Scientific Programming 11 (2003) 3–37
IOS Press

Interprocedural definition-use chains of dynamic pointer-linked data structures

Yuan-Shin Hwang[a,*] and Joel Saltz[b]
[a]*Department of Computer Science, National Taiwan Ocean University, Keelung 20224, Taiwan*
E-mail: shin@cs.ntou.edu.tw
[b]*Department of Biomedical Informatics, The Ohio State University, Columbus, OH 43210, USA*
E-mail: saltz-@medctr.osu.edu

Abstract. This paper presents a flow-sensitive algorithm to compute interprocedural definition-use chains of dynamic pointer-linked data structures. The goal is to relate the statements that construct links of dynamic pointer-linked data structures (i.e. definitions) to the statements that might traverse the structures through the links (i.e. uses). Specifically, for each statement S that defines links of pointer-linked data structures, the algorithm finds the set of statements that traverse the links which are defined by S. This algorithm solves the definition-use chaining problem by performing backward iterative data flow analysis to compute the set of upward exposed uses at each statement. The results of this algorithm can be used to identify parallelism in programs even with cyclic pointer-linked data structures.

Keywords: Definition-use chains, pointer-linked data structures, data flow analysis, interprocedural analysis, pointer analysis, dependence analysis

1. Introduction

A *definition* of a variable v is a statement that assigns, or may assign, a value to v, and a *use* of v is a reference of v in a statement that reads, or may read, the value of v. A definition of v *reaches* a use of v if there is a path such that the definition is not killed. Similarly, a use u of a variable, say v, is *reachable* from a program point s if there is path from s to u that does not redefine v. The definition-use chaining problem is to compute for a program point s the set of uses u of a variable, say v, such that there is a path from s to u that does not redefine v [2].

Such notions can be overloaded for pointer-linked data structures. A *definition* of a pointer-linked data structure p is a statement that assigns, or may assign, a value to a storage location that can be accessed by traversing the links starting from p. Similarly, a *use* of a pointer-linked data structure p is a statement that reads, or may read, the value of a storage location that can be accessed by traversing the links starting from p. A definition of a pointer-linked data structure p *reaches* a use of p if there is a path such that the definition is not killed. Similarly, a use u of p is *reachable* from a program point s if there is path from s to u that does not redefine p. The definition-use chaining problem is to compute for a program point s the set of uses u of a pointer-linked data structure, say p, such that there is a path from s to u that does not redefine p [27]. This paper presents an algorithm to identity the definition-use chains of dynamic pointer-linked data structures.

The algorithm solves the definition-use chaining problem by performing backward iterative data flow analysis to compute the set of upward exposed uses at each statement. The advantages of this approach are as follows: (1) the uses that are propagated to the statements where the links of dynamic pointer-linked data structures are defined represent the traversal patterns of the constructed data structures, (2) it avoids

*Corresponding author: Yuan-Shin Hwang, Department of Computer Science, National Taiwan Ocean University, Keelung 20224, Taiwan. Tel.: +886 2 24622192, ext.6602; Fax: +886 2 24623249; E-mail: shin@cs.ntou.edu.tw.

ISSN 1058-9244/03/$8.00

the problem of building alias/shape graphs to summarize all possible traversal paths, many of which are not realized, of subsequent statements on the constructed pointer-linked data structures, and it can handle cyclic data structures and destructive update operations, such as list reverse or tree branch swap, and (3) it does not generate the spurious definition-use chains that will be created by the forward approach based on reaching definitions and alias information of pointers.

A flow-sensitive algorithm will be proposed to compute interprocedural definition-use chains of pointer-linked data structures. This algorithm follows the iterative data flow technique [33]. It first gathers the local uses and definitions at program points right before and after procedure calls. It then solves data flow equations for reachable uses by propagating local information using iterative techniques. Once the global information converges, this algorithm computes interprocedural definition-use chains by associating the local information with the propagated information.

One application of definition-use chains will be to facilitate the dependence analysis on programs with cyclic pointer-linked data structures [29]. The dependence analysis can be broken into the following three steps:

- Traversal patterns which loops or recursive procedures traverse the pointer-linked data structures are identified, and the statements that construct the links of traversal patterns will be located by definition-use chains of recursive data structures.
- Traversal-pattern-sensitive shape analysis will be performed to estimate possible shapes of traversal patterns.
- Dependence analysis will then be performed to identify parallelism using the result of shape analysis.

This technique can identify parallelism in programs with cyclic data structures due to the fact that many programs follow acyclic structures (i.e. traversal patterns) to access nodes on the cyclic data structures. For instance, this technique can be applied to identify the parallelism in the Barnes-Hut tree code [7]. The most time-consuming loops in Barnes-Hut that traverse the bipartite graph and perform computations can be parallelized to achieve good speedup [29].

The above technique only recognizes parallelism in the traversal references of the Barnes-Hut code, and hence only the graph traversal operations are parallelized while the graph construction operations are left to be executed in sequential. Therefore, further improvement can be achieved if the graph traversal operations of the Barnes-Hut code are parallelized as well. Furthermore, the definition-use chains of pointer-linked data structures provide the essential piece of information for the parallelization of the graph construction operations [28].

The remainder of the paper is organized as follows. Section 2 outlines the background information of this paper, such as the programming model and program representations. Section 3 gives the problem specifications and describes the algorithm to compute definition-use chains of dynamic pointer-linked data structures. Section 4 presents the flow-sensitive algorithm to handle interprocedural analysis. Experimental results will be presented in Section 6 and the related work is compared in Section 7.

2. Background

2.1. Programming model

The algorithms presented in this paper are designed to analyze programs with dynamic pointer-linked data structures which 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 (or records) 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, 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 pointer-linked 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

Programs will be transformed into an SSA (Static Single Assignment) intermediate representation [15]. It has been proved that several optimization techniques can be applied efficiently on SSA representation, such as constant propagation [44], redundancy elimination [3,38], induction variable identification [49], etc. 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.

Although SSA form is originally designed for programs with fixed-location variables only, e.g. Fortran-77 programs, same transformation can be applied to programs with pointer variables since contents (location addresses) of pointer variables can be treated as values in regular variables. Once normalized programs are transformed into the 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. Therefore, the possible forms of pointer assignment statements in SSA form are

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

Figure 1(a) presents a loop and its SSA representation. Each definition of the pointer variable ptr is given a new name in the example. Note that although statement S2 should be transformed to the following SSA form

S2 do
S2′ $\text{ptr}_2 = \phi\ (\text{ptr}_1, \text{ptr}_3)$
S2″ while (ptr_2)

it is represented by the SSA form in Fig. 1(a).

If programs contain procedure calls, each actual parameter will be represented by two new names, one for actual parameter that passes to the callee and the other (e.g. ptr_3 enclosed by parentheses in Fig. 1(b)) for the parameter that returns from the callee. Similarly, each formal parameter of a procedure is represented by two new variables, for the formal parameter at the entry of procedure and the other for the formal parameter at the end of procedure, as shown in Fig. 1(b).

The SSA representation is chosen for two reasons. First, each pointer instance is uniquely named in the SSA form. This format simplifies the notations used in this paper. The other reason is because the algorithm proposed in this paper has been implemented on the *ParaScope* parallel programming environment, which uses the SSA form as the intermediate representation [14].

2.3. Access path expressions

Pointers will be represented by *access path expressions*, each of which is a pointer instance followed by a string of field names connected by the field component operator "." [13,18]. The pointer instance of an access path expression e is called the *entry* of the access path expression e, and can be denoted as $entry(e)$. The string of field names connected by the field component operator of e will be called as the *path string* of the access path expression e. For example, $p\colon\ list.next$ is an access path expression of pointer p, which means the pointer instance $list$ is the entry of the access path of p and the path string contains only one field name $next$. Multiple occurrence of the same field names can be represented by "$+$" (at least once) or "$\star$" (zero or more than once) operators employed by representation of regular expressions, e.g. $list(.next)^{\star}$.

To compute access path expressions, pointer assignment statements must be examined. Fig. 2 presents a sequence of statements that traverse a linked list and access path expressions of pointers at each statement. The statements in the program are examined backward from the end to the entry of the program and access path expressions are computed following the order. The computation of access path expressions starts at the statement S3, and the access path expression of r, $q.n$, will be passed to the predecessor of S3. At S2, the ac-

Program		SSA Representation	
S1	ptr = list	S1	$ptr_1 = list_1$
S2	do while (ptr)	S2	do while (ptr_2)
		S2′	$ptr_2 = \phi(ptr_1, ptr_3)$
S3	ptr = ptr.next	S3	$ptr_3 = ptr_2$.next
S4	end do	S4	end do

(a) List Traversal Loop

Program		SSA Representation	
S1	ptr = list	S1	$ptr_1 = list_1$
S2	do while (ptr)	S2	do while (ptr_2)
		S2′	$ptr_2 = \phi(ptr_1, ptr_3)$
S3	call advance(ptr)	S3	call advance(ptr_2 (ptr_3))
S4	end do	S4	end do
S5	procedure advance(ptr)	S5	procedure advance(ptr_4)
S6	ptr = ptr.next	S6	$ptr_5 = ptr_4$.next
S7	end	S7	end (ptr_5)

(b) List Traversal Loop with Procedure Call

Fig. 1. SSA representation of programs with pointers.

cess path expression of q will be computed and the result is $p.n$. Meanwhile, the access path expression $q.n$ of r will be transformed by substituting q by the access path expression of q, and hence the new path expression of r is $p.n.n$. Finally, S1 will further transform path expressions by replacing p with $list$, as shown in Fig. 2(c).

This example demonstrates that access path expressions can show the relative positions of pointers on pointer-linked data structures. Furthermore, the access path expressions also represent the traversal patterns of programs. To determine if two path expressions can reach same locations, comparison operations similar to the $Match$ operation defined in Deutsch [18] can be applied.

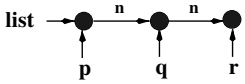
3. Definition-use chains of pointer-linked data structures

3.1. Problem specifications

Linked data structures, such as lists, tree, graphs, etc., are declared by recursively defined data types in programming languages. Each node is a fixed-size storage location represented by a record (or a node) with a fixed number of fields, some of them are pointers of recursively defined data types. As a result, a field of a node can be accessed by a binary access path, which has the form $v.f$ where v is a pointer variable and f is a field name.

As described in Section 2, there are three basic types of pointer assignment statements when programs are normalized such that each pointer assignment statement contains only binary access paths. An *aliasing statement* $p = q$ will direct p to point to the same location pointed to by q. This statement introduces an alias pair but does not constitute a definition or a use to pointer-linked data structures. When a link traversing statement $p = q.f$ is executed, the value stored in $q.f$ will be fetched and be assigned to p, and hence it is a use of $q.f$. Furthermore, this statement is a traversal through the link specified by $q.f$. Therefore, this statement is a use of the pointer-linked data structures some of whose nodes are pointed to by q. Similarly, the execution of a link defining statement $p.f = q$ is a definition of the field $p.f$, and hence it constitutes a definition of the pointer-linked data structures some of whose nodes are pointed to by p.

Figure 3 demonstrates the effects of different types of pointer assignment statements on a pointer-linked data structure whose starting node is pointed to by the pointer $list$. The first statement $t = list$, which is an aliasing statement, directs the pointer t to the node pointed to by the pointer $list$. This statement references the pointer $list$, but does not induce any references to



(a) List

S1: p = list
S2: q = p.n
S3: r = q.n

(b) Program

S1: p: $list$ q: $list.n$ r: $list.n.n$
S2: q: $p.n$ r: $p.n.n$
S3: r: $q.n$

(c) Path Expressions

Fig. 2. Backward computation of path expressions.

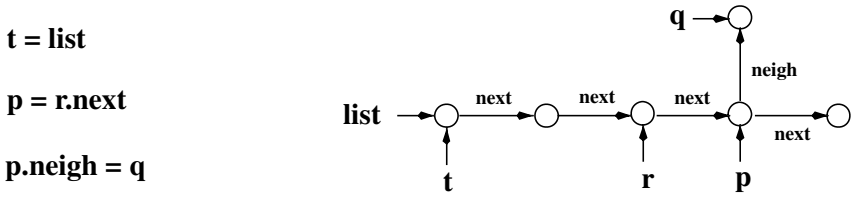


Fig. 3. Effects of pointer assignment statements on a recursive data structure.

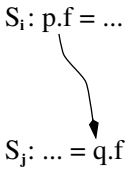


Fig. 4. Definition-use association.

the pointer-linked data structure $list$. The second statement $p = r.next$ references a link of the pointer-linked data structure, which is stored in the field $r.next$, and assigns the pointer p the destination node of the link $r.next$. As a result, the execution of this statement introduces a use on the structure $list$. On the other hand, the execution of the statement $p.neigh = q$ creates a link to connect the pointer-linked structure $list$ and the node pointed to by the pointer q. This statement changes the connections of the structure by adding one link, and consequently is a definition of the structure $list$.

Since link defining statements and link traversing statements correspond to definitions and uses of pointer-linked data structures respectively, the definition-use chains between link defining statements and link traversing statements will be computed to represent the definition-use chains of pointer-linked data structures. Consequently, the problem to compute the definition-use chains of dynamic pointer-linked data structures is formulated as

> For each statement S that defines links of pointer-linked data structures, find the set of statements that access the links which are defined by S.

Specifically, when S_i is a definition of $p.f$ and S_j contains a use of $q.f$, as shown in Fig. 4, then $q.f$ is a use of $p.f$ if

- There is at least one path from S_i to S_j such that $p.f$ can reach S_j, and
- p and q access the same locations.

3.2. Approach

The computation of definition-use chains for programs with regular fixed-location variables can be solved by data flow analysis in the direction opposite to the flow of program control [2]. Let $IN[B]$ and $OUT[B]$ be the set of uses that are reachable from the beginning and end of the basic block B, respectively. Let $USE[B]$ be the set of *upward exposed uses* of B, which are the set of pairs (s, x) such that s is a statement in B that uses variable x and such that no prior definition of x occurs in B, and let $DEF[B]$ be the set of pairs (s, x) such that s is a statement which uses x, s is not in B, and B has a definition of x. The data flow equations for the definition-use chaining problem are:

$$OUT[B] = \bigcup_{S \in succ(B)} IN[S] \tag{1}$$

$$IN[B] = USE[B] \cup (OUT[B] - DEF[B]) \quad (2)$$

The same analysis approach can be applied to the problem of computing the definition-use chains of dynamic pointer-linked data structures. The distinction is the set of upward exposed uses of binary access paths, instead of regular fixed-location variables, will be computed by the data flow analysis, and then the definition-use chains will be established by processing the upward exposed uses at statements where the binary access paths are defined. The set of upward exposed uses of binary access paths is the set of pairs $(S,\ p.f)$ such that S is a statement which uses the binary access path $p.f$ and such that no prior definition of $p.f$ occurs.

As each pair $(S,\ p.f)$ is propagated backward, the pointer p will be represented by an access path expression. The access path expressions in the use pairs will be transformed by statements, and the definition-use chains will be identified at link defining statements. Take the example listed in Fig. 5(a), which creates and traverses a linked list as shown in Fig. 5(b). The set of upward exposed uses at each statement will be computed by collecting the uses of statements and propagating them backward, as shown in Fig. 5(c). At statement S6, the only use is $(S6,\ p_2.n)$. When this use is propagated to S5, $(S6,\ p_2.n)$ will be transformed to $(S6,\ list_2.n.n)$, i.e. p_2 of the use $p_2.n$ will be replaced by $list_2.n$, because the statement $S6:\ p_2 = list_2.n$ assigns the value $list_2.n$ to p_2. Furthermore, a new use $(S5,\ list_2.n)$ is also generated. Similarly, S4 replaces $list_2$ in both uses by p_1, and the set of upward exposed uses at S4 consists of $(S5,\ p_1.n)$ and $(S6,\ p_1.n.n)$. When the uses reach S3, the use $(S5,\ p_1.n)$ matches the definition of S3, $p_1.n$, and hence a definition-use chain is identified between S3 and S5. Furthermore, the use will be transformed to $(S5,\ list_1)$ and then be discarded since the pattern $list_1$ is not involved in any structural traversal. Similarly, the other use $(S6,\ p_1.n.n)$ will be changed to $(S6,\ list_1.n)$ and be propagated to S2. Statement S2 does not modify the pattern of the use because $list_1$ of the pattern $list_1.n$ is not aliased to p_1 of S2. Finally when the use reaches S1, it will be killed since it matches the definition of S1, and the definition-use chain between S1 and S6 is recognized.

The example in Fig. 5 shows that as the upward exposed uses are propagated backward, the access path expressions of upward exposed uses are transformed to reflex their relative positions on the pointer-linked data structures. It also reveals special features of this approach:

- Uses are gathered and propagated to the statements where the definitions are created. The advantage is that no alias or connection graphs will be required to describe the connections of pointer-linked data structures. Furthermore, the uses that are propagated to the statements where the links of dynamic pointer-linked data structures are defined represent the traversal patterns of the constructed data structures.
- Each unique pointer in the set of access path expressions of reachable uses can be assumed to point to a distinct location node (storage location) until it reaches the statement where it is defined. It simplifies the process of comparing access path expressions.
- Aliases will be identified by the propagation and transformation process on access path expressions. For example, pointers $list_2$ and p_1 are identified as aliases at the statement S4, and pointer p_2 is determined to be aliased to $list_1$ after transformations by S3 is performed.

