Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects

Jong-Deok Choi* Michael Burke* Paul Carini*

IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

We present practical *approximation* methods for computing interprocedural aliases and side effects for a program written in a language that includes pointers, reference parameters and recursion. We present the following results: 1) An algorithm for *flow-sensitive* interprocedural alias analysis which is more precise and efficient than the best interprocedural method known. 2) An extension of traditional *flow-insensitive* alias analysis which accommodates pointers and provides a framework for a family of algorithms which trade off precision for efficiency. 3) An algorithm which correctly computes side effects in the presence of pointers. Pointers cannot be correctly handled by conventional methods for side effect analysis. 4) An alias naming technique which handles dynamically allocated objects and guarantees the correctness of data-flow analysis. 5) A compact representation based on transitive reduction which does not result in a loss of precision and improves precision in some cases. 6) A method for intraprocedural alias analysis which is based on a sparse representation.

1 Introduction

Aliasing occurs when two or more l-value [2] expressions reference the same storage location at the same program point p. Aliasing complicates data-flow analysis, which is a basis for program transformations such as optimization and parallelization. The problem of exactly computing interprocedural aliases in the presence of general pointers is known to be NP-hard [16]. In this paper, we describe *approximation* methods for computing interprocedural aliases and side effects for a program written in a language that includes pointers, reference parameters and recursion. We present the following new results:

- An algorithm for *flow-sensitive* interprocedural alias analysis which is more precise and efficient than the best interprocedural method known [16, 17]. The *flow-sensitive* interprocedural alias analysis method interleaves intraprocedural and interprocedural analyses.
- An extension of traditional *flow-insensitive* alias analysis [4, 10] to accommodate pointers. Our *flow-insensitive* alias analysis method provides a framework for a family of algorithms with varying degrees of precision.
- An algorithm which correctly computes side effects in the presence of pointers. Conventional methods for side-effect analysis [4, 9, 5] decompose the computation into separate direct side-effect and alias analyses. Pointers cannot be correctly handled by algorithms based on this decomposition. Apart from the cost of the alias analysis, this side-effect analysis does not incur any cost beyond the conventional algorithms.
- A naming technique for dynamically allocated objects. This technique improves the precision of existing methods for alias analysis [7, 12, 13, 18].
- An alias naming technique which, when aliases are factored into data-flow analysis, guarantees the correctness and enhances the precision of data-flow analysis. This technique also allows a compact representation which, based on transitive reduction, does not result in a loss of precision. For some cases, this technique results in improved precision.
- A method for intraprocedural alias analysis which is based on a sparse representation.

The algorithms we develop in this paper compute aliases of *access paths* [18] – l-value expressions which are constructed from variables, pointer indirection operators, and structure field select operators. (In C these expressions would include the '*' indirection operator and the '->' and '.' field select operators.) Two access paths are *must-aliases* at p if they refer to the same storage location in all execution instances of p. Two access paths are *may-aliases* at p if they refer to the same storage location in some execution instances of p. This paper concerns computing may-aliases, of which

*E-mail addresses: {jdchoi,burkem,carini}@watson.ibm.com

ACM-20th PoPL-1/93-S.C., USA

```
int **p, **q, *r, *s, X, Y;

SUB1() {
  p = &s;
  r = &X;
  SUB3();
  *r = 10;
  Y = X;
}; /* SUB1 */

SUB2() {
  p = q;
  SUB3();
  ...
}; /* SUB2 */

SUB3() {
  *p = r;
}; /* SUB3 */
```

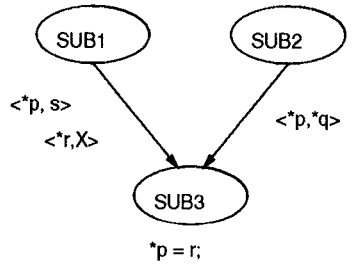

Figure 1: Example Program Segment and Its PCG.

must-aliases are a subset. We will refer to may-aliases as aliases, whenever the meaning is clear from context.

Dynamic intraprocedural aliases occur due to pointers in languages like LISP, C, C++ or Fortran 90, and *static* intraprocedural aliases occur due to constructs such as the union construct in C and Fortran `EQUIVALENCE`. *Interprocedural aliases* occur due to the passing of *reference* parameters and pointers at call sites, which propagates intraprocedural aliases across procedures and introduces new aliases. Thus, aliases are propagated through the *procedure call graph* (PCG), whose nodes represent procedures and edges represent call sites. Static alias information is typically computed during the semantic analysis phase of compilation, and we assume that this information is readily available. Where the meaning is clear, we will refer to dynamic intraprocedural aliases as simply intraprocedural aliases. A *flow-sensitive* interprocedural analysis makes use of the intraprocedural control flow information associated with individual procedures. A *flow-insensitive* interprocedural analysis does not make use of intraprocedural control flow information.

Through pointer assignment, a called procedure can affect the aliasing of the calling procedure at the point just after the call site. Computing aliases requires *bidirectional* iteration over the PCG, with aliases propagated forward from the call site to the entry point of the called routine, and backward from the return point of a called procedure to the point just after the call site.

Section 2 contains examples which illustrate some of our improvements in precision over other interprocedural methods. Section 3 presents an algorithm for computing intraprocedural alias information which is based on a sparse representation. Section 4 presents our interprocedural alias analysis methods. Section 5 presents the method for computing call site side effects in the presence of pointers. Section 6 describes a technique for naming memory locations which guarantees the correctness of data flow analysis. It also presents a representation of alias information which enhances the precision and efficiency of alias analysis methods. The complexity of the alias analysis algorithms is presented in Section 7. Comparison of our method with relevant work is given in Section 8. Finally, we draw conclusions in Section 9.

2 Examples

In this section, we present two examples. The first example demonstrates the improved precision of our flow-sensitive interprocedural method over the Landi-Ryder approach [16, 17]. The second example demonstrates the improved precision of our method for naming dynamically allocated objects over previous approaches, including those described in [7, 18].

We use $< x, y >$ to denote that access paths x and y are may-aliases at a program point. Note that the *trivial* alias $< x, x >$ always holds for all access paths x at all program points. Also note that, if x and y are *non-null* pointer-valued access paths, $< x, y >$ *implies* $< *x, *y >$. We refer to these *implied* aliases and *trivial* aliases as *implicit* aliases.

Figure 1 shows three procedures and the corresponding PCG. Alias relations holding at the entry node of $SUB3$ consist of $< *p, s >, < *r, X >$, and $< *p, *q >$.[1] Of these three alias relations, $< *p, s >$ and $< *r, X >$ are propagated along the PCG edge from $SUB1$ to $SUB3$, and $< *p, *q >$ is propagated along the PCG edge from $SUB2$ to $SUB3$. The alias relations holding at the exit of $SUB3$ are different from those at the entry node, due to the assignment to $*p$ in $SUB3$, and are propagated to the points in $SUB1$ and $SUB2$ immediately following the invocations of $SUB3$. Our interprocedural flow-sensitive analysis will correctly compute the following:

$$< *p, s >, < *r, X >, < **p, *r >, < **p, X >, < *s, *r >, < *s, X >$$

as the alias relations holding immediately after the call to $SUB3$ in $SUB1$, and the following:

$$< *p, *q >, < **p, *r >, < **q, *r >$$

as the alias relations holding immediately after the call to $SUB3$ in $SUB2$.

In contrast, the Landi-Ryder method [16, 17] will imprecisely compute, in addition to the above valid alias relations, that the (invalid) alias relation $< **q, X >$ holds immediately after the call site invoking $SUB3$ from either $SUB1$ or $SUB2$ (but not both). This invalid alias relation results from the spurious coupling of

[1] We assume that no alias relations hold before any of the procedures are called.

```
int *p, *q, *r, *s, *t;

main() {
  p = SUB1();
  q = SUB2();
}; /* main */

int * SUB1() {
  r = SUB3();
  SUB4 ();
  return(r);
}; /* SUB1 */

int * SUB2() {
  s = SUB3();
  SUB4 ();
  return(s);
}; /* SUB2 */

int * SUB3() {
  t = malloc(...);
  SUB4 ();
  return(t);
}; /* SUB3 */
```

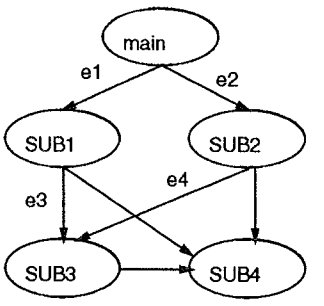

Figure 2: Example Program Segment and Its PCG.

