal structures of dynamic recursive data structures, by utilizing this DEF/USE information. The advantage of this shape analysis approach is that it can identify the linear shapes from cyclic data structures if the patterns of traversal are acyclic. This information will facilitate compilers to perform optimizations or parallelization for programs even with cyclic recursive data structures. Consequently, a simple dependence test algorithm will also be presented in this paper to exploit the possibility of parallelization from this information.

The rest of this paper is organized as follows. Section 2 describes the programming model and background information of traversal-pattern-aware pointer analysis. Section 3 presents the algorithm to build DEF/USE chains of pointer statements. Section 4 describes algorithms to perform traversal-pattern-aware shape estimation and dependence test. Summary is presented in Section 5.

## 2 Background

This section describes the programs accepted by the algorithms proposed in this paper, and outlines the traversal-pattern-aware pointer analysis approach.

### 2.1 Programming Model

The algorithms presented in this paper are designed to analyzes 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 in languages such as C is 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. $p = q$
2. $p = q.n$
3. $p.n = q$

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 $p_i.m = q_j.n$ is also valid, it is represented by two consecutive statements, $t_k = q_j.n$ and $p_i.m = t_k$, for the reason of simplicity.

The DEF/USE chains will be constructed to identify the relationships between statements of second and third forms. The execution of a statement $p = q.n$ constitutes a *USE* of $q.n$, and it traverses the link from node $q$ to node $q.n$. On the other hand, executing a statement $p.n = q$ establishes a link from node $p$ to node $q$, and this link information is stored in the field $p.n$ and hence constitutes a *DEF*. Therefore, the second form of statements can be called *link traversing statements*, and the third can be called *link defining statements*. The first type of statements will not be part of DEF/USE chains since they merely introduce aliases among declared pointer variables and allocated locations (if right hand side is $new()$). No link traversal is performed by the first form of statements, and hence they can be named as *aliasing statements*.

### 2.2 Traversal-Pattern-Aware Pointer Analysis

Most pointer analysis algorithms have to give up further optimizations or parallelization when cyclic data structures are encountered, since conservative estimation on cyclic structures might not provide any useful information to compilers. However, with awareness of future traversal patterns, pointer analysis can still gather helpful hints for compilers to perform optimizations or parallelization even on programs with cyclic data structures. The knowledge of traversal patterns can direct compilers to focus analysis on statements that are crucial to optimizations or parallelization.

```
C    Build a doubly-linked list   C     Forward traverse the doubly-linked list
S1   list = new()                 S11   ptr = list.next
S2   do i = 1, N                  S12   do while (ptr.next)
S3      node = new()              S13      prv = ptr.prev
S4      node.value = ...          S14      ptr.result += prv.value
S5      node.next = list          S15      ptr = ptr.next
S6      list.prev = node          S16   end do
S7      list = node
S8   end do
```

Figure 2: Build and Traverse a Doubly Linked List

Consider the example in Figure 2. When the traversal pattern is unknown, pointer analysis has to be performed on all pointer statements, and the cycles connected by $next$ and $prev$ edges will force compilers to make conservative summary estimation. This information will not provide much help for compiler optimizations or parallelization because of properties of cyclic structures. However, if the traversal pattern of the second loop is identified before pointer analysis, this knowledge will instruct compilers to focus on analyzing the data structures connected by the $next$ links, while still keeping track of $prev$ edges. Compilers will be able to identify that the main traversal structure comprised of $next$ edges is a singly-linked list. Consequently, this knowledge enables compilers to exploit parallelism from a recursive data structure which appears to be cyclic overall.

```
C    {list is a cyclic list as Figure 1(c)}
C    Reverse the cyclic list        C     Traverse the reversed cyclic list
S1   rev-list = nil                 S11   ptr = rev-list
S2   do while (list)                S12   do while (ptr)
S3      tmp = rev-list              S13      neigh = ptr.neigh
S4      rev-list = list             S14      ptr.result += neigh.value
S5      list = list.next            S15      ptr = ptr.next
S6      rev-list.next = tmp         S16   end do
S7   end do
```

Figure 3: Reverse and then Traverse a Cyclic List

The awareness of future traversal patterns can simplify pointer analysis on programs with destructive updating as well. For instance, the program shown in Figure 3 reverses a cyclic list with the shape like Figure 1(c), and then traverses the list through the link $next$. Without the knowledge of this linear traversal pattern, pointer analysis on the loop that reverses the cyclic list will be difficult. On the other hand, a traversal-pattern-aware pointer analysis algorithm will be able to identify the linear main traversal structure and collect useful information for compilers with help of DEF/USE chains.