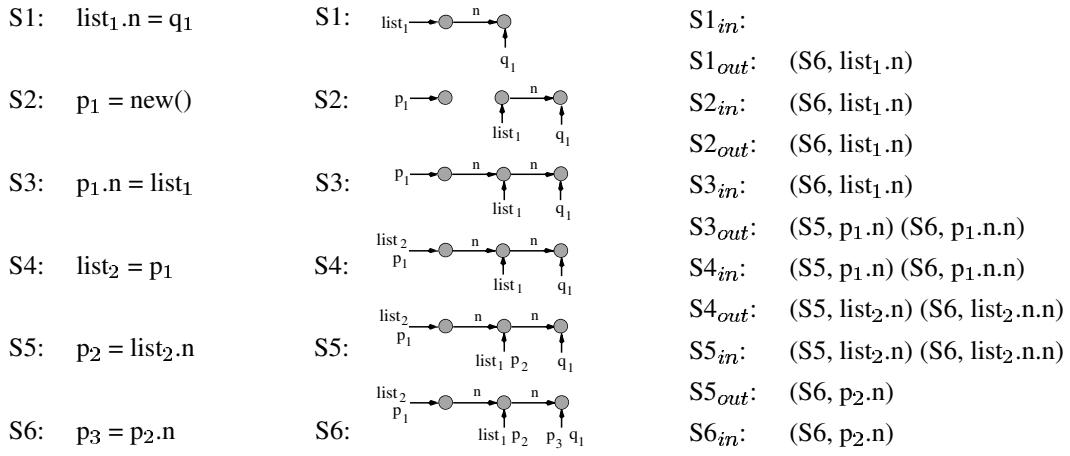
This approach can handle programs even with cyclic pointer-linked data structures or DAGs (directed-acyclic graphs). Consider the examples in Fig. 6. The uses that are propagated to S4 access the same link $r.n$ of a DAG through different pointers p and q, as shown in Fig. 6(a). The statement S3 transforms the use $(S5,\ q.n.n)$ to $(S5,\ r.n)$ and S2 transforms the use $(S4,\ p.n.n)$ to $(S4.\ r.n)$. Both uses match the definition $r.n$ of S1, and hence definition-use chains are identified. Similarly, the uses on a cyclic graph by S3 and S4 of the example shown in Fig. 6(b) are propagated to the statements that define the cyclic graph, and definition-use chains can be easily established.

In summary, the set of upward exposed uses $UPEXP[S]$ at statement S can be computed by the following data flow equations:

$$UPEXP_{out}[S] = \bigcup_{d \in succ(S)} UPEXP_{in}[d] \quad (3)$$

$$UPEXP_{in}[S] = USE[S] \cup F_S(UPEXP_{out}[S]) \quad (4)$$

where $UPEXP_{out}[S]$ is the set of upward exposed uses at the exit of statement S and $UPEXP_{in}[S]$ is the set of upward exposed uses at the entry of S, $USE[S]$ is the set of uses generated by the statement S, F_S is the transformation that will be performed by S on each use, and $succ(S)$ is the set of the successors of S. The transformation F_S will be called the *transfer function* of the statement S and it is defined by statement type of S.



(a) Program (b) Graphs (c) Upward Exposed Uses

Fig. 5. Backward propagating and transforming uses to identify definition-use chains.

S1:	r.n = y	
S2:	p.n = r	(S4, r.n) (S5, r.n)
S3:	q.n = r	(S4, p.n.n) (S5, r.n)
S4:		(S4, p.n.n) (S5, q.n.n)

(a) Program with DAG

S1:	q.n = p	
S2:	p.n = q	(S4, q.n)
S3:	t_1 = p.n	(S3, p.n) (S4, p.n.n)
S4:	t_2 = t_1.n	(S4, t_1.n)

(b) Program with Cyclic Graph

Fig. 6. Programs with DAGs or cyclic graphs.

3.2.1. Transformations of access path expressions

The transformations by transfer function F_S of the statement S on each access path expression are presented as follows.

Aliasing statements $S: p_i = q_j$

For an aliasing statement, $S: p_i = q_j$, the pointer instance p_i is defined after the statement S is executed, and furthermore it points to the same locations as q_j. Therefore, during the backward propagation process, if the entry of an access path expression e is p_i, i.e. $e \equiv p_i.T$ where T is a path string, then entry p_i will be replaced by q_j, and the new access path expression will be $q_j.T$. Otherwise, the access path expression e remains unchanged. In other words, an access path expression e will be transformed by $F_{S:\, p_i = q_j}$ to

$$F_{S:\, p_i = q_j}(e) = \begin{cases} q_j.T & \text{if } e \equiv p_i.T \\ e & \text{otherwise} \end{cases}$$

Link traversing statements $S: p_i = q_j.f$

Similar to aliasing statements, the pointer instance p_i of a link traversing statement $S: p_i = q_j.f$ is not defined before S. As a result, the transformation of link traversing statements can be formulated similarly to that of aliasing statements: an access path expression e will be transformed by $F_{S:\, p_i = q_j.f}$ to

$$F_{S:\, p_i = q_j.f}(e) = \begin{cases} q_j.f.T & \text{if } e \equiv p_i.T \\ e & \text{otherwise} \end{cases}$$

In other words, if the entry of an access path expression e is p_i, i.e. $e \equiv p_i.T$, the entry will be replaced by $q_j.f$.

Figure 7 shows an example and the access path expressions of its uses at each statement to demonstrate the meaning of the transformations with respect to the relative positions on pointer-linked data structures. The use (S3, p_2.n) is created at S3 and is stored in UPEXP$_{in}$[3]. The path $p_2.n$ represents the link n from the node pointed to by p_2. The use is propagated back to S2, and then path $p_2.n$ will be transformed by replac-

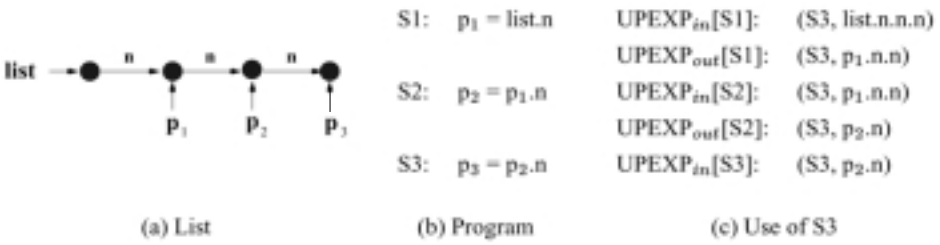


Fig. 7. Transformations by link traversing statements.

ing p_2 with $p_1.n$ and the new access path expression will be $p_1.n.n$. The new access path expression $p_1.n.n$ at S2 represents the path from the node pointed to by p_1 to the link referenced at S3. Similarly, the access path expression will be transformed to $list.n.n.n$ at S1, which means the use $p_2.n$ of S3 references the third link of the data structure $list$.

Link defining statements $S: p_i.f = q_j$

In contrast to the above two types of statements, a link defining statement $S: p_i.f = q_j$ does not create any new pointer instances. Instead, a new link definition $p_i.f$ is generated by S. However, similar transformations on access path expressions that are employed by aliasing statements and link traversing statements will be performed by link defining statements. The distinction is that substitution is performed on the pattern $p_i.f$, not a pointer instance as done by the previous two types of statements. Therefore, an access path expression e will be transformed $F_{S: p_i.f=q_j}$ to

$$F_{S: p_i.f=q_j}(e) = \begin{cases} q_j.T & \text{if } e \equiv p_i.f.T \\ e & \text{otherwise} \end{cases}$$

In other words, if the prefix of an access path expression e is $p_i.f$, i.e. $e \equiv p_i.f.T$, the access path expression will be transformed to $q_j.T$.

Figure 8 shows an example and its access path expressions to demonstrate that the transformations can transfer access path expressions to correctly reflect the relative positions. The example creates a list with one edge, then inserts another edge to the list, and finally traverses the two links of the list. The transformations on access path expressions must reflect the change of structure configuration. The use reference $p_1.n$ on the list at S4 is propagated and transformed by the link traversing statement S3, and the access path expression will be $r_1.n.n$, which means the second edge of the list r_1. The first edge of the list r_1 is connected by statement S2 and is not part of the list before statement S2, that is, the second edge of the list r_1 at S2 is in turn the first edge of the list t_1 at S1. The transformation on the access path expression by the link defining statement S2 reflects the situation. The access path expression $r_1.n.n$, which corresponds to the second edge of the list r_1, will be transformed to $t_1.n$, which represents the first edge of the list t_1 at S1.

ϕ-statements $S: p_i = \phi(p_j, p_k)$

Statements with ϕ-functions are created by SSA representation to merge values of the same pointers from different branches of program control flow. Therefore, the execution of ϕ-statements is similar to aliasing statements. Each statement of both types creates a pointer instance and furthermore neither of statement types is involved in traversal operations on pointer-linked data structures. Consequently, the same transformation employed by aliasing statements can be applied to ϕ-statements. However, since each ϕ-function takes values from different branches, the transformation will create more than one new access path expressions, one for each branch, from the original access path expression. As a result, the transformation of a ϕ-statement $S: p_i = \phi(p_j, p_k)$ can be formulated as follows: for an access path expression e, the following access path expressions will be created:

$$F_{S: p_i = \phi(p_j, p_k)}(e) = \begin{cases} p_j.T \quad p_k.T & \text{if } e \equiv p_i.T \\ e & \text{otherwise} \end{cases}$$

3.2.2. Drawback and solution

One drawback of this approach is that the order of link construction must match the order of link traversal. Some definition-use chains will not be identified if the orders of link construction and traversal are not matched. For example, Fig. 9 shows the effects of different orders of link construction and traversal operations by switching the statements of examples in Fig. 6. When uses in the example of Fig. 9(a) are propagated to the statement S3, they are not identified as the uses of S3 since their access path expressions do not match the definition. The same situation happens when the

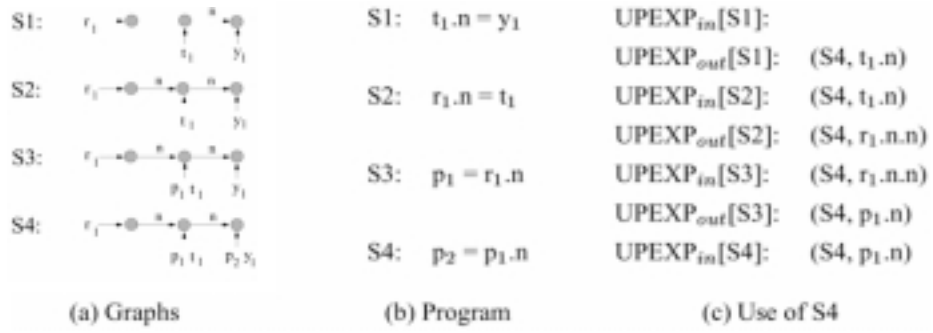


Fig. 8. Transformations by link defining statements.

use from S4 of the example in Fig. 9(b) reaches its corresponding definition statement S2.

The reason is because some aliases would not be identified when the order of link traversal is different from the order of link construction, e.g. the aliases q and t_1 in Fig. 9(b). The situation is caused by the fact that, after the link defining statement $S1: p.n = q$ of Fig. 9(b) is executed, the same destination node q can be reached by two different paths, $p.n$ and q. The definition $q.n$ of S2 references the node q through the path q whereas t_1 of the use $t_1.n$ of S4 accesses q after traversing the path $p.n$. Although the pointers q and t_1 point to the same node and the fact will be identified at the statement S1, the information is not available at S2.

The solution is to propagate definitions as well. For instance, the definition $r.n$ of S3 in Fig. 9(a) is propagated to S2, it matches the use of S5 after the use is transformed by S2. Consequently, the definition-use chain between S3 and S5 can be recognized. The definition $(S3, r.n)$ will be further propagated back to S1 and its definition-use relationship with S4 will be identified. Similarly, the definition-use chain between S2 and S4 of the example in Fig. 9(b) will be established if the definition $(S2, q.n)$ is propagated along with uses, as shown in Fig. 10.

3.3. Data flow equations

In summary, the set of upward exposed uses of every statement will be collected to compute the definition-use chains of dynamic pointer-linked data structures. The set of uses that can be reached from statement S can be solved by the following data flow equations:

$$UPEXP_{out}[S] = \bigcup_{d \in succ(S)} UPEXP_{in}[d] \quad (5)$$

$$UPDEF_{out}[S] = \bigcup_{d \in succ(S)} UPDEF_{in}[d] \quad (6)$$

$$\begin{aligned} &UPEXP[S] \\ &= USE[S] \cup F_S(UPEXP_{out}[S]) \end{aligned} \quad (7)$$

$$\begin{aligned} &UPDEF[S] \\ &= DEF[S] \cup F_S(UPDEF_{out}[S]) \end{aligned} \quad (8)$$

$$UPDEF_{in}[S] = F_{UPDEF[S]}(UPDEF[S]) \quad (9)$$

$$UPEXP_{in}[S] = F_{UPDEF[S]}(UPEXP[S]) \quad (10)$$

where $UPEXP_{out}[S]$ and $UPEXP_{in}[S]$ are the sets of upward exposed uses after and before statement S respectively, $USE[S]$ is the set of uses generated by the statement S, F_S is the *transfer function* of S, $succ(S)$ is the set of successors of S, $UPDEF_{out}[S]$ and $UPDEF_{in}[S]$ are the sets of upward exposed definitions after and before S respectively, $DEF[S]$ is the set of definitions generated by the statement S, and $F_{UPDEF}[S]$ is the transfer function of the set $UPDEF[S]$. The transfer function $F_{UPDEF}[S]$ can be represented by the following equation:

$$F_{\langle s_1, p_1, f, q_1 \rangle \in UPDEF[S]}(\langle s, p, n \rangle) = \begin{cases} - & \text{if} s_1 \gg s \wedge p_1.f \equiv p.n \\ \langle s, q_1.T, n \rangle & \text{if} s_1 \gg s \wedge p \equiv p_1.f.T \\ \langle s, p, n \rangle \ \langle s, q_1.T, n \rangle & \text{if } s_1 \not\gg s \wedge p \equiv p_1.f.T \\ \langle s, p, n \rangle & \text{otherwise} \end{cases} \quad (11)$$

if $\langle s, p, n \rangle \in UPEXP[s_1]$ and F_{s_1} does not transform the tuple, where $s_1 \gg s$ means statement s_1 dominates s.

3.4. Algorithm

The algorithm follows the iterative data flow analysis technique to compute the definition-use chains of dynamic pointer-linked data structures. It solves

S1:	p.n = r	(S4, r.n) (S5, r.n)
S2:	q.n = r	(S4, p.n.n) (S5, r.n)
S3:	r.n = y	(S4, p.n.n) (S5, q.n.n)
S4:		(S4, p.n.n) (S5, q.n.n)

(a) Program in Fig. 6(a)

S1:	p.n = q	(S4, q.n)
S2:	q.n = p	(S3, p.n) (S4, p.n.n)
S3:	t_1 = p.n	(S3, p.n) (S4, p.n.n)
S4:	t_2 = t_1.n	(S4, t_1.n)

(b) Program in Fig. 6(b)

Fig. 9. Effects of order of link construction and traversal.

S1:	p_1.n = q_1	UPDEF$_{in}$[S1]:	(S1, p_1.n) (S2, q_1.n)	UPEXP$_{in}$[S1]:	(S4, q_1.n)
		UPDEF$_{out}$[S1]:	(S2, q_1.n)	UPEXP$_{out}$[S1]:	(S3, p_1.n) (S4, p_1.n.n)
S2:	q_1.n = y_1	UPDEF$_{in}$[S2]:	(S2, q_1.n)	UPEXP$_{in}$[S2]:	(S3, p_1.n) (S4, p_1.n.n)
		UPDEF$_{out}$[S2]:		UPEXP$_{out}$[S2]:	(S3, p_1.n) (S4, p_1.n.n)
S3:	t_1 = p_1.n	UPDEF$_{in}$[S3]:		UPEXP$_{in}$[S3]:	(S3, p_1.n) (S4, p_1.n.n)
		UPDEF$_{out}$[S3]:		UPEXP$_{out}$[S3]:	(S4, t_1.n)
S4:	t_2 = t_1.n	UPDEF$_{in}$[S4]:		UPEXP$_{in}$[S4]:	(S4, t_1.n)

(a) Program (b) Denitions (c) Upward Exposed Uses

Fig. 10. Propagating definitions to handle structural aliases.

the data flow equations of upward exposed use problem and then applies the solutions to the computation of definition-use chains. Backward analysis is conducted on the *interval flow graph* of the procedure, i.e. $IG = [V_{IG},\ E_{IG}]$, where V_{IG} is the set of nodes and E_{IG} is the set of edges, to solve the data flow equations [43]. The algorithm is shown in Fig. 11. The outcome of the algorithm will be definition-use chains between link defining statements and link traversing statements.