$< *r, X >$ and $< *p, *q >$ in $SUB3$, although these alias relations cannot hold at the same time. Their method uses *assumed aliases* at the entry node and, in general, can compute $O(2^{n-1})$ invalid alias relations in the presence of n alias relations holding at the entry node, although the total number of aliases they compute is no more than $O(m^3)$ in the presence of $\Omega(m)$ valid aliases [17]. The generation of invalid alias relations by their method is discussed in Section 8.

Figure 2 shows an example program, where all five pointer variables – p, q, r, s, t – point to objects dynamically allocated by the `malloc` statement in $SUB3$. For the method described in [7], all the pointer variables are assumed to point to the same location. Our method determines that $*p$ is not aliased to $*q$ or $*s$; and that $*q$ is not aliased to $*p$ or $*r$. Our method thereby improves the precision of alias information for dynamically allocated objects. A more detailed description is provided in Section 6.

3 Intraprocedural Alias Computation

The intraprocedural alias computation can be formulated as a data flow framework [20, 22], in which the solution at a given program point is related to the solution at other points. A *control flow graph* is a directed graph $CFG =< N_{CFG}, E_{CFG}, \texttt{Entry}, \texttt{Exit} >$. The nodes N_{CFG} are the statements of a procedure and two additional nodes, `Entry` and `Exit`. The edges E_{CFG} represent transfers of control between the statements. We assume that each node is on a path from `Entry` to `Exit`.

The data flow framework for computing aliases includes *transfer functions*, which describe the effect of nodes on aliasing. IN_Y denotes the set of aliases assumed to hold on entrance to Y; OUT_Y denotes the set of aliases assumed to hold on exit from Y. The effect of node Y on aliasing is captured by its transfer function:

$$OUT_Y = f_Y(IN_Y)$$

After a framework has been globally *evaluated*, each node Y has a solution OUT_Y that is consistent with IN_Y and the transfer function at every node.

We use a sparse representation, the *Sparse Evaluation Graph* (SEG) [8], for intraprocedural alias analysis. The set of nodes in a Sparse Evaluation Graph $SEG =< N_{SEG}, E_{SEG}, \texttt{Entry}, \texttt{Exit} >$ for computing aliases is a subset of N_{CFG} and is the union of the following two sets: $GenNodes$, which is the set of nodes where pointer variables are potentially modified; and $MeetNodes$, which is the set of nodes where alias information is combined at join points in the SEG.

Intraprocedural aliasing can be computed by applying well-known iterative or interval-based techniques to the SEG. This computation requires three components: the aliases holding at the `Entry` node of the SEG ($ALIAS_{Entry}$), the aliases holding at (i.e. immediately after) the call-site nodes in the SEG, and the transfer function for each node that contains a (potential) pointer assignment. The aliases holding at the `Entry` node and the call-site nodes are interprocedurally computed, as described in Section 4. If interprocedural alias analysis is not performed, interprocedural aliases holding at the `Entry` node and the call-site nodes would represent the pessimistic assumption that all possible (interprocedurally induced) aliases hold. In this section, we describe the transfer function for a general assignment to an access path of a pointer type.

3.1 Transfer Function for Pointer Assignment

Let A^Y_{IN} and A^Y_{OUT} be the set of alias relations holding at the entry and exit of node Y, respectively. A^Y_{IN} and A^Y_{OUT} correspond to IN_Y and OUT_Y as defined before. We use A_{IN} and A_{OUT} when the node Y in question is clear from context. As described before, a $MeetNode$ in the SEG has multiple predecessor nodes. For a $MeetNode$ Y, A^Y_{IN} is the union of the A_{OUT} sets of its predecessor nodes in the SEG:

$$A^Y_{IN} = \bigcup_{X \in Preds(Y)} A^X_{OUT}$$

For an access path x, let $A_{IN}(x)$ be the set of alias relations in A_{IN} that contain access paths which include x. As an example, for

$$A_{IN} = \{< x, y >, < *x, z >, < w, z >,$$

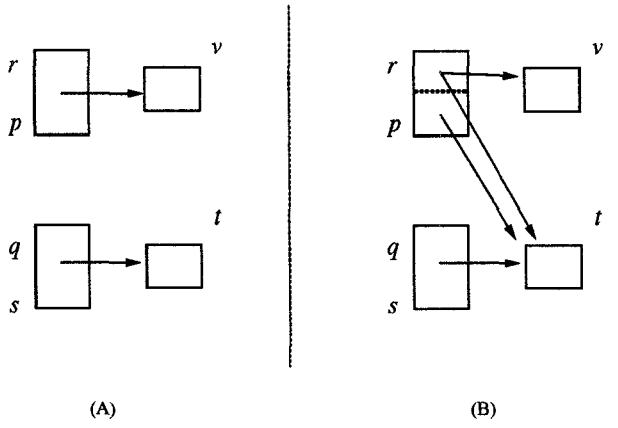

Figure 3: Graphical View of Aliasing.

$$< *x, *y >, < x, x >, < y, y >,$$
$$< w, w >, < z, z >\}$$

$A_{IN}(x)$ is $\{< x, y >, < *x, z >, < *x, *y >, < x, x >\}$. The first three alias relations of A_{IN} are *explicit* aliases, while the remaining are *implicit* aliases. For efficiency, our representation includes only *explicit* aliases. For clarity, we sometimes include implicit aliases in our exposition. The above alias sets become

$$A_{IN} = \{< x, y >, < *x, z >, < w, z >\}$$

and

$$A_{IN}(x) = \{< x, y >, < *x, z >\}$$

respectively.

For an alias relation $AR \in A_{IN}(x)$, we use $AR(y/x)$ to denote the new alias relation obtained by replacing each occurrence of access path x in AR with access path y. We also use $A_{IN}(y/x)$ to denote the new set of alias relations obtained by replacing each occurrence of access path x in A_{IN} with access path y. For A_{IN} in the above example,

$$A_{IN}(q/x) = \{< q, y >, < *q, z >\}.$$

For the following assignment statement to an l-value with a pointer type

$$S_i : p \leftarrow q\ , \tag{1}$$

the transfer function is as follows:

$$A_{OUT} = (A_{IN} - A_{IN}(*p)) \cup \tag{2}$$
$$\bigcup_{<p,u> \in A_{IN},\ AR \in A_{IN}(*q)} \{AR(*u/*q)\}. \tag{3}$$

The *killing* rule (Rule 2) says that the value of p may change after the assignment. The *generation* rule (Rule 3) says that any alias of p, including *implicit* ones, may point to what q may point to: if u is an alias of p, $*u$ becomes aliased to what $*q$ is aliased to. For example, assume A_{IN} at statement S_i in (1) is:

$$\{\ < p, r >, < *p, v >, < *r, v >,$$
$$< q, s >, < *q, t >, < *s, t >\ \} \tag{4}$$

```
S1:   struct cell {...}
                cell1, cell2, cell3, *p, *q;
S2:   p = &cell1;        q = &cell2;
S3:   if () {
S4:             p->left = &cell3
S5:             p->right = q;
...
S6:   } else if () {
S7:             p->left = q;
S8:             p->right = q;
...
S9:   } else
S10:            p = q;
S11:  p->val = ...;
S12:  p->left->val = ...;
```

Figure 4: Example C Program Segment.

which is shown in Figure 3-(A) (along with the following implicit aliases: $< *p, *r >, < *q, *s >, < p, p >, < *p, *p >, < r, r >, \ldots$). Now, A_{OUT} at that statement, ignoring its implicit aliases, becomes:

$$\{\ < p, r >, < *r, v >, < q, s >,$$
$$< *q, t >, < *s, t >, \tag{5}$$
$$< *p, t >, < *r, t >, < *p, *q >,$$
$$< *r, *q >, < *p, *s >, < *r, *s >\ \} \tag{6}$$

which is shown in Figure 3-(B). Elements of (5) came from applying the killing rule (2) to A_{IN}. Elements of (6) came from applying the generation rule (3) to the following subset of A_{IN}:

$$\{< p, p >, < p, r >, < *q, t >, < *q, *q >, < *q, *s >\},$$

among which $< p, p >$, $< *q, *q >$ and $< *q, *s >$ are implicit alias relations. Recall that, in the above example, $< p, r >$ in A_{IN} at S_i is a *may-alias*: p and r *may* refer to the same memory location. After the assignment to p in 1, r might still point to v.

