An Improved Intra-procedural May-alias Analysis Algorithm

Deepak Goyal
Department of Computer Science
New York University
deepak@cs.nyu.edu

February 6, 1999

Abstract

Hind et al. ([5]) use a standard data flow framework [15, 16] to formulate an intra-procedural may-alias computation. The intra-procedural aliasing information is computed by applying well-known iterative techniques to the Sparse Evaluation Graph (SEG) ([3]). The computation requires a transfer function for each node that causes a potential pointer assignment (relating the data flow information flowing into and out of the node), and a set of aliases holding at the entry node of the SEG. The intra-procedural analysis assumes that precomputed information in the form of summary functions is available for all function-call sites in the procedure being analyzed. The time complexity of the intra-procedural may-alias computation for the algorithm presented by Hind et al. ([5]) is $O(N^6)$ in the worst case (where N is the size of the SEG). In this paper we present a worst case $O(N^3)$ time algorithm to compute the same may-alias information.

1 Preliminaries

Consider the C assignment statement

$$a = \&b; \tag{1}$$

The effect of the assignment is to assign the address of b to a. Therefore, after execution of Statement (1), expressions $*a$, and b refer to the same storage location. Consequently, any modification to the value stored at the location corresponding to $*a$ will result in the modification of the value stored at the location corresponding to b, and vice-versa. In this case, $*a$ and b are said to be *aliased.* We call expressions such as $*a$, and b, *access-paths* ([10]). More precisely, an access-path is the l-value of an expression constructed from variables, pointer dereferences, and structure field selection operators. The aliasing of $*a$ and b is represented by the alias-pair $\langle *a, b\rangle$.

An *alias relation* R at statement S is a set of alias-pairs $\langle x, y\rangle$. Such a relation R is the *must-alias* relation at statement S, if, an alias-pair $\langle x, y\rangle$ belongs to R, iff access-paths x and y refer to the same storage location in all execution instances of statement S. The absence of an alias-pair $\langle u, v\rangle$ from R implies that access-paths u and v refer to different locations in at least one execution instance of statement S. An alias-relation R is the *may-alias* relation at statement S, if, an alias-pair $\langle x, y\rangle$ belongs to R, iff access-paths x and y refer to the same storage location in *some* execution instance of statement S. Thus, the absence of an alias pair $\langle u, v\rangle$ from R implies that u and v must refer to different storage locations in all execution instances of statement S.

The computation of the actual must-alias and may-alias relations at all program points is undecidable ([9, 14]). Therefore, we must be satisfied with computing safe approximations to these relations. In order to understand what it means for an alias relation to be a *safe approximation* to the must-alias relation (respectively, the may-alias relation), we must understand how the information contained in the must-alias relation (repectively the may-alias relation) is used. For each alias pair $\langle x, y\rangle$ in the computed must-alias relation, we want to be sure that x and y refer to the same storage location in all execution instances. Therefore, any under-approximation (subset) of the actual must-alias relation is a safe approximation *e.g.* the empty set $\{\}$. For each alias pair $\langle u, v\rangle$ *not* in the computed may-alias relation, we want to be sure that

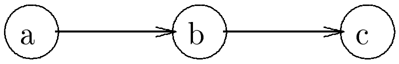

Figure 1: An example alias graph

u and v refer to different storage locations in all execution instances. Therefore, any over-approximation (*i.e.* superset) of the actual may-alias relation is a safe approximation *e.g.* the complete relation. It is fairly obvious that the actual must-alias relation is reflexive, symmetric, and transitive, and that the actual may-alias relation is reflexive, and symmetric (though not necessarily transitive). It is not difficult to prove that if relation R is a safe approximation to the actual must-alias relation, then the reflexive, symmetric and transitive closure of R is also safe. Correspondingly, it is not difficult to see that if relation R is a safe approximation to the actual may-alias relation, then the largest reflexive and symmetric subset of R is also safe.

In this paper, we will be concerned with computing a safe approximation to the may-alias relation at every program point, and from now on whenever we refer to an alias relation, we will mean a may-alias relation. May-alias relations, in addition to being reflexive and symmetric, also satisfy a congruence property, *viz.* the existence of a may-alias pair $\langle x, y\rangle \in R$ implies the existence of another may-alias pair $\langle *x, *y\rangle$ in R, if x and y are non-null, pointer-valued access paths. It is again not difficult to see that if R is a safe approximation to the actual may-alias relation, then the largest subset of R that satisfies the congruence property is also safe.

Since the may-alias analysis requires the computation of a may-alias relation at every program point, it is beneficial to have a compact representation for may-alias relations. The goal is to minimize space usage for alias information by not explicitly storing alias pairs that are inferable from the reflexivity, symmetry, or congruence properties of may-alias relations. The next sub-section gives a brief description of the compact representation used in [5].

1.1 Compact Representation of Alias Information

Hind et al. ([5]) use a compact representation of alias information in which the memory locations are associated with names, and are referred to as *named objects* ([6, 2]). The names are either user variable names or new names created by the analysis. The compact representation of an alias relation requires that for each alias-pair:

1. at least one access path component is a named object *i.e.*, does not contain a dereference, and
2. the other access path component has no more than one level of dereferencing.