Each use will be represented by a tuple $\langle s,\ p,\ n\rangle$, where s is the statement that creates the use, p is the access path expression of the pointer, and n is the field name. Therefore, a link traversing statement $S: p_i = q_j.n$ will create a use tuple $\langle S,\ q_j,\ n\rangle$. Similarly, each definition will be represented by a tuple $\langle s,\ p,\ n, q\rangle$ where s is the statement that creates the definition, p and q are the access path expressions, and n is the field name. A link defining statement $S: p_i.n = q_j$ will introduce a definition tuple $\langle S,\ p_i,\ n,\ q_j\rangle$.

Each statement S stores the definitions and uses that are created at S in $DEF[S]$ and $USE[S]$ respectively. The definitions and uses of successor statements of S will be gathered and then be stored in $UPDEF_{out}[S]$ and $UPEXP_{out}[S]$. The tuples in $UPDEF_{out}[S]$ and $UPEXP_{out}[S]$ will be transformed by the transfer function F_S of statement S and then stored in temporary sets $UPDEF[S]$ and $UPEXP[S]$ along with the local definitions and uses in $DEF[S]$ and $USE[S]$. The definitions in $UPDEF[S]$ will be transformed by the transfer function $F_{UPDEF}[S]$ and then stored in the set $UPDEF_{in}[S]$. Similarly, the uses in the set $UPEXP[S]$ will then be transformed by the transfer function $F_{UPDEF}[S]$ and stored in $UPEXP_{in}[S]$.

The backward phase computes the data flow equations by traversing the nodes of the interval flow graph in reverse topological order. At each iteration, the sets on each statement are computed by the Eqs (5)–(10). This iterative process terminates when none of data flow equations produces new tuples. Definition-use chains between link defining statements and link defining statements will be identified when the data flow analysis converges, and the results will be stored in the set $DefUse[S]$, which lists the uses of the definition defined at the statement S.

3.4.1. Normalization of access path expressions

Since loops can have unbounded numbers of iterations, the access path expressions of definition and use tuples might have unbounded numbers of fields after being transformed by transfer functions of statements in loops. In order to ensure the iterative data flow algorithm will terminate, the length of every access path expression must be finite. Therefore, access path expressions will be normalized at header nodes of loops. Fig. 12 shows the algorithm $Factor(p)$ that normalizes

algorithm ComputeDefUseChains (IG)
input:
$IG = [V_{IG}, E_{IG}]$: interval flow graph of the procedure
output:
$DefUse[S]$: The set of statements that use the definition of S
begin
{initialize sets}
for each node S in IG **do** $UPEXP_{in}[S] = UPDEF_{in}[S] = DefUse[S] = \emptyset$
{iteratively analyze a procedure}
while changed
for each node S in IG in reverse topological order **do**
{Compute definition and use sets}
$UPEXP_{out}[S] = \bigcup_{d \in succ(S)} UPEXP_{in}[d]$
$UPDEF_{out}[S] = \bigcup_{d \in succ(S)} UPDEF_{in}[d]$
$UPEXP[S] = USE[S] \cup F_S\ (UPEXP_{out}[S])$
$UPDEF[S] = DEF[S] \cup F_S\ (UPDEF_{out}[S])$
$UPDEF_{in}[S] = F_{UPDEF[S]}(UPDEF[S])$
$UPEXP_{in}[S] = F_{UPDEF[S]}(UPEXP[S])$
{Factor definition and use sets at loop header}
if S is the header node of a loop **then**
Factor($UPEXP_{in}[S]$)
Factor($UPDEF_{in}[S]$)
end
end
end
{compute definition-use chains}
for each node S in IG **do**
Compute definition-use chains and store results in $DefUse[S]$
end
end

Fig. 11. Algorithm for computing definition-use chains within a procedure.

an access path expression p. The algorithm is a simplified algorithm which is adapted from the normalization algorithm for symbolic access paths proposed by Deutsch [18] and the normalization algorithm for path expressions proposed by Hendren [25].

The algorithm examines all the fields of the path string of an access path expression p. Initially the first field f_1 will be stored in e, and then fields will be examined from left to right. If f_i is a recursively defined type and its type is the same as the type of e, then f_i will be merged to e. Otherwise, e will be appended to the end of path string of p' and then e will be initialized as f_i. The result p' of the normalization $Factor(p)$ will have the form $p_i.e_1.f_i.e_2.f_j.\cdots.e_k$, where $e_i = (f|\cdots)^+$. For example, the access path expression $p_i.n.n$ will be normalized to $p_i.(n)^+$, while $h_i.left.right$ will be transformed to $h_i.(left|right)^+$.

The set of all possible access path expressions is partitioned into finite equivalent classes after the normalization function $Factor$ is performed. It is the result of unitary-prefix decomposition theorem developed by Eilenberg [20], which has been applied by Deutsch to

algorithm Factor(p)
input:
p: an access path expression
output:
p': a normalized access path expression
begin
Let $p \equiv p_i.f_1.f_2.\cdots.f_n$
$p' = p_i$; $e = (f)^1$ where $f \equiv f_1$
for each f_i $i = 2, n$ **do**
if f_i is a recursively defined field and $type(e) \equiv type(f_i)$ **then**
$e = (f)^+$ where $f \equiv f \mid f_i$
else
Append e to p'
$e = (f)^1$ where $f \equiv f_i$
end
end
return p'
end

Fig. 12. Normalization algorithm $Factor(p)$.

represent sets of possible alias relations [17,18]. Eilenberg's unitary-prefix decomposition theorem will be recapped as follows [17,20]. A subset A of $\Sigma^\star$ is a *unitary set* if either A is an empty set or A is the language generated by a deterministic automaton with a single terminal state that is accessible. A *unitary monoid* A is a submonoid of $\Sigma^\star$ that is a unitary subset of $\Sigma^\star$. A subset A of $\Sigma^\star$ is a *prefix* set if $A \cap A\Sigma^+ = \emptyset$.

Definition 3.1 UNITARY-PREFIX MONOMIALS [20]
A unitary-prefix monomial of degree n is a recognizable set of the form: $U = M_n\sigma_n M_{n-1}\sigma_{n-1}\cdots M_1\sigma_0 M_0$ *in which* $M_n, M_{n-1}, \cdots, M_0$ *are unitary monoids,* $\sigma_n, \sigma_{n-1}, \cdots, \sigma_0$ *are letters of* Σ*, and each of the sets* $M_n\sigma_n \cdots M_i\sigma_i$ $(1 \leqslant i \leqslant n)$ *is a prefix.*

Theorem 1 UNITARY-PREFIX DECOMPOSITION [20]
Each recognizable subset $\mathcal{L}$ *of* $\Sigma^\star$ *admits a disjoint decomposition:* $\mathcal{L} = U_1 \cup \cdots \cup U_m$ *where* U_i $(1 \leqslant i \leqslant m)$ *is a unitary-prefix monomial. Furthermore, there exists a computable algorithm that determines* $\{U_1, \cdots, U_m\}$.

3.4.2. Computing definition-use chains

The set $UPEXP_{out}[S]$ of each link defining statement $S : p_i.f = q_j$ will be examined to compute definition-use chains. If there exists a use tuple $\langle S_1, p_i, f\rangle \in UPEXP_{out}[S]$, then S is a definition of S_1, and consequently the definition-use information will be stored in the set $DefUse[S]$:

$$DefUse[S] = DefUse[S] \cup \{S_1\}$$

Furthermore, the tuples in sets $UPEXP[S]$, $UPEXP_{in}[S]$, and $UPDEF[S]$ will be compared to identify possible definition-use chains. If a use tuple of $\langle S_1, p_i, n\rangle$ is in $UPEXP[S]$ but the tuple is not in $UPEXP_{in}[S]$, i.e. the tuple $\langle S_1, p_i, n\rangle$ is killed by the transfer function $F_{UPDEF[S]}$ at statement S, then the tuple $\langle S_2, p_i, n, q\rangle \in UPDEF[S]$ that kills the tuple $\langle S_1, p_i, n\rangle$ is the definition. Therefore, the information of definition-chain between S_2 and S_1 will be stored in the set $DefUse[S_2]$:

$$DefUse[S_2] = DefUse[S_2] \cup \{S_1\}$$

The computation of definition-use chains can be incorporated into the backward propagation process, since a definition-use pair will be identified when a use tuple is killed either by F_S or $F_{UPDEF[S]}$ at statement S, i.e. when the use tuple matches the definition of a link defining statement or a definition tuple in $UPDEF[S]$.

If two definition tuples of two link defining statements, say S_1 and S_2, in $UPDEF_{in}[S]$ are matched and if statement S_1 dominates S_2, the sets $DefUse[S_1]$ and $DefUse[S_2]$ will be compared to identify possible definition-use chains. If s is in $DefUse[S_1]$ and if there exists a use tuple $\langle s, p, n\rangle$ in $UPEXP_{out}[S_2]$, then s will be added to $DefUse[S_2]$, i.e.

Program	Iteration 1	Iteration 2	Iteration 3
S1: $rlist_1$ = nil	⟨S9, $list_1$, v⟩ ⟨S5, $list_1$, n⟩	⟨S9, $list_1$, v⟩ ⟨S9, $list_1$.n, v⟩ ⟨S5, $list_1$, n⟩ ⟨S5, $list_1$.n, n⟩	⟨S9, $list_1$, v⟩ ⟨S9, $list_1.n^+$, v⟩ ⟨S5, $list_1$, n⟩ ⟨S5, $list_1.n^+$, n⟩
S2: do while ($list_2$) $list_2 = \phi$ ($list_1$, $list_3$) $rlist_2 = \phi$ ($rlist_1$, $rlist_3$)	⟨S8, $rlist_2$, n⟩ ⟨S8, $rlist_2.n^+$, n⟩ ⟨S9, $rlist_2$, v⟩ ⟨S9, $rlist_2.n^+$, v⟩ ⟨S9, $list_2$, v⟩ ⟨S5, $list_2$, n⟩	⟨S8, $rlist_2$, n⟩ ⟨S8, $rlist_2.n^+$, n⟩ ⟨S9, $rlist_2$, v⟩ ⟨S9, $rlist_2.n^+$, v⟩ ⟨S9, $list_2$, v⟩ ⟨S9, $list_2$.n, v⟩ ⟨S5, $list_2$, n⟩ ⟨S5, $list_2$.n, n⟩	⟨S8, $rlist_2$, n⟩ ⟨S8, $rlist_2.n^+$, n⟩ ⟨S9, $rlist_2$, v⟩ ⟨S9, $rlist_2.n^+$, v⟩ ⟨S9, $list_2$, v⟩ ⟨S9, $list_2.n^+$, v⟩ ⟨S5, $list_2$, n⟩ ⟨S5, $list_2.n^+$, n⟩
S3: t = $rlist_2$	⟨S8, $rlist_2$, n⟩ ⟨S8, $rlist_2.n^+$, n⟩ ⟨S9, $rlist_2$, v⟩ ⟨S9, $rlist_2.n^+$, v⟩ ⟨S9, $list_2$, v⟩ ⟨S5, $list_2$, n⟩	⟨S8, $rlist_2$, n⟩ ⟨S8, $rlist_2.n^+$, n⟩ ⟨S9, $rlist_2$, v⟩ ⟨S9, $rlist_2.n^+$, v⟩ ⟨S9, $list_2$, v⟩ ⟨S9, $list_2$.n, v⟩ ⟨S5, $list_2$, n⟩ ⟨S5, $list_2$.n, n⟩	⟨S8, $rlist_2$, n⟩ ⟨S8, $rlist_2.n^+$, n⟩ ⟨S9, $rlist_2$, v⟩ ⟨S9, $rlist_2.n^+$, v⟩ ⟨S9, $list_2$, v⟩ ⟨S9, $list_2.n^+$, v⟩ ⟨S5, $list_2$, n⟩ ⟨S5, $list_2.n^+$, n⟩
S4: $rlist_3$ = $list_2$	⟨S8, t, n⟩ ⟨S8, t.n^+, n⟩ ⟨S9, t, v⟩ ⟨S9, t.n^+, v⟩ ⟨S9, $list_2$, v⟩ ⟨S5, $list_2$, n⟩	⟨S8, t, n⟩ ⟨S8, t.n^+, n⟩ ⟨S9, t, v⟩ ⟨S9, t.n^+, v⟩ ⟨S9, $list_2$, v⟩ ⟨S9, $list_2$.n, v⟩ ⟨S5, $list_2$, n⟩ ⟨S5, $list_2$.n, n⟩	⟨S8, t, n⟩ ⟨S8, t.n^+, n⟩ ⟨S9, t, v⟩ ⟨S9, t.n^+, v⟩ ⟨S9, $list_2$, v⟩ ⟨S9, $list_2.n^+$, v⟩ ⟨S5, $list_2$, n⟩ ⟨S5, $list_2.n^+$, n⟩
S5: $list_3$ = $list_2$.n	⟨S8, t, n⟩ ⟨S8, t.n^+, n⟩ ⟨S9, t, v⟩ ⟨S9, t.n^+, v⟩ ⟨S9, $rlist_3$, v⟩ ⟨S5, $list_2$, n⟩	⟨S8, t, n⟩ ⟨S8, t.n^+, n⟩ ⟨S9, t, v⟩ ⟨S9, t.n^+, v⟩ ⟨S9, $rlist_3$, v⟩ ⟨S9, $list_2$.n, v⟩ ⟨S5, $list_2$, n⟩ ⟨S5, $list_2$.n, n⟩	⟨S8, t, n⟩ ⟨S8, t.n^+, n⟩ ⟨S9, t, v⟩ ⟨S9, t.n^+, v⟩ ⟨S9, $rlist_3$, v⟩ ⟨S9, $list_2.n^+$, v⟩ ⟨S5, $list_2$, n⟩ ⟨S5, $list_2.n^+$, n⟩
S6: $rlist_3$.n = t	– ⟨S8, t, n⟩ ⟨S8, t.n^+, n⟩ ⟨S9, t, v⟩ ⟨S9, t.n^+, v⟩ ⟨S9, $rlist_3$, v⟩	– ⟨S8, t, n⟩ ⟨S8, t.n^+, n⟩ ⟨S9, t, v⟩ ⟨S9, t.n^+, v⟩ ⟨S9, $rlist_3$, v⟩ ⟨S9, $list_3$, v⟩ ⟨S5, $list_3$, n⟩	– ⟨S8, t, n⟩ ⟨S8, t.n^+, n⟩ ⟨S9, t, v⟩ ⟨S9, t.n^+, v⟩ ⟨S9, $rlist_3$, v⟩ ⟨S9, $list_3$, v⟩ ⟨S9, $list_3$.n, v⟩ ⟨S5, $list_3$, n⟩ ⟨S5, $list_3$.n, n⟩
S7: end do	⟨S8, $rlist_3$, n⟩ ⟨S8, $rlist_3.n^+$, n⟩ ⟨S9, $rlist_3$, v⟩ ⟨S9, $rlist_3.n^+$, v⟩	⟨S8, $rlist_3$, n⟩ ⟨S8, $rlist_3.n^+$, n⟩ ⟨S9, $rlist_3$, v⟩ ⟨S9, $rlist_3.n^+$, v⟩ ⟨S9, $list_3$, v⟩ ⟨S5, $list_3$, n⟩	⟨S8, $rlist_3$, n⟩ ⟨S8, $rlist_3.n^+$, n⟩ ⟨S9, $rlist_3$, v⟩ ⟨S9, $rlist_3.n^+$, v⟩ ⟨S9, $list_3$, v⟩ ⟨S9, $list_3$.n, v⟩ ⟨S5, $list_3$, n⟩ ⟨S5, $list_3$.n, n⟩
S8:	⟨S8, $rlist_2$, n⟩ ⟨S8, $rlist_2.n^+$, n⟩ ⟨S9, $rlist_2$, v⟩ ⟨S9, $rlist_2.n^+$, v⟩	⟨S8, $rlist_2$, n⟩ ⟨S8, $rlist_2.n^+$, n⟩ ⟨S9, $rlist_2$, v⟩ ⟨S9, $rlist_2.n^+$, v⟩	⟨S8, $rlist_2$, n⟩ ⟨S8, $rlist_2.n^+$, n⟩ ⟨S9, $rlist_2$, v⟩ ⟨S9, $rlist_2.n^+$, v⟩

Fig. 13. List reverse example and its upward exposed uses.

$DefUse[S_2] = DefUse[S_2] \cup \{s\}$
if $s \in DefUse[S_1] \wedge \langle s, p, n\rangle$
$\in UPEXP_{out}[S_2]$

The comparison is required to handle the case like the following program fragment:

S1 $p_1 = q_2$
S2 $p_1.n =$
S3 $q_2.n =$
S4 $= p_1.n$

S4 will be in the set $DefUse[S2]$, but not in $DefUse[S3]$, though S3 is a definition of the use of S4.