If $< p, r >$ is a *must-alias*, we can apply the killing rule to r as well as p. This will delete $< *r, v >$ from A_{OUT}, resulting in better precision. This corresponds to the case when there is no edge from r to v in Figure 3-(B).

As another example, consider the program segment in Figure 4. Assuming no aliasing holds before S_2 in the figure, the alias relations after S_2 are as follows:

$$\{< *p, cell1 >, < *q, cell2 >\}.$$

There are 8 alias relations after S_5:

$$\{\ < *p, cell1 >, < *q, cell2 >,$$
$$< *(cell1.left), cell3 >, < *(cell1.right), *q >,$$
$$< *(cell1.right), cell2 >,$$
$$< *((*p).left), cell3 >, < *((*p).right), *q >,$$
$$< *((*p).right), cell2 >\}, \tag{7}$$

```
S1:     if () {
S2:             p = &y;
S3:             q = null;
S4:     } else {
S5:             p = &x;
S6:             q = &z;
S7:     };
S8:     *p = q;
```

Figure 5: Example C Program Segment.

and 14 alias relations after S_{10}:

$$\{ \; < *p, cell1 >, < *q, cell2 >, \\ < *(cell1.left), cell3 >, < *(cell1.right), cell2 >, \\ < *(cell1.right), *q >, \\ < *((*p).left), cell3 >, < *((*p).right), cell2 >, \\ < *((*p).right), *q >, \\ < *(cell1.left), cell2 >, < *(cell1.left), *q >, \\ < *((*p).left), cell2 >, < *((*p).left), *q >, \\ < *p, *q >, < *p, cell2 >\}. \qquad (8)$$

Later in Section 6, we show how to represent alias relations more compactly.

In Figure 5, the alias information at S_7 generated along the CFG path $S_1 \rightarrow S_2 \stackrel{*}{\rightarrow} S_7$ is

$$A1 = \{< *p, y >\}$$

and the alias information generated along the CFG path $S_1 \rightarrow S_5 \stackrel{*}{\rightarrow} S_7$ is

$$A2 = \{< *p, x >, < *q, z >\}.$$

Among the aliases generated by applying the transfer function of S_8 to the alias information $A1 \cup A2$ resulting from the merge at S_7, we have $< *y, z >$, which results from the coupling of $< *p, y > \in A1$ and $< *q, z > \in A2$. However, $< *p, y >$ and $< *q, z >$ never hold for the same execution instance, and thus $< *y, z >$ never holds immediately after S_8.

Unioning alias information at join points of control flow such as S_7 can generally result in a loss in precision, unless path-specific information is kept for alias information [19]. A similar case occurs during interprocedural analysis in Section 4. A similar technique to the one developed in Section 4.4 can be used for limited distinction between aliases generated along different CFG paths. With this technique, we can identify that $< *p, y > \in A1$ and $< *q, z > \in A2$ do not hold for the same execution instance and, thereby, do not create $< *y, z >$ at S_8, resulting in improved precision.

3.2 Cyclic Data Structures

Consider the C program segment given in Figure 6 and the cyclic data structure given in Figure 7, which shows the data structures, in terms of r and $cell1$, of the user program resulting from the execution of statements $S1$ and $S2$, respectively, in Figure 6. For simplicity, we ignore the (anonymous) object pointed to by $cell1$. Also, we assume no non-trivial aliasing holds before $S1$. The alias relationship holding after $S1$ is, thus, $< *r, cell1 >$. At $S2$, $*cell1$ and $\&cell1$ play the role of p and q, respectively, in Rule 2. Thus, the set $\{< p, u > \in A_{IN}\}$ of Rule 3 becomes $\{< *cell1, *cell1 >, < *cell1, **r >\}$, and the set $A_{IN}(*q)$ of Rule 3 becomes $\{< cell1, *r >, < cell1, cell1 >\}$. Rules 2 and 3 yield the following

```
S1:     r = &cell1;
S2:     *cell1 = &cell1;
```

Figure 6: Example C Program Segment.

$$\{ \; < *r, cell1 >, < *r, ***r >, < *r, **cell1 >, \\ < ***r, cell1 >, < cell1, **cell1 >\}. \qquad (9)$$

Note that the above alias set does not represent the complete (infinite) set of alias relations for the cyclic data structure $S2$ in Figure 7. However, combined with the approach described in Section 6.3, the alias relations in the above set are sufficient for deducing the complete set of aliases implied by the cyclic structure.

4 Interprocedural Alias Computation

In this section, we describe two methods for computing interprocedural aliases. The interprocedural alias solution supplies $ALIAS_{Entry}$ and $ALIAS_{Exit}$ for each procedure. In the first method, we compute flow-insensitive interprocedural aliasing over the PCG. In the second method, we compute flow-sensitive interprocedural aliasing over the PCG, where each node of the PCG is associated with the SEG for that procedure. For a given interprocedural problem, flow-insensitive algorithms are efficient but imprecise in comparison to flow-sensitive algorithms. We first illustrate the difference between our flow-sensitive and flow-insensitive approaches with an example. We then describe our flow-insensitive and flow-sensitive analysis methods, and finally describe the technique for identifying realizable execution paths, which applies to both methods.

4.1 Interprocedural Alias Example

In Figure 8, a flow-insensitive analysis would determine that at some point in P, a pointer assignment ($S3$) induces the alias $< *u, a >$. This alias is presumed to hold everywhere in P, including its call site. It is passed forward along the PCG edge to Q, and is presumed to hold everywhere in Q. A flow-insensitive analysis of Q

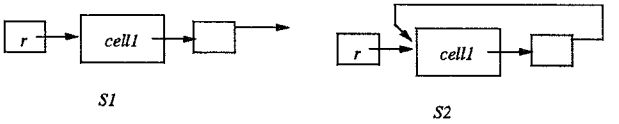

Figure 7: Cyclic Data Structure.

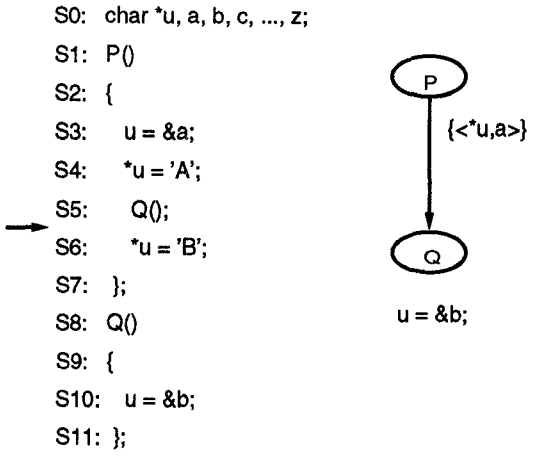

Figure 8: Example Program Segment and Its PCG.

would determine that, at some point in Q, a pointer assignment induces the alias $< *u, b >$. This alias is also presumed to hold everywhere in Q, including its exit. The two aliases are passed backward along the PCG edge to P, and presumed to hold everywhere in P. No further changes are introduced, and the iteration over the call graph stops. The flow-insensitive alias analysis has determined that the alias set $\{< *u, a >, < *u, b >\}$ holds everywhere in P and Q.

A flow-sensitive analysis of P would associate alias relation $< *u, a >$ with statements S_3, S_4 and S_5. Since Q has not been analyzed yet, no assumptions are made about the call to Q. $ALIAS_{Exit}(P)$ is also associated with $< *u, a >$. The alias associated with the call site is then propagated forward along the PCG edge to $ALIAS_{Entry}(Q)$. A flow-sensitive analysis of Q would associate $< *u, b >$ with $ALIAS_{Exit}(Q)$, because the pointer assignment at S_{10} has both killed the alias $< *u, a >$ and generated $< *u, b >$. The alias $< *u, b >$ is then passed backward along the call graph to the point just following S_5 (the call to Q). Since information has changed in P, another analysis of P takes place, and propagates $< *u, b >$ to S_6 and $ALIAS_{Exit}(P)$. No further changes take place. At the conclusion of the flow-sensitive analysis, $< *u, a >$ holds at S_3, S_4, and S_8, and $< *u, b >$ holds at S_5, S_6, and S_{10}.

