A Safe Approximate Algorithm for Interprocedural Pointer Aliasing*

William Landi (landi@cs.rutgers.edu)
Barbara G. Ryder (ryder@cs.rutgers.edu)

Department of Computer Science
Rutgers University, New Brunswick, NJ 08903

Abstract

During execution, when two or more names exist for the same location at some program point, we call them *aliases*. In a language which allows arbitrary pointers, the problem of determining aliases at a program point is $\mathcal{P}$-space-hard [Lan92]. We present an algorithm for the Conditional May Alias problem, which can be used to safely approximate Interprocedural May Alias in the presence of pointers. This algorithm is as precise as possible in the worst case and has been implemented in a prototype analysis tool for C programs. Preliminary speed and precision results are presented.

1 Introduction

Programming language environments feature software tools that improve the quality, efficiency, understandability, and reusability of code. Optimizers, debuggers, testers and parallelizers use *data flow analysis* to statically extract semantic information from programs to increase their efficacy. Aliases represent important semantic information whose precision can greatly affect the quality of optimized code and the precision of various compile-time interprocedural analyses [Cal88, CK89, PRL91].

An *alias* occurs at some program point during program execution when two or more names exist for the same location. The aliases of a particular name at a program point t are all other names that refer to the same memory location on some path to t. When this execution path traverses more than one procedure, we are solving the Interprocedural May Alias Problem.

While the calculation of aliases for FORTRAN is well understood [Ban79, Coo85, CK89, Mye81], aliasing in C is different than aliasing in Fortran in two respects. First, aliases can change due to side effects of intraprocedural execution flow. Second, aliases created during execution of a called procedure can affect aliases which hold on return to the calling procedure. Arbitrary pointers cause the problem of computing aliases to become $\mathcal{P}$-space-hard; currently, there are no good approximation algorithms.

In this paper, we present an approximation algorithm for interprocedural pointer-induced aliasing based upon Conditional May Alias information that describes aliasing within a procedure assuming certain conditions hold at its entry. We report data on algorithm performance and accuracy on real C programs. We define a appropriate precision measure and show that our algorithm is as precise as possible in the worst case.

2 Related Work

Weihl devised an approximation algorithm for finding aliases in the presence of pointers [Wei80], which unfortunately was very imprecise. Our attempts to use his

*The research reported here was supported, in part, by Siemens Research Corporation and NSF grant CCR-8920078.

ACM SIGPLAN '92 PLDI-6/92/CA
© 1992 ACM 0-89791-476-7/92/0006/0235...$1.50

algorithm for interprocedural analysis of C programs in ISMM [Ryd89] were unsuccessful because of the degree of imprecision in the reported aliases. Chow and Rudmik [CR82] also presented an algorithm for finding aliases in the presence of pointers; however, their algorithm traced non-executable interprocedural execution paths and handled local variables incorrectly. Coutant [Cou86] extended Weihl's work by keeping his restriction of finding program aliases, but allowing the tracing of aliases through more than one level of dereference and adding additional language constructs (e.g., structures and arrays). Benjamin Cooper [Coo89] developed an algorithm which used *alias histories* to insure that a procedure returns to the call site that invoked it.

There also has been some work [Deu90, Deu92, NPD87] in detecting aliases in higher order programming languages. [NPD87] only considers programs with single level dereferences and has the added difficulty of tracking the binding of functions to names. The problem addressed by [Deu90] is an order of magnitude complication over general aliasing; he allows closures (partially evaluated functions) and continuations (storing of runtime environment for later reuse). In [Deu92], an algorithm for finding aliases in polymorphically typed programs is presented.

The work done on dependence analysis and conflict detection in programs with recursive structures [CWZ90, Gua88, HA90, HN89, HPR89, JM82, LH88] is also related, although it is directed at finding access patterns into structures rather than explicitly finding aliases. A *conflict* [LH88] occurs between two statements when one statement writes a location and the other accesses (reads or writes) the same location (*loc*), thus preventing the possibility of those statements being executed in arbitrary order. A data dependence exists between two statements iff they conflict and there is an execution path from one program point to the other on which *loc* is not written.