3.5. Examples

Figure 13 presents the iterations to compute the set of upward exposed uses of each statement of a loop that reverses a linked list. The uses of statements S8 and S9 enter the loop through the header of the loop S3 and reach the end of loop body S7. The use tuple $\langle S8, rlist_3, n\rangle$ matches the definition of S6 and hence is killed by the statement, and consequently a pair of definition and use is found. Another use tuple $\langle S8, rlist_3.(n)^+, n\rangle$ is transformed to $\langle S8, t.(n)^\star, n\rangle$, which will be represented by two tuples $\langle S8, t, n\rangle$ and $\langle S8, t.(n)^+, n\rangle$, by replacing the prefix $rlist_3.n$ by t. Similarly, the use tuple $\langle S9, rlist_3.(n)^+, v\rangle$ will be transformed to $\langle S9, t, v\rangle$ and $\langle S9, t.(n)^+, v\rangle$, while the last use tuple $\langle S9, rlist_3, v\rangle$ is unchanged. When the new tuples are propagated to S5, they will not be transformed. However, a new use $\langle S5, list_2, n\rangle$ is created. At the statement S4, the aliasing statement changes the entry $rlist_3$ of the tuple $\langle S9, rlist_3, v\rangle$ to $\langle S9, list_2, v\rangle$ while it leaves the others unchanged. The use tuples will be propagated to the header of the loop after being transformed by S3.

The tuples that are propagated to the header node S2 of the loop will be transformed by the ϕ statements to two sets of use tuples, one set will be propagated out of the loop to S1 and the other set will be passed back to the end of the loop body. The set that is propagated to S1 will be transformed by the aliasing statement, and those use tuples with entry $rlist_1$ will be removed after transformation since their access path expressions are empty strings. This process will repeat until no new patterns are created.

(a) Acyclic List (b) Cyclic List

Fig. 14. Possible shapes of list $list_1$.

This example demonstrates that the algorithm proposed in this paper can handle programs with destructive update operations [40]. It correctly identifies the definition-use chain between S6 and S8, and does not introduce the spurious definition-use chain between S6 and S5 even though $list_2$ and $rlist_3$ will be deemed as aliases by existing aliases algorithms [13,18,21,30, 48]. The use tuples at S1 mean that the uses of S5 and S9 are defined by the statements before the list reverse loop. Furthermore, the use tuples of S9 show that this approach works regardless of the shape of the linked list $list_1$, which can be either an acyclic list or a cyclic list as shown in Fig. 14. The same set of tuples will be produced by this approach even when $list_1$ is a cyclic list.

3.6. Complexity

For a normalized program with N statements, there are at most N definitions and uses in total. The pointer p of each use tuple $\langle s, p, n\rangle$ and pointers p and q of each definition tuple $\langle s, p, n, q\rangle$ will be represented by access path expressions and be transformed by transfer functions F_S and $F_{UPDEF[S]}$ during backward propagation process. Since access path expressions will be normalized (see Section 3.4.1), each access path expression p will have the form $p \equiv p_i.e_1.f_i.e_2.f_j.\cdots.e_m$ and consequently p has 2^m possible patterns, i.e. $p_i.e_1.f_i.e_2.f_j.\cdots.e_k$ $(1 \leqslant k \leqslant m)$. The number m is determined by the hierarchical configurations of data structures that are declared in programs. For example, if a linked list is declared in a program, then $m \equiv 1$ since access patterns will be factored into the forms $list$ and $list.(next)^+$. On the other hand, if a list of lists is declared, then m would be 2.

The algorithm *ComputeDefUseChains* follows the iterative data flow technique, and consequently it takes $d + 2$ iterations for a tuple to reach all its destinations, where d is the level of loop connectedness [24]. Furthermore, once each tuple reaches its destinations, it requires $3m$ more iterations to compute all possible access path expressions. Therefore, the time complexity of the algorithm is $d + 3m + 2$.

4. Flow-sensitive interprocedural algorithm

This algorithm to compute interprocedural definition-use chains of dynamic pointer-linked data structures is adapted from the interprocedural algorithm of computing definition-use chains of variables proposed by Harrold and Soffa [23]. Individual procedure will be analyzed using the algorithm presented in Section 3.4 to abstract intraprocedural information, which is used to construct an interprocedural flow graph (IFG). Intraprocedural information will then be propagated through the program via the IFG to obtain interprocedural information. The propagation will be performed on realizable paths only, i.e. the calling context will be observed. Finally, interprocedural definition-use chains of pointer-linked data structures will be computed using the local definition at each node in the IFG along with the propagated information.

4.1. Interprocedural flow graph (IFG)

A program will be represented by an *interprocedural flow graph* (*IFG*), which is based on the interprocedural flow graph [23] and the program summary graph [11]. An IFG is comprised of subgraphs, each of which abstracts local information of a procedure. The connections among subgraphs are determined by the call graph of the program [22].

There are four types of nodes in an IFG, *ENTRY*, *EXIT*, *CALL*, and *RETURN*. An ENTRY node represents the entry of a procedure and an EXIT node represents the exit at the end of a procedure. A CALL node represents the program point prior a procedure call while a RETURN node represents the program point after returning from the procedure call. Consequently, an ENTRY node and an EXIT node are created for each procedure, while a CALL node and a RETURN node are created for each call site. Intraprocedural information will be computed and annotated to appropriate nodes in every subgraph, and will be propagated along the IFG edges for interprocedural analysis.

A *reaching edge* in a subgraph indicates that definitions that reach the source of the edge might also reach the sink of the edge. In other words, a reaching edge indicates that there are flow paths from the source of the edge to the sink of the edge. For example, a reaching edge from an ENTRY node to a CALL node means that definitions that reach the entry of the procedure might reach the call site as well. Reaching edges represent the intraprocedural flow information between nodes of IFG subgraphs, and hence can be computed by intraprocedural analysis.

Binding edges represent the interactions among procedures by connecting subgraphs of the IFG. Let the ENTRY node and EXIT node of the procedure P be denoted as $entry^P$ and $exit^P$ respectively, and the CALL node and RETURN node of the procedure P to procedure Q be denoted as $call^{P \to Q}$ and $return^{P \to Q}$ respectively. A binding edge from node $call^{P \to Q}$ to node $entry^Q$ represents a procedure call from P to Q at a call site in P and the definitions of the actual parameters in P reach the formal parameters of Q. Similarly, a binding edge from node $exit^Q$ to node $return^{P \to Q}$ represents a return from procedure call and the last definitions of the formal parameters of Q reach the actual parameters of P after the call site.

Interreaching edges from CALL nodes to RETURN nodes represent the effects of called procedures on actual parameters at call sites [23,36]. An interreaching edge indicates definitions that reach the program point before the procedure call may reach the program point right after the return from the called procedure. Interreaching edges will be computed by an iterative algorithm executing on partially constructed IFG.

In summary, an IFG is a directed graph $IFG = [V_{IFG}, E_{IFG}]$, where V_{IFG} is the set of nodes, which can be further divided into CALL, RETURN, ENTRY, and EXIT nodes, and E_{IFG} is the set of edges, which can be categorized as call binding, return binding, reaching, and interreaching edges. Local definitions and uses will be gathered by intraprocedural algorithms and annotated to appropriate nodes in the subgraphs of IFG. The local information will be propagated through edges of IFG to compute the interprocedural definition-use chains.

4.2. Algorithm

The algorithm, *ComputeIPDefUseChains*, that computes the interprocedural definition-use chains of dynamic pointer-linked data structures is given in Fig. 15. This algorithm follows the iterative data flow technique. It first gathers the local uses and definitions at program points right before and after procedure calls and annotates the information on nodes of IFG subgraphs. It then solves data flow equations for reachable uses by propagating local information through the IFG using iterative techniques. Once the global information converges, this algorithm computes interprocedural definition-use chains by associating the local information with the propagated information. The algorithm, shown in Fig. 15, can be broken into four steps:

- Constructing the IFG subgraphs
- Constructing the IFG
- Propagating the local information
- Computing the definition-use chains

The program in Fig. 16 will be used as an example to demonstrate the interprocedural analysis. The example first creates a linked list, then calls a recursive procedure $reverse$ to reverse the linked list, and finally traverses the reversed list by a procedure $advance$ in a loop. This example is used to demonstrate how the flow-sensitive interprocedural algorithm handles recursive procedure calls and procedure calls within loops.

4.2.1. Constructing the IFG subgraphs

Each procedure is represented by an IFG subgraph. The steps to construct an IFG subgraph for a procedure are as follows:

- Creating IFG nodes
- Computing local information
- Building reaching edges

Creating IFG nodes

A pair of nodes, an ENTRY node and an EXIT node respectively, will be created for each procedure. The ENTRY node corresponds to the beginning of the procedure, while the EXIT node corresponds to the end of procedure. Similarly, a CALL node and a RETURN node are created for each procedure call in the procedure. The CALL node corresponds to the program point before the procedure call, and the RETURN node corresponds to the program point after the call. Separate CALL/RETURN pairs will be generated for procedure calls at different call sites in the procedure. Figure 17 shows the IFG nodes of the example program listed in Fig. 16. The first procedure call at S9 in $main$ is represented by the pair of nodes 1 and 2, and the second call site at S13 is represented by nodes 3 and 4. The ENTRY nodes of the procedures $reverse$ and $advance$ are node 5 and node 9 respectively and the EXIT nodes of $reverse$ and $advance$ are node 6 and

algorithm ComputeIPDefUseChains (P)
input:
  P: procedures
output:
  $IFG = [V_{IFG}, E_{IFG}]$
begin
{Step 1: construct subgraph for each procedure}
  for each P_i in P **do**
    create $entry^{P_i}$ and $exit^{P_i}$
    for each call $P_i \rightarrow Q$ at a call site **do**
      create $call^{P_i \rightarrow Q}$ and $return^{P_i \rightarrow Q}$
    Create dummy uses for each formal and actual parameter
    Perform intraprocedural analysis on P_i and annotate local information
    Create reaching edges and compute transfer functions
  end
{Step 2: construct the IFG}
  Create call and return binding edges
  Compute transfer functions and build interreaching edges
{Step 3: propagate USE through IFG}
  {Phase 1}
  $V = V_{IFG} - \{n | n \in V_{IFG} \wedge type(n) \equiv EXIT\}$
  $E = E_{IFG}$
  Propagate(V, E)
  {Phase 2}
  $V = V_{IFG}$
  $E = E_{IFG} - \{e | e \in E_{IFG} \wedge type(e) \equiv CallBinding\}$
  Propagate(V, E)
{Step 4: compute interprocedural DEF/USE chains}
  for each P_i in P
    Retrieve global information from IFG
    Compute interprocedural definition-use chains
  end
end

Fig. 15. Interprocedural algorithm.

node 10 respectively. The procedure call within the procedure $reverse$ are represented by the pair of nodes 7 and 8.

Computing local information

Once nodes of IFG subgraphs are created, local information will be gathered by intraprocedural data flow analysis and be annotated to appropriate IFG nodes. Since during backward iterative analysis local information of each procedure can be passed to callees through the RETURN nodes in the IFG and to callers through the ENTRY node, the sets of local upward exposed uses and definitions will be annotated to appropriate ENTRY and RETURN nodes. Therefore, the sets of definitions and uses that can be reached from the beginning of the regions represented by ENTRY and RETURN nodes in the IFG will be computed. The intraprocedural algorithm presented in the previous section will be performed to compute the sets of reachable definitions and uses. The use sets will be annotated to appropriate ENTRY and RETURN nodes as *UPEXP* sets while the definition sets will be annotated to appropriate ENTRY and RETURN nodes as *UPDEF* sets. The UPEXP or UPDEF set of an ENTRY node of a procedure is the set of uses or definitions that can be reached from the beginning of the procedure, while the UPEXP or UPDEF set of a RETURN node at a call site represents the set of uses or definitions that can be reached from the

```
{main procedure}
S1:    program main
S2:      list1 = nil
S3:      do i = 1, N
         list2 = φ(list1, list3)
S4:        temp = new()
S5:        temp.n = list2
S6:        list3 = temp
S7:      end do
S8:      rlist1 = nil
S9:      call reverse(list2 (list4), rlist1 (rlist2), t1 (t2))
S10:     list5 = t2
S11:     ptr1 = list5
S12:     do while(ptr2)
         ptr2 = φ(ptr1, ptr3)
S13:       call advance(ptr2 (ptr3))
S14:     end do
S15:   end
```

```
{reverse procedure}
S21:   procedure reverse(X1, Y1, q1)
S22:     if (X1 ≡ nil) then
S23:       q2 = Y1
S24:     else
S25:       p1 = X1.n
S26:       X1.n = Y1
S27:       call reverse(p1 (p2), X1 (X2), q1 (q3))
S28      end if
         X3 = φ(X1, X2); q4 = φ(q2, q3)
S29:   end (X3, Y1, q4)

{advance procedure}
S31:   procedure advance(ptr4)
S32:     ptr5 = ptr4.n
S33:   end (ptr5)
```

Fig. 16. Example program for interprocedural analysis.

return of a procedure call. The UPEXP and UPDEF sets are defined as follows, where P and Q represent procedures,

$$UPEXP[n] =$$

$$\begin{cases} \text{uses in P} & \text{reachable from the beginning of } P \\ & \text{if } n \text{ is } \textit{entry}^P \\ \text{uses in } P & \text{reachable from the return from} \\ P \rightarrow Q & \text{if } n \text{ is } \textit{return}^{P \rightarrow Q} \\ \emptyset & \text{otherwise} \end{cases}$$

$$UPDEF[n] =$$

$$\begin{cases} \text{definitions in } P \text{ reachable} & \text{from the beginning of } P \\ & \text{if } n \text{ is } \textit{entry}^P \\ \text{definitions in } P \text{ reachable} & \text{from the} \\ \text{return from } P \rightarrow Q & \text{if } n \text{ is } \textit{return}^{P \rightarrow Q} \\ \emptyset & \text{otherwise} \end{cases}$$

Building reaching edges

A reaching edge can be created between an ENTRY node and either a CALL or an EXIT node to indicate that there is flow path from the entry of a procedure to the program point before a procedure call or to the end of the procedure. Similarly, a reaching edge can be created from an RETURN node to an EXIT node or a CALL node to indicate there is a flow path from the return of a procedure call to the end of the procedure or the program point before a procedure call.

A reaching edge between two nodes in an IFG subgraph represents a region of code in the procedure that is abstracted by the IFG subgraph. Furthermore, it means definitions that reach the beginning of the code region might reach the end of region. Similarly, the reaching edge means uses that are reachable at the end of region might also be reachable at the beginning of the region. Therefore, in order to compute the reaching edges between nodes in the IFG subgraphs, local reaching definitions or reachable uses will be gathered.

In contrast to definitions and uses of fixed-location variables, each of which is a read or write reference to a single location, references to any locations that can be accessed through the links of pointer-linked data structures constitute definitions or uses of the data structures. Consequently, definitions of p that reach an IFG node can reach another IFG node if there are definitions of q at the destination node and one of the following conditions holds, where S and T are path strings:

- $p = q.S$
  p is a child of q, as shown in Fig. 18(a). In other words, the definitions stored in the set of locations that are reachable from p can be accessed through q, namely $q.S$.
- $q = p.S$
  q points to a child of p, as shown in Fig. 18(b).
- $p.S = q.T$
  Some children of p are also children of q, as shown in Fig. 18(c).

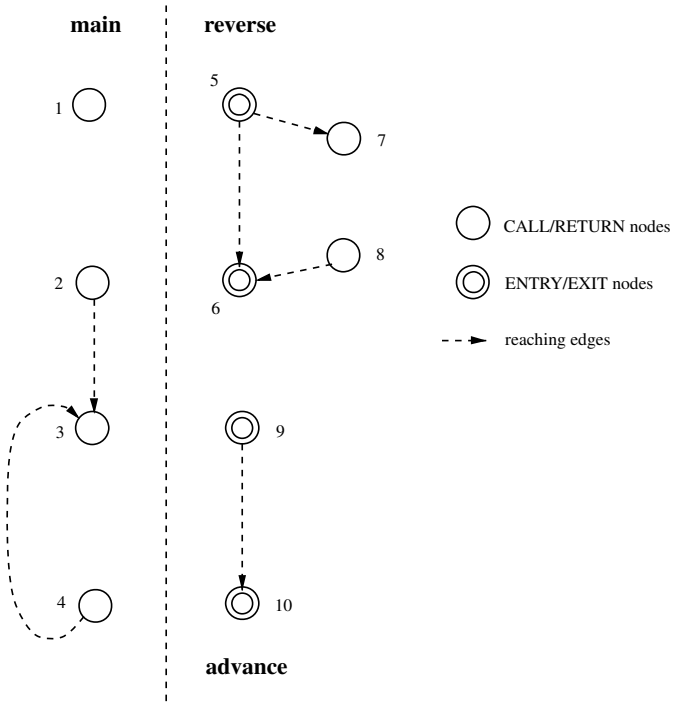


Fig. 17. IFG subgraphs of the example program.