4.2 Flow-Insensitive Interprocedural Alias Computation

Flow-insensitive interprocedural algorithms are based on the PCG, associating summary information with its nodes and propagating this information through the PCG. As a class they are highly efficient because they do not require a representation or analysis of individual routines during the interprocedural component of the analysis.

The basic model of computing flow-insensitive interprocedural aliasing is similar to that of the PTRAN system [3], which computes this information for FORTRAN 77 programs in a single topological-order traversal over the PCG. The major difference is that iteration over the PCG is required due to cycles resulting from recursion and also due to pointer-induced aliasing.[2] Iteration due to cycles in the PCG is accommodated by topological-order traversals. However, as described in Section 1, iteration due to pointer-induced aliasing is bidirectional – such aliasing is propagated from a callee to its callers as well as from a caller to its callees. To accommodate pointer-induced aliasing, iterations over the PCG must alternate between topological and reverse-topological order.

In flow-insensitive analysis, each node of the PCG is associated with a transfer function which is a conservative summary of all the intraprocedural transfer functions of the procedure. We assume that static alias information is available; the static alias information for a procedure is used to initialize its $ALIAS_{Entry}$ set. Iterating through the PCG in alternate directions continues until the $ALIAS_{Entry}$ and $ALIAS_{Exit}$ sets converge for all nodes.

If the transfer function for a procedure is computed in a flow-insensitive manner, then it does not kill aliases, but simply generates new ones. However, where a flow-sensitive analysis of each procedure precedes the flow-insensitive interprocedural analysis, transfer functions which generate *kill* information can be computed and used to yield a more precise solution.

Summary kill information is associated with the call sites of a node. $ForwardKill(C_i)$ represents aliases killed between the entry node of a procedure and call site C_i in the procedure. $BackwardKill(C_i)$ represents aliases killed along paths from call site C_i to the exit node of the procedure. Summary generation information is associated with a PCG node. For node n, $AliasGen(n)$ denotes the set of aliases generated by n. The following equations formulate the flow-insensitive alias computation as a data flow problem over the PCG which can utilize kill information if it is available:

$$ALIAS_{Entry}(node_j) = \bigcup_{e_{ij}} ((ALIAS_{Entry}(node_i) - ForwardKill(e_{ij})) \cup AliasGen(node_i)) \quad (10)$$

$$ALIAS_{Exit}(node_j) = \bigcup_{e_{jk}} ((ALIAS_{Exit}(node_k) - BackwardKill(e_{jk}))) \cup AliasGen(node_j), \quad (11)$$

where node i denotes the ith node in topological order and e_{ij} denotes an edge (call site) from node i to node j. This problem is solved in a straightforward manner by iteration over the call graph (alternating topological and reverse-topological iterations to accommodate the bidirectional character of the problem). For efficiency, the iterative algorithm could be implemented with a worklist representing the nodes yet to be processed.

Using dominator trees and postdominator trees [8], we compute safe kill information, in time linear with respect to the number of nodes in the control flow graph,

[2] In the presence of cycles, topological order is defined by the removal of *back edges* from the PCG [15].

```
S1:  foreach procedure,
                set ALIAS_Entry = STATIC_ALIASES;
S2:             set ALIAS_Exit = {};
S3:  repeat;
S4:     foreach procedure P visited in a topological traversal over the PCG;
S5:        compute ALIAS_Entry of P using A^C_IN at each call site C which invokes P;
S6:        compute flow-sensitive intraprocedural aliasing of P,
           including ALIAS_Exit of P, for each SEG node S_i;
              if S_i is a call site C
                 compute A^C_OUT using A^C_IN and
                 ALIAS_Exit of the called procedure;
              else
                 compute A_OUT using A_IN and its transfer function;
S8:  until aliasing converges;
```

Figure 9: Algorithm to Compute Flow-Sensitive Interprocedural Aliasing

as follows:

$$ForwardKill(C_i) = \bigcup_{node_j \in DOM(C_i)} AliasKill(node_j)$$

$$BackwardKill(C_i) = \bigcup_{node_j \in PDOM(C_i)} AliasKill(node_j)$$

where $AliasKill(node_j)$ is the set of aliases killed at CFG node $node_j$ in accordance with Rule 2 in Section 3, $DOM(C_i)$ is the set of CFG nodes that dominate C_i, and $PDOM(C_i)$ is the set of CFG nodes that post-dominate C_i.

4.3 Flow-Sensitive Interprocedural Alias Computation

Figure 9 describes our algorithm for computing flow-sensitive interprocedural aliasing. As above, the static aliasing for a procedure is used to initialize its $ALIAS_{Entry}$ set (see Statement S_1 in Figure 9). Interleaving the inter- and intraprocedural analyses results in more precise alias information than with the flow-insensitive method.

As described in Figure 9, traversals over the PCG in topological order are repeated.[3] At step S_5 of Figure 9, the alias information at the entry node of P, $ALIAS_{Entry}$, is computed by first unioning the intraprocedural aliases at the call sites which invoke procedure P and then propagating the resulting set into the entry node of P. At step S_6, $ALIAS_{Exit}$ of the called procedure is propagated back to the call site C for each call in P. $ALIAS_{Exit}$ sets computed during the previous iteration are used. Once again, for efficiency, the iterative algorithm could be implemented with a worklist representing the nodes yet to be processed.

Although a called procedure can affect the aliases holding on return from the call sites which invoke it, $ALIAS_{Exit}$ of a procedure does not uniformly affect all the procedures which call it. Rather, the aliases holding on return from a call site C_a are affected only by a subset of $ALIAS_{Exit}$ of the called procedure. This subset includes only those aliases *induced* by aliases holding at C_a or generated in the called procedure independently of the aliases holding at the invoking call sites. Identifying call sites whose aliases have induced an alias relation in $ALIAS_{Exit}$ is related to identifying *realizable execution paths* [16].

[3] Due to pointers, alias information is propagated bidirectionally. However, in this algorithm, topological-order iterations over the PCG are sufficient.

4.4 Realizable Execution Paths

In this section, we describe a method which correctly identifies, for each alias relation of $ALIAS_{Exit}$, the call sites whose aliases have induced the alias relation. We also show how we extend the transfer functions described in Section 3 to accomplish this. This method is applicable to both our flow-insensitive and flow-sensitive frameworks.

Assume there are n subroutines $Q_1 \ldots Q_n$ each of which invokes P. Also assume alias relation A_i ($1 <= i <= n$), which is propagated from Q_i to P as part of $ALIAS_{Entry}$ of P, reaches the exit node of P and becomes part of $ALIAS_{Exit}$ of P. When $ALIAS_{Exit}$ of P is propagated to the callers of P, $Q_1 \ldots Q_n$, all of $A_1 \ldots A_n$ might be propagated to all of $Q_1 \ldots Q_n$; not remembering the path(s) in the PCG along which particular data-flow information has been propagated can substantially reduce the precision of the data-flow solution.

To distinguish the call sites to which alias relations need be propagated, we define an *alias instance* (AI) as a triple:
[*alias relation AR, source alias set* SAS_C, *call site* C], where each member SA of SAS_C induces the alias relation AR via C. At the entry node of P, we compute the set of alias instances $ALIAS_{Entry}$ of P which hold with respect to each call site C that invokes P. For example, if $SAS_C = \{< u, v >\}$, and u and v are global to P, an alias instance $AI = [< u, v >, \{< u, v >\}, C]$ will be in $ALIAS_{Entry}(C)$. By convention, we will use *null* for a trivial source alias $< u, u >$. Once we compute $ALIAS_{Entry}(C)$ for each call site C which invokes P, we compute $ALIAS_{Entry}$ of P as follows:

$$ALIAS_{Entry} = \bigcup_{C \in CS(P)} ALIAS_{Entry}(C),$$

where $CS(P)$ is the set of call sites which invoke P. We will use $ALIAS_{Entry}$ as a shorthand for $ALIAS_{Entry}(C)$ when call site C in question is clear from the context. Note that alias instances generated independently of any other (non-trivial) alias have *null* (no) source aliases and *null* call sites.

We extend the notations defined in terms of alias relations in Section 3 to alias instances. For example, $A_{IN}(x)$, which is defined to be the set of *alias relations* in A_{IN} which contain access path x, becomes the set of *alias instances* in A_{IN} whose alias relations contain x in them.