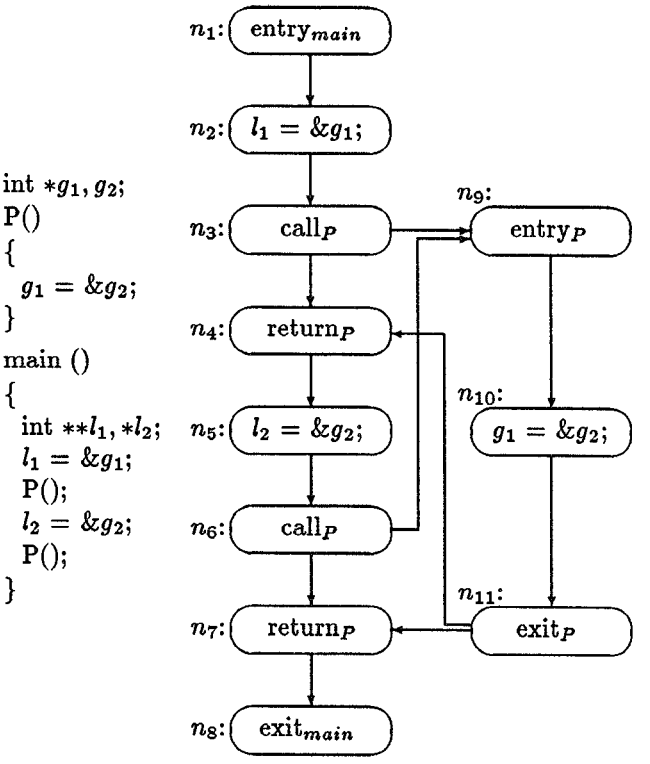


Figure 1: A program and its ICFG

3 Problem Representation

We analyze C-like imperative programming languages with sophisticated pointer usage and data structures, no type casting, explicit function calls (without function variables), and arrays which we treat as aggregates. We represent a program by an *Interprocedural Control Flow Graph* or *ICFG*[LR91] which intuitively is the union of statement-level control flow graphs for each procedure augmented by call, return, entry and exit nodes. Call nodes are connected to the entry nodes of procedures they invoke; exit nodes are connected to return nodes corresponding to these calls. A sample ICFG can be found in Figure 1.

Objects are locations that can store information, and *object names* provide ways to refer to objects. An object name is a variable and a (possibly empty) sequence of dereferences and field accesses. Object names can be defined by three simple BNF rules:

- *object_name* → **object_name*
- *object_name* → *object_name.field_of_structure*
- *object_name* → *variable*

If there are any recursively defined data structures (e.g., linked lists) then the number of object names is potentially infinite. We will limit to some constant, k, the number of dereferences allowable in any object name and obtain a finite number of potential object names. We consider any object name with $l > k$ dereferences to be represented by the object name obtained by ignoring the last $l-k$ dereferences yielding a unique k-limited name. Thus, for $k = 1$, $p\text{->}f_1\text{->}f_2$ would be represented by $p\text{->}f_1$ (and not by $*p$). We borrow terminology from [JM79] and call this *k-limiting*, because their process is analogous even though they *k-limit* dynamic structures while we *k-limit object names.* We need the following functions for object names:

is_prefix(on_1, on_2) returns *true* iff on_1 can be transformed into on_2 by a (possibly empty) sequence of dereferences and field accesses.

apply_trans(on_1, on_2, on_3)**:** $is\_prefix(on_1, on_2)$ must be *true.* The function applies to on_3 the sequence of dereferences and field accesses necessary to transform on_1 into on_2 and returns the result. For example, $apply\_trans(p\text{->}n, p\text{->}n\text{->}d, r)$ returns $r\text{->}d$.

