Identifying DEF/USE Information of Statements that Construct and Traverse Dynamic Recursive Data Structures *

Yuan-Shin Hwang and Joel Saltz

Department of Computer Science, University of Maryland, College Park, MD 20742

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 perform pointer analysis without the knowledge of traversal patterns of dynamic recursive data structures to be constructed. This paper presents an algorithm to identify the traversal patterns of recursive data structures and propagate this information back to those statements that define the data structures. This approach recognizes the DEF/USE relationships between the statements that define and traverse dynamic recursive data structures. The outcome of this technique will be useful for pointer analysis and parallelization. Algorithms to perform pointer analysis and dependence test using the knowledge of traversal patterns will also be presented.

1 Introduction

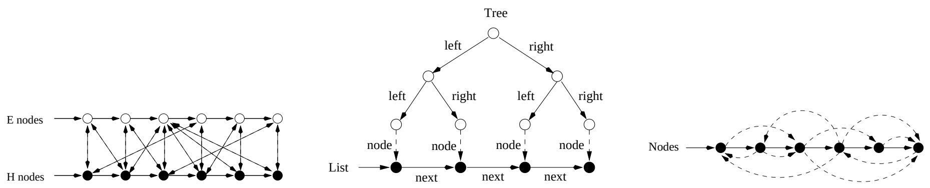
Pointer analysis is essential for optimizing and parallelizing compilers that support languages with pointers like C and Fortran 90. There has been a considerable number of techniques being proposed in this field. Researchers have developed algorithms to detect pointer-induced aliases [3, 5, 6, 9, 17], 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, 15, 16]. The shape information can be exploited to parallelize or optimize programs by providing compilers more insights 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 programs with cyclically linked data structures [7]. The main reason is because these proposed shape analysis techniques are performed

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

without taking account of traversal patterns on recursive data structures by programs. They might gather information that is inappropriate for parallelization and desired optimizations.

Although many programs create data structures which are 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 specify neighboring nodes. Therefore, this paper classifies the links of data structures into two groups based on the way they are referenced and traversed by programs. *Traversing links* are the edges that programs use to advance from nodes to nodes of recursive data structures, whereas *referencing links* are the connections that are used to reference values of neighboring nodes. The structures connected only by traversing links can be called *traversing structures*. Although lots of programs create data structures with cyclic links, most of them have acyclic traversing structures. It is usually referencing links that cause graphs to be cyclic and make shape information unable to provide hints for parallelism exploitation.



(a) Bipartite(EM3D)　　(b) DAG (B-H)　　(c) Cyclic List

Fig. 1. Complex Graphs with Acyclic Traversal Patterns

One example is the bipartite graph shown in Figure 1(a) constructed by the program EM3D [13], 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 traversing (solid) links and reads information from H nodes via referencing (dashed) links, and then similarly traverses the H node list and references E nodes. Although the bipartite graph is cyclic, the traversal patterns (on 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 connected by traversing links are a list and a tree, respectively. Figure 1(c) shows the cyclic structures used to represent sparse matrixes, which are frequently used 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 traversing 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 hints for compilers to perform proper pointer analysis on statements that construct dynamic recursive data structures. Therefore, the relationships between the statements that define and traverse dynamic recursive data structures respectively must be established. This paper proposes an algorithm to identify this DEF/USE information by gathering the traversal patterns of statements that reference recursive data structures and then propagating the information back to link defining statements.

Algorithms to use this DEF/USE information to perform pointer analysis and dependence test will also be presented. One special feature of the dependence test algorithm is that it first performs analysis to determine if loop iterations are inherently dependent by assuming every unique path will lead to a distinct node. If loops are identified as dependent, then pointer analysis on the statements that define the data structures referenced by these loop will not be necessary since no parallelization can be done on these loops no matter what information is gathered by pointer analysis. In other words, DEF/USE information can not only direct compilers to identify appropriate information from pointer analysis, but also avoid performing unnecessary pointer analysis.