Now, we extend the (aliasing) transfer function (Rule 2 and Rule 3) in Section 3 that is based upon alias relations to the following that is based upon alias instances. For the following assignment to an l-value with a pointer type:

$$S_i : p \leftarrow q \ , \tag{12}$$

the new transfer function is as follows:

$$A_{OUT} = (A_{IN} - A_{IN}(*p)) \cup \tag{13}$$
$$\{ \ [AR_2(*u/*q), SA_g, C_g] \mid$$
$$[< p, u >, SA_1, C_1] \in A_{IN},$$
$$[AR_2, SA_2, C_2] \in A_{IN}(*q),$$
$$SA_g = SA_1 \cup SA_2, \tag{14}$$
$$(C_g = C_1 = C_2) \vee ((C_g = C_1) \wedge (C_2 = null)) \vee$$
$$((C_1 = null) \wedge (C_g = C_2))\}. \tag{15}$$

Note that some alias instances do not have source aliases (*null*), while some alias instances have more than one source alias. An example of the former is when an alias relation is generated independently of any other alias relations. An alias instance of this case has *null* for its source alias and call site. An example of the latter is when an alias relation is induced by more than one alias relations. Informally, Expression 15 states that the generation rule be applied only when the two alias instances have conforming call sites: either the two call sites are the same or at least one of them is *null*.

In Section 3.1 we observed that unioning alias information at join points of control flow results in a loss of precision. A similar technique to the one developed here can be used for limited distinction between aliases generated along different CFG paths.

4.5 Reference Parameters

In this and the following sections, we describe how to compute $ALIAS_{Entry}$ from A^C_{IN} sets and how to compute A^C_{OUT} from A^C_{IN} and $ALIAS_{Exit}$ of the called procedure. We assume that static aliasing information is available; alias instances due to static aliasing are without source aliases and are used to initialize $ALIAS_{Entry}$. We describe the accommodation of reference parameters in this section, and call-by-value parameters in the following section. We will use A_{IN} for A^C_{IN} and A_{OUT} for A^C_{OUT} when the call site C in question is clear from the context.

Let a_i and f_i be the ith actual and formal parameters of a subroutine call at call site C in Q which invokes P. Let $\gamma_i \circ AR$ denote the new alias relation obtained by replacing each occurrence of a_i (if any) in AR with f_i, and $\gamma_i^{-1} \circ AR$ denote the new alias relation obtained by replacing each occurrence of f_i (if any) in AR with a_i. Formally,

$$\gamma_i \circ AR = AR(f_i/a_i), \tag{16}$$
$$\gamma_i^{-1} \circ AR = AR(a_i/f_i). \tag{17}$$

We extend the γ operation to alias instances as follows:

$$\gamma_i \circ [AR, SAS, C] = [\gamma_i \circ AR, SAS, C]$$

Also, define $\Gamma \circ AR$ as the *set* of alias relations obtained by applying every combination of $\gamma_1 \ldots \gamma_N$ to AR, where N is the number of parameters at C, and define $\Gamma_p^{-1} \circ AR$ similarly (i.e., $\gamma_1^{-1} \ldots \gamma_N^{-1}$).[4] In the following, when the particular source alias sets or call site are not relevant, they will be shown simply as '$\cdot$'. $ALIAS_{Entry}$ of P is generated at the call site C in routine Q as follows:

- For each $[< u, v >, \cdot, \cdot] \in A_{IN}$, add the following to $ALIAS_{Entry}$:

$$\{[AR_j, \{< u, v >\}, C] \mid AR_j \in \Gamma \circ < u, v >\}.^5$$

We use the following examples to illustrate the above rule. In the examples, we do not show trivial alias instances such as $[< u, u >, null, null]$ in the generated alias instances.

1. For $[< u, v >, \cdot, \cdot] \in A_{IN}$, where u and v are global to the callee:[6]
   - alias instance: $AI_k = [< u, v >, < u, v >, C]$;
2. From trivial alias $[< a_i, a_i >] \in A_{IN}$:
   - alias instance: $AI_k = [< a_i, f_i >, null, C]$.
3. Assume $A_{IN} = \{[< *x, z >, \cdot, \cdot], [< y, *w >, \cdot, \cdot]\}$, where x,y, w, and z are all global variables. Also, assume x and y are passed as the first and the second actual parameters to P: a_1 is x, and a_2 is y.
   - alias instances:
     (a) $[< x, f_1 >, null, C]$;
     (b) $[< *f_1, z >, \{< *x, z >\}, C]$;
     (c) $[< y, f_2 >, null, C]$;
     (d) $[< f_2, *w >, \{< y, *w >\}, C]$;

Note that in computing $ALIAS_{Entry}$, we do not explicitly screen out *non-reachable aliases of* P – aliases that contain access paths not reachable in P, because such non-reachable aliases of P may become reachable aliases of routines (transitively) called by P [6].

A_{OUT} at a call site C of P in Q consists of two components: $A_{OUT}(null)$ which is the set of aliases in $ALIAS_{Exit}$ that are generated independently of

[4] Since an alias relation consists of a pair of access paths, it is sufficient to apply every γ combination with length up to and including two.

[5] Non-reachable aliases, which are harmless to keep, can be screened out after interprocedural alias analysis is done.

[6] The zero length γ combination is the identity operator, which is applied in this case.

any aliases propagated from the call sites of P, and $A_{OUT}(induced)$ which is the set of aliases in $ALIAS_{Exit}$ that are induced by one or more aliases propagated from the call site C that invokes P. $A_{OUT}(null)$ can be computed as follows:

$$A_{OUT}(null) = \{[< u, v >, null, null] \in \Gamma^{-1} \circ AI \mid AI \in ALIAS_{Exit}, CallSite(AI) = null\}. \quad (18)$$

To compute $A_{OUT}(induced)$, we first compute $ALIAS_{Exit}(C)$, the subset of $ALIAS_{Exit}$ which need be propagated to call site C, as follows:

$$ALIAS_{Exit}(C) = \{[< u, v >, SAS, C] \in \Gamma^{-1} \circ AI \mid AI \in ALIAS_{Exit}\}. \quad (19)$$

Now, each alias instance

$$AI_p = [AR_p, SAS_p, C] \in ALIAS_{Exit}(C)$$

potentially generates a set of new alias instances, each of which has the same alias relation (AR_p), to be propagated to call site C that invokes P as follows:

$$\{AI_n = [AR_p, SAS_n, C_n]\},$$

where SAS_n and C_n are the new source alias sets and call sites. Note that some instances in $ALIAS_{Exit}(C)$ may fail to generate any new instances to be propagated to C. Also note that SAS_p is a subset of A_{IN}, since $ALIAS_{Exit}(C)$, from which AI_p is computed, is a set of alias instances induced by alias relations propagated at call site C in Q which invokes P.

We now describe how to compute SAS_n and C_n, and also describe when alias instances in $ALIAS_{Exit}(C)$ fails to generate instances to be propagated back to C. Informally, SAS_n is the union of the source alias sets of the alias instances in A_{IN} with the same call sites whose alias relation matches one or more of the alias relations in SAS_p. Formally, SAS_n and C_n are as follows:

$$SAS_n = \bigcup_i SAS_i \text{ such that}$$
$$\forall AR_i \in SAS_p, \exists (AI_i = [AR_i, SAS_i, C_n]) \in A_{IN}. \quad (20)$$

Then, $A_{OUT}(induced)$ is as follows:

$$A_{OUT}(induced) = \bigcup_n \{AI_n\}, \quad (21)$$

and A_{OUT} is as follows:

$$A_{OUT} = A_{OUT}(null) \cup A_{OUT}(induced). \quad (22)$$

Some AR_p may fail to be propagated back to call site C. For example, assume there are two call sites C_1 and C_2 that invoke Q, and that two alias instances AI_1 and AI_2 have been propagated to Q from C_1 and C_2: AI_1 and AI_2 cannot hold during the same execution instance. Also, assume AI_1 and AI_2 both are propagated to P at C in Q which invokes P. Now in P, AI_1 and AI_2 both have C as their call sites and can generated AI_p as follows:

$$[AR_p, \{AR(AI_1), AR(AI_2)\}, C].$$

When AR_p is propagated back to the caller, Q, we find out that AI_1 and AI_2 have different call sites, C_1 and C_2, and that AI_p is an invalid alias instance. However, we do not attempt to *invalidate* all the alias relations in P (and subroutines transitively called by P) that are induced by AR_p.