Thus, all alias-pairs are of the form $\langle *a, b\rangle$ (respectively $\langle b, *a\rangle$), or $\langle a, b\rangle$, where a and b are named objects. It is important to note that alias pairs of the form $\langle a, b\rangle$, where both a and b are named objects can arise only because of ***structural*** aliasing, *i.e.* because of program constructs such as C's union or FORTRAN's EQUIVALENCE statement. These may-alias pairs are in-fact must-alias pairs at all program points. Such pairs of named objects are therefore closed under transitivity[1], *i.e.* if $\langle a, b\rangle \in R$, and $\langle b, c\rangle \in R$, then $\langle a, c\rangle \in R$. However, other alias pairs of the form $\langle *a, b\rangle$ are not closed under transitivity.

The compact representation of the alias relation, can be mapped to and from a directed-graph-based representation. Each node in the directed graph corresponds to a distinct location. Thus, each equivalence class of named objects is mapped to a node in the graph. Furthermore, every alias-pair of the form $\langle *a, b\rangle$ contributes a directed edge from the node corresponding to a to the node corresponding to b. Such a graph is called an *alias-graph.* For example, look at the alias-graph in Figure 1 (taken from [5]). The corresponding compact representation for this alias graph is $\{\langle *a, b\rangle, \langle *b, c\rangle\}$. The explicit representation of the alias information may be extracted from the compact representation (or the alias graph) by using Definition 2. If relation R is a compact representation of an alias relation, we define the corresponding

[1] remember that must alias relations are transitively closed.

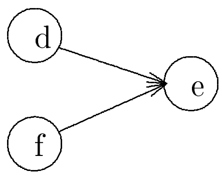

Figure 2: Another alias graph

explicit aliases of a_i, where a is a named object and i is the number of dereferences (*e.g.* $a_2 \stackrel{def}{=} **a$) by

$$ExplicitAliases(a,i) = \begin{cases} \{a\} & \text{if } i = 0 \\ \{r : \langle *s, r\rangle \in R | s \in ExplicitAliases(a, i-1)\} & \text{otherwise} \end{cases} \quad (2)$$

In terms of the alias graph, we can think of $ExplicitAliases(a,i)$ as returning the set of vertices reachable by paths of length i from vertex a. It is easy to verify that the explicit alias information of the alias graph in Figure 1 includes the alias-pairs $\langle *a, b\rangle, \langle *b, c\rangle, \langle **a, c\rangle, \langle a, a\rangle, \langle b, b\rangle, \langle c, c\rangle$, and all other alias-pairs that can *further* be inferred by reflexivity and congruence. Note that in this example, none of the named objects are aliased to each other, and hence each equivalence class of named-objects is a singleton set.

It is however, important to realize that the compact representation comes at the cost of a possible loss in precision. For example, consider the explicit alias relation

$$R = \{\langle *a, b\rangle, \langle *b, c\rangle, \langle **a, *b\rangle, \langle a, a\rangle, \langle b, b\rangle, \langle c, c\rangle\}. \quad (3)$$

Note, that relation R contains the alias pairs $\langle *a, b\rangle$, and $\langle *b, c\rangle$, but does not contain the alias pair $\langle **a, c\rangle$. This implies that although it is possible for $*a$ to be aliased to b, and $*b$ to be aliased to c, it is never possible for $**a$ to be aliased to c. This information, unfortunately, cannot be captured by the compact representation. The best compact representation of relation R will still be the alias-graph in Figure 1, and it is easily verified that $ExplicitAliases(a, 2) = \{c\}$, *i.e.* the explicit alias representation constructed from relation R contains the alias pair $\langle **a, c\rangle$. This is a consequence of the fact that a certain amount of transitivity is built into the way we extract explicit alias information from the compact representation using Definition 2. However, extraction of explicit alias pairs from the compact representation does not involve taking a full transitive closure. For example, looking at the alias graph in Figure 2, we see that the presence of alias pairs $\langle *d, e\rangle$ and $\langle *f, e\rangle (\equiv \langle e, *f\rangle)$ in the compact representation does not imply the existence of the pair $\langle *d, *f\rangle$ in the corresponding explicit representation. An interesting discussion on the relative merits and precision of both the compact and explicit representations may be found in [11] and the appendix of [5].

Definition 2 is a modified form of the definition given in [5], where an explicit recursive procedure $ComputeAliases(a, i)$ is given to compute the explicit aliases of named object a with i levels of dereference. The set of explicit aliases computed by Defintion 2 is the same as those computed by the procedure *ComputeAliases* in [5]. In the rest of the paper, we always use the compact representation of may-alias relations, and use Definition 2 to extract explicit alias information from the compact representations.

2 May-Alias Analysis

The intra-procedural may-alias analysis is performed on the *sparse evaluation graph* (SEG) ([3]). The SEG is a sparse representation of the *control flow graph* (CFG) comprising of *gen* nodes, *i.e.* nodes that may potentially modify the alias information, and *join* nodes, *i.e.* nodes where alias information is merged at join points. The SEG also contains a special entry node N_{Entry}, and an exit node N_{Exit}, such that every node is on some path from N_{Entry} to N_{Exit}. The computation of the may-alias information at each program point (*i.e.* on entry to, and, on exit from each SEG node n) is done by an iterative dataflow analysis ([15, 16]). Equations 4 and 5 (taken from the definitions in [5]) define the relationship between the alias information flowing into and out of an SEG node n.