The rest of this paper is organized as follows. Section 2 describes the intermediate representation of programs. Section 3 presents the algorithm to gather traversal patterns and propagate DEF/USE information. Section 4 outlines the approaches to use the DEF/USE information to perform pointer analysis and dependence test. Summary is presented in Section 6.

2 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, ϕ-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 ϕ-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.

This SSA transformation is designed specially for programs with fixed-location variables only, e.g. Fortran-77 programs, since any definitions to a variable will create a new value in one location without affecting the existing values of any other locations. It does not apply for programs with pointers because every pointer can be dynamically assigned to any locations. Definitions via a pointer would potentially change any existing values. However, same transformation can be performed on pointers if pointer variables are treated simply as regular variables.

For programs with pointers, assignment statements can be categorized into two classes: pointer assignment statements and value assignment statements. *Pointer assignment statements* define the associations of pointer variables, that is, each pointer assignment redirects a pointer variable to a certain location in program address space. On the other hand, *value assignments statements* assign values to the locations corresponding to the referenced variables. Although it seems that these two classes of assignment statements work in totally different ways, an observation is that once all addresses of storage locations are treated as values, the pointer assignment statements work in exactly the same way as value assignment statements. Consequently, pointer assignment statements can be transformed into SSA representations just as value assignment statements. For instance, Figure 2 presents a list append function and its SSA representation. Note that this paper only considers pointer assignment statements and hence only SSA form of pointer assignment statements will be displayed.

Program	SSA Representation
S1 ptr = list	S1 $\text{ptr}_1 = \text{list}_1$
S2 do while (ptr.next)	S2 do while (ptr_2.next)
	S2′ $\text{ptr}_2 = \phi\ (\text{ptr}_1, \text{ptr}_3)$
S3 ptr = ptr.next	S3 $\text{ptr}_3 = \text{ptr}_2$.next
S4 end do	S4 end do
S5 ptr.next = node	S5 ptr_2.next = node_1

Fig. 2. List Append Function and Its SSA Form

Every statement is 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, the only possible types of pointer assignment statements in pointer SSA form are

1. $p_i.n = q_j$
2. $p_i = q_j.n$
3. $p_i = q_j$
4. $p_i = \phi\ (p_j,\ p_k)$

Although the other type, $p_i.n = q_j.l$, is valid, it is represented by two consecutive statements, $t_k = q_j.l$ and $p_i.n = t_k$.

Among these types of pointer assignment statements, only $p_i.n = q_j$ will change connections of dynamic recursive data structures, and hence can be called as *link defining statements*. In contrast, the rest types of pointer assignment statements only cause aliases without modifying any connections of dynamic data structures. They can be called as *link referencing statements*, and are commonly used to traverse and reference elements of dynamic recursive data structures.

The process of structure traversal can be defined by the execution of series of link referencing statements. Furthermore, a new pointer instance is created when each these 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, 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

$$p:\ e(.f)^{\star}.r$$

Similarly, for nested loops that traverse lists of lists, the traversal patterns can be described as

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

where $< p,\ e,\ f,\ - >$ represents the induction pointer of the outer loop while $< p\,',\ e\,',\ f\,',\ r\,' >$ is the traversal patterns of the inner loop. Note that the link defining statements play no role in defining path expressions.

3 Identifying DEF/USE Information

This section presents an algorithm to identify the traversal patterns of the loops that reference the recursive data structures, and then propagate this information back to the statements that construct and/or modify the data structures. The results of this operations will be the DEF/USE relationships between statements that construct and traverse dynamic recursive data structures respectively. The algorithm can be broken into the following steps:

- Perform forward analysis following the links of SSA form to gather aliases among pointer variable instances and identify path expression of each pointer variable instance. The induction pointers will be recognized during this process. Note that link defining statements will not be examined at this stage.
- Gather traversal patterns and propagate the information back to link defining statements by performing backward analysis through the edges of CFG (control flow graph), and construct DEF/USE chains between link defining statements and link referencing statements.