| Program | | SSA Representation | |
|---|---|---|---|
| S1 | ptr = list | S1 | ptr$_1$ = list$_1$ |
| S2 | do while (ptr.next) | S2 | ptr$_2$ = $\phi$ (ptr$_1$, ptr$_3$) |
| | | S2′ | do while (ptr$_2$.next) |
| S3 | ptr = ptr.next | S3 | ptr$_3$ = ptr$_2$.next |
| S4 | end do | S4 | end do |
| S5 | ptr.next = node | S5 | ptr$_2$.next = node$_1$ |

Figure 4: List Append Function and Its SSA Form

## 2.3 Intermediate Program Representation

Programs will be transformed into an SSA (Static Single Assignment) intermediate representation [4]. A program is defined to be in SSA form if, for every original variable $V$, trivial merging functions, $\phi$-functions, for $V$ have been inserted and each mention for $V$ has been changed to mention of a new name $V_i$ such that the following conditions hold:

- If a program flow graph node $Z$ is the first node common to two non-null paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$ that start at nodes $X$ and $Y$ containing assignments to $V$, then a $\phi$-function for $V$ has been inserted at $Z$.
- Each new name $V_i$ for $V$ is the target of exactly one assignment statement in the program.
- Along any program flow path, consider any use of a new name $V_i$ for $V$ (in the transformed program) and the corresponding use of $V$ (in the original program). Then $V$ and $V_i$ have the same value.

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 (location addresses) of pointer variables can be treated as values in regular variables. Once normalized programs are transformed into SSA representation, a new form of pointer assignments will be introduced:

$$p_i = \phi\,(p_j,\ p_k)$$

which will be placed at merging points of programs. For instance, Figure 4 presents a list append function and its SSA representation.

The process of traversal can be defined by the execution of series of link referencing statements. Furthermore, a new pointer instance is created when each link referencing statement is executed. Consequently, 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. Similarly, for nested loops that traverse lists of lists, the traversing patterns can be described as

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

where <*p, e, f,* –> represents the induction pointer of outer loop while <*p′, e′, f′, r′*> is the traversal patterns of inner loop.

## 3 Building DEF/USE Chains

This sections presents the algorithm to build DEF/USE chains between link defining statements and link traversing statements. It consists of two passes:

- Forward Pass
  Static aliases induced by aliasing statements are identified. These static aliases will not be affected by link defining or traversing statements.
  Path expression of each pointer variable will be specified also.

- Backward Pass
  Traversal patterns represented by path expressions are gathered and propagated to build DEF/USE chains between link defining and traversing statements.

The second pass conducts backward analysis to avoid constructing graphs that summarize connections of dynamic recursive data structures without the awareness of traversal patterns. Each traversal pattern is assumed performing references on a linear structure, either a list or a tree. During the process of analysis, multiple traversal patterns will reach the same locations if dynamic recursive data structures are either DAGs (directed acyclic graphs) or cyclic graphs.

### 3.1 Identifying Static Aliases and Path Expressions

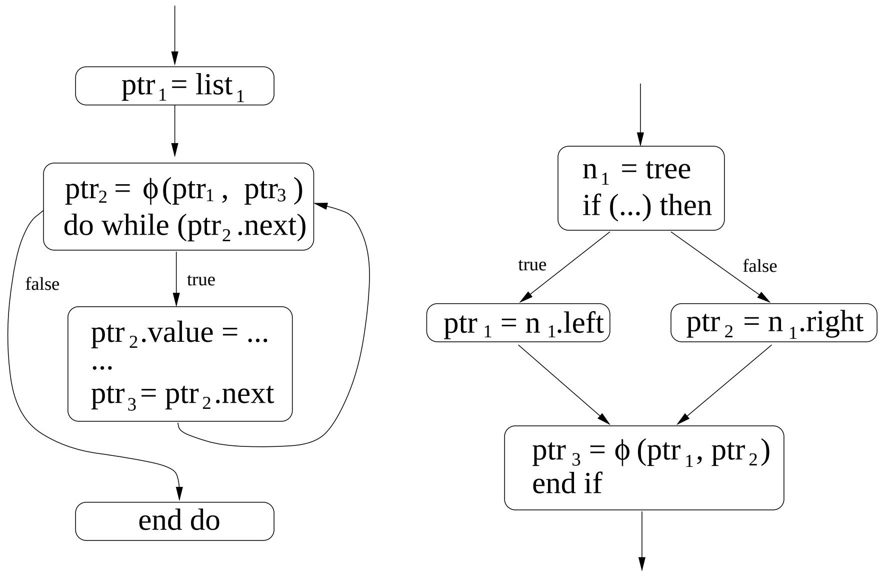
This phase performs forward analysis through the links of SSA representation to identify aliases among pointer variable instances and allocated locations, and to compute path expression of every pointer variable instance. The algorithm works as follows:

- Examine all pointer assignment statements. If a statement has the form $p_i = q_j$ and $q_j$ is either a storage allocation call $new()$ or $nil$, initialize the MustAlias set of $p_i$ to $\{q_j\}$ and set both MayAlias set and PathExpression set as $\emptyset$. All SSA edges with their source at this node will be added to the WorkList. For other statements, initialize MustAlias, MayAlias, and PathExpression sets to $\emptyset$, with the exception that the MustAlias sets of statements $p_i = \phi\,(p_j,\ p_k)$ are set as $U$ (i.e. all variables).

- The algorithm terminates when the WorkList becomes empty.
- An SSA edge is taken off from the WorkList. The sets at the definition end of the SSA edge are compared with those at the use end of the SSA edge.
- If the sets are different, update the sets at the use end of the SSA edge, compute new sets for the pointer variable at left-hand-side of the statement at the use end of the SSA edge based on the type of statement, and then add all SSA edges with their source at this node to the WorkList.
  - $p_i = q_j$
    $MustAlias_{p_i} = MustAlias_{q_j} \cup \{q_j\}$
    $MayAlias_{p_i} = MayAlias_{q_j}$
    $PathExpression_{p_i} = \{< p_i, e, f, r > \mid \forall < q_j, e, f, r > \in PathExpression_{q_j}\}$
  - $p_i = q_j.n$
    $MustAlias_{p_i} = \emptyset$
    $MayAlias_{p_i} = \emptyset$
    $PathExpression_{p_i} = \begin{array}{l} \{< p_i, q_j, -, n >\} \quad if\ PathExpression_{q_j} \equiv \emptyset \\ \{< p_i, e, -, r.n > \mid \forall < q_j, e, -, r > \in PathExpression_{q_j}\} \quad Otherwise \end{array}$
  - $p_i.n = q_j$
  - $p_i = \phi\,(p_j,\ p_k)$ (conditionals)
    $MustAlias_{p_i} = MustAlias_{p_j} \cap MustAlias_{p_k}$
    $MayAlias_{p_i} = MayAlias_{p_j} \cup MayAlias_{p_k}$
    $\cup\, (MustAlias_{p_j} \cup MustAlias_{p_k} - MustAlias_{p_i})$
    $PathExpression_{p_i} = PathExpression_{p_j} \uplus PathExpression_{p_k}$
  - $p_i = \phi\,(p_j,\ p_k)$ (loops: p$_j$ is from the entry edge and p$_k$ the iteration edge)
    $MustAlias_{p_i} = MustAlias_{p_j} \cap MustAlias_{p_k}$
    $MayAlias_{p_i} = MayAlias_{p_j} \cup MayAlias_{p_k}$
    $\cup\, (MustAlias_{p_j} \cup MustAlias_{p_k} - MustAlias_{p_i})$
    $PathExpression_{p_i} = \{< p_i, p_j, r, - > \mid \forall < p_k, p_i, -, r > \in PathExpression_{p_k}\}$
    $\cup\ \{< p_i, e, f, r > \mid \forall < p_k, \neg p_i, f, r > \in PathExpression_{p_k}\}$

The special union operator $\uplus$ is to calculate path expressions of pointer instances after merging by $\phi$-functions at end of conditionals. It first examines the entry points of each pair of path expressions from the Reference sets of both operands of a $\phi$-functions. If the entries are identical, the operator will create a new path expression with the same entry point such that relative path is the union of relative paths from both path expressions. For instance, the result of $\{$<*p$_j$, node, –, left*>$\}$ $\uplus$ $\{$<*p$_k$, node, –, right*>$\}$ will be $\{$<*p$_i$, node, –, left$|$right*>$\}$. Otherwise, the operator simply performs set union operations.

The identification of induction pointers is performed only on the statements with $\phi$-functions at the headers of loops. A pointer instance at definition side of a statement with a $\phi$-function represents an induction pointer in the original program when the entry of its path expression is itself before the execution of the statement. Furthermore, the relative path of the path expression is the path that the iteration pointer will advance at each iteration. In other words, if the path expression of $p_k$ of the statement $p_i = \phi\,(p_j,\ p_k)$ has the form <$p_k$, $p_i$, –, $r$>, then $p_i$ is the instance of an induction pointer and its path expression will be <$p_i$, $p_j$, $r$, –>. It means the induction pointer will start from $p_j$, and advance along the structure by path $r$ at every iteration.