4.6 Call-by-Value Parameters

Let $\hat{\gamma}_i \circ A_{IN}$ denote the new aliasing information obtained by replacing each occurrence of $*a_i$ in A_{IN} with $*f_i$. Formally,

$$\hat{\gamma}_i \circ A_{IN} = A_{IN}(*f_i / *a_i). \quad (23)$$

Note that, in this section, $*a_i$ and $*f_i$ imply that a_i and f_i are access paths of a pointer type. We also define $\hat{\Gamma} \circ AR$ similar to $\Gamma \circ AR$ for reference parameters, but in terms of $\hat{\gamma}$'s. Then we compute $ALIAS_{Entry}$ of P for call site C in Q as follows:

- For each $[< u, v >, \cdot, \cdot] \in A_{IN}$, add the following to $ALIAS_{Entry}$:

$$\{[AR_j, \{< u, v >\}, C] \mid AR_j \in \hat{\Gamma} \circ < u, v >\}.^7$$

An example of this rule is the trivial alias relation $< u, u >$ of a pointer-type global variable u passed as ith parameter, which results in the following alias instance at the entry node:

$$[< *u, *f_i >, null, C].$$

Although correct for computing alias information, the rule is not enough for correct *side-effect analyses*, such as *MOD* and *USE* [4], of a subroutine call. We describe later why this rule is not enough for correct side-effect analyses and how to augment the rule for correct side-effect analyses.

As before with reference parameters, A_{OUT} at a call site C of P in Q consists of two components: $A_{OUT}(null)$ and $A_{OUT}(induced)$. $A_{OUT}(null)$ can be computed as follows:

$$A_{OUT}(null) = \{AI = [< u, v >, null, null] \mid AI \in ALIAS_{Exit}, CallSite(AI) = null\}. \quad (24)$$

Notice that unlike in Equation 18, there is no Γ^{-1} operation involved in Equation 24. $ALIAS_{Exit}(C)$ is computed as follows:

$$ALIAS_{Exit}(C) = \{AI = [< u, v >, SAS, C] \mid AI \in ALIAS_{Exit}\}. \quad (25)$$

Also, notice the lack of Γ^{-1} operation in Equation 25. Then, we compute $A_{OUT}(induced)$ and finally A_{OUT} the same way we compute for reference parameters described in the previous section.

[7] As with the reference parameter case, non-reachable aliases can be screened out.

```
S1:   SUB1 {
S2:     int A, B;
S3:     SUB2 (&A,&B);
S4:   };
S5:   SUB2 (int *f1, *f2) {
S6:     int *tmp = f1;
S7:     f1 = f2;
S8:     f2 = tmp;
S9:     *f1 = *f2;
S10:  };
```

Figure 10: Example Program Segments with Value Parameters

5 Side-Effect Analysis with Pointer Parameters

The conventional methods for side-effect analysis [4, 5, 9] decompose the computation into separate direct side-effect and alias analyses. Aliasing is factored into direct side effects after they have been computed. In the presence of pointers, side-effect analysis cannot be performed separately from alias analysis, and this decomposition produces an incorrect solution. In this section, we illustrate that the conventional method does not correctly handle pointers, and describe our solution.

Figure 10 shows procedure $SUB1$ passing two pointer parameters to procedure $SUB2$. With conventional side-effect analysis, the USE and MOD sets of procedures $SUB2$ are as follows [4]:

$$USE(SUB2) = \{*f2\},\ MOD(SUB2) = \{*f1\},$$

which shows that the access path involving the first parameter is in MOD and the access path involving the second parameter is in USE.

With the side-effect information of $SUB2$ computed as above, by the conventional method, the first actual parameter (A) is regarded as modified, and the second actual parameter (B) is regarded as used at S_3, which is *incorrect*; B is the one that is modified, and A is the one that is used at S_3.

Our solution is to compute side-effects of a procedure as a part of alias analysis by introducing a *representative parameter* r_i for each ith actual parameter local to the caller. Thus, with $SUB2$, we introduce $r1$ and $r2$ along with $f1$ and $f2$. With that, we create $[< r1, *f1 >, null, C]$ and $[< r2, *f2 >, null, C]$ as part of $ALIAS_{Entry}$ of $SUB2$, where C denotes the particular call site that invokes $SUB2$. Now, $ALIAS_{Exit}(SUB2)$ will have the following alias relations:

$$< *tmp, *f2 >, < r2, *f1 >, and < r1, *f2 >,$$

and the USE and MOD sets of $SUB2$ will be as follows:

$$USE(SUB2) = \{r1\},\ MOD(SUB2) = \{r2\},$$

which do not include access paths local to $SUB2$: $*f1$ and $*f2$. In mapping the USE and MOD sets of $SUB2$ to its call sites, we map $r1$ to the first parameter and $r2$ to the second parameter. With this, we correctly compute that B is modified and A is used at S_3.

```
S1:   int cell1, cell2, *p;
S2:   cell1 = ...;
S3:   cell2 = ...;
S4:   p = &cell1;
S5:   *p = ...;
S6:   p = &cell2;
S7:   ...= *p;
```

Figure 11: Example C Program Segment.

Apart from the cost of alias analysis, this side-effect analysis does not incur any cost beyond conventional side-effect algorithms. The precision of the side effect analysis algorithm depends on the underlying alias analysis method.

6 Naming for Correctness, Precision and Efficiency

The naming of memory locations is required for the correctness of data-flow analysis. We address this issue in the beginning of this section. Further, naming techniques described in the remainder of this section improve the precision and efficiency of our alias analysis methods.

6.1 Data-Flow Analysis and Aliasing

In classical data-flow analysis without pointers, *uses* and *defs* at each statement are first computed in terms of (variable) names associated with the accessed memory locations. Then, alias information for each variable is used to identify additional variables potentially used or defined at each statement. While the association between a name and its memory location does not change within a name scope, the association between a pointer expression and its memory location can (repeatedly) change within a name scope. For example, during an execution instance of the code segment in Figure 11, the memory location associated with access path $*p$ changes from *cell*1 to *cell*2, while the memory locations associated with names *cell*1 and *cell*2 do not change. The result is that while there should be a *def-use* data-flow chain [2] from S_3 to S_7, there should be no def-use chain from S_5 to S_7.

We propose the use of memory locations (variable names) aliased to a pointer expression rather than the pointer expression per se for data-flow analysis. With this scheme, S_5 performs a write-access of *cell*1, which is aliased to $*p$ at that statement, and S_7 performs a read-access of *cell*2, which is aliased to $*p$ at that statement; correctly identifying the def-use chain from S_3 to S_7 becomes straightforward with this method. We refer to memory locations associated with names, such as *cell*1 and *cell*2 in the example, as *named objects*.

Since we use the named object, rather than the access path through pointers aliased to the named object, we do not lose any data-flow information by not keeping alias pairs such as $< *p, *q >$ that do not contain a named object. However, with such alias pairs discarded, a transitive closure operation is required for correctness. With the compact representation described

in Section 6.3, the transitive closure does not result in a loss of precision. Therefore, we do not keep alias pairs without named objects.[8]

6.2 Naming Anonymous Objects

Using aliased memory locations is possible only when pointers point to named objects. For *anonymous* objects in a heap storage (memory) returned by a storage allocator such as malloc in C, we use a naming scheme based on the place (statement) in the program where an anonymous object is created as in [7], combined with the *k-limited* approach: we allow up to k instances of the object to have distinct names for each statement, where k is a pre-determined constant (as in [14, 21, 18, 13]). We thus ensure that a pointer expression is always associated with at least one named object to which it is aliased. The named object, instead of the access path of the expression itself, is used for data-flow analysis.

A potential drawback of this naming scheme is that different instances of anonymous objects created at the same malloc statement, but along different paths in the PCG, become indistinguishable, as shown in Figure 2. Our naming method qualifies these named anonymous objects with an additional name string that captures call path information. With this scheme, we identify two qualified names as identical if and only if one qualified name is a prefix substring of the other.