Let In_n and Out_n be the alias relations holding immediately before and immediately after SEG node n. For a node n, In_n is the union of the Out sets of its predecessor nodes:

$$In_n = \bigcup_{p \in Preds(n)} Out_p. \tag{4}$$

The intuition behind Equation 4 is as follows. If an alias pair $\langle x, y \rangle \in Out_p$ for some predecessor (in the SEG) node p of node n, then, it may be the case that x and y are aliased in some execution instances in which control reaches node n from node p. Therefore, all such alias pairs $\langle x, y \rangle$ must be in the alias relation In_n holding at entry to node n.

The alias relation Out_n holding on exit from node n is defined as a function f_n of the alias relation In_n holding on entry to node n by Equation 5. Let the SEG node n correspond to the assignment $p_i = q_j$, where p_i is a pointer expression with i levels of indirection from variable p, and q_j is a pointer expression with j levels of indirection from variable q.[2] Out_n is defined as

$$Out_n = f_n(In_n) \stackrel{def}{=} In_n - Must(EA(p,i)) \ \cup \Big(\bigcup_{a \in EA(p,i), b \in EA(q,j+1)} \{\langle *a, b \rangle\} \Big), \tag{5}$$

where $EA(p,i)$ stands for $ExplicitAliases(p,i)$. The functions EA and $Must$ are defined below by Equations 6 and 7 respectively.

$$EA(p,i) = \begin{cases} \{p\} & \text{if } i = 0 \\ \{r : \langle *s, r \rangle \in In_n | s \in EA(p, i-1)\} & \text{otherwise} \end{cases} \tag{6}$$

$$Must(S) = \begin{cases} In_n & \text{if } S = \{\} \\ \{\langle *p, q \rangle \in In_n | p \in S\} & \text{if } |S| = 1 \text{ and } S = \{p\} \text{ for some } p \\ \{\} & \text{if } |S| > 1 \end{cases} \tag{7}$$

Notice that the alias set In_n is implicitly used in the computation of both $EA(p,i)$ and $Must(S)$ (for any set of named-objects S). It would in fact be better to use $EA_{In_n}(p,i)$ and $Must_{In_n}(S)$ instead of $EA(p,i)$ and $Must(S)$ respectively. However, we will continue to use $EA(p,i)$ and $Must(S)$ whenever In_n is clear from the context.

Equation 5 may be intuitively understood as follows. In a typical dataflow analysis framework ([1]), Out_n is defined by an equation of the form

$$Out_n = (In_n - Kill_n) \cup Gen_n,$$

where $Kill_n$ and Gen_n denote the dataflow information killed and generated at node n respectively. Let us look at the Gen set for the assignment $p_i = q_j$ given by the expression

$$\bigcup_{a \in EA(p,i), b \in EA(q,j+1)} \{\langle *a, b \rangle\}. \tag{8}$$

The objects in $EA(p,i)$[3] and $EA(q,j)$ are respectively the possible left hand sides and the possible right hand sides of the assignment. The effect of the assignment $p_i = q_j$ is to assign the value of some member of $EA(q,j)$ to some member of $EA(p,i)$. Thus, the corresponding effect of the assignment on the may-alias graph must be to add edges from every object in $EA(p,i)$ to every object that can be pointed to by a member of $EA(q,j)$, *i.e.* to every object in $EA(q,j+1)$. In Figure 3, the situation is described pictorially, and the dashed edges correspond to the Gen set of the assignment $p_i = q_j$.

The $Kill$ set is given by the expression $Must(EA(p,i))$ which corresponds to those alias pairs that must necessarily be killed by the assignment. If the set $EA(p,i)$ is a singleton set, say $\{a\}$, then a must necessarily get modified by the assignment, and hence all alias pairs of the form $\langle *a, x \rangle$ must be killed. If, however, $EA(p,i)$ is a set with more than one object, say $\{a, b, \ldots\}$, then at most one of these objects will

[2] $\&q$ is treated as q_{-1}.

[3] Recall that $EA(p,i)$ is the set of nodes reachable by a path of length i from node p.

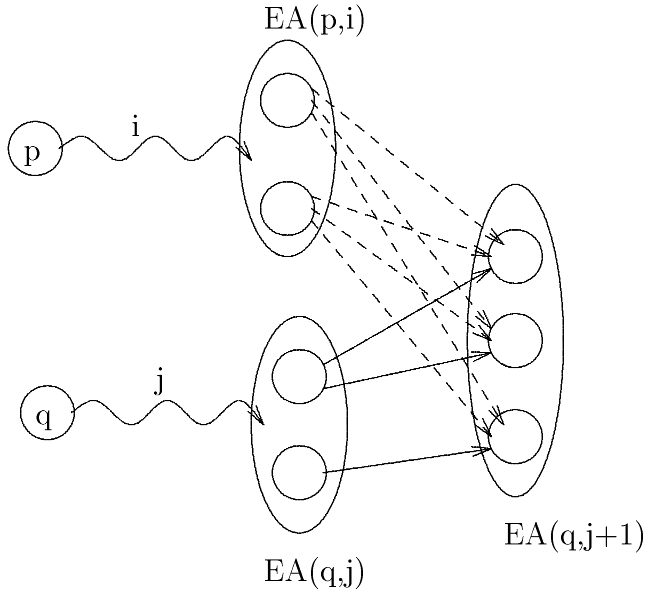

Figure 3: Dashed edges correspond to the alias pairs generated by the assignment $p_i = q_j$

get modified by the assignment, and the others will remain unchanged. Since it is not possible to know which object gets modified, therefore, it is not possible to determine which alias-pairs get killed. Hence, $Must(S)$ is conservatively defined to be the empty set $\{\}$, whenever $|S| > 1$. An interesting problem arises when $EA(p, i)$ itself evaluates to the empty set. In this case, we define the $Kill$ set to be the input alias set In_n, so that Out_n evaluates to the empty set. This stops the flow of alias information through node n, until some more alias information reaches node n, such that $EA(p, i)$ becomes non-empty. If, on termination of the iterative analysis, there are one or more SEG nodes for which the corresponding sets $EA(p, i)$ are empty, then it can be proven that if an execution path ever reaches any of these nodes, then a null pointer dereference error will occur.