Similarly, uses of q that are reachable from an IFG node can be reachable at another IFG node if there are uses of p and if one of the following conditions holds:

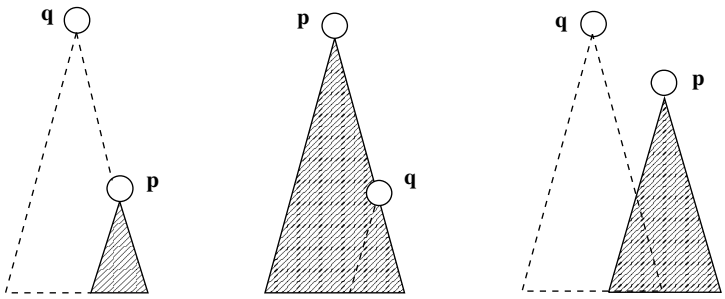
- $p = q.S$
- $q = p.S$
- $p.S = q.T$

In order to compute reaching edges between nodes in the IFG subgraph of the procedure, *dummy uses* will be added to the USE set for each formal parameter at the end of procedure and each actual parameter at call sites before the intraprocedural analysis is performed. Namely, for each parameter p which has recursively defined fields $f_1, f_2, \ldots, f_n$, a dummy use $p.T\ [p]$, where T is call the *dummy path* and $T = (f_1|f_2|\ldots|f_n)^+$, will be created. This form represents all possible patterns that can be accessed through parameter p. For example, the dummy use of the formal parameter at the end of procedure $advance$ in Fig. 16 is $ptr_5.(n)^+\ [ptr_5]$. The dummy uses will be propagated and transformed along with uses in $UPEXP$ sets. The same transformation defined by the intraprocedural algorithm proposed in the previous section will be performed on access path expressions of dummy uses. If any dummy uses reach ENTRY nodes or RETURN nodes of the procedure, reaching edges will be added to the IFG subgraph. Furthermore, the transfer functions of the reaching edges will be derived from the dummy uses that reach these ENTRY or RETURN nodes. For example, reaching edges are added between nodes 5 and 6, 5 and 7, and 8 and 6 of procedure $reverse$, and nodes 9 and 10 of procedure $traverse$ respectively, as shown in Fig. 17. The reaching between nodes 2 and 3 and nodes 4 and 3 will be added in the main procedure.

Transfer functions

The algorithm to compute transfer functions between IFG nodes is shown in Fig. 19. First, dummy uses will be added to the USE set for each formal parameter at the end of procedure and each actual parameter at call sites before the intraprocedural analysis is performed. These dummy uses will be propagated and transformed along with uses in $UPEXP$ sets. The same transformation that is applied to reachable uses by the intraprocedural algorithm will be applied to dummy uses as well, except for the link defining statements since the dummy paths can be shortened only by link defining statements and the pattern of transformation will be stored in the $[\,]$ field of each dummy use.

When a dummy pattern $p.S\,[v.W]$ reaches a link defining statement $p.f = q$, if the first field of S is f,



(a) $p = q.S$ (b) $q = p.S$ (c) $p.S = q.T$

Fig. 18. Reaching definitions and reachable uses.

i.e. $S = f.T$, the pattern $p.S$ will be transformed to $q.T$. That is, the entry point p of the path $p.S$ is replaced by q and the first field of path string S is removed. Furthermore, if $S = f.T$ is a dummy path, f will be appended to the path pattern $[v.W]$ and the new dummy use will be $q.T\,[v.W.f]$. For example, the dummy use $ptr_5.(n)^+\,[ptr_5]$ of procedure $advance$ of the example in Fig. 16 will be transformed by the link traversing statement $S32: ptr_5 = ptr_4.n$ to $ptr_4.n.(n)^+\,[ptr_5]$, whereas the dummy use $X_1.(n)^+\,[X_1]$ of the recursive call in procedure $reverse$ will be transformed by the link defining statement $S26: X_1.n = Y_1$ to $Y_1\,[X_1.n]$ and $Y_1.(n)^+\,[X_1.n]$, as shown in Fig. 20.

Once a dummy use $p.S\,[v.W]$ reaches the entry of the procedure or the program point at the return of a procedure call, a reaching edge will be added from the corresponding ENTRY or RETURN node in the IFG graph to the IFG node where the dummy use is originated. Furthermore, the transfer function of the edge can be derived from the reaching dummy use $p.S[v.W]$. The transfer function of the reaching edge will be derived from the dummy use by the following rules:

- If the reached dummy use contains a dummy path, it means the original patterns can be preserved by the region of code that is represented by the reaching edge. Therefore, the dummy use will be used as part of the transfer function of the reaching edge. Let T be the remaining dummy path, i.e. $S \equiv S'.T$, then T will be removed from S and the transfer function will be $p.S'\,[v.W]$. Furthermore, the unique field names in T, say $(f_1|f_2|\ldots|f_n)$, will be gathered and be put after the pattern in the $[\,]$ field. As a result, the transfer function will be $p.S'\,[v.W\,(f_i)]$, where $f_i = f_1|f_2|\ldots|f_n$.
- On the other hand, if the reached dummy use does not contain any dummy path, it means the original patterns will be killed by the reaching edge. Therefore, the dummy use will be discarded.

For example, the dummy use $(S29,\ Y_1.(n)^+\,[Y_1])$ will introduce $Y_1\,[Y_1\,(n)]$ to the transfer function of the reaching edge $\langle 5,\ 6\rangle$, while the dummy use $(S27,\ Y_1\,[X.n])$ will be discarded.

The transfer function of each reaching edge might contain multiple patterns that are derived from dummy uses. For example, the transfer function of the edge $\langle 5,\ 6\rangle$ in Fig. 17 has two patterns $Y_1\,[Y_1\,(n)]$ and $Y_1\,[q_4\,(n)]$, while the transfer function of the edge $\langle 5,\ 7\rangle$ is $X_1\,[p_1\,(n)]$, $q_1\,[q_1\,(n)]$, and $Y_1\,[X_1.n\,(n)]$. When a use reaches the sink node of the reaching edge, it will be compared by every pattern of the transfer function and be transformed if applicable. For a pattern $p.S\,[v.W\,(f_i)]$, where $f_i = (f_1|f_2|\ldots|f_n)$, of the transfer function of a reaching edge, the $[v.W\,(f_i)]$ part specifies the input pattern of the transfer function and $p.S$ represents the output pattern. In other words, the function will perform transformation at the sink node of the reaching edge only on the uses that have the form $v.W.T$, i.e. the prefix of the uses must be $v.W$. Furthermore, if the first field of T is f, f must be one of the fields of f_i, i.e. $f \in (f_1|f_2|\ldots|f_n)$, then the use $v.W.T$ will be transformed by the function to $p.S.T$. On the other hand, if the first field of T is not in f_i, no output will be produced. Therefore, a transfer function $F = p.S\,[v.W\,(f_i)]$, where $f_i = (f_1|f_2|\ldots|f_n)$, can be represented by the follow equation:

$$F_{p.S\,[v.W\,(f_i)]}(v.X) = \tag{12}$$

$$\begin{cases} p.S.T \text{ if } X \equiv W.T \wedge (T \equiv f.Y \wedge f \in f_i) \\ - \qquad \text{otherwise} \end{cases}$$

Figure 21 shows another example and its dummy uses at each statement. The procedure of the example swaps the branches of the actual parameter. A reaching edge will be created from the ENTRY

```
procedure ComputeTransferFunction (P, G)
input:
  P: procedure
  G: the IFG subgraph of P
begin
{Initialize dummy patterns}
  for each formal or actual parameter v of P do
    create a dummy use v.T[v],
      where T = (f1|f2|...|fn)+ and f1, f2, ..., fn are recursively defined fields of v
    add the dummy use to the USE set of the corresponding node
  end
{Propagate and transform dummy uses}
  while changed do
    for each node n of P in reverse topological order do
      for each dummy use p.S[v.W] in UPEXP_out[n], where S and W are sequences of fields, do
        case (statement type) do
        p = q: p.S ⟹ q.S
        p = q.f: p.S ⟹ q.f.S
        p.f = q:
          if S = f.T then
            p.S ⟹ q.T
            if S is a dummy path then W ⟹ W.f
          end
        end
        Union the new pattern to UPEXP_in[n]
      end
    end
  end
{Build reaching edges and compute transfer functions}
  for each dummy pattern p.S[v.W] that reaches a node n that is a call site or the entry do
    Identify the corresponding IFG node s in G of node n
    Identify the IFG node d where the dummy use is originated
    Build a reaching edge ⟨s, d⟩
    Compute the transfer function of the reaching edge
  end
end
```

Fig. 19. Algorithm for building reaching edges and computing transfer functions.

node to EXIT node of the swap procedure. The patterns $h\,[h]$, $h.left\,[h.right]$, and $h.right\,[h.left]$ that are derived from the dummy tuples without dummy paths can be discarded since they do not transform uses. The transfer function of the reaching edge contains two patterns: $h.right\,[h.left\,(left|right)]$ and $h.left\,[h.right\,(left|right)]$, which are derived from dummy uses $(S6,\ h.right.(left|right)^+\,[h.left])$ and $(S6,\ h.left.(left|right)^+\,[h.right])$. The transfer function means if the access path expression of a use that reaches S6 has the pattern $h.left.T$ (or $h.right.T$), then it will be changed to $h.right.T$ (or $h.left.T$). On the other hand, if the use pattern is exactly $h.left$ (or $h.right$), it will be killed by the reaching edge.

4.2.2. Constructing the IFG

For each procedure call from procedure P to procedure Q, a call binding edge $E_{call}^{P\rightarrow Q}$ will be added from the CALL node $call^{P\rightarrow Q}$ to the corresponding ENTRY node $entry^{Q}$. Furthermore, a return binding edge $E_{return}^{P\rightarrow Q}$ will be added from the EXIT node $exit^{Q}$ to the RETURN node $return^{P\rightarrow Q}$. For example, the edges $\langle 1,\ 5\rangle$, $\langle 3,\ 9\rangle$, and $\langle 7,\ 5\rangle$ in Fig. 22 are call bind-

(a) Program	(b) Dummy Uses
S21: procedure reverse(X_1, Y_1, q_1)	(S29, Y_1.(n)$^+$ [Y_1]) (S29, Y_1.(n)$^+$ [q_4]) (S27, X_1.n.(n)$^+$ [p_1]) (S27, q_1.(n)$^+$ [q_1]) (S27, Y_1 [X_1.n]) (S27, Y_1.(n)$^+$ [X_1.n])
S22: if ($X_1 \equiv$ nil) then	(S29, Y_1.(n)$^+$ [Y_1]) (S29, Y_1.(n)$^+$ [q_4])
S23: $q_2 = Y_1$	(S29, X_1.(n)$^+$ [X_3]) (S29, Y_1.(n)$^+$ [Y_1]) (S29, Y_1.(n)$^+$ [q_4])
S24: else	(S29, X_1.(n)$^+$ [X_3]) (S29, Y_1.(n)$^+$ [Y_1]) (S29, q_2.(n)$^+$ [q_4])
S25: $p_1 = X_1$.n	(S29, Y_1.(n)$^+$ [Y_1]) (S27, X_1.n.(n)$^+$ [p_1]) (S27, Y_1 [X_1.n]) (S27, Y_1.(n)$^+$ [X_1.n]) (S27, q_1.(n)$^+$ [q_1])
S26: X_1.n = Y_1	(S29, Y_1.(n)$^+$ [Y_1]) (S27, p_1.(n)$^+$ [p_1]) (S27, Y_1 [X_1.n]) (S27, Y_1.(n)$^+$ [X_1.n]) (S27, q_1.(n)$^+$ [q_1])
S27: call reverse(p_1 (p_2), X_1 (X_2), q_1 (q_3))	(S29, Y_1.(n)$^+$ [Y_1]) (S27, p_1.(n)$^+$ [p_1]) (S27, X_1.(n)$^+$ [X_1]) (S27, q_1.(n)$^+$ [q_1])
S28 end if	(S29, X_2.(n)$^+$ [X_3]) (S29, Y_1.(n)$^+$ [Y_1]) (S29, q_3.(n)$^+$ [q_4])
$X_3 = \phi(X_1, X_2)$; $q_4 = \phi(q_2, q_3)$	(S29, X_1.(n)$^+$ [X_3]) (S29, Y_1.(n)$^+$ [Y_1]) (S29, q_2.(n)$^+$ [q_4]) (S29, X_2.(n)$^+$ [X_3]) (S29, q_3.(n)$^+$ [q_4])
S29: end (X_3, Y_1, q_4)	(S29, X_3.(n)$^+$ [X_3]) (S29, Y_1.(n)$^+$ [Y_1]) (S29, q_4.(n)$^+$ [q_4])

Fig. 20. Propagation and transformation on dummy uses.

(a) Program	(b) Dummy Uses
S1: procedure swap(h)	(S6, h [h]) (S6, h.right [h.left]) (S6, h.right.(left|right)$^+$ [h.left]) (S6, h.left [h.right]) (S6, h.left.(left|right)$^+$ [h.right])
S2: left = h.left	(S6, h [h]) (S6, h.right [h.left]) (S6, h.right.(left|right)$^+$ [h.left]) (S6, h.left [h.right]) (S6, h.left.(left|right)$^+$ [h.right])
S3: right = h.right	(S6, h [h]) (S6, h.right [h.left]) (S6, h.right.(left|right)$^+$ [h.left]) (S6, left [h.right]) (S6, left.(left|right)$^+$ [h.right])
S4: h.right = left	(S6, h [h]) (S6, right [h.left]) (S6, right.(left|right)$^+$ [h.left]) (S6, left [h.right]) (S6, left.(left|right)$^+$ [h.right])
S5: h.left = right	(S6, h [h]) (S6, right [h.left]) (S6, right.(left|right)$^+$ [h.left]) (S6, h.right.(left|right)* [h])
S6: end (h)	(S6, h [h]) (S6, h.(left|right)$^+$ [h])

Fig. 21. Branch swap procedure and its dummy uses.

ing edges, while $\langle 6, 2\rangle$, $\langle 10, 4\rangle$, and $\langle 6, 8\rangle$ are return binding edges.

An interreaching edge will be then created to connect the CALL node and RETURN node of each site. The transfer function of the interreaching edge between the CALL node $call^{P\to Q}$ and the RETURN node $return^{P\to Q}$ is the composition of the transfer functions of the call binding edge $E_{call}^{P\to Q}$, the procedure Q, and the return binding edge $E_{return}^{P\to Q}$. In other words, the transfer function of the interreaching edge can be computed by the following equation:

$$F_{E_{return}^{P\to Q}} \circ F_Q \circ F_{E_{call}^{P\to Q}} \tag{13}$$

The operation $F_{E_1} \circ F_{E_2}$ to combine two transfer functions $F_{E_1} = p.W\,[v.R\,(f_1)]$ and $F_{E_2} = q.T\,[p.S\,(f_2)]$ is defined as:

$$p.W\,[v.R\,(f_1)] \circ q.T\,[p.S\,(f_2)] = \tag{14}$$

$$\begin{cases} q.T.X\,[v.R\,(f_1)] \text{ if } W = S.X \wedge car(X) \in f_2 \\ q.T\,[v.R.Y\,(f_2)] \text{ if } S = W.Y \wedge car(Y) \in f_1 \\ - \qquad\qquad \text{otherwise} \end{cases}$$

where p, q, and v are pointer variables, and R, S, T, W, and X are path strings. Specifically, if S is a

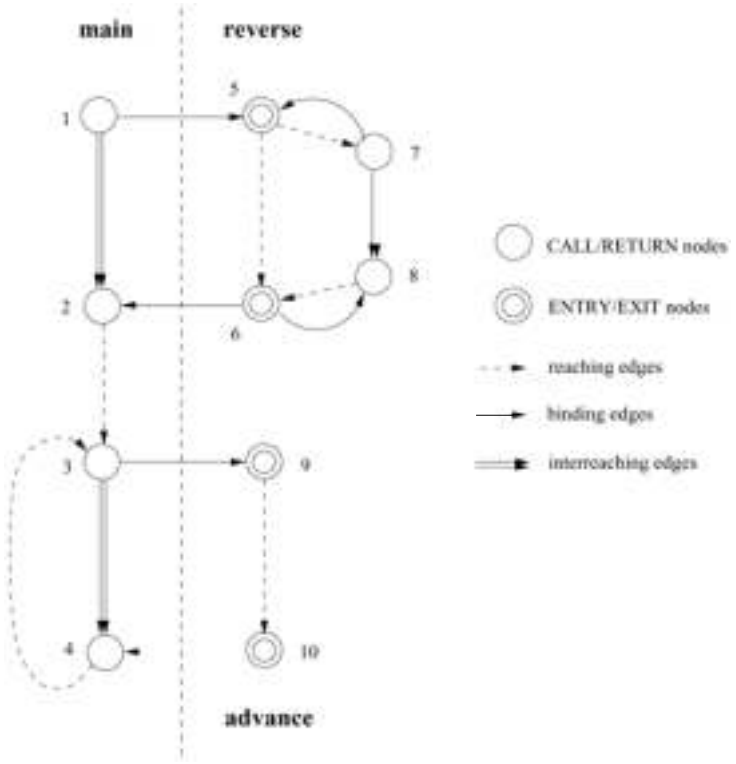


Fig. 22. IFG of the example program.

prefix of W, then the prefix S will be removed from W and if the first field of the rest of W is in f_2, i.e. $car(X) \in f_2$, then X will be inserted to $q.T$. On the other hand, if W is a prefix of S, the prefix W will be deleted from S and if the first field of the rest of S is in f_1, i.e. $car(Y) \in f_1$, then Y is appended to $v.R$. For example, the transfer function of the procedure $advance$ in Fig. 16(a) is $ptr_4.n\,[ptr_5\,(n)]$, and the transfer functions of call binding edge and return binding edge are $ptr_2\,[ptr_4\,(n)]$ and $ptr_5\,[ptr_3\,(n)]$, respectively. The transfer function of the interreaching edge will be $ptr_2.n\,[ptr_3\,(n)]$.