For example in Figure 2, we first name the object allocated in $SUB3$ as, say, N_{S1} based on the `malloc` statement. At the call site of $SUB3$ in $SUB1$, the name becomes e_3N_{S1}, and $*r$ becomes aliased to e_3N_{S1}; and at the call site of $SUB1$ in *main*, the name becomes $e_1e_3N_{S1}$, and $*p$ becomes aliased to $e_1e_3N_{S1}$. At the call site of $SUB3$ in $SUB2$, the name becomes e_4N_{S1}, and $*s$ becomes aliased to e_4N_{S1}; and at the call site of $SUB2$ in *main*, the name becomes $e_2e_4N_{S1}$, and $*q$ becomes aliased to $e_2e_4N_{S1}$. Note that names are prefixed with PCG edges when they are propagated from the callee to the caller, but not when and after they are propagated from the caller to the callee. Therefore, object names, propagated from a caller to a callee, do not change when they are propagated back from the callee to the caller.[9]

In *main*, we identify $*r$ and $*p$ as accessing the same named object since $*r$ is aliased to $e_1e_3N_{S1}$, which is a prefix substring of (in fact identical to) the named object to which $*p$ is aliased. Note that $*r$ is aliased to $e_1e_3N_{S1}$ in *main*, while it is aliased to e_3N_{S1} in $SUB1$. We also identify $*s$ and $*q$ as aliases for the same reason. However, $*p$ and $*q$ are not aliases since the object name to which $*p$ is aliased is not a prefix substring of the object name to which $*q$ is aliased, and vice versa. In $SUB4$, we identify $*t$ and $*r$ as accessing the same named object since $*t$ is aliased to N_{S1}, which is a substring of e_3N_{S1}, to which $*r$ is aliased.

We handle call chains with cycles (due to recursive calls) by not allowing a name string to have duplicate call graph edges in it. With this scheme, the number and length of name strings are bounded by the program size, while the capability of distinguishing different instances of the objects created at the same place is compromised.

6.3 Compact Representation of Alias Information

Alias information such as given in (7) and (8) can be regarded as *exhaustive* information: it contains all explicit alias relations holding at *each* statement. However, exhaustive alias information holding at each statement is rarely needed. In most cases for a statement, alias relations of only those access paths referenced at that statement are needed, and having exhaustive information for all the access paths at each statement incurs unnecessary time and space cost.

Pointer-induced alias relations determine a directed graph as shown in Figure 3: $< *p, a >$ implies that there is a (de-referencing) edge from object p to object a. Likewise, $< **q, b >$ implies that there exists an object c such that there is an edge from q to c and one from c to b. Since every object pointed to by a pointer has a unique name in our analysis, $< **q, b >$ implies the existence of $< *q, c >$ and $< *c, b >$ for some (named) object c, from which we can deduce $< **q, b >$. Each named object corresponds to a unique node in the directed graph.

We improve the time and space efficiency during alias analysis by keeping only alias relations with no more than one level of de-referencing. Full alias information is expanded later as needed when alias information is factored into general data-flow analysis. For a given access path, the corresponding path(s) in the directed graph is (are) traversed to determine the named object(s) to which it is aliased. During this traversal, the number of de-references encountered must be counted. Note that this is equivalent to performing *transitive reduction* [1] over the directed graph of alias relations during alias analysis, and later computing full alias information on demand.

With this compact representation, alias information of (7) in Section 3 will become:

$$\begin{aligned}\{\ & < *p, cell1 >, < *q, cell2 >, \\ & < *(cell1.left), cell3 >, \\ & < *(cell1.right), cell2 >\},\end{aligned} \tag{26}$$

and alias information of (8) will become:

$$\begin{aligned}\{\ & < *p, cell1 >, < *q, cell2 >, \\ & < *(cell1.left), cell3 >, \\ & < *(cell1.right), cell2 >, \\ & < *(cell1.left), cell2 >, \\ & < *p, cell2 >\}.\end{aligned} \tag{27}$$

Later at S_{11} in Figure 4, we readily obtain alias relations of $*p$ as $\{< *p, cell1 >, < *p, cell2 >\}$. At S_{12}, we can compute alias relations of $*(*p).left$ as follows:

$$\{< *(*p.left), cell3 >, < *(*p.left), cell2 >\}$$

by combining the following alias relation in 27:

$$\begin{aligned}\{\ & < *p, cell1 >, < *(cell1.left), cell3 >, \\ & < *(cell1.left), cell2 >\}.\end{aligned} \tag{28}$$

[8] Alias information such as $< *p, *q >$ can be useful for data-flow analysis within a set of contiguous statements during which no alias information changes with respect to $*p$ or $*q$. If so, we can easily maintain such alias pairs.

[9] Note that alias information propagated from a caller to a callee can be propagated from the callee only to that caller, not to other callers.

```
S1:   p = &x;
S2:   q = p;
S3:   *p = &y;
```

Figure 12: Example C Program Segment.

```
S1:   x = &y;
S2:   p = q = &x;
S3:   x = &z;
```

Figure 13: Example C Program Segment.

An advantage of the transitive reduction/closure method is demonstrated by the code fragment listed in Figure 12. In our compact representation the alias relations at S_2 are $< *q, x >$ and $< *p, x >$. The alias relations at S_3 are $< *q, x >, < *p, x >$, and $< *x, y >$. We generate the alias relation $< **q, y >$ by applying our transitive closure operation at S_3. A full representation would require that the additional alias relation $< *p, *q >$ be added at S_2 in order to determine that the alias relation $< **q, y >$ exists at S_3.

6.4 Compact Representation and Precision of Alias Information

Performing the transitive reduction of alias relations can improve the precision of alias information. Consider the program segment of Figure 13. Assuming no aliases hold before S_1, $< *x, y >$ is the only alias after S_1. After S_2, the exhaustive alias relations holding are as follows:

$$\{ \quad < *x, y >, < *p, x >, < *q, x >, < *p, *q >, \\ < **p, y >, < **q, y >, < **p, *x >, < **q, *x >, \\ < **p, **q >\}, \qquad (29)$$

all of which can be represented by the following transitive reduction:

$$\{< *p, x >, < *q, x >, < *x, y >\}. \qquad (30)$$

After S_3, the transitive reduction information becomes:

$$\{< *p, x >, < *q, x >, < *x, z >\}, \qquad (31)$$

from which the exhaustive information can be computed:

$$\{ \quad < *x, z >, < *p, x >, < *q, x >, < *p, *q >, \\ < **p, z >, < **q, z >, < **p, *x >, < **q, *x >, \\ < **p, **q >\}. \qquad (32)$$

The difference between (29) and (32) is that every access path y in (29) has been replaced by z in (32).

However, the exhaustive alias information computed by applying S_3 directly to (29) is as follows:

$$\{ \quad < **p, y >< **q, y >\} \cup \qquad (33)$$
$$\{ \quad < *x, z >, < *p, x >, < *q, x >, < *p, *q >, \\ < **p, z >, < **q, z >, < **p, *x >, < **q, *x >, \\ < **p, **q >\}, \qquad (34)$$

which contains additional alias relations (33), which are carried over from (29).[10] These alias relations are originated from $< *x, y >$ at S_1 and $\{< *p, x >, < *q, x >\}$ at S_2. Once generated, however, they have lost their sources and are carried over after S_3, although S_3 invalidates one of their sources: $< *x, y >$. Therefore, performing the transitive reduction not only improves the efficiency of the analysis, but can improve the precision as well.

Note that the same improved precision can also be obtained by keeping must-alias information: If we keep the information that $< *p, x >$ and $< *q, x >$ are must-aliases, we can regard $*p$ and $*q$ as being modified at S_3 in the figure, and can apply the killing rule to them as well as x.

[10] Note that $< **p, *x >$ and $< **q, *x >$ in 34 are implicit aliases derived from $< *p, x >$ and $< *q, x >$, respectively.

7 Complexity Analysis

There are three distinct phases of our flow-insensitive alias algorithm: initial intraprocedural analysis for generation and optional killing information, interprocedural iteration over the PCG, and final intraprocedural analysis. The space complexity of the analysis during the first and second phases is $O(V * T * P)$, where V is the total number of distinct objects (i.e. variables and named anonymous objects), T is the maximum number of aliases for an object, and P is number of procedures in the program. The space complexity can also be expressed as $O(A * P)$, where A is the maximum number of alias relations holding at a statement. This is expected to be smaller than $V * T$. The space complexity during the final phase is $O(A * S)$, where S is the number of statements in the program.