In the next section, we describe three successively more efficient algorithms to compute may-alias information. The termination of each of these algorithms relies on the function f_n (given by Equation 5) being monotonic (increasing) with respect to the addition of alias pairs to In. It is easily verified that function $Must$, as defined by Equation 7, is anti-monotonic (decreasing), and function $ExplicitAliases$, as defined by Equation 6, is monotonic with respect to the addition of alias pairs to In. Then, it is easily verified from Equation 5 that function f_n is monotonic.

3 Computing May-Alias Information

This section describes three algorithms to compute the may-alias information at each SEG node n. The first sub-section describes a very simple iterative algorithm used by [5] that computes a safe approximation to the may-alias relation at entry to every SEG node n, which runs in $O(N^6)$ time where N is the size of the SEG. The next sub-section explains how the simple iterative algorithm can be improved by using a worklist based strategy to run in $O(N^5)$ time. Finally, we present a much more efficient algorithm based on the ideas of *dominated convergence* and *finite differencing* ([7]), that runs in worst case $O(N^3)$ time.

3.1 Simple Iterative Algorithm

From the previous section, it is clear that the may-alias relations to be computed at the entry and exit to each SEG node n, *i.e.* In_n, and Out_n must satisfy Equations 4 and 5. In fact, Equations 4 and 5 may be

Input: $In(0)$
Output: $In(i)$ for all $i = 0, \ldots, N$

$$
\begin{array}{ll}
\text{Init :} & \forall i = 1, \ldots N \quad In(i) = \{\} \\
& \forall i = 0, \ldots N \quad Out(i) = f_i(In(i)) \\
\text{Loop :} & \text{repeat} \\
& \forall i = 1, \ldots, N \ \mathit{loop} \\
& \quad In(i) = \cup_{p \in Pred(i)} Out(p) \\
& \quad Out(i) = f_i(In(i)) \\
& \mathit{end}\forall \\
& \text{until all the } Out(i)\text{'s converge.}
\end{array}
$$

Figure 4: Simple iterative algorithm for computing may-aliases

combined into the single equation

$$In_n = \bigcup_{p \in Pred(n)} f_p(In_p), \tag{9}$$

for each SEG node n, where f_p is the transfer function for node p. In other words, the computed may-alias information must be a fixed-point of the following system of equations:

$$
\begin{array}{lcl}
In_1 & = & \bigcup_{p \in Pred(1)} f_p(In_p) \\
In_2 & = & \bigcup_{p \in Pred(2)} f_p(In_p) \\
& \cdots & \\
In_N & = & \bigcup_{p \in Pred(N)} f_p(In_p).
\end{array} \tag{10}
$$

It is not difficult to see that the may-alias information that we seek to compute, is in fact, the least fixed point of the system of equations 10. All other fixed points of the system of equations 10 are also safe approximations of the may-alias relations at every node n, although they are not as accurate (precise) as the least fixed point.

Let the nodes of the SEG be numbered from $0, \ldots, N$, where node 0 denotes the entry node N_{Entry} of the SEG. Then, if each of the transfer functions f_p is monotonic, the least fixed point of the system of equations 10 can be computed by Kildall's iterative algorithm ([8]) shown in Figure 4.

Before we are ready to calculate the run-time complexity of the algorithm in Figure 4, we need to understand a few details about the low level implementation. Since an alias relation is implemented as a set of alias pairs, the efficient computation of Out_n from In_n using Equation 5 requires that the primitive operations such as set membership test and insertion into a set, be performed efficiently. The implementation of Hind et al. ([5]) relies on hashing to perform the set-based operations efficiently. Thus, the complexity of their algorithm is average-case $O(N^6)$ time, under the assumption that hashing takes $O(1)$ time on the average. However, it is possible to use a more sophisticated data-structure for implementing sets that allows primitive operations such as set membership test and insertion into a set to be implemented in worst case $O(1)$ time without the use of hashing. We do not present the details of these data structures in this paper but refer the interested reader to [12, 4] which describe data-structres that can be used to perform all set membership tests and set insertions in our algorithm in worst case $O(1)$ time.

Let us make the simplifying assumption that the SEG has been pre-processed so that the maximum number of predecessors of any node is no more than two. This can easily be done by adding a series of dummy join nodes to the SEG. The resulting number of nodes in the new SEG is still linear in the number of edges in the old SEG. We will let N denote the number of vertices in the processed SEG. Note, however, that both the number of vertices and the number of edges in the original SEG are linear in the size of the procedure. Let A_{max} denote the maximum number of aliases holding at any program point, and let L be a bound on i and j, *i.e.* the maximum number of dereferences to any of the named objects appearing in the assignment $p_i = q_j$.