(a) Loop (b) Conditional

Figure 5: Basic Blocks in CFG

## 3.2 Building DEF/USE Chains

This phase performs backward analysis to gather traversal patterns and propagate this information back to link defining statements. The process will construct the DEF/USE chains between link defining statements and link referencing statements. The analysis is operated on CFG: $[V_{CFG},\ E_{CFG}]$, where $V_{CFG}$ is the set of basic blocks of CFG and $E_{CFG}$ is the set of edges. Statements will be grouped into basic blocks, while statements with $\phi$-functions will be placed in the header nodes of loops or merging nodes of conditionals, as shown in Figure 5.

The algorithm works as follows:

- Initialize all sets of nodes and StatList to $\emptyset$ and
  set WorkList = $\{B \mid B \in V_{CFG} \wedge$ <$B$, $EXIT$> $\in E_{CFG}\}$.

- The algorithm terminates when the both WorkList and StatList become empty. Execution may proceed by processing items from either list. If any new sets of any statements in the block are different from old sets, then add all preceding basic blocks to the WorkList.

- If a basic block is taken off from the WorkList. Compute new sets of every statement in the basic block based on the statement types. [1]

  - $S: p_i = q_j$
    $Reference_S = Reference_{in} - Reference_{p_i} \cup Reference_{q_j}$
    $where\ \ Reference_{p_i} = \{< s, p, e, n > \mid < s, p, e, n > \in Reference_{in} \ \wedge E(e) \equiv p_i\}$
    $\qquad Reference_{q_j} = \{< s, p, e', n > \mid < s, p, e, n > \in Reference_{p_i} \wedge e' = e : E(e) \leftarrow q_j\}$
    $Define_S = Define_{in}$
  - $S: p_i = q_j.n$
    $Reference_S = Reference_{in} \cup \{< S, q_j, P(q_j), n >\}$
    $Define_S = Define_{in}$
  - $S: p_i.n = q_j$
    $Reference_S = Reference_{in} - Reference_{p_i} - Reference_{prefix} \cup Reference_{suffix}$
    $Use_S = Use_S \cup \{s \mid < s, p, e, n > \in (Reference_{p_i} \cup Reference_{may} \cup Reference_{path})\}$
    $where\ \ Reference_{p_i} = \{< s, p, e, n > \mid < s, p, e, n > \in Reference_{in} \wedge (p \equiv p_i \vee e \in MustAlias(p_i))\}$
    $$Reference_{may} = \left\{ < s, p, e, n > \;\middle|\; \begin{array}{l} < s, p, e, n > \in Reference_{in} \wedge \\ (\exists a \in MayAlias(E(e)) \ni p_i \in D(a.SX(e)) \end{array} \right\}$$
    $$Reference_{prefix} = \left\{ < s, p_i, e, n > \;\middle|\; \begin{array}{l} < s, p_i, e, n > \in Reference_{in} \wedge \\ PX(e) \equiv a.n \wedge a \in MustAlias(p_i) \end{array} \right\}$$
    $$Reference_{suffix} = \left\{ < s, p, e_s, n > \;\middle|\; \begin{array}{l} < s, p, e, n > \in Reference_{prefix} \wedge \\ e \equiv a.n.SX(e) \wedge e_s = q_j.SX(e) \end{array} \right\}$$
    $\qquad Reference_{path} = \{< s, p_i, e, n > \mid < s, p_i, e, n > \in Reference_{in} \wedge p_i \in D(e)\}$
    $Define_S = Define_{in} \cup \{< s, p_i, n >\}$
    $StatList = \ StatList \cup \{< s, q_j, p_i.n > \mid < s, p, t > \in Define_{in} \wedge P(q_j) \equiv PX(P(p))\}$
    $\qquad \cup \{< s, p_i.n, q_j > \mid < s, p, t > \in Define_{in} \wedge P(p_i.n) \equiv PX(P(p))\}$
  - $S: p_i = \phi(p_j, p_k)$
    $Reference_{S_{p_j}} = Reference_{in} - Reference_{p_i} \cup Reference_{p_j}$
    $Reference_{S_{p_k}} = Reference_{in} - Reference_{p_i} \cup Reference_{p_k}$
    $where\ \ Reference_{p_i} = \{< s, p, e, n > \mid < s, p, e, n > \in Reference_{in} \wedge E(e) \equiv p_i\}$
    $\qquad Reference_{p_j} = \{< s, p, e', n > \mid < s, p, e, n > \in Reference_{p_i} \wedge e' = e : E(e) \leftarrow p_j\}$
    $\qquad Reference_{p_k} = \{< s, p, e', n > \mid < s, p, e, n > \in Reference_{p_i} \wedge e' = e : E(e) \leftarrow p_k\}$