After this process is performed, every pointer assignment statement that defines or modifies the connections of dynamic recursive data structures will have the knowledge of the set of statements that will traverse the data structures. The program represented in SSA form shown in Figure 3 will be used as an example to demonstrate the forward and backward analysis processes, which will be described in the rest of this section.

C	Build a doubly-linked list	C	Forward traverse the doubly-linked list
S1	list$_1$ = new()	S11	ptr$_1$ = list$_2$.next
S2	do while (...)	S12	do while (ptr$_2$.next)
S2$'$	list$_2$ = ϕ (list$_1$, list$_3$)	S12$'$	ptr$_2$ = ϕ (ptr$_1$, ptr$_3$)
S3	node$_1$ = new()	S13	prv$_1$ = ptr$_2$.prev
S4	node$_1$.value = ...	S14	ptr$_2$.result += prv$_1$.value
S5	node$_1$.next = list$_2$	S15	ptr$_3$ = ptr$_2$.next
S6	list$_2$.prev = node$_1$	S16	end do
S7	list$_3$ = node$_1$		
S8	end do		

Fig. 3. Build and Forward Traverse a Doubly-Linked List

3.1 Identifying 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 determine path expression of every pointer variable instance. The algorithm works as follows:

- Examine all link referencing statements. If a statement has the form $p_i = q_j$ and q_j is either *nil* or *new*(), then initialize the MustAlias set of p_i to $\{q_j\}$ and set both MayAlias set and PathExpression set as $\emptyset$. All the SSA edges with their source at this node will be added to the WorkList. For other statements, initialize sets MustAlias, MayAlias, and PathExpression 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.

- $p_i = q_j$
  $MustAlias_{p_i} = MustAlias_{q_j} \cup \{q_j\}$
  $MayAlias_{p_i} = MayAlias_{q_j}$
  $PathExpression_{p_i} = \{< p_i, e, f, r > \mid \forall < q_j, e, f, r > \in PathExpression_{q_j}\}$
- $p_i = q_j.n$
  $MustAlias_{p_i} = \emptyset$
  $MayAlias_{p_i} = \emptyset$
  $$PathExpression_{p_i} = \begin{array}{l} \{< p_i, q_j, -, n >\} \quad if\ PathExpression_{q_j} \equiv \emptyset \\ \{< p_i, e, -, r.n > \mid \forall < q_j, e, -, r > \in PathExpression_{q_j}\} \end{array}$$
- $p_i.n = q_j$
- $p_i = \phi\ (p_j,\ p_k)$ (conditionals)
  $MustAlias_{p_i} = MustAlias_{p_j} \cap MustAlias_{p_k}$
  $$\begin{aligned} MayAlias_{p_i} = {} & MayAlias_{p_j} \cup MayAlias_{p_k} \cup \\ & (MustAlias_{p_j} \cup MustAlias_{p_k} - MustAlias_{p_i}) \end{aligned}$$
  $PathExpression_{p_i} = PathExpression_{p_j} \uplus PathExpression_{p_k}$
- $p_i = \phi\ (p_j,\ p_k)$ (loops: p_j is from the entry edge and p_k from the iteration edge)
  $MustAlias_{p_i} = MustAlias_{p_j} \cap MustAlias_{p_k}$
  $$\begin{aligned} MayAlias_{p_i} = {} & MayAlias_{p_j} \cup MayAlias_{p_k} \cup \\ & (MustAlias_{p_j} \cup MustAlias_{p_k} - MustAlias_{p_i}) \end{aligned}$$
  $$\begin{aligned} PathExpression_{p_i} = {} & \{< p_i, p_j, r, - > \mid \forall < p_k, p_i, -, r > \in PathExpression_{p_k}\} \\ & \cup\ \{< p_i, e, f, r > \mid \forall < p_k, \neg p_i, f, r > \in PathExpression_{p_k}\} \end{aligned}$$

The special union operator $\uplus$ is to calculate path expressions of pointer instances after merging by ϕ-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 ϕ-functions. If the entries of path expressions 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 ϕ-functions at the headers of loops. A pointer instance at definition side of a statement with a ϕ-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 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.