As in [LR91] we will represent *aliases* by unordered pairs of object names (e.g., $\langle v, *p\rangle$). The order is unimportant because the alias relation is symmetric. Since we have k-limited object_names, to safely represent aliases we must assume that $\langle a, b_k\rangle$ with k-limited component b_k, represents not only $\langle a, b_k\rangle$ but also any alias $\langle a, b'_k\rangle$ such that b_k can be transformed into b'_k by a sequence of dereferences and field accesses. Also $\langle a_k, b_k\rangle$ with two k-limited components represents itself and all aliases $\langle a'_k, b'_k\rangle$ such that a_k is a prefix of a'_k and b_k is a prefix of b'_k.

The following definitions will be used throughout the paper:

realizable: A path is **realizable** iff it is a path in the ICFG and whenever a procedure on this path returns, it returns to the call site which invoked it.

holds: Alias $\langle a, b\rangle$ **holds** on the realizable path $\rho n_1 n_2 ... n_i$ iff a and b refer to the same location after execution of program point n_i whenever the execution defined by the path occurs.

Interprocedural May Alias: The precise[1] solution for **Interprocedural May Alias** is $\{[n, \langle a, b\rangle] \mid \exists$ a realizable path, $\rho n_1 n_2 ... n_{i-1} n$, in the ICFG on which $\langle a, b\rangle$ holds$\}$.

visible: At a call site, an object name (e.g., $*x$) of the calling procedure is **visible** in the called procedure iff the called procedure is in the scope of the object name *and* at run time the object name refers to the same object in both the calling and called procedure. (e.g., If x is a local variable of procedure P, then the x in P before a recursive call is not visible after the call, since at execution time it is a different instantiation.)

4 Approximating May Alias

The may-hold Relation We have used our precise algorithm for computing aliases in the presence of single level pointers [LR91] as a basis for a safe interprocedural aliasing algorithm for arbitrary pointers. The key idea in both algorithms is to use **Conditional May Alias** information that answers the question: *If there is a path from program entry to the entry node of the procedure containing n_i on which every alias in the set $\mathcal{AA}$ holds, then may object name a be aliased to object name b on some path to n_i?* Fortunately, it is safe to consider only $\mathcal{AA}$ (sets of aliases) with cardinality less than or equal to one [Lan92]. To insure efficiency, we only concern ourselves with $\mathcal{AA}$ sets that actually occur. We use $may\text{-}hold([(n_i, \mathcal{AA}), \langle a, b\rangle])$ to encode the answer to the Conditional May Alias question:

$may\text{-}hold([(n_i, \mathcal{AA}), \langle a, b\rangle])$ is *true* iff $\langle a, b\rangle$ holds on some path from $entry(n_i)$, the entry of the procedure containing n_i, to n_i assuming there is a path from entry of main to $entry(n_i)$ on which the assumed alias $\mathcal{AA}$ holds **and there is a path from entry of main to the $entry(n_i)$ on which $\mathcal{AA}$ holds.**

[1] We are using the standard data flow definition of precise which means "precise up to symbolic execution", assuming all paths through the program are executable [Bar78].

For even further efficiency gains we have designed our algorithm so that work is performed only when *may-hold*$([(n_i, \mathcal{AA}), \langle a, b\rangle])$ is *true*. Since most of the *may-holds* will be *false*, this improves the average time complexity of our algorithm considerably.

Finally, given Conditional May Alias, May Alias is easily computable;

may-alias(*node*) =

$$\{\mathcal{PA} | (\exists \mathcal{AA}) \textit{may-hold}(node, AA, PA) = true\}.$$

This can be computed in time linear in the size of the *may-hold* solution; thus, we will only concern ourselves with the computation of *may-hold*.

Computing may-hold The algorithm for computing *may-hold* is simple at a high level. First, we find all the *may-hold* relations which are trivially true, (e.g., *may-hold*$([(\text{"}p = \&v\text{"}, \emptyset), (*p, v\rangle])$ is true). Once we have this initial set of *true may-holds*, we compute the set of all *true may-holds* using a worklist algorithm (Figure 2) which propagates alias information from exits of nodes to their immediate successors in the ICFG.