- If the item is an element <*S, x, y*> from StatList, compute new sets for the statement *S:* $p_i.n = q_j$.
  $Reference_y = \{< s, p, e, n > \mid < s, p, e, n > \in References_S \wedge PX(e) \equiv P(y)\}$
  $Reference_{x \leftarrow y} = \{< s, p, e', n > \mid < s, p, e, n > \in Reference_y \wedge PX(e') = PX(e) : P(y) \leftarrow P(x)\}$
  $Reference_{p_i} = \{< s, p, e, n > \mid < s, p, e, n > \in Reference_{x \leftarrow y} \wedge p_i \in D(e)\}$
  $Use_S = Use_S \cup \{s \mid < s, p, e, n > \in Reference_{p_i}\}$
  $StatList = \ StatList \cup \{< s, q_j, e.n > \mid < s, p, t > \in Define_S \wedge P(q_j) \equiv PX(P(p))\}$
  $\qquad \cup \{< s, e.n, q_j > \mid < s, p, t > \in Define_S \wedge e.n \equiv PX(P(p))\}$
  $References_S = References_S \cup \{< s, p, e_s, n > \mid < s, p, e, n > \in Reference_{p_i} \wedge e_s = q_j.SX(e)\}$

[1] Abbreviations: E = ENTRY, P = PATH, D = DESTINATION, SX = SUFFIX, and PX = PREFIX

## 4 Traversal-Pattern-Aware Pointer Analysis

This section presents an algorithm to perform traversal-pattern-aware shape analysis and outlines the approach to apply results of shape analysis to dependence test.

### 4.1 Shape Analysis

This algorithm is adapted from the shape analysis algorithm proposed by Sagiv et al [14]. 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.

This algorithm has two phases. First phase gathers traversal patterns and builds the DEF/USE chains. The technique to perform the first phase has been presented in Section 3. The remaining of this section will present the second phase of this shape analysis algorithm.

#### 4.1.1 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. [2], the Abstract Storage Graph (ASG) by Plevyak et al. [13], and Shape-Graphs by Sagiv et al. [14]

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* [14]). 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.

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 6 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

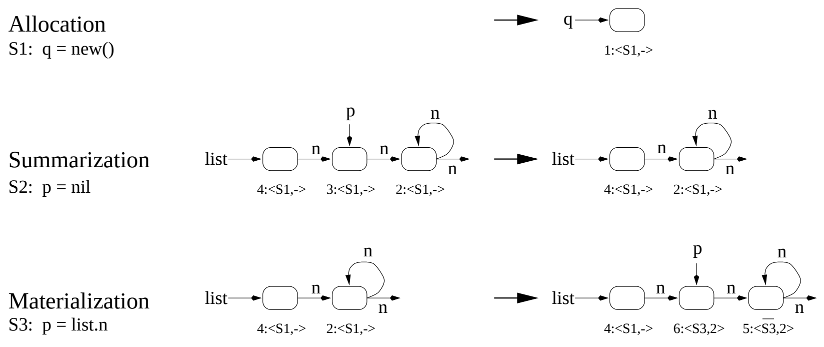


Figure 6: Storage Nodes and Tags

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 7(a) represents an unshared list specified by the path $(n)^\star$. Similarly, the summary node with tag $\{l, r\}$ of Figure 7(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 7(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 7(d) means multiple paths of the form $(l|r)^\star$ might reach the same node.

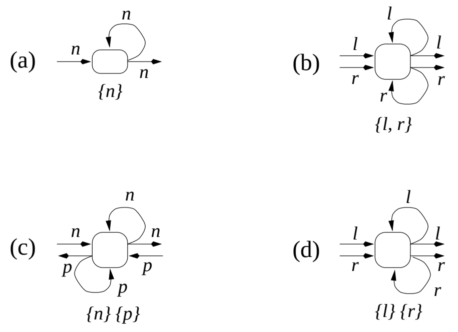


Figure 7: Sharing Information of Summary Nodes

Tail nodes of recursive data structures usually are not summarized into summary nodes to distinguish cyclic structures from acyclic structures. Figure 8 depicts the shape graphs that characterize different types of data structures. The difference between the tail nodes of Figure 8(a) and Figure 8(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 8(f).