Consider the example in Figure 3. The instances of pointer variables *list* and *node* do not have path expressions since they are not results of traversal through any links. On the other hand, instances of *ptr* and *prv* are the consequences of traversing along the links *next* and *prev*, and hence their path expressions are $< ptr_1, list_2, -, next >$, $< ptr_2, ptr_1, next, - >$, $< ptr_3, ptr_2, -, next >$, and $< prv_1, ptr_2, -, prev >$, respectively.

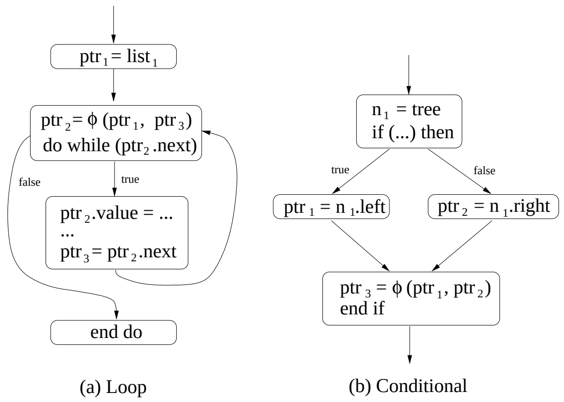


Fig. 4. Basic Blocks in CFG

3.2 Propagating Traversal Patterns back to Link Defining Statements

This phase performs backward analysis to gather traverse 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 ϕ-functions will be placed in the header nodes of loops or merging nodes of conditionals, as shown in Figure 4. The algorithm works as follows:

- Initialize sets *Define*, *Use*, and *Reference* of each node to $\emptyset$. Note that *Define* and *Use* will only be modified by statement $p_i.n = q_j$. Other statement types will simply copy from incoming sets.
- The iterative algorithm terminates when no new elements are introduced to any sets. At each iteration, basic blocks are examined in reverse topological order, and all tuples in *WorkList* will be examined once.
- Compute new sets of every statement in the basic block based on the statement type.
  - $S : p_i = q_j.n$
    If tuple $< S, q_j, n >$, where s is the program point of this statement, and q_j and n represent the traversal pattern $q_j.n$, is not in *Reference*, create one and inset it to the set.
    Examine each tuple $< s, p, f >$ in *Reference*. If p $\equiv$ p$_i$.e, where e is a sequence of field names, replace $p_i.e$ by $q_j.n.e$.

- $S : p_i = q_j$
  Examine each tuple $< s, p, n >$ in the set Reference. If the entry of its path expression starts from p_i, then replace p_i by q_j.
- $S : p_i.n = q_j$
  Examine each tuple $< s, p, n >$ in the set *Reference*, and perform operations if it meets one of the following conditions. If $p \equiv p_i$, then insert s to the set *Use* of this statement and remove the tuple from *Reference*. If p and p_i are may-aliases, insert s to *Use*. If path expression e reaches p_i, insert s to *Use*. If p passes $p_i.n$ (i.e. $p \equiv$ path(p_i).n.e$'$), replace p by $q_j.e'$.
  If $< S, p_i, n >$ is not in *Define*, create one. Examine each type $< s, p, f >$ in *Define*, if q_j is prefix of p, add $< s, p_i.n, q_j >$ to WorkList.
- $S : p_i = \phi(p_j, p_k)$
  Examine each tuple $< s, p, n >$ in the set *Reference*. Create a new set Reference$_{p_j}$ by replacing entry point of e by p_j if the entry is p_i or simply copying the tuple otherwise. Similarly, set Reference$_{p_k}$ will be created. Note that if S is a ϕ-statement at the header of a loop and p_i is an induction pointer, then p will be replaced by the path expression of p_i.

– For each tuple $< S, x, y >$ in Worklist, compute *Reference* set for the statement $S : p_i.n = q_j$. If any incoming traversal patterns have path expression with prefix y, they are replaced by prefix x and then substitution will be performed based on the new path expressions.