Lemma 3.1 *$EA(p, i)$ can be computed in $O(i \times A_{max})$ time.*

Proof: We compute $EA(p,i)$ by successively computing $EA(p,0), EA(p,1), EA(p,2), \ldots, EA(p,i)$. The computation of $EA(p,k+1)$ from $EA(p,k)$ can be done by just computing the neighbors of all nodes in $EA(p,k)$, and can take at most A_{max} time. Thus, the successive computations of $EA(p,0), \ldots, EA(p,i)$ take $O(i \times A_{max})$ time. $\square$

Lemma 3.2 *The function $f_n(In_n)$ given by Equation 5 can be computed in $O(L \times A_{max})$ time.*

Proof: The computation of $f_n(In_n)$ is done by first computing $EA(p,i)$ and $EA(q,j+1)$. Since L is an upper bound on i and j, we see from Lemma 3.1 that $EA(p,i)$ and $EA(q,j+1)$ can be computed in $O(L \times A_{max})$ time. Next, it is easy to see that the computation of $Must(EA(p,i))$ takes at most $O(A_{max})$ time. Finally, since the size of Expression (8) is bounded by A_{max}, we see that the entire computation of $f_n(In_n)$ takes at most $O(L \times A_{max})$ time. $\square$

Corollary 3.3 *Each iteration of the algorithm in Figure 4 takes $O(N \times L \times A_{max})$ time.*

Lemma 3.4 *The number of iterations in the algorithm in Figure 4 is bounded by $(N \times A_{max}) + 1$.*

Proof: Each iteration of the algorithm, except for the last one, results in the addition of at least one new edge to the Out set of some node i. The total number of such edges for each node is bounded by A_{max}. Thus, the total number of iterations is bounded by $(N \times A_{max}) + 1$. $\square$

Corollary 3.5 *The time complexity of the algorithm in Figure 4 is $O(L \times A_{max}^2 \times N^2)$ time.*

It is quite reasonable to assume that L is actually a small constant. Also, in the worst case, A_{max} may be as large as N^2. Thus, a coarser estimate of the time complexity of the simple algorithm is $O(N^6)$. Next, we show how a simple worklist based strategy can improve the worst case time complexity of the algorithm to $O(N^5)$. Finally, we present a much more efficient algorithm for computing the same may-alias information in worst case $O(N^3)$ time.

3.2 A Worklist Based Strategy

One of the main sources of inefficiency in the simple iterative algorithm is the repetitive computation of $f_k(In(k))$ for all SEG nodes k in each iteration, including even all those nodes for which there is no change in the value of their In sets in the previous iteration. A simple optimization that eliminates needless repeated computations of $Out(k) = f_k(In(k))$, is the use of a worklist based strategy. A worklist of nodes, for which the In sets changed in the previous iteration is maintained. In each iteration, the computation of Out is performed only for the nodes in the worklist, and another worklist comprising of those nodes for which the In sets change in this iteration, is created for use in the next iteration. A careful implementation of the worklist strategy can improve the worst case time complexity of the may-alias computation to $O(L \times A_{max}^2 \times N)$ time. This is because, for each SEG node k, the re-computation of $Out(k)$ is done only whenever at least one new alias pair is added to $In(k)$. Using the fact that the size of $In(k)$ is bounded by A_{max}, and that the computation of $Out(k)$ takes $O(L \times A_{max})$ time, we get the worst case time complexity $O(L \times A_{max}^2 \times N)$ for the worklist strategy based algorithm. Again, assuming that L is a small constant, and that $A_{max} = O(N^2)$, we see that the complexity of the worklist based algorithm is $O(N^5)$ time.

3.3 Going Beyond the Worklist Strategy

The goal of using worklists is to eliminate needless re-computation of $Out(k)$, whenever $In(k)$ does not change for an SEG node k. The same idea can be extended to further improve the algorithm by using finite differencing ([13]) to compute $Out(k)$ incrementally for small changes to $In(k)$. The benefit of this strategy stems from the fact that although the computation of $Out(k) = f_k(In(k))$ from scratch may be quite expensive, the incremental computation of $Out(k)$ for a small incremental change in the value of $In(k)$ may involve relatively inexpensive computations. Figure 5 describes the outline of an algorithm based on this idea. Assuming that the new value of $Out(k)$ can be computed incrementally from the old values of $Out(k)$, $In(k)$, and the new edge x being added to $In(k)$, we first prove the correctness of the algorithm in Figure 5.

Input: $In(0)$
Output: $In(i)$ for all $i = 0, \ldots, N$

$$\begin{array}{ll} \text{Init}: & \forall i = 1, \ldots N \quad In(i) = \{\} \\ & \forall i = 0, \ldots N \quad Out(i) = f_i(In(i)) \\ \text{Loop}: & \text{while } \exists k : In(k) \subset \cup_{p \in Pred(k)} Out(p) \text{ loop} \\ & \quad x = \text{arbitrary element from } (\cup_{p \in Pred(k)} Out(p) - In(k)) \\ & \quad \text{Add } x \text{ to } In(k) \\ & \quad \text{Incrementally compute } Out(k) = f_k(In(k)) \\ & \text{end loop} \end{array}$$

Figure 5: Outline of the improved algorithm using Dominated Convergence

Lemma 3.6 *The algorithm in Figure 5 correctly computes the same may-alias information that is computed by the simple iterative algorithm in Figure 4.*

Before we sketch a proof of Lemma 3.6, we first make the following assertions:

Assertion 3.7 *Inside the while loop of Figure 5, the condition*

$$In(k) \subseteq \cup_{p \in Pred(k)} Out(p) \tag{11}$$

always holds for all $k = 1, \ldots, N$.