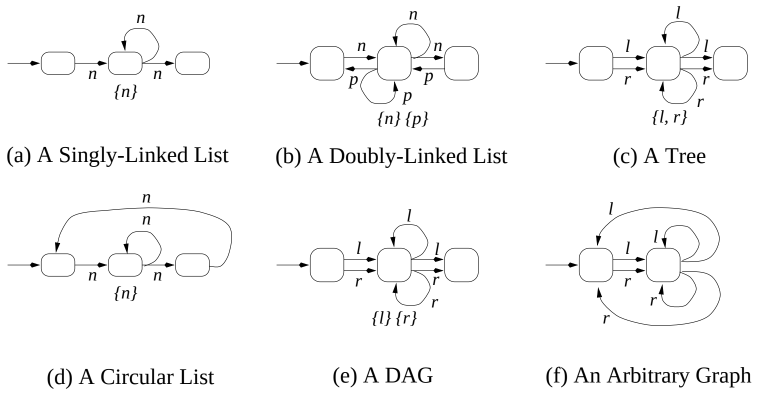


Figure 8: Data Structures Represented by Shape Graphs

The edges presented on shape graphs are used to symbolize the links of main traversal structures. The secondary traversal links are not represented as edges on shape graphs, but the connections caused by these links are described by node tags. Figure 9 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.

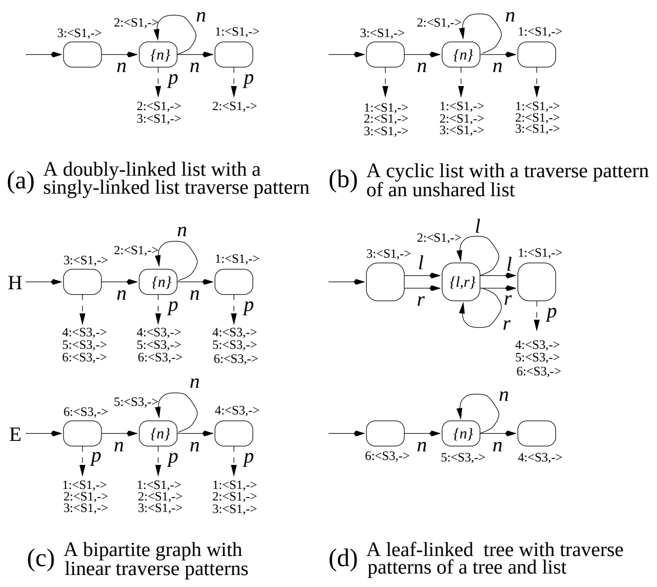

1:<S1,->
2:<S1,->
3:<S1,->
4:<S3,->
5:<S3,->
6:<S3,->
{n}
{l,r}
H
E
(a) A doubly-linked list with a singly-linked list traverse pattern
(b) A cyclic list with a traverse pattern of an unshared list
(c) A bipartite graph with linear traverse patterns
(d) A leaf-linked tree with traverse patterns of a tree and list

Figure 9: 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.

### 4.1.2 Algorithm

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

- The algorithm presented in the previous section is performed to establish the DEF/USE relationships of pointer statements. The traversal patterns by iterative or recursive program constructs will be gathered during the DEF/USE construction process.
- 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.
  - $S : p_i.n = q_j$
    $E_f = E_{f_{in}} \cup \{[s, n, t] | [p_i, s] \in E_{p_{in}} \wedge [q_j, t] \in E_{p_{in}}\}$
  - $S : p_i = \phi(p_j, p_k)$
    $V = V_{in_j} \cup V_{in_k}$
    $E_i = E_{in_j} \cup E_{in_k}$
    $E_p = E_i \cup \{[p_i, s] | [p_j, s] \in E_i\} \cup \{[p_i, s] | [p_k, s] \in E_i\} - \{[p_j, \star]\} - \{[p_k, \star]\}$