Figure 5 depicts the process of backward propagation. It lists the new tuples introduced by the traversal pattern of statement S15 after serious of substitutions. At iteration 2, this process identifies that the links specified by traversal pattern of S15 are defined by statement S5. The tuple $< S15, node_1, next >$ is removed and a USE is identified, while the other tuple $< S15, node_1.next^+, next >$ is replaced by $< S15, list_2.next^\star, next >$, which is then transformed into two tuples $< S15, list_2, next >$ and $< S15, list_2.next^+, next >$.

The backward process will collect the traversal patterns of statements in the the loop S12 of the program in Figure 3 and establish their DEF/USE relationships with statements in loops S2 and S12. The traversal patterns are $< S15, ptr_2, next >$, $< S13, ptr_2, prev >$, and $< S11, list_2, next >$. When these traversal patterns are propagated back to loop S2, this process will identify that the link constructed by statement S5 might be traversed by statements S11 and S15, and S6 might be referenced by statement S13.

Although the constructed recursive data structure by loop S2 is a doubly-linked list, i.e. it contains cycles, the traversal pattern defined by loop S12 is in fact only traversing through a singly-linked list. The back links are merely used to reference values from neighboring nodes, and play no role in structure traversal. In other words, the traversing links, specified by list traversing statement S15, of this recursive structures constitutes a singly-linked list. This example demonstrates that DEF/USE chains can provide valuable information for compilers to perform pointer analysis which is essential for optimizations and parallelization.

Program		Iteration 1	Iteration 2
S1	list_1 = new()		
S2 S2′	do while (...) $\mathrm{list}_2 = \phi$ (list_1, list_3)	$< S15,\ list_2,\ next >$ $< S15,\ list_2.next^+,\ next >$	
S3	node_1 = new()	$< S15,\ list_2,\ next >$ $< S15,\ list_2.next^+,\ next >$	
S4	node_1.value = ...	$< S15,\ list_2,\ next >$ $< S15,\ list_2.next^+,\ next >$	
S5	node_1.next = list_2	$< S15,\ list_2,\ next >$ $< S15,\ list_2.next^+,\ next >$	$< S15,\ list_2.next^\star,\ next >$ $\Downarrow$ $< S15,\ list_2,\ next >$ $< S15,\ list_2.next^+,\ next >$
S6	list_2.prev = node_1	$< S15,\ node_1.next,\ next >$ $< S15,\ node_1.next.next^+,\ next >$	$< S15,\ node_1,\ next >$ $< S15,\ node_1.next^+,\ next >$
S7	list_3 = node_1	$< S15,\ node_1.next,\ next >$ $< S15,\ node_1.next.next^+,\ next >$	$< S15,\ node_1.,\ next >$ $< S15,\ node_1.next^+,\ next >$
S8	end do	$< S15,\ list_3.next,\ next >$ $< S15,\ list_3.next.next^+,\ next >$	$< S15,\ list_3,\ next >$ $< S15,\ list_3.next^+,\ next >$
S11	ptr_1 = list_2.next	$< S15,\ list_2.next,\ next >$ $< S15,\ list_2.next.next^+,\ next >$	
S12 S12′	do while (ptr_1) $\mathrm{ptr}_2 = \phi$ (ptr_1, ptr_3)	$< S15,\ ptr_1.next,\ next >$ $< S15,\ ptr_1.next^+,\ next >$	
S13	prv_1 = ptr_2.prev	$< S15,\ ptr_2,\ next >$	
S14	ptr_2.result += prv_1.value	$< S15,\ ptr_2,\ next >$	
S15	ptr_3 = ptr_2.next	$< S15,\ ptr_2,\ next >$	
S16	end do		

Fig. 5. Example (Only DEF/USE information pertaining S15 is shown)

4 Applications

This section outlines algorithms to perform pointer analysis and dependence test by using the DEF/USE information.