The time complexity of the intraprocedural killing analysis is linear in the height of the dominator tree and in the height of the postdominator tree of each procedure and is, thus, $O(P * S)$. The time complexity during the second phase is $O(A^2 * P^2)$. The time complexity of the final intraprocedural phase is $O(d*(A*(C+M)+T^2*N))$, where C is the total number of call sites, M is the total number of *MeetNodes* in the SEGs, N is the total number of nodes of all the SEGs (minus call sites and *MeetNodes* of the program), and d is the *loop-connectedness parameter*, the largest number of back edges on any cycle-free path of the program's control flow graphs [15]. M and N can grow linearly with the size of the program, but are expected to be small in practice. The overall time complexity of the flow-insensitive algorithm is, thus, $O(P*S+A^2*P^2+d*A*C+d*A*M+d*T^2*N)$.

There are three distinct phases of our flow-sensitive alias algorithm: the SEG construction phase, during which the SEG for intraprocedural alias analysis is constructed for each procedure; the interprocedural phase during which, by iteration over the PCG, $ALIAS_{Exit}$ and $ALIAS_{Entry}$ of each procedure are computed; the final intraprocedural phase, during which aliasing is computed for the access paths of each statement in each

procedure. The final intraprocedural analysis is the same as with the flow-insensitive algorithm, and is analyzed above. In the construction of the SEG for a procedure, the time (and space) complexity is CFE^2+CFN^2, where CFE (CFN) is the number of edges (nodes) in the control flow graph [8]. For this phase, then, the complexity is $EE + NN^2$, where EE (NN) is the total number of nodes (edges) in the control flow graphs of the procedures in the program. In practice, the complexity of SEG construction is $O(CFE + CFN)$ for a procedure and so $O(EE + NN)$ for a program [11]. The space complexity during the interprocedural phase is $O(A * P)$. The interprocedural phase dominates the time complexity. The worst case requires $O(A * P)$ iterations over the PCG. The time complexity to compute new $ALIAS_{Entry}$ of the procedures for each iteration over the PCG is $O(d*(A*(P+C+M)+T^2*N))$. The overall (inter- and intraprocedural) time complexity of the flow-sensitive algorithm is $O(d*A^2*P^2+d*A^2*P*C+d*A^2*P*M+d*A*P*T^2*N)$, where the last term is dominant. In practice, we expect the iteration over the PCG to converge rapidly, resulting in the overall time complexity of $O(k(A_a*(P+C+M)+T_a^2*N))$, where k is a small constant, A_a is the average number of aliases holding at a statement, and T_a is the average number of aliases for an object.

8 Related Work

The most closely related work is that of Landi and Ryder [16, 17], which present algorithms for computing aliasing for the same language constructs that we consider here. Our method for computing intraprocedural aliases differs from theirs in that it employs a sparse representation, which provides a more efficient framework than control flow graphs. Our transfer functions are simpler to use than their alias-generating rules based on case analysis.

Our method for computing interprocedural aliases, and for relating *inter*-procedural and *intra*-procedural alias relations, also differs from theirs. Their method computes may-aliasing based on *may-hold* information. May-hold is a relation whose arguments are a set of aliases AA, a node in the control flow graph n, and an alias pair $< a, b >$. It represents whether $< a, b >$ holds at n, assuming that there is a path from the program entry node to the entry node of the procedure containing n, for which every alias in AA holds. The algorithm of [17] finds the may-hold relations which are trivially true, and computes the set of all true may-holds using a worklist technique. May-aliasing can be computed from the may-hold solution in time linear with respect to the size of the solution. The worst case time complexity of their method is $O(S * V^4 + P * V^6)$, where V is the number of variables, S the number of statements, and P the number of procedures in the program. The worst case complexity of our method, corresponding to setting $T = V$ and $A = V^2$ in Section 7, is $O(N*V^4*P)$, where P is the number of procedures and N is the total number of SEG nodes of the program, which is substantially smaller than S, the total number of statements in the program.

To reduce the number of alias sets considered in the may-hold computation, which is potentially a powerset of 2^n elements (with n alias relations $(A_1 \ldots A_n)$ holding at the entry node), their method approximates by allowing for only singleton assumed-alias sets, each of which has at most one alias relation. With this approximation, they keep only $n + 1$ assumed alias sets $(AA(0) = \{\}, AA(1) = \{A_1\}, \ldots, AA(n) = \{A_n\})$ out of 2^n possible sets, resulting in a loss of precision (which we do not incur) in the presence of more than one level of pointer de-referencing.

Another related area is the work done in conflict detection between dynamically allocated recursive structures [7, 18]. In treating recursive structures, we use a naming scheme based on the statement in the program where an anonymous object is created as in [7], combined with the *k-limited* approach (as in [14, 21, 18, 13]). A previous drawback of this naming scheme is that different instances of anonymous objects created at the same malloc statement but along different paths in the PCG become indistinguishable. We have improved this by qualifying named anonymous objects with a name string that captures call path information. With this scheme, we substantially improve alias information by distinguishing different instances of dynamically allocated objects.

Intraprocedurally, our method is similar to the work done in conflict analysis [7, 12, 13, 18]. However, our method is the first to accommodate recursive data structures without explicitly building graphs. Our method, which uses Sparse Evaluation Graphs, is more efficient than methods based on control flow graphs. Again, our method includes an efficient and accurate computation of interprocedural alias information, while these graph-based methods in general either do not address or poorly address the interprocedural aspects of computing aliases.

Finally, we are the first to show that the conventional method for computing side effects of procedure calls (MOD and USE) based on [4] does not work in the presence of pointers. We develop an algorithm which handles pointers without loss of efficiency or precision.

9 Conclusions

We have provided accurate and efficient methods for computing intra- and interprocedural aliases and side effects for languages like LISP, C, C++ and Fortran 90 which contain reference parameters, pointers and recursion.

Without interprocedural alias analysis and procedure side effect information, compilers must make worst-case assumptions about formal parameters and variables global to a procedure. These assumptions impede optimizations and can result in inefficient code, particularly for a language like C for which pointer usage and procedure calls are generally frequent.

We have demonstrated that our flow-sensitive interprocedural alias method is more efficient and precise than existing methods. We have developed a new flow-insensitive interprocedural alias analysis method which accommodates pointers. This method is highly efficient and also provides a framework for computing alias solutions with varying degrees of precision.

The efficiency of our interprocedural analyses is improved by identifying realizable execution paths and by

distinguishing different instances of dynamically allocated objects created along different call paths. We have presented a technique for naming memory locations which guarantees the correctness of data flow analysis. We have also developed techniques for compacting alias information.

We have developed an efficient technique for precisely computing procedure call side effects in the presence of pointers, including the passing of pointers as reference or value parameters.

Acknowledgements

The authors are grateful to their colleagues in the PTRAN group for their contributions to this paper. Special thanks to Ron Cytron, Jeanne Ferrante, Michael Hind and Edith Schonberg for their advice and suggestions. We would also like to acknowledge Fran Allen, for her advice and support.

References

[1] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

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

[3] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the ptran analysis system for multiprocessing. *Proceedings of the ACM 1987 International Conference on Supercomputing*, 1987. Also published in The Journal of Parallel and Distributed Computing, Oct., 1988, 5(5) pages 617-640.

[4] John Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, pages 29–41, January 1979.

[5] Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[6] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. *Proceedings of the Sigplan '86 Symposium on Compiler Construction*, 21(7):162–175, July 1986.

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

[8] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, January 1991.

[9] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. *Proceedings of the Sigplan '88 Conference on Programming Language Design and Implementation*, 23(7):57–66, July 1988. Atlanta,Georgia.

[10] Keith D. Cooper and Ken Kennedy. Fast interprocedural alias analysis. *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 49–59., January 11-13 1989. Austin, Texas.

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, October 1991.

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

[13] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, 1989.

[14] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, January 1982.

[15] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *JACM*, 23,1:158–171, January 1976.

[16] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem classification. *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, January 1991.

[17] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

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

[19] Eugene W. Myers. A precise inter-procedural data flow algorithm. *Conference Record of Eighth ACM Symposium on Principles of Programming Languages*, 1981.

[20] Barry K. Rosen. Data flow analysis for procedural languages. *JACM*, 26(2):322–344, April 1979.

[21] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. *Conference Record of Fifteenth ACM Symposium on Principles of Programming Languages*, 1988.

[22] Robert Tarjan. Fast algorithms for solving path problems. *Journal of the Association for Computing Machinery*, 28(3):594–614, 1981.