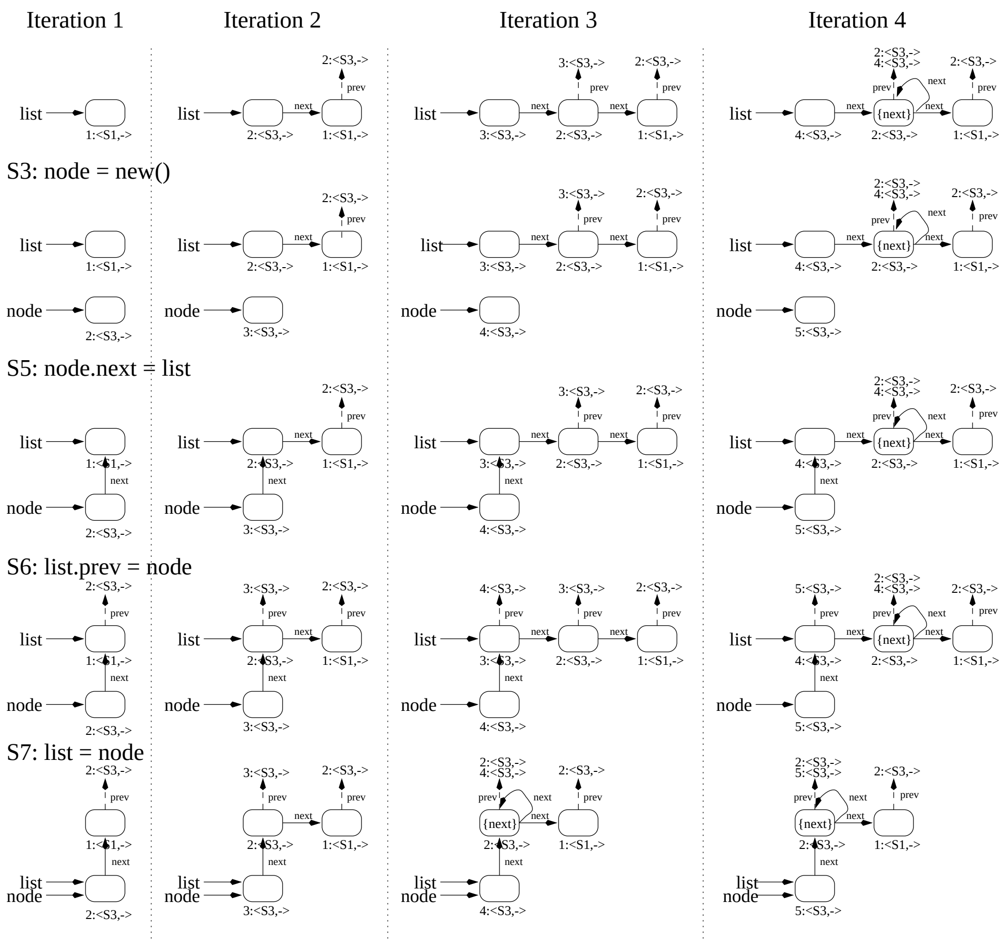
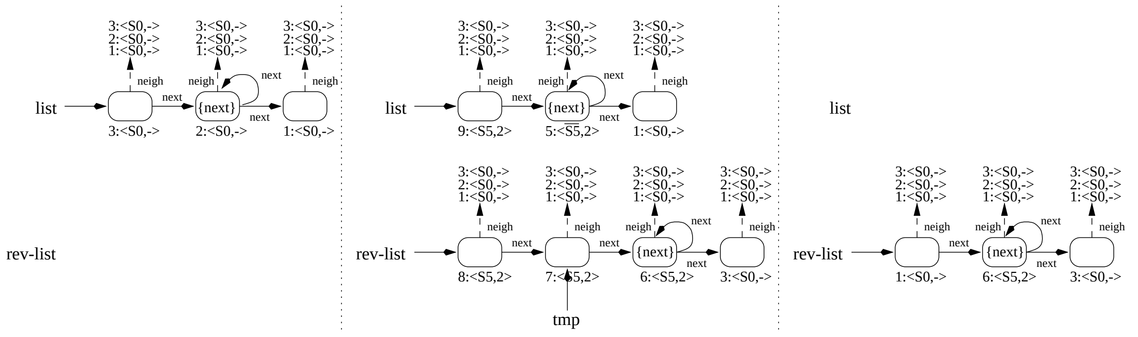


Figure 10: Shape Graphs Generated by Doubly-Linked List Building Program

### 4.1.3 Examples and Features

Take the example shown in Figure 2 which builds a doubly-linked list and forward traverses the list through the $next$ link. The main traverse structure is determined by statements S11 and S15, and the statement that constructs this structure is S5. This fact will be identified by the first phase of the algorithm for traversal-pattern-aware shape analysis. Consequently, only the edges constructed by statement S5 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 the second phase of algorithm is applied to this example are listed in Figure 10. This process estimates that the shape of main traversal structure is a singly-linked list.