4.1 Pointer Analysis

The features of this pointer analysis approach are as follows:

- Pointer analysis is performed with the knowledge of traversal patterns of the recursive data structures to be constructed. The main goal is to estimate if every traversing link leads to a unique location.
- It only examines the *link defining statements* and determines the possible shapes of traversing structures.
- Alias information can be inferred by the path expressions and shape information.

This section outlines a simple algorithm that will estimate the possible shapes of dynamic recursive data structures that are constructed by programs without destructive updating, such as list reverse [16].

This algorithm performs forward analysis on CFG starting from ENTRY node. It keeps track of connections that are constructed by the link defining statements. This algorithm first categorizes link defining statements based on the following observation. For pointer instances p_i and q_j of a link defining statement $p_i.n = q_j$, each can be either a newly allocated location or a node on an existing recursive data structure. Consequently, there are four possibilities as shown in Figure 6, where solid boxes represent newly allocated locations and dashed ones correspond to the nodes on existing recursive data structures. The algorithm then estimates the possible shapes by examining the types of link defining statements and current shape information.

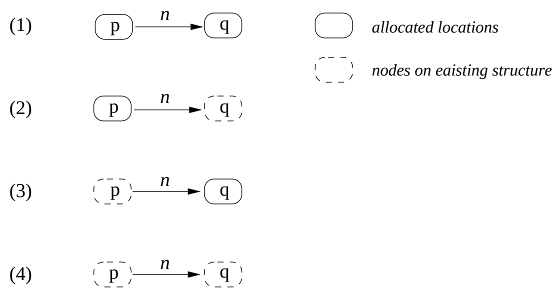


Fig. 6. Possible Outcomes of Link Defining Statement $p.n = q$

When both p_i and q_j are allocated locations (case 1), the assignment creates a new linked data structure and will not cause new cycles on any existing data structures. Similarly, case 2 and case 3 are an append and an insert operation, respectively. Neither operations will introduce cycles. However, case 4 might generate cycles or DAGs (directed acyclic graphs), depending on the connections relationship between p_i and q_j:

- $q_j \xrightarrow{\star} p_i$
  A cycle is generated.
- $p_i \not\rightarrow q_j$
  Two separate structures are connected. The operation does not introduce any new cycles.
- $p_i \xrightarrow{\star} q_j$
  - $p_i.n \xrightarrow{\star} q_j$
    Same shape.
  - $p_i.n \not\rightarrow q_j$
    A DAG if the structure is tree originally.
    Otherwise, it is a cyclic structure.
- $q_j \xrightarrow{?} p_i$
  This situation happens when p_i and q_j are defined in different loops with the same entry point. Although p_i and q_j reference the same data

structure, their relative position is unknown. Therefore, conservative estimation will be made.

Consequently, case 4 statements are the only ones that will generate cycles or turn tree structures into DAGs. Note the case 4 has a variant which is commonly used to insert elements between nodes of existing recursive data structures, depicted in Figure 7. The relationship between p_i and q_j can be identified from their path expressions on SSA representation and the connection information of dynamic recursive data structures gathered up to this point.

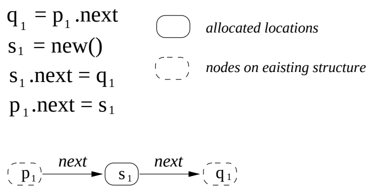


Fig. 7. An Insert Operation

The algorithm to estimate shapes of traversing structures of dynamic recursive data structures works as follows:

- Transform programs to SSA form.
- Apply the algorithm in the previous section to construct DEF/USE chains of link defining statements and link traversing statements.
- Categorize link defining statements that define traversing links.
- Estimate the possible shapes based on the rules defined for the classes of link defining statements.

Apply the algorithm to perform shape analysis on the program shown in Figure 3. The link defining statement that defines traversing links is S5 in loop S2 and it is a class 1 statement, whereas referencing links are defined by statements S6, which is class 4. Consequently, this algorithm will estimate the shape of the structure connected by traversing links is a singly-linked list, whereas the structure defined by traversal patterns $< S13, ptr_2, prev >$ and $< S15, ptr_2, next >$ contains cycles. This information is critical for dependence test which determines if the loop iterations can be executed independently.