Proof: Assertion 3.7 is true initially on entry to the loop for the first time since for all $k = 1, \ldots, N$, the $In(k)$'s are equal to $\{\}$. The addition of $x \in \cup_{p \in Pred(k)} Out(p) - In(k)$ to $In(k)$ does not violate Condition 11. Since each function f_k is monotonic, the new value for $Out(k)$ must be a superset of its previous value, and hence cannot result in the violation of Condition 11. □

Assertion 3.8 *The condition*

$$Out(k) = f_k(In(k)) \tag{12}$$

is always true on entry to the while loop, for all $k = 0, \ldots, N$.

Proof: Condition 12 is clearly true on entry to the loop for the first time. The (incremental) re-computation of $Out(k)$ following the addition of x to $In(k)$ ensures that Condition 12 is also always satisfied on re-entry to the loop. □

We are now ready to see a proof sketch of Lemma 3.6.
Proof Sketch of Lemma 3.6: The algorithm in Figure 5 always terminates. Each iteration results in the addition of a new edge x to $In(k)$. Since the maximal size of each $In(k)$ is finitely bounded (by A_{max}), the termination of the algorithm is guaranteed. The termination condition of the algorithm ensures that for all $k = 1, \ldots, N$,

$$\cup_{p \in Pred(k)} Out_p \subseteq In(k) \tag{13}$$

Condition 13, together with Assertions 3.7 and 3.8, implies that on termination, the values of $In(k)$ are actually a fixed point of the system of equations 10. An argument based on the *Dominated Convergence Theorem* ([7]) can be used to prove that this solution is in fact the least fixed point and hence, the same as the solution computed by the simple iterative algorithm. □

Let us try to see how the repeated operations in the loop of the algorithm in Figure 5 can be performed efficiently. For each iteration of the loop, we first need to find an SEG node k for which

$$In(k) \subset \cup_{p \in Pred(k)} Out_p, \tag{14}$$

and extract an arbitrary element x from $\cup_{p \in Pred(k)} Out_p - In(k)$. Next, we add this edge to $In(k)$ and try to compute the new value for $Out(k)$ incrementally. Before we go into a detailed explanation of how to compute the new value of $Out(k)$ efficiently in an incremental fashion, let us see how we can efficiently find an SEG node k satisfying Condition 14, and extract an edge x from $\cup_{p \in Pred(k)} Out_p - In(k)$. We do this by maintaining a worklist of pairs $[k, x]$ of SEG node k, and edge x, such that $x \in \cup_{p \in Pred(k)} Out_p - In(k)$. We initialize this list before the entry to the loop for the first time. In each iteration, we extract a pair $[k, x]$ from this list and add edge x to $In(k)$. After computing the new value of $Out(k)$, for each edge y that is newly added to $Out(k)$, we go through every successor p of node k (in the SEG), and add the pair $[p, y]$ to the worklist, if $y \notin In(p)$, and the pair $[p, y]$ is not in the worklist already.

It is clear that each edge x that appears in the final value of $In(k)$ for some SEG node k, is inserted into the worklist at most once. Thus, the total cumulative cost of maintaining the worklist inside the loop for all the iterations is $O(N \times A_{max})$. Also, the worklist is initialized once before the entry to the loop for the first time. This initialization cost is also $O(N \times A_{max})$.

3.3.1 Incremental Computation of Out

Let us see how to recompute $Out(k)$ incrementally as a result of the addition of a new edge x to $In(k)$, for a fixed, but arbitrary SEG node k. Let In and In' denote the values of $In(k)$ before and after the addition of edge x, *i.e.* $In' = In \cup \{x\}$, and let $Out = f_k(In)$, and $Out' = f_k(In')$. Let the statement corresponding to SEG node k be $p_i = q_j$. From now on, let us abbreviate $EA(p, l)$ to S_l, and $EA(q, m)$ to T_m. Also, since Expression (8) resembles a cross-product, we will use $S_i \times T_{j+1}$ to denote $\cup_{a \in S_i, b \in T_{j+1}} \{\langle *a, b\rangle\}$. Then the transfer function f_k may be rewritten more compactly as

$$f_k(In(k)) \stackrel{def}{=} (In(k) - Must(S_i)) \cup (S_i \times T_{j+1}). \tag{15}$$

Let us also define a function $Neighbors(S)$ (Equation 16) to be the set of nodes reachable from the nodes in set S by a path of length 1 in the graph In[4].

$$Neighbors(S) \stackrel{def}{=} \{v : \langle *u, v\rangle \in In | u \in S\} \tag{16}$$

From Equation 6 it is obvious that for $l > 0$, $EA(p, l) = Neighbors(EA(p, l-1))$.

The incremental computation of Out' requires some extra storage costs. We will need to compute and store the auxiliary expressions S_l (or $EA(p, l)$) and T_m (or $EA(q, m)$) for $l = 0, \ldots, i$, and $m = 0, \ldots, j+1$. Let S_k and S'_k denote the values of $EA(p, k)$ computed relative the relations In and In' respectively. Initially, when $In = \{\}$, we have $S_0 = \{p\}$, and $S_l = \{\}$, for $l > 0$. Similarly, $T_0 = \{q\}$, and $T_m = \{\}$ for $m > 0$.