In order to compute the transfer function of each interreaching edge, the transfer function of its callee will have to be computed first. The transfer function of a procedure is the combination of transfer functions of edges in its IFG subgraph, which in turn might contain interreaching edges. Therefore, the process to compute transfer functions of procedures in a program is performed by a depth-first traversal through the call graph. Once the transfer functions of all the callees of a procedure are computed, the transfer functions of the corresponding interreaching edges can computed by the Eq. (13), and consequently the transfer function of the procedure can be easily computed by iteratively traversing the edges of its IFG subgraph to combine the transfer functions of the edges. However, when a program contains recursive procedure calls, cycles in the call graph will occur. Each cycle in the call graph can be identified by the depth-first traversal and isolated to compute the transfer functions of the recursive procedures on the corresponding IFG subgraphs.

The algorithm that computes the transfer functions of interreaching edges of an IFG *ComputeIE* is shown in Fig. 23. The call graph of a program is the input to the algorithm and the algorithm will traverse the procedures of the program following the depth-first search order of the call graph. The algorithm *ComputeIE* first initializes the number $DFS(N)$ of each call graph node to 0, and then calls the function *ComputeIERecursive* to recursively compute the transfer functions of the called procedures of the main procedure.

The function *ComputeIERecursive* assigns the number $DFS(N)$ of the call graph node N and increments the counter $count$ if N is visited the first time. Otherwise, the function returns without performing any computations. The function will recursively call itself to compute the transfer functions of the callees of N. Each recursive call to a callee of N returns the DFS ordering of the callee. If the DFS number is greater

```
procedure ComputeIE (CG)
input:
  CG: call graph
begin
{Initialization}
  count = 1
  for each node N of CG do DFS(N) = 0
{Traverse call graph in depth-rst-search  order}
  for each outgoing edge e of the root of CG do
    call ComputeIERecursive(callee(e))
    build an interreaching edge and compute the transfer function
  end
end
procedure ComputeIERecursive (N)
input:
  N: node of call graph
begin
{Return if the node has been visited}
  if DFS(N) > 0 return DFS(N)
{Recursively traverse the called procedures}
  DFS(N) = order = count++
  for each outgoing edge e of N do
    Active(e) = true
    child = ComputeIERecursive(callee(e))
    if order > child then order = child
    {Compute the transfer function if callee's transfer function is computed}
    if child > DFS(N) then Build an interreaching edge and compute the transfer function
  end
{Return if N is part of recursive calls}
  if order < DFS(N) then return order
{Isolate N and possibly its callees and compute the transfer function}
  for each incoming edge e of N do Active(e) = false
  call ComputeTransferFunction(N)
end
```

Fig. 23. Algorithm for building interreaching edges and their transfer functions.

than $DFS(N)$, it means the transfer function of the callee is computed, and hence the transfer function of the corresponding interreaching edge can be computed. Otherwise, it means N and the callee are in a cycle of the call graph that is induced by recursive calls. For example, when *ComputeIERecursive* visits the node 2 of the call graph in Fig. 25, which corresponds to the procedure $reverse$ of the example in Fig. 16, it assigns $DFS(2)$ the value 1. It then calls itself to compute the transfer function of the callees of the procedure $reverse$, and the returned DFS number is also 1. Since the returned DFS number is not greater than $DFS(2)$, the procedure $reverse$ is recursive.

After recursively visiting all the callees of N, *ComputeIERecursive* then calls the function *ComputeTransferFunction* to compute the transfer function of N. The function first identifies the IFG nodes and edges that correspond to the active nodes and edges of the call graph and marks the IFG nodes and edges active. The function uses a set $ReachList[s]$ to store the result of transfer function composition up to each IFG node s. It first initializes the $ReachList[s]$ to the transfer function of the edge $e \equiv \langle s, d \rangle$, $TransFunc[e]$, if the sink node

```
procedure ComputeTransferFunction (N)
input:
  N: node of call graph
begin
{Identify corresponding IFG nodes and edges}
  for each call graph node that is connected by active edges starting from N do
    Set the corresponding IFG nodes and edges active
    if the destination node d of the IFG edge e ≡ ⟨s, d⟩ is an EXIT node
    then ReachList[s] = TransFunc[e]
    else ReachList[s] = ∅
  end
{Compose edge transfer functions to compute the procedure transfer function}
  while changed do
    for each active IFG node d do
      for each active IFG edge e ≡ ⟨s, d⟩ do
        ReachList[s] = ReachList[s] ∪ (ReachList[d] ∘ TransFunc[e])
      end
    end
  end
{Compute transfer functions of procedures}
  for each active ENTRY node s do
    for each pattern p.S [v.W (f_i)] in ReachList[s] do
      if v is a formal parameter of the EXIT node of the same procedure
      then TransFunc[s] = TransFunc[s] ∪ p.S [v.W (f_i)]
    end
  end
end
```

Fig. 24. Algorithm for computing transfer function of procedure.

d is an EXIT node, and initializes to a empty set otherwise. The function then iteratively traverses the active IFG nodes and edges to compose patterns of transfer functions until no new patterns are added to any sets.

For example, when the procedure $reverse$ is analyzed, *ComputeTransferFunction* first identifies the IFG nodes 5, 6, 7, and 8, reaching edges $\langle 5, 6\rangle$, $\langle 5, 7\rangle$, and $\langle 8, 6\rangle$, and binding edges $\langle 7, 5\rangle$ and $\langle 6, 8\rangle$. The transfer function of edge $\langle 5, 6\rangle$ will be copied to $ReachList[5]$, i.e. $ReachList[5] = \{Y_1\,[Y_1\,(n)],\ Y_1\,[q_4\,(n)]\}$, and the transfer function of edge $\langle 8, 6\rangle$, $\{X_2\,[X_3\,(n)],\ q_3\,[q_4\,(n)]\}$, will be copied to $ReachList[8]$. *ComputeTransferFunction* then traverses the active IFG edges to compute the $ReachList$ of each node until no changes occur. Finally, it examines the $ReachList[5]$ of the ENTRY node of the procedure *reverse* and obtains the transfer function of the procedure, $Y_1\,[X_3\,(n)]$, $Y_1\,[Y_1\,(n)]$, $Y_1\,[q_4\,(n)]$, $Y_1\,[X_3.n^+\,(n)]$, $Y_1\,[Y_1.n^+\,(n)]$, and $Y_1\,[q_4.n^+\,(n)]$.

Once the transfer function of the callee is computed, the interreaching edge that connects the CALL node and RETURN node of the call site can be created, and the transfer function of the interreaching edge can be computed by the Eq. (13). Therefore, the interreaching edges will be created between nodes 1 and 2, 3 and 4, and 7 and 8, as shown in Fig. 22. The transfer function of the interreaching edge $\langle 1, 2\rangle$ will be $rlist_1\,[list_4\,(n)]$, $rlist_1\,[rlist_2\,(n)]$, $rlist_1\,[t_2\,(n)]$, $rlist_1\,[list_4.n^+\,(n)]$, $rlist_1\,[rlist_2.n^+\,(n)]$, and $rlist_1\,[t_2.n^+\,(n)]$.

Take the branch swap example in Fig. 21. Its transfer function can be computed by a single iteration since its IFG does not have cycles, and the transfer function contains the patterns $h.left\,[h.right\,(left|right)]$ and $h.right\,[h.left\,(left|right)]$. The transfer function means if the access path expression of a use that reaches S6 has the pattern $h.left.T$ (or $h.right.T$), then it will be changed to $h.right.T$ (or $h.left.T$). On the other hand, if the use pattern is exactly $h.left$ (or $h.right$), it will be killed by the reaching edge. The only pattern that can pass the procedure without any

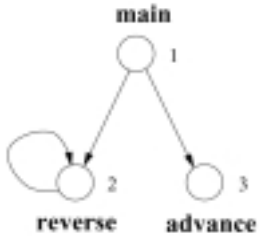


Fig. 25. Call graph of the example program.

transformations is h, but this pattern will not be propagated since it is not a definition or use of any links of pointer-linked data structures.

If the procedure *swap* is modified to a recursive procedure, as shown in Fig. 26, its IFG (and call graph too) contains cycles. The transfer functions of reaching edges e_1 and e_2 from the ENTRY to CALL nodes are $h.left\,[left_1\,(left|right)]$ and $h.right\,[right_1\,(left|right)]$, respectively. Similarly, the transfer functions of e_5 and e_6 from RETURN nodes to the EXIT node are $left_2\,[h.right\,(left|right)]$ and $right_2\,[h.left\,(left|right)]$, respectively. The transfer function of the edge e_3 is $\emptyset$ since no dummy paths can pass the procedure. As a result, the transfer function of the procedure *swap* will be $\emptyset$. It means any patterns, say $h.T$ where T is a path string, that is reachable at the end of the procedure S10 are not reachable at the entry of the procedure S1.

4.2.3. Propagating the local information

After the IFG graph is complete and local definition and use sets are annotated on IFG nodes, the next step is to propagate local information throughout the IFG to obtain interprocedural information. In other words, local definitions and uses annotated on nodes in IFG subgraphs will be propagated to compute the interprocedural reachable definition and use sets of each node in the IFG, which represent the definitions and uses of nonlocal variables in other procedures. These interprocedural reachable definitions and uses are computed by propagating UPDEF[n] and UPEXP[n] sets backward throughout the IFG, while taking into account of the calling context of the called procedures.

Preserving the calling context of called procedures is important, since for interprocedural data flow analysis not all paths in the graph representation correspond to real program executions. To preserve the call context, only the paths that agree with call sequence should be traversed, that is, only the *realizable* paths on IFG will be traversed [30]. A realizable path is a path whenever a procedure on this path returns, it returns to the call site which invokes it. The propagation process will be performed on realizable paths in IFG.

To preserve the calling context of called procedures, a two-phase process will be performed to propagate definitions and uses. In the first phase, all EXIT nodes and incoming and outgoing edges connected to any EXIT nodes will be excluded. Information is allowed to flow throughout the rest of the IFG. In the second phase, information that reaches EXIT nodes will be propagated to IFG nodes through all IFG edges except for call binding edges. Consequently, the first phase only processes the ENTRY, CALL, and RETURN nodes, and propagates the definitions and uses that can be reached in called procedures over the call binding edges. In this phase, only definitions and uses of called procedures will be propagated to calling procedures. No definitions and uses will be passed to the called procedures since the EXIT node of every procedure is excluded in this phase. Then the second phase propagates the definitions and uses that can be reached in calling procedures over the return binding edges, reaching edges, and interreaching edges. In this phase, each procedure accepts the definitions and uses from the calling procedures, and propagates the incoming definition and uses throughout its IFG subgraph.

This two-phase propagation process preserves the calling context of called procedures and ensures only realizable paths in IFG are traversed. As demonstrated by Fig. 22, the edges $\langle 6, 2\rangle$ and $\langle 6, 8\rangle$ will be disabled during the first phase. Consequently, information will not propagated backward through the unrealizable paths like $7 \to 3 \to 6 \to 2$ and $1 \to 3 \to 6 \to 8$.

The following set of data flow equations are used to compute the IPUSE set and IPDEF set before and after node S of the IFG:

$$IPUSE_{out}[S] = \bigcup_{e \equiv \langle S, D\rangle \in E} F_e(IPUSE_{in}[D]) \tag{15}$$

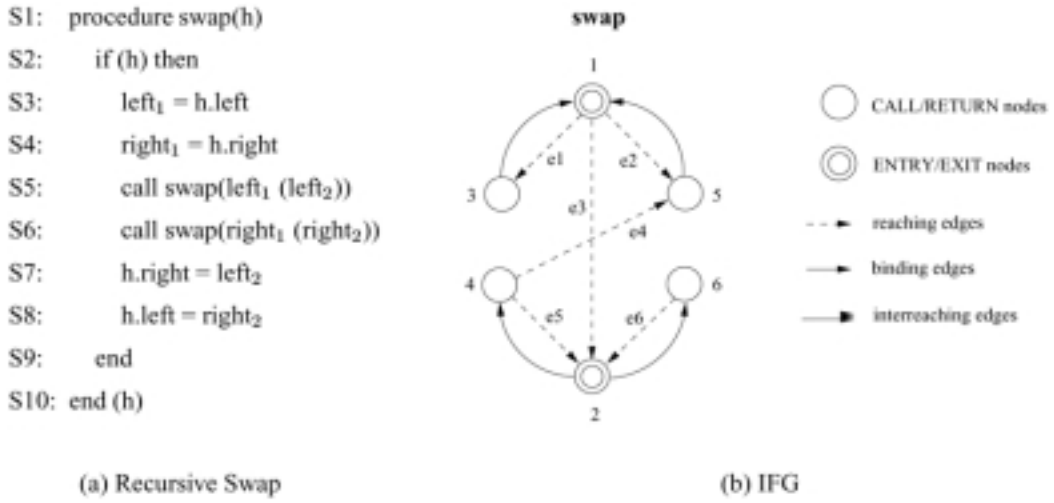
$$IPDEF_{out}[S] = \bigcup_{e \equiv \langle S, D\rangle \in E} F_e(IPDEF_{in}[D]) \tag{16}$$

$$IPDEF[S] = F_{IPDEF_{out}[S]}(IPDEF_{out}[S]) \tag{17}$$

$$IPUSE[S] = F_{IPDEF_{out}[S]}(IPUSE_{out}[S]) \tag{18}$$

$$IPUSE_{in}[S] = IPUSE[S] \cup UPEXP[S] \tag{19}$$

$$IPDEF_{in}[S] = IPDEF[S] \cup UPDEF[S] \tag{20}$$



(a) Recursive Swap (b) IFG

Fig. 26. Recursive swap procedure.

where F_e is the transfer function of edge e, as defined in Equation 13. The algorithm to propagate upward exposed uses is shown in Fig. 27. In the first phase, the inputs to the *propagate* procedure, i.e. the set of nodes V and the set of edges E, are

$$V = V_{IFG} - \{n | n \in V_{IFG} \wedge type(n) = EXIT\}$$

$$E = E_{IFG}$$

In the second phase, the set of nodes V and the set of edges E are

$$V = V_{IFG}$$

$$E = E_{IFG} - \{e | e \in E_{IFG} \wedge type(e) = CallBinding\}$$

Figure 28 lists the set of uses of each node after phases of propagation. Before the first phase, the sets of local uses of the example program that are annotated on the IFG are $\{\langle S25, X_1, n\rangle\}$ at node 5 and $\{\langle S32, ptr_2, n\rangle\}$ at node 9. During the first phase, the use $\langle S32, ptr_2, n\rangle$ will propagated from the ENTRY node of the procedure *advance* to the main procedure, and then be propagated and transformed along the cycle that connects the IFG nodes 3 and 4. As a result, the sets of uses at nodes 3 and 4 are $\{\langle S32, ptr_2, n\rangle\ \langle S32, ptr_2.(n)^+, n\rangle\}$, and $\{\langle S32, ptr_3, n\rangle\ \langle S32, ptr_3.(n)^+, n\rangle\}$, respectively. Furthermore, the uses will be propagated along the reaching edge $\langle 2, 3\rangle$ and the interreaching edge $\langle 1, 2\rangle$ to reach nodes 1 and 2. Similarly, the use $\langle S25, X_1, n\rangle$ will be propagated along the cycle of nodes 5 and 7, and the sets of uses at nodes 5 and 7 are $\{\langle S25, X_1, n\rangle\ \langle S25, X_1.(n)^+, n\rangle\}$, and $\{\langle S25, p_1, n\rangle\ \langle S25, p_1.(n)^+, n\rangle\}$, respectively. The set of uses at node 5 will then be passed to node 1 through the call binding edge $\langle 1, 5\rangle$.

During the second phase, the set of uses at node 4 will be propagated back to the procedure *advance* via the return binding edge $\langle 10, 4\rangle$, and hence the set of uses of node 10 is $\{\langle S32, ptr_3, n\rangle\ \langle S32, ptr_3.(n)^+, n\rangle\}$. Similarly, the set of uses at node 3 will be passed to the procedure *reverse* through the edge $\langle 6, 3\rangle$, and consequently the use set of node 6 is $\{\langle S32, q_4, n\rangle\ \langle S32, q_4.(n)^+, n\rangle\}$. The set of uses of node 6 will then be propagated along the edges of the IFG subgraph to another nodes of the same procedures, as shown by the last column of the table in Fig. 28.

4.2.4. Computing the definition-use chains

After the first three steps, interprocedural reachable information is stored on CALL and EXIT nodes of IFG. The set of definitions and uses will be retrieved from each CALL or EXIT node and will be annotated to the corresponding IG node, and then the final step will be performed on IG to compute interprocedural definition-use chains. The intraprocedural algorithm presented in the previous section will be performed on each procedure of IG to compute interprocedural definition-use chains.