4.2 Identifying Independent Operations

This section outlines an approach to check if iterations of a loop that traverses a dynamic recursive data structure and performs operations on each node can be executed independently. The result of this analysis can be used to parallelize loops with independent iterations. Similar test can be performed to determine if all the loops that reference the same recursive data structure have independent iterations. If this property is confirmed, program transformations might be able to perform on structure defining loop to facilitate parallel access to this data structure.

Iterations can be executed independently if no access conflicts occur. It can be determined by examining the sets of references and definitions in a traversing loop, which are represented by path expressions. This analysis is performed in two stages. Access conflict analysis is first conducted on the assumption that each unique path expression leads to a distinct location, without concerning the actual connections of recursive data structures. The rationale is that if any loop is deemed to have dependent iterations, it is inherently sequential no matter if the actual data structures contain cycles or not. Only those independent loops will be passed to the second stage, which uses the results of pointer analysis to determine if iterations are free of access conflicts.

The algorithm to determine if the iterations of a loop that traverse elements of recursive data structures can be executed independently works as follows:

– First Stage:
  . Gather the read and write sets of the statements in the loop body and present them relative to the induction pointer.
  . Compare the read and write sets. If conflicts occur, the iterations are not independent.
– Second Stage
  . If a loop passes the first stage of dependence test, the iterations are independent under the assumption that the actual data structures are linear. This stage first checks if the traversing structures of the referenced data structures by the loop are linear, and then analyzes if references through referencing links cause access conflicts with traversing structures.
  . Perform pointer analysis algorithm presented in the previous section. If traversing structure is a DAG or contains cycles, the loop is not independent.
  . If referencing links reference any nodes on the traversing structure, check if any access conflicts occur.

Consider the loop S12 of the program in Figure 3. At each iteration this loop reads from and writes to the same location of a node on the traversing structure, and references a value from the neighboring node. The read set consists of tuples $< S14, ptr_2, result >$ and $< S14, ptr_2.prev, value >$, while the write set is $\{< S14,\ ptr_2,\ result >\}$. The result of analysis shows that the iterations of this loop can be executed independently. However, if the statement S24 is replaced by

$$\text{S24}\ \ \text{ptr}_2.\text{result} += \text{prv}_1.\text{result}$$

then iterations of the loop will have to be executed sequentially.

5 Related Work

Various pointer analysis techniques have been proposed, such as alias analysis [3, 5, 6, 9, 17], side effect analysis [3, 10], conflict/interference analysis for programs with dynamic pointer-linked data structures [8, 11], and shape analysis [2, 7, 15, 16]. Another interesting field is pointer analysis on objected-oriented programs [12, 14]. The distinction of this work from others is that it performs pointer analysis and dependence test with the knowledge of traversal patterns by utilizing the DEF/USE information of pointer statements. The advantage is this approach can simplify pointer analysis when programs construct cyclic recursive data structures but exhibit acyclic traversal patterns.

Symbolic path expressions have been proposed by other researchers [5, 11]. Larus and Hilfinger used path expressions to specify nodes in alias graphs [11], whereas Deutsch paired symbolic access paths to represent alias information between recursive data structures [5]. In contrast, 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 an approach to identify the traversal patterns on dynamic recursive data structures and propagates this information to the statements that construct the structures. This DEF/USE information can be applied to direct compilers to gather important knowledge for optimizations and parallelization.

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. Jenq Kuen Lee, Dan Ho, and Yue-Chee Chuang. Data distribution analysis and optimization for pointer-based distrib uted programs. In *Proceedings of the 26th International Conference on Parallel Proces sing (ICPP)*, Bloomingdale, IL, August 1997.
13. 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.
14. Pedro C. Diniz Martin C. Rinard. 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.*
15. 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.
16. 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.
17. 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.*

This article was processed using the LaTeX macro package with LLNCS style