Let $\triangle Out$ denote $Out' - Out$. Similarly, let $\triangle S_k$ denote $S'_k - S_k$, and $\triangle T_m$ denote $T'_m - T_m$. Figure 6 gives the details of an efficient algorithm for computing $\triangle Out, \triangle S_0, \ldots, \triangle S_i, \triangle T_0, \ldots, \triangle T_{j+1}$ efficiently from the values of $In, Out, x, S_0, \ldots, S_i, T_0, \ldots, T_{j+1}$. (Note, that $\triangle S_0$ and $\triangle T_0$ are always equal to $\{\}$).

The algorithm described in Figure 6 is correct though it omits some details about how to perform Step 1 efficiently, which shall be explained shortly. Let us first try to understand why the algorithm in Figure 6 computes $\triangle Out$ correctly. Let the edge x being added to In be the alias pair $\langle *a, b\rangle$. It is easy to prove that if $S_i = \{r\}$ for some named object r, and $\triangle S_i = \{\}$, then

$$\triangle Out = f_k(In') - f_k(In) = \begin{cases} \{\langle *a, b\rangle\} \cup (S_i \times \Delta t_{j+1}) - Out & \text{if } a \neq r \\ (S_i \times \Delta t_{j+1}) - Out & \text{if } a = r \end{cases} \tag{17}$$

Similarly, if $|S_i| > 1$, then

$$\triangle Out = f_k(In') - f_k(In) = (\{\langle *a, b\rangle\} \cup (\triangle S_i \times T_{j+1}) \cup (S_i \times \triangle T_{j+1}) \cup (\triangle S_i \times \triangle T_{j+1})) - Out \tag{18}$$

The correctness of the algorithm in Figure 6 can be understood by looking at how the algorithm proceeds as edges are successively added to In. Initially, when In is empty, either S_i is empty or contains exactly one element. If S_i is empty, then the condition for either the first *if* or the second *if* in Step 2 must evaluate to true, and it is obvious that $\triangle Out$ is computed correctly. If the condition for the second *if* evaluates to true, then for all subsequent insertions of edges to In, the conditions for the first and second *if's* can never

[4] Recall that each alias relation can be thought of as an alias graph.

Let x be the edge $\langle *a, b\rangle$
Inputs: $In, Out, x, S_0, \ldots, S_i, T_0, \ldots, T_{j+1}$
Outputs: $\triangle Out, \triangle S_0, \ldots, \triangle S_i, \triangle T_0, \ldots, \triangle T_{j+1}$

Step 1 : for $(l = 1, \ldots, i)$ use $S_{l-1}, S_l, \triangle S_{l-1},$ and x to compute $\triangle S_l$
for $(l = 1, \ldots, j+1)$ use $T_{l-1}, T_l, \triangle T_{l-1},$ and x to compute $\triangle T_l$
$\triangle Out = \{\}$;

Step 2 :
if $S_i = \{\}$ and $\triangle S_i = \{\}$ then
  return; (because Out and Out' are both empty)
else
if $S_i = \{\}$ and $\triangle S_i \neq \{\}$ then
  compute Out' from In' directly from the definition and let $\triangle Out = Out' - Out$;
else
if $S_i = \{r\}$ (for some r) and $\triangle S_i = \{\}$ then
  Add $(x = \langle *a, b\rangle)$ to $\triangle Out$ if $a \neq r$ and $x \notin Out$;
  Add $\{\langle *c, d\rangle : c \in S_i, d \in \triangle T_{j+1} | \langle *c, d\rangle \notin Out\}$ to $\triangle Out$
else
if $S_i = \{r\}$ (for some r) and $\triangle S_i \neq \{\}$ then
  compute Out' from In' directly from the definition and let $\triangle Out = Out' - Out$;
else
if $|S_i| > 1$ then
  Add $(x = \langle *a, b\rangle)$ to $\triangle Out$ if $x \notin Out$
  Add $\{\langle *c, d\rangle : c \in \triangle S_i, d \in T_{j+1} | \langle *c, d\rangle \notin Out\}$ to $\triangle Out$
  Add $\{\langle *c, d\rangle : c \in S_i, d \in \triangle T_{j+1} | \langle *c, d\rangle \notin Out\}$ to $\triangle Out$
  Add $\{\langle *c, d\rangle : c \in \triangle S_i, d \in \triangle T_{j+1} | \langle *c, d\rangle \notin Out\}$ to $\triangle Out$
endif

Figure 6: Incremental computation of Out'

evaluate to true again. Equation 17 explains how $\triangle Out$ is correctly computed in the case for the third *if*. The correctness for the fourth *if* is obvious. If the condition for the fourth *if* evaluates to true, then for all subsequent insertions of edges to In, only the condition for the fifth *if* can evaluate to true. Equation 18 explains how $\triangle Out$ is correctly computed in the case for the fifth *if*.

Before we describe how to perform Step 1 efficiently, let us look at the cost of Step 2.