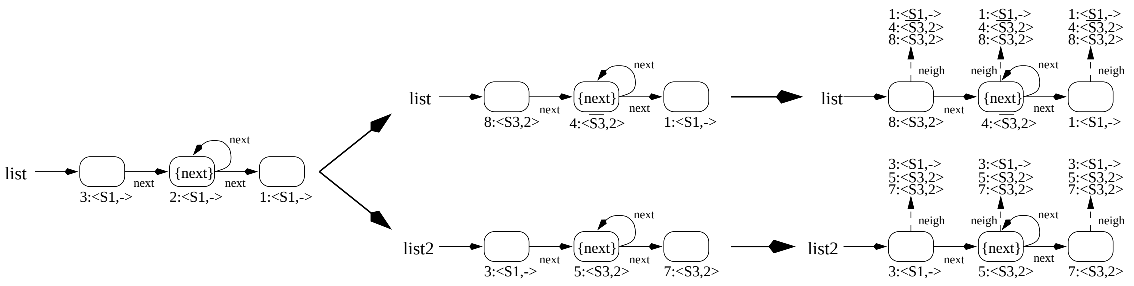
The same process can be applied to the program that reverses a cyclic list and then traverses the reversed list, as shown in Figure 3. The algorithm recognizes the linear traversal structure of the cyclic list and performs shape analysis on the cyclic list reverse program. Note that the $neigh$ links do not change during the course of analysis process. The destinations of the these links can be recognized through the tags of storage nodes. For instance, the tags of summary nodes in Figure 11(b), *6:<S5,2>* and *5:<$\overline{S5}$,2>*, mean they represent sets of storage locations retrieved from the summary node *2:<S0,–>*. This message reflects the fact that both sets of storage locations are the possible destination of any $neigh$ link.



(a) Before S2 (b) After S6 (c) After S7

Figure 11: Shape Graphs Generated by Cyclic List Reverse Program

A similar example can be presented and compared with this cyclic list reverse program 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 12).



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

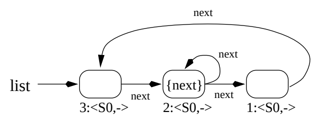
Figure 12: Shape Graphs Generated by Program that Splits a List and Constructs Cyclic Lists

One feature of this algorithm is that shape graphs will not be computed for certain statements in programs if these statements can be identified that they will not affect the shapes of constructed recursive data structures. Typical examples are the loops that only traverse data structures. They only reference nodes of recursive data structures and will not cause any changes in the shapes of data structures. Therefore, the construction of shape graphs for these type of statements will not affect the outcome of shape analysis. The statements which have no effects on shape analysis will be identified during the first phase of traversal-pattern-aware pointer analysis.

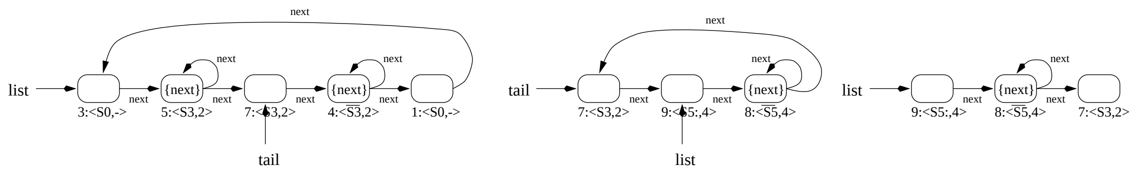
Another interesting feature of this shape analysis algorithm is its ability to detect that a circular list is broken into an unshared list, as the outcome of execution of the program shown in Figure 13(a). This program traverse the circular list and stops at a node when the condition is met. It then 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 13, 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 13: Shape Graphs Generated by Program that Breaks a Circular List

## 4.2 Dependence Test

The results of traversal-pattern-aware shape analysis can be applied to dependence test because the properties of main traversal structures defined by iterative or recursive program constructs can be inferred. Access conflict analysis is first conducted for each instance of iterative or recursive traversal on the assumption that each unique path expression leads to a distinct location. The rationale is that if any loops or recursive functions are deemed to have dependent iterations, they are inherently sequential no matter if the actual data structures contain cycles or not. Once the access conflict analysis reports that loops or recursive functions are not inherently sequential, traversal-pattern-aware shape analysis will be performed to confirm the results.

Consider the loop S11 of the program in Figure 2. At each iteration this loop reads from and writes to the same location of a node on the main traversal structure, and references a value from the neighboring node. The result of analysis shows that the iterations of this loop can be executed independently. However, if the statement S14 is replaced by

```
S14   ptr.result += prv.result
```

then iterations of the loop will not be independent.

## 5 Summary

This paper presents algorithms to apply the DEF/USE information to traversal-pattern-aware pointer analysis. Instead of performing analysis on the whole recursive data structures, this technique can first identify structures specified by traversal patterns and then perform analysis on the structures. This approach can perform pointer analysis on programs even with cyclic data structures.

## References

[1] J. Barnes and P. Hut. A hierarchical O(NlogN) force-calculation algorithm. *Nature*, pages 446–449, December 1976.

[2] 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*.

[3] 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.

[4] 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.

[5] 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*.

[6] 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*.

[7] 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.

[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] 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*.

[10] 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*.

[11] 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*.

[12] 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.

[13] 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.

[14] 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.

[15] 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*.