From the table in Fig. 28, the uses that will be retrieved from the IFG node 1 and included in the set $UPEXP[S9]$ are $\langle S25, list_2, n\rangle$, $\langle S25, list_2.(n)^+, n\rangle$, $\langle S32, rlist_1, n\rangle$, and $\langle S32, rlist_1.(n)^+, n\rangle$. These uses will be propagated backward on the IG and trans-

```
procedure Propagate (V, E)
input:
  V: set of nodes
  E: set of edges
begin
  while changed
    for each node S of type V
      for each node D that is sink of edge e = ⟨S, D⟩ in E
        IPUSE_out[S] = IPUSE_out[S] ∪ F_e(IPUSE_in[D])
        IPDEF_out[S] = IPDEF_out[S] ∪ F_e(IPDEF_in[D])
      end
      IPDEF[S] = F_IPDEF_out[S](IPDEF_out[S])
      IPUSE[S] = F_IPDEF_out[S](IPUSE_out[S])
      IPUSE_in[S] = IPUSE[S] ∪ UPEXP[S]
      IPDEF_in[S] = IPDEF[S] ∪ UPDEF[S]
    end
  end
end
```

Fig. 27. Algorithm for propagating reachable use sets.

formed by the statements. When the uses are passed to S8, the uses $\langle S32, rlist_1, n\rangle$ and $\langle S32, rlist_1.(n)^+, n\rangle$ will be killed while the other two remain unchanged and be passed to the header node S3 of the loop. The two uses will be changed to $\langle S25, temp, n\rangle$ and $\langle S25, temp.(n)^+, n\rangle$ by the statement S6. When the use $\langle S25, temp, n\rangle$ is propagated to S5, it will be killed since it matches the definition of S5 and consequently a definition-use chain between S5 and S25 is identified. The other use $\langle S25, temp.(n)^+, n\rangle$ will be transformed by S5 to $\langle S25, list_2, n\rangle$ and $\langle S25, list_2.(n)^+, n\rangle$, and the new patterns will be passed back to the end of loop body for another iteration.

The set of uses that will be retrieved from the IFG node 7 are $\langle S25, p_1, n\rangle$, $\langle S25, p_1.(n)^+, n\rangle$, $\langle S32, X_1, n\rangle$, and $\langle S32, X_1.(n)^+, n\rangle$. Figure 29 shows the process of propagation and transformation on the set uses by the code fragment of the procedure *reverse*. Only the $UPEXP_{in}[n]$ sets are shown in the table. When the uses are propagated to the statement S26, the first two uses are not transformed since they do not match the definition $X_1.n$. On the other hand, the $\langle S32, X_1, n\rangle$ is killed by S26 since it matches the definition, and consequently S26 is the definition of the use of S32. The last use $\langle S32, X_1.(n)^+, n\rangle$ will also be transformed by S26, and the patterns $\langle S32, Y_1, n\rangle$ and $\langle S32, Y_1.(n)^+, n\rangle$ are created.

4.3. Complexity

Each procedure and call site is modeled by a pair of IFG nodes. Therefore, if a program contains P procedures and C call sites in total, there are $N \equiv 2P + 2C$ nodes on the IFG of the program. The total number of call binding edges and return binding edges is $2C$, and the number of interreaching edges is C. The number of reaching edges is determined by the number of call sites in each procedure. If the number of call sites per procedure is denoted as c, the maximum number of reaching edges in a procedure would be $2c + c^2 + 1$, where $2c$ is the number of reaching edges from the Entry node to CALL nodes and from Return nodes to the EXIT node, c^2 is the number of reaching edges between EXIT nodes and CALL nodes, and the last item is the reaching edge from ENTRY to EXIT.

The first step and last step of the algorithm *ComputeIPDefUse* basically perform the intraprocedural analysis described in Section 3.4, which has the time complexity of $d + 3m + 2$ iterations. In Step 2, the computation of interreaching edges and their transfer functions is performed by a depth-first traversal through the call graph. The computation is linear in the number of procedure if there are no recursive procedure calls. The time complexity of Step 3 is $\mathcal{O}(N^2)$ because of the *Propagate* procedure.

5. Applications

This section presents a couple of applications of definition-use information of pointer-linked data structures.

5.1. Identifying parallelism in programs with cyclic graphs

5.1.1. Identifying parallelism in traversal references

This application is motivated by the observation – although many programs create pointer-linked data structures which appear to be cyclic overall, they usually follow acyclic structures to access all nodes on the data structures. For instance, graph algorithms frequently extract acyclic structures, such as spanning trees, from cyclic graphs and traverse the graphs following the links of the acyclic structures, while the rest of edges are merely used to reference values on neighboring nodes. Furthermore, pointer-linked data structures can have unbounded numbers of nodes and are commonly traversed by loops or recursive procedures. The edges

Node	Before Phase 1	After Phase 1	After Phase 2
1		$\langle S25, list_2, n\rangle$ $\langle S25, list_2.(n)^+, n\rangle$ $\langle S32, rlist_1, n\rangle$ $\langle S32, rlist_1.(n)^+, n\rangle$	$\langle S25, list_2, n\rangle$ $\langle S25, list_2.(n)^+, n\rangle$ $\langle S32, rlist_1, n\rangle$ $\langle S32, rlist_1.(n)^+, n\rangle$
2		$\langle S32, t_2, n\rangle$ $\langle S32, t_2.(n)^+, n\rangle$	$\langle S32, t_2, n\rangle$ $\langle S32, t_2.(n)^+, n\rangle$
3		$\langle S32, ptr_2, n\rangle$ $\langle S32, ptr_2.(n)^+, n\rangle$	$\langle S32, ptr_2, n\rangle$ $\langle S32, ptr_2.(n)^+, n\rangle$
4		$\langle S32, ptr_3, n\rangle$ $\langle S32, ptr_3.(n)^+, n\rangle$	$\langle S32, ptr_3, n\rangle$ $\langle S32, ptr_3.(n)^+, n\rangle$
5	$\langle S25, X_1, n\rangle$	$\langle S25, X_1, n\rangle$ $\langle S25, X_1.(n)^+, n\rangle$	$\langle S25, X_1, n\rangle$ $\langle S25, X_1.(n)^+, n\rangle$ $\langle S32, Y_1, n\rangle$ $\langle S32, Y_1.(n)^+, n\rangle$
6			$\langle S32, q_4, n\rangle$ $\langle S32, q_4.(n)^+, n\rangle$
7		$\langle S25, p_1, n\rangle$ $\langle S25, p_1.(n)^+, n\rangle$	$\langle S25, p_1, n\rangle$ $\langle S25, p_1.(n)^+, n\rangle$ $\langle S32, X_1, n\rangle$ $\langle S32, X_1.(n)^+, n\rangle$
8			$\langle S32, q_3, n\rangle$ $\langle S32, q_3.(n)^+, n\rangle$
9	$\langle S32, ptr_4, n\rangle$	$\langle S32, ptr_4, n\rangle$	$\langle S32, ptr_4, n\rangle$ $\langle S32, ptr_4.(n)^+, n\rangle$
10			$\langle S32, ptr_5, n\rangle$ $\langle S32, ptr_5.(n)^+, n\rangle$

Fig. 28. Propagation of uses on IFG.

Statements	Iteration 1	Iteration 2
S25: $p_1 = X_1.n$	$\langle S25, X_1, n\rangle$ $\langle S25, X_1.(n)^+, n\rangle$ $\langle S32, Y_1, n\rangle$ $\langle S32, Y_1.(n)^+, n\rangle$	$\langle S25, X_1, n\rangle$ $\langle S25, X_1.(n)^+, n\rangle$ $\langle S32, Y_1, n\rangle$ $\langle S32, Y_1.(n)^+, n\rangle$
S26: $X_1.n = Y_1$	$\langle S25, p_1, n\rangle$ $\langle S25, p_1.(n)^+, n\rangle$ $-\langle S32, Y_1, n\rangle$ $\langle S32, Y_1.(n)^+, n\rangle$	$\langle S25, p_1, n\rangle$ $\langle S25, p_1.(n)^+, n\rangle$ $-\langle S32, Y_1, n\rangle$ $\langle S32, Y_1.(n)^+, n\rangle$
S27:	$\langle S25, p_1, n\rangle$ $\langle S25, p_1.(n)^+, n\rangle$ $\langle S32, X_1, n\rangle$ $\langle S32, X_1.(n)^+, n\rangle$	$\langle S25, p_1, n\rangle$ $\langle S25, p_1.(n)^+, n\rangle$ $\langle S32, X_1, n\rangle$ $\langle S32, X_1.(n)^+, n\rangle$

Fig. 29. Propagating and transforming uses in procedure *reverse*.

along which loops or recursive procedures traverse a pointer-linked data structure constitute traversal patterns, and can be viewed as the skeleton of the pointer-linked data structure. On the other hand, the remaining edges of the recursive data structure are generally used to reference values on other nodes. Accordingly, it will be beneficial if compilers can differentiate links of recursive data structures based on the traversal patterns when pointer analysis is performed.

The approach to identify parallelism in programs with cyclic graphs can be broken into three steps [29].

- *Gather Traversal Patterns and Compute Definition-Use Chains of Recursive Data Structures*
  Definition-use chains of recursive data structures will be computed and meanwhile traversal patterns of iterative or recursive program constructs, such as loops or recursive functions, will be gathered. For each statement that references a link of a graph, all corresponding statements that might define the link will be identified using the information of definition-use chains.
- *Perform Traversal-Pattern-Sensitive Shape Analysis*
  Once the statements that construct the graphs accessed by traversal patterns are identified, traversal-pattern-sensitive shape analysis will be performed to estimate possible shapes of the traversal patterns.
- *Perform Dependence Analysis*
  Dependence test is performed to determine if access conflicts occur between the sets of read and write references based on data reference patterns and the result of shape analysis on traversal patterns and overall data structures.

5.1.2. Identify parallelism in construction operations

Although the above approach can usually parallelize the most time-consuming part of these programs (the graph traversal operations account for over 90% of total execution times in most programs), the rest of programs that accounts for less than 10% will dominate the execution on multiprocessor systems according to Amdahl's law [5]. For example, with only 10% of ex-

ecution being sequential, the maximum speedup is 10 irrespective of the number of processors. Similarly, if 20% of the computation is executed sequentially, the maximum speedup will then be limited to 5. In order to achieve good scalability, the graph construction operations must be parallelized as well.

The requirement that iterations of a loop which constructs the adjacency lists can be parallelized is that all traversal references on the adjacency lists are independent as well. The approach can be broken into two steps.

- Identify parallelism in traversal references on cyclic graphs
- Identify parallelism in graph construction operations

The first step is performed by the method described in the above section. Once the dependence analysis on traversal references is performed, the second step can be performed to examine the following conditions to determine if construction operations are independent as well:

- Every graph that is connected by the primary traversal edges is a list of lists,
- The iterations of all loops that traverse the adjacency lists are independent once the graphs are constructed,
- The adjacency lists accessed by the traversal references are created by the construction operations, and
- The main lists are not modified between the traversal references and the construction operations.

If the conditions are met, the construction operations of graphs can be parallelized to enhance the scalability of programs.

5.2. Slicing on programs with dynamic recursive data structures

Slicing is a technique that extracts from a program statements relevant to a particular criteria [8,47]. It has been applied to many fields, such as program debugging [1,45], parallelization [16,46], and program integration [26], etc.

The definition-use chains of dynamic recursive data structures will be a useful information to compute slices of program with dynamic recursive data structures. Based on the technique to find program slices proposed by Weiser [47], a slice of a program P that meets the *slicing criterion* $C = \langle i, V \rangle$, where i is a statement in P and V is a subset of variables in P, can be computed by the function R_C^0 that maps statements to sets of variables:

R_C^0 = all variables v such that:

1. $n \equiv i \wedge v \in V$, or
2. n is an immediate predecessor of a node m such that
   (a) $v \in USE[n]$ and there is a w such that $w \in DEF[m] \wedge w \in R_C^0(m)$, or
   (b) $v \notin DEF[n] \wedge v \in R_C^0(m)$.

If $p \in V$ of a slicing criterion $C = \langle i, V \rangle$ is a pointer, the computation of R_C^0 function can follow SSA edges, which represent the definition-use relationships of pointers. On the other hand, if $p.f$ is in V of $C = \langle i, V \rangle$, the sets $DEF[n]$ and $DefUse[n]$ of each statement n can be used to compute $R_C^0(n)$.

6. Implementation and experimental results

This section presents the experimental results of the algorithms that are proposed in previous sections. The intraprocedural and interprocedural algorithms that compute the definition-use chains of dynamic pointer-linked data structures have been implemented on the *ParaScope* parallel programming environment developed at Rice University [14]. The compiler accepts programs with dynamic pointer-linked data structures that are written in Fortran 90, and computes definition-use pairs between link defining statements and link traversing statements.

6.1. Benchmarks

A set of programs that create and traverse various types of pointer-linked data structures is chosen as the benchmarks, and the data structures that are built by the benchmark programs are listed in Table 1. The following is a brief description of each benchmark program.

Barnes-Hut It is a hierarchical N-body program that computes gravitational forces with asymptotic complexity of $\mathcal{O}(N \log N)$ [7]. It creates a hierarchical octree to represent spatial locations of N bodies in a three-dimensional space. Each leaf node of the octree points to a body The simulation proceeds over time steps. At each time step, the program builds an octree, computes the net force of every body, and then updates the positions of N bodies.

Table 1
Data structures built by benchmark programs

Program	Description	Data Structures
Barnes-Hut	N-body simulation	Leaf-linked tree
EM3D	Simulation of electromagnetic waves	Bipartite
Moldyn	Molecular dynamics	Cyclic list
Power	Power system optimization	Hierarchical tree
Reverse	List reversal	Cyclic list
TreeAdd	Tree addition	Binary tree

EM3D This program models the propagation of electromagnetic waves through objects in three dimensions [32]. It builds a bipartite graph that contains nodes representing electric and magnetic field values. The dependences between E nodes (electric field) and H nodes (magnetic field) are represented by the edges between them. At each time step, new values of E nodes are computed from a weighted sum of neighboring H nodes, and then the same computation is performed for the H nodes.

Moldyn It is a molecular dynamics simulation program [9,34]. It computes interactions of a set of molecules that are initially uniformly distributed over a three-dimensional space with a Maxwellian distribution of velocities. The computation of forces is limited to the interactions between molecules within a cutoff radius. The molecules of the system are represented by the nodes of a linked list, and edges that represent the interactions of neighboring molecules within cutoff range form a cyclic list, which is updated periodically to reflect the movements of molecules.

Power This program solves the power system optimization problem, which determines the prices that will optimize the benefit to the community [31]. The power lines from a power station to customers are represented by a hierarchical tree: root (power station) → lateral nodes → branch nodes → leaf nodes (customers).

Reverse It is similar to the example shown in Fig. 16 that uses a recursive procedure to reverse a linked list, which in turn is adapted from the destructive list-reversal function in Deutsch [18]. The difference is that this example first creates a cyclic list calls a recursive procedure to reverse the cyclic list, and then traverses the reversed cyclic list.

TreeAdd The program recursively walks a tree and computes the sum of values of tree nodes [37]. It first calls a recursive procedure to create a balanced binary tree, and then calls the other recursive procedure to traverse the tree and adds the sums of subtrees.

Table 2
Characteristics of benchmark programs

Program	Procedures	Lines	DEF/USE Pairs
Barnes-Hut	15	731	20
EM3D	5	289	16
Moldyn	6	476	10
Power	17	976	5
Reverse	3	80	5
TreeAdd	3	56	2

This set of benchmark examples is chosen because they use different types of program constructs, e.g. recursive procedures, procedures in loops, destructive operations, etc., to create and traverse various types of pointer-linked data structures, ranging from tree-like structures to cyclic pointer-linked data structures. Table 2 lists more detailed information of these programs. The numbers of procedures in these programs vary from 3 to 17 and the numbers of statements range from 56 to 976 (including declaration statements and comments). The numbers of definition-use pairs between link defining statements and link traversing statements in these benchmark examples are from 2 to 20.

6.2. Experimental results

After intraprocedural analysis phase is performed, the IFG subgraphs of procedures will be constructed and local definition and use tuples will be gathered. Table 3 presents the statistics after the intraprocedural phase. The first section shows the numbers of IFG nodes and IFG edges, and the maximum and average numbers of tuples in transfer functions of IFG reaching edges. The next section presents the maximum and average numbers of use tuples in $UPEXP$ sets and the third section presents the maximum and average numbers of definition tuples in $UPDEF$ sets. If the programs contain definition-use pairs between link defining statements and link traversing statements within the same procedures, they will be identified in this phase and the last column shows the numbers of intraprocedural definition-use pairs.

Interprocedural analysis phase propagates local information gathered in intraprocedural analysis phase to

Table 3
Statistics after intraprocedural analysis

Program	IFG	IFG	TransFunc		UPEXP		UPDEF		DEF/USE
	Nodes	Edges	Max	Avg	Max	Avg	Max	Avg	Pairs
Barnes-Hut	74	33	3	1.36	4	0.80	10	1.05	5
EM3D	20	7	2	1.14	11	4.97	14	7.10	8
Moldyn	22	11	1	1.00	11	4.30	9	1.84	2
Power	94	27	1	1.00	2	0.11	2	0.03	0
Reverse	12	7	4	2.00	6	1.46	11	2.08	2
TreeAdd	18	10	1	1.00	2	0.17	2	0.17	0

Table 4
Statistics after interprocedural analysis

Program	IFG	IFG	TransFunc		UPEXP		UPDEF		DEF/USE
	Nodes	Edges	Max	Avg	Max	Avg	Max	Avg	Pairs
Barnes-Hut	74	99	10	1.30	14	3.18	14	1.67	20
EM3D	20	22	2	0.82	25	18.00	14	7.10	16
Moldyn	22	26	1	0.96	21	12.50	11	4.78	10
Power	94	117	14	1.04	21	4.14	2	0.03	5
Reverse	12	16	18	4.06	9	4.35	11	2.08	5
TreeAdd	18	28	4	1.21	8	2.38	2	0.17	2

compute interprocedural definition-use chains. Table 4 shows measurements of the interprocedural phase. The layout of the table is the same as Table 3. Binding edges between IFG subgraphs and interreaching edges between CALL nodes and RETURN nodes are created during interprocedural, as shown in the first section of the table. As reachable definition and use tuples that are originated from other procedures are propagated, interprocedural definition-use chains can be identified. The last column shows the total numbers of (intraprocedural and interprocedural) definition-use pairs that are identified in the benchmark examples.