Lemma 3.9 *Let In_{final} and Out_{final} be the final values of $In(k)$ and $Out(k)$ on termination of the algorithm. The cumulative cost of Step 2 in the algorithm in Figure 6 over the addition of all the edges x to $In(k)$ (starting from $In(k) = \{\}$, and ending when $In(k) = In_{final}$), for some arbitrary but fixed SEG node k, is $O((|In_{final}| + |Out_{final}|) \times L)$.*

Proof Sketch: The conditions in the second and fourth *if's* (in Figure 6) are true at most once each during all the additions of edges to $In(k)$, and therefore the expensive computation of $\triangle Out$ from In' using the definition directly is done at most twice. The cost of each of these two computations are $O((In_{final} + Out_{final}) \times L)$ (or $O(A_{max} \times L)$ from Lemma 3.2). The rest of the work is done whenever the condition of either the third or fifth *if* is true. Suppose an edge $\langle *c, d\rangle$ is added to $\triangle Out$ inside the fifth *if* statement. The at least one of the following cases must be true:

1. the edge x is $\langle *c, d\rangle$
2. $c \in \triangle S_i, d \in T_{j+1}$, or
3. $c \in S_i, d \in \triangle T_{j+1}$, or
4. $c \in \triangle S_i, d \in \triangle T_{j+1}$.

The last three conditions can never be true for the same edge $\langle *c, d\rangle$ twice. Consider the case when $c \in \triangle S_i$. For each of the subsequent iterations, the condition $c \in S_i$ is true, and hence, the condition $c \in \triangle S_i$ can never be true again. Thus, all the edges considered for addition into $\triangle Out$ in the third or fifth *if* statements, are different from each other. Furthermore, each of these edges are members of Out_{final}. Thus, the cumulative cost of all operations performed in the third and fifth *if* statements is $O(|In_{final}| + |Out_{final}|)$.

Putting all the costs together, we see that the cumulative cost of Step 2 is $O((|In_{final}| + |Out_{final}|) \times L)$. $\square$

Now, let us see how to perform Step 1 efficiently. Let the edge x being added to In be $\langle *a, b\rangle$. It is fairly obvious that

$$Neighbors(A \cup B) = Neighbors(A) \cup Neighbors(B).$$

Using the fact that $S_k = Neighbors(S_{k-1})$, it is not difficult to prove that

$$\triangle S_k = \{y \in Neighbors(\triangle S_{k-1}) | y \notin S_k\} \cup \{y : y \in \{b\} | a \in S'_{k-1}\}$$

Lemma 3.10 *The cumulative cost of computing all $\triangle S_l$'s (for an arbitrary but fixed l) is $O(|In_{final}|)$.*

Proof: Any node v appears in $\triangle S_{k-1}$ at most once during the execution. Only at this time, we look at all the edges going out of node v (in the computation of $Neighbors(\triangle S_{k-1})$). Since each edge in In_{final} is looked at, at most once, the total cost is $O(|In_{final}|)$. $\square$

Thus the total cumulative cost of performing Step 1 is bounded by $O(L \times |In_{final}|)$. Since $|In_{final}|$ and $|Out_{final}|$ are both bounded by A_{max}, we see that the total work done for any fixed but arbitrary SEG node k is $O(L \times A_{max})$. Thus, the total cost of the algorithm is $O(L \times A_{max} \times N)$. The coarse estimate for the complexity of our algorithm is $O(N^3)$, where N is the size of the SEG. Another interesting fact worth mentioning is that except for the factor L (which in practice should be a small constant), the algorithm presented here is asymptotically optimal since its worst case time complexity is linear in the size of its output.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1988.

[2] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Programming Language Design and Implementation*, pages 296–310, 1990.

[3] J. D. Choi, M. Burke, and P. Carini. Automatic construction of sparse data flow evaluation graphs. In *18th annual ACM symposium on Principles of Programming Languages*, pages 55–66, 1991.

[4] D. Goyal and R. Paige. The formal reconstruction and speedup of the linear time fragment of willard's relational calculus subset. In Bird and Meertens, editors, *Algorithmic Languages and Calculi*, pages 382–414. Chapman and Hall, 1997.

[5] M. Hind, M. Burke, P. Carini, and J. D. Choi. Interprocedural pointer alias analysis. Accepted for publication in TOPLAS in 1999. An earlier version appeared as IBM Technical report #21055, December 1997.

[6] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Programming Language Design and Implementation*, pages 28–40, 1989.

[7] J. Keller and R. Paige. Program derivation with verified transformations – a case study. *CPAM*, 48(9-10), 1995.

[8] G. A. Kildall. A unified approach to global program optimization. In *ACM Symp. on Principles of Prog. Lang.*, pages 194–206, 1973.

[9] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[10] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Programming Language Design and Implementation*, pages 21–34, 1988.

[11] T. Marlowe, W. Landi, B. Ryder, J. Choi, and P. Carini. Pointer induced aliasing: A clarification. *SIGPLAN Notices*, 28(9):67–70, September 1993.

[12] R. Paige. Real-time simulation of a set machine on a ram. In N. Janicki and W. Koczkodaj, editors, *Computing and Information*, volume II, pages 69–73. Canadian Scholars' Press, Toronto, May 1989.

[13] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. on Programming Languages and Systems*, 4(3):401–454, 1982.

[14] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(6):1467–1471, November 1994.

[15] B. K. Rosen. Data flow analysis for procedural languages. *Journal of the ACM*, 26(2):322–344, April 1979.

[16] R. Tarjan. Fast algorithms for solving path problems. *Journal of the ACm*, 28(3):594–614, 1981.