The remainder of this section gives details of this algorithm. We discuss our implementation of *may-hold*, parameter binding, and the functions used in Figure 2. We will use the following macro throughout:

```
make_true(node,AA,PA)
{  if may-hold([(node,AA),PA]) is false
   {  set may-hold([(node,AA),PA]) to true
      add (node,AA,PA) to the worklist
   }
}
```

Representation In order to have an efficient implementation for our alias algorithm, we must be able to do the following operations in constant time:

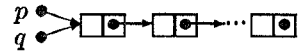
- Set *may-hold*$([(node, \mathcal{AA}), \mathcal{PA}])$ to *false* for all possible *node*, $\mathcal{AA}$, and $\mathcal{PA}$.
- Find the value of *may-hold*$([(node, \mathcal{AA}), \mathcal{PA}])$ for a given *node*, $\mathcal{AA}$, and $\mathcal{PA}$.
- Set the value of *may-hold*$([(node, \mathcal{AA}), \mathcal{PA}])$ for a given *node*, $\mathcal{AA}$, and $\mathcal{PA}$.

We use dynamic hashing [KS86] to do this, giving us constant time operations in the average case. We assume *may-hold* for any $[(node, \mathcal{AA}), \mathcal{PA}]$ triple is *false* if it is not in the hash table; otherwise it is *true* [Lan92].

```
find_aliases()
{ worklist = ∅
  /* Alias Introduction */
  for each node (N) in the ICFG
  { if N is an assignment to a pointer
      aliases_intro_by_assignment(N)
    if N is a call node
      aliases_intro_by_call(N)
  }
  /* Implied Aliases */
  while worklist is not empty
  { remove (N,AA,PA) from worklist
    if N is a call node
      aliases_at_call_implies(N,AA,PA)
    else if N is an exit node
      alias_at_exit_implies(N,AA,PA)
    else any_other_alias_implies(N,AA,PA)
  }
}
```

Figure 2: Computing may-hold

Implicit Assumptions Consider the assignment "$p = q$" where p and q are pointers to linked lists. This assignment results in the following:



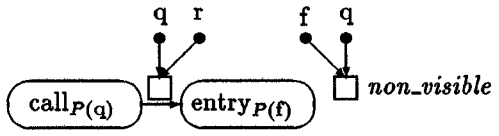
Thus yielding the aliases $\langle *p, *q\rangle$, $\langle p\text{->}next, q\text{->}next\rangle$, $\langle p\text{->}next\text{->}next, q\text{->}next\text{->}next\rangle$, and so on. However, these aliases depend on $*q$, q->$next$, and q->$next$->$next$ being non-$NULL$ before the assignment. Our algorithm implicitly assumes that this is the case, and we would create all of the above aliases that k-limiting allows. This assumption is not an inherent part of our algorithm and can be removed, but we use it here because it is generally reasonable and increases efficiency of the algorithm. For conciseness, throughout this paper, whenever we create an alias $\langle *p, *q\rangle$ we assume that the aliases $\langle p\text{->}next, q\text{->}next\rangle$, $\langle p\text{->}next\text{->}next, q\text{->}next\text{->}next\rangle$, ... are also created although we do not explicitly state this everywhere.

Modeling Parameter Bindings For interprocedural analysis, we need to model the affects of param-

eter bindings on aliases. We do this with a function $bind_{call}(\mathcal{PA})$. Intuitively, $bind_{call}(\emptyset)$ will be all the aliases on entry of a called procedure that must exist because of parameter bindings, while $bind_{call}(\langle a, b\rangle)$ will be the set of aliases at entry of a called procedure whose existence is implied by a being aliased to b at *call*.