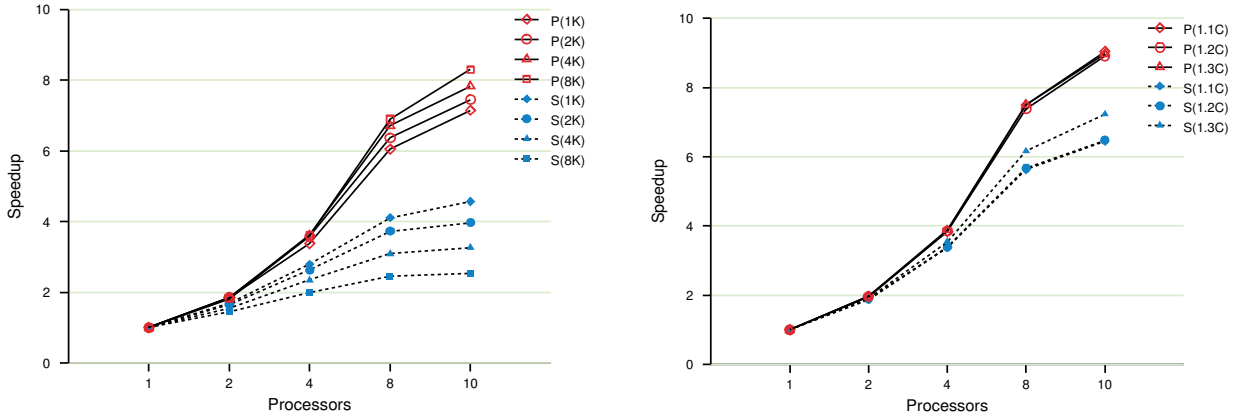
Figure 30 depicts the speedup ratios of EM3D and Moldyn as the numbers of processors grow from 1 to 10 [28]. In the S version, the traversal references of EM3D and Moldyn are parallelized and the construction operations are left to be sequential, whereas in the P version both the traversal references and construction operations of EM3D Moldyn are both parallelized. Both EM3D and Moldyn with graph construction parallelized (i.e. P version) scale very well.

The reason of less scalability for the S versions of both EM3D and Moldyn is that the sequential execution of graph construction operations takes more and more percentages of execution times as the number of processors grows, as shown in Fig. 31. Figure 31(a) shows that the sequential graph construction for 1K E nodes and 1K H nodes takes 9% of total execution time on 1 processor and increases to 43% on 10 processors, while building the graph for 8K E nodes and 8K H nodes on 1 processor occupies 30% of execution time and grows to over 77% on 10 processors. Similarly, the percentages of execution times spending on graph construction for Moldyn increase from less than 5% on 1 processor to over 31% on 10 processors, as shown in Fig. 31(b). On the other hand, Fig. 31(a) and (b) show that after the construction operations are parallelized (P version) the percentages of graph construction times over execution times keep flat even as the numbers of processors increase.

7. Related work

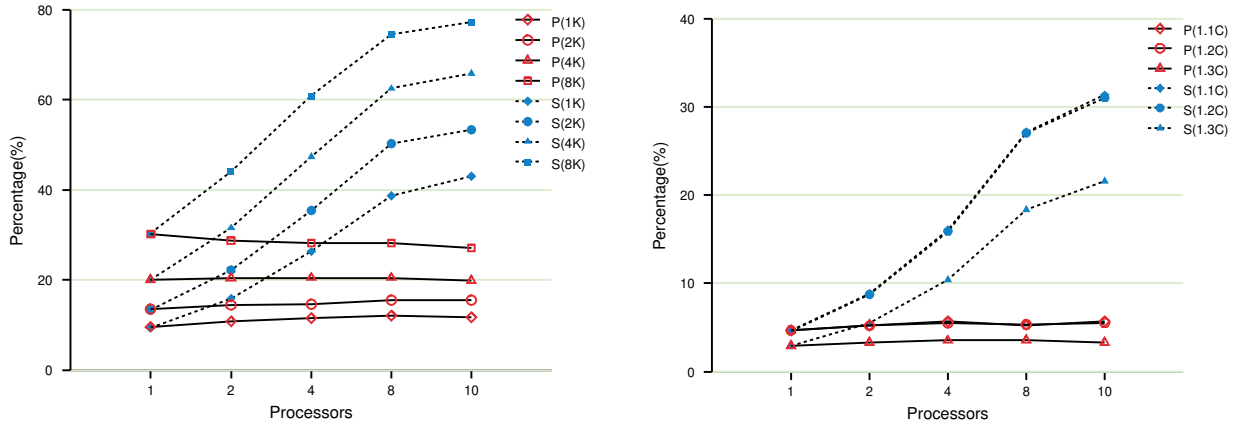
Definition-use chains of variables are commonly used in optimizing and parallelizing compilers [50] and even software engineering tools [19]. Analysis techniques for computing definition-use chains for individual procedures are well known [2] and interprocedural algorithms have also been proposed [23]. Definition-use chains can be extended to cover heap locations and pointer variables. Algorithms have been proposed to compute definition-use chains for heap locations and single level pointers in C [4,35].

Pande et al. propose an algorithm to solve the interprocedural definition-use chaining problem for single level pointers in C [35]. The technique first computes the set of interprocedural reaching definitions and then establishes definition-use chains using the reaching definitions. Altucher and Landi have developed a new naming scheme to improve definition-use information of dynamically allocated locations [4]. The new naming scheme is used to compute *extended must aliases*



(a) EM3D (b) Moldyn

Fig. 30. Speedup.



(a) EM3D (b) Moldyn

Fig. 31. Percentage of build time over execution time.

of dynamically allocated locations and the information will be applied to improve definition-use information.

The algorithm that is proposed in this paper to compute definition-use chains of pointer-linked data structures is based on the computation of upward exposed uses of simple binary access paths. The flow-sensitive interprocedural algorithm is adapted from the interprocedural algorithm of computation of definition-use chains of variables proposed by Harrold and Soffa [23]. The intraprocedural and interprocedural algorithms presented in this paper are basically similar to techniques to compute definition-use chains of variables within individual procedures and across procedure boundaries [2,23]. The algorithm developed by Chase et al. to compute definitions and uses of SSGs (storage shape graphs), each of which in turn summarizes all pointer paths into and through allocated storage at a statement, presents an interesting parallel [12]. However, the method is not designed to handle a program that reverses a list in place.

Recently, the problem of identifying pointer-induced aliases has received significant attention from researchers [10,13,18,21,30,39,41,42,48]. Burke et al. have developed a flow-insensitive algorithm [10]. Since flow-insensitive algorithms do not take intraprocedural control flow into account, they are less accurate than flow-sensitive approach, but more efficient. Steensgaard has proposed an interprocedural flow-insensitive points-to analysis based on type inference methods with an almost linear time complexity [42]. Although Steensgaard's algorithm has a better time complexity, it is less precise than the flow-insensitive algorithm proposed by Andersen [6]. Shapiro and Horwitz have developed a flow-insensitive points-to anal-

ysis algorithm that improves the precision of Steensgaard's algorithm and is much faster than Andersen's algorithm [41].

Landi and Ryder propose an interprocedural flow-sensitive analysis approach that computes the conditional may alias problem and uses the result to approximate the interprocedural may aliases [30]. They represent aliases with unordered pairs of object names, each of which is k-limited. Choi et al. present a flow-sensitive interprocedural alias analysis algorithm that interleaves intraprocedural and interprocedural analysis to compute the alias pairs [13]. In order to avoid the unrealized path problem [30], Landi and Ryder use the set of *reaching aliases* (*RAs*) that exist at the entry of procedure p when p is invoked as the encoding of the runtime stack, and the RAs can be used to determine to which call sites aliases at the exit of a called procedure should be propagated, whereas Choi et al. compute at the entry node of P the set of *alias instances* which hold with respect to each call site that invokes P. Furthermore, the above approaches impose the k-limited rule on name objects. Deutsch proposes a storeless approach that can avoid such limitation [18].

Recently, several context-sensitive pointer analysis algorithms, which treat multiple calls to the same procedure independently rather than constructing a single approximation, have been proposed [21,48]. Emami et al. build an *invocation graph* that explicitly represent all invocation paths to compute interprocedural points-to relationships between accessible stack locations [21]. They compute a separate result for each invocation graph node, but each recursive call is represented by a pair of recursive and approximation nodes. On the contrary, Wilson and Lam propose to compute *partial transfer functions* (*PTFs*) to represent the effects of called procedures [48]. Although context-sensitive analysis is generally more precise than context-insensitive analysis, Ruf presents a context-insensitive algorithm and performs analysis on pointer-intensive benchmark programs to demonstrate that context-insensitivity exerts little to no precision penalty [39].

The main difference of the alias analysis approach in this paper is that it follows the backward data flow analysis approach. It gathers the uses of pointers to where they are defined to identify aliases. Another feature is that this approach does not compute all pairs of aliases, since it only identify aliases that are needed to compute definition-use chains of pointer-linked data structures. That is, if a use p_i meets its definition at statement S and hence is killed by S, it is redundant to compute the aliases of p_i before S.

References

[1] H. Agrawal, R.A. DeMillo and E.H. Spafford, Debugging with dynamic slicing and backtracking, *Software – Practice and Experience* **23**(6) (June 1993), 589–616.

[2] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, 1986.

[3] B. Alpern, M.N. Wegman and F.K. Zadeck, Detecting equality of variables in programs, in: *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages,* San Diego, California, January 1988, pp. 1–11.

[4] R. Altucher and W. Landi, An extended form of must alias analysis for dynamic allocation, in: *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* San Fran-cisco, California, January 1995, pp. 74–84.

[5] G. Amdahl, Validity of the single-processor approach to achieving large-scale computing capabilities, in: *Proceedings of 1967 AFIPS Conference,* (Vol. 30), 1967.

[6] L.O. Andersen, Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen, 1994.

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

[8] D.W. Binkley and K.B. Gallagher, Program slcing, in: *Advances in Computers,* (Vol. 43), M. Zelkowitz, ed., Academic Press, San Diego, California, 1996.

[9] B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D.J. States, S. Swaminathan and M. Karplus, CHARMM: A program for macromolecular energy, minimization, and dynamics calculations, *Journal of Computational Chemistry* **4**(2) (1983), 187–217.

[10] M. Burke, P. Carini, J.-D. Choi and M. Hind, Flow-insensitive interprocedural alias analysis in the presence of pointers, in: *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing,* Columbus, Ohio, August 1995.

[11] D. Callahan, The program summary graph and flow-sensitive interprocedural data flow analysis, *SIGPLAN Notices* **23**(7) (July 1988), 47–56, *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

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

[13] J.-D. Choi, M. Burke and P. 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,* Charleston, South Carolina, January 1993, pp. 232–245.

[14] K.D. Cooper, M.W. Hall, R.T. Hood, K. Kennedy, K.S. McKinley, J.M. Mellor-Crummey, L. Torczon and S.K. Warren, The ParaScope parallel programming environment, *Proceedings of the IEEE* **81**(2) (February 1993), 244–263, in Special Section on Languages and Compilers for Parallel Machines.

[15] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman and F.K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, *ACM Transactions on Programming Languages and Systems* **13**(4) (October 1991), 451–490.

[16] R. Das, J. Saltz and R. von Hanxleden, Slicing analysis and indirect access to distributed arrays, in: *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing,* Springer-Verlag, August 1993, pp. 152–168. Also available as University of Maryland Technical Report CS-TR-3076 and UMIACS-TR-93-42.

[17] A. Deutsch, A storeless model of aliasing and its abstractions using finite representations of right-regular equivalent relations, in: *Proceedings of the IEEE 1992 International Conference on Computer Languages,* San Francisco, California, April 1992, pp, 2–13.

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

[19] E. Duesterwald, R. Gupta and M.L. Soffa, A demand-driven analyzer for data flow testing at the integration level, in: *IEEE International Conference on Software Engineering,* Berlin, Germany, March 1996, pp. 575–586.

[20] S. Eilenberg, *Automata, Languages, and Machines,* Academic Press, 1974.

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

[22] M.W. Hall and K. Kennedy, Efficient call graph analysis, *ACM Letters on Programming Languages and Systems,* **1**(3) (September 1992), 227–242.

[23] M.J. Harrold and M.L. Soffa, Efficient computation of interprocedural definition-use chains, *ACM Transactions on Programming Languages and Systems* **16**(2) (March 1994), 175–204.

[24] M.S. Hecht, *Flow Analysis of Computer Programs,* Elsevier North-Holland, 1977.

[25] L.J. Hendren, *Parallelizing Programs with Recursive Data Structures,* PhD thesis, Cornell University, 1990.

[26] S. Horwitz, J. Prins and T. Reps, Integrating noninterfering versions of programs, *ACM Transactions on Pro-gramming Languages and Systems* **11**(3) (July 1989), 345–387.

[27] Y.-S. Hwang, *Interprocedural Definition-Use Chains of Dynamic Recursive Data Structure,* PhD thesis, University of Maryland, 1998.

[28] Y.-S. Hwang, Parallelizing graph construction operations in programs with cyclic graphs, in: *Proceedings of the 2001 International Conference on Parallel and Distributed Computing and Systems* (*PDCS 2001*), Anaheim, USA, August 2001.

[29] Y.-S. Hwang and J. Saltz, Identifying parallelism in programs with cyclic graphs, in: *Proceedings of the 2000 International Conference on Parallel Processing,* Toronto, Canada, August 2000, pp. 201–208.

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

[31] S. Lumetta, L. Murphy, X. Li, D. Culler and I. Khalil, Decentralized optimal power pricing: The development of a parallel program, in: *Proceedings of Supercomputing '93,* Portland, Oregon, November 1993, pp. 240–249.

[32] 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.

[33] S.S. Muchnick, *Advanced Compiler Design & Implementation,* Morgen Kaufmann, 1997.

[34] S.S. Mukherjee, S.D. Sharma, M.D. Hill, J.R. Larus, A. Rogers and J. Saltz, Efficient support for irregular applications on distributed-memory machines, in: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* (*PPOPP*), ACM Press, *ACM SIGPLAN Notices* **30**(8) (July 1995), 68–79.

[35] H.D. Pande, W.A. Landi and B.G. Ryder, Interprocedural defuse associations for c systems with single level pointers, *IEEE Transactions on Software Engineering* **20**(5) (May 1994), 385–402.

[36] T. Reps, S. Horwitz and M. Sagiv, Precise interprocedural dataflow analysis via graph reachability, in: *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* San Francisco, California, January 1995, pp. 49–61.

[37] A. Rogers, M.C. Carlile, J.H. Reppy and L.J. Hendren, Supporting dynamic data structures on distributed-memory machines, *ACM Transactions on Programming Languages and Systems* **17**(2) (March 1995), 233–263.

[38] B.K. Rosen, M.N. Wegman and F.K. Zadeck, Global value numbers and redundant computations, in: *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages,* San Diego, California, January 1988, pp. 12–27.

[39] E. Ruf, Context-insensitive alias analysis reconsidered, *SIGPLAN Notices* **30**(6) (June 1995), 13–22, *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[40] M. Sagiv, T. Reps and R. 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,* St. Petersburg Beach, Florida, January 1996, pp. 16–31.

[41] M. Shapiro and S. Horwitz, Fast and accurate flow-insensitive points-to analysis, in: *Conference Record of POPL '97: 24nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* Paris, France, January 1997, pp. 1–14.

[42] B. Steensgaard, Points-to analysis in almost linear time, in: *Conference Record of POPL '96: 23nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* St. Petersburg Beach, Florida, January 1996, pp. 32–41.

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

[44] M.N. Wegman and F.K. Zadeck, Constant propagation with conditional branches, *ACM Transactions on Program-ming Languages and Systems* **13**(2) (April 1991), 181–210.

[45] M. Weiser, Programmers use slices when debugging, *Communications of the ACM* **25**(7) (July 1982), 446–452.

[46] M. Weiser, Reconstructing sequential behavior from parallel behavior projections, *Information Processing Letter* **17**(5) (October 1983), 129–135.

[47] M. Weiser, Program slicing, *IEEE Transactions on Software Engineering* **10** (1984), 352–357.

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

[49] M. Wolfe, Beyond induction variables, *SIGPLAN Notices* **27**(7) (July 1992), 162–174, *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[50] M. Wolfe, *High Performance Compilers for Parallel Computing,* Addison-Wesley, 1995.