Unfortunately, this definition is not sufficient because a procedure call can both create and destroy an alias in the *calling* procedure, involving an object name not visible in the called procedure. For example, the first call of P in Figure 1 creates the alias $\langle **l_1, g_2\rangle$ in main even though l_1 is not in the scope of P. However, only references to the visible object name in an alias pair can affect whether the alias holds on a path. A procedure has the same effect on all alias pairs which contain visible object name w and any non-visible object name. Therefore, we use the object name *non_visible* to represent all non-visible object names[2].

In the *bind* function, if any of the aliases in the bind set involve *non_visible*, we need to know the corresponding object name in the calling procedure. For example, assume q is global to P, r is not visible to P, and q,r, and f are all type "int *":



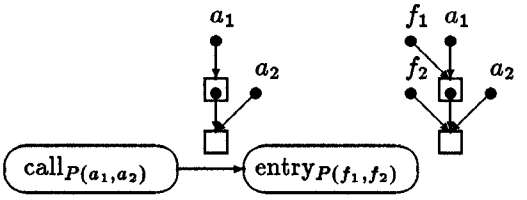
$$bind_{call_{P(q)}}(\emptyset) = \{(*q, *f)\}$$

$$bind_{call_{P(q)}}(\langle *q, *r\rangle) = \left\{ \begin{array}{l} (\langle *f, *q\rangle, \_), \\ (\langle *q, \textit{non\_visible}\rangle, *r), \\ (\langle *f, \textit{non\_visible}\rangle, *r) \end{array} \right\}$$

The occurrence of $(\langle *q, \textit{non\_visible}\rangle, *r)$ in $bind_{call_{P(q)}}(\langle *q, *r\rangle)$ represents the fact that $*q$ is aliased to non-visible object name $*r$ at the entry of the called procedure P.

Computing $bind_{call}(\emptyset)$*:* There are two ways aliases can be implied by parameter bindings. The first alias corresponds to a simple formal to actual pairing. For example, if P is a function with formal f of type "int *" and *call* is an invocation of P with actual a then $(*f, *a)$[3] is in $bind_{call}(\emptyset)$. The second occurs if two distinct formals are passed two actuals where one actual is a prefix of the other. For example, if P is a function with two formals f_1 (type "int **") and f_2 (type "int *") and *call* is P(a,$*a$) then $\langle **f_1, *f_2\rangle$ is in $bind_{call}(\emptyset)$.

Computing $bind_{call}(\langle x, y\rangle)$*:* There are three ways that an alias at a call site may imply an alias on entry to a procedure. The first is trivial: if the two object names are global to the called procedure then they are also aliased on entry to the called procedure. The other two can be illustrated by the following example (both a_1 and a_2 are global to P):



In this example, since $*a_2$ is aliased to $**a_1$ at call$_{P(a_1,a_2)}$, $*f_2$ is aliased to $**a_1$. This example can be generalized to the second way an alias at a call site can imply an alias at the entry of a procedure. Whenever an actual has an alias to an object name, its corresponding formal picks up an alias to that object name or *non_visible*, if the object name is not visible in the called procedure. Also in the example, since $*a_2$ is aliased to $**a_1$ at call$_{P(a_1,a_2)}$, $*f_2$ is aliased to $**f_1$ at entry$_{P(f_1,f_2)}$. This is typical of the third case; when two actuals are aliased at a call site, the corresponding formals are aliased on entry to the called procedure. The algorithm for computing $bind_{call}(\langle x, y\rangle)$ is a straightforward encoding of these three cases.

The remainder of this section gives a more detailed description of the algorithm in Figure 2 by discussing which *may-holds* are set to *true* by each its functions.

[2] In [LR91] we referred to *non_visible* as ".".

[3] $(\langle *f, \textit{non\_visible}\rangle, *a)$ if a is not visible in P.

4.1 Aliases_intro_by_assignment(*node*)

Let *node* be the pointer assignment "$p = q$".[4] Clearly, *may-hold*($[(node, \emptyset), \langle *p, *q \rangle]$) is *true* unless p is a prefix of q. To understand this exception, consider the assignment "$p = p{->}next$". It does not create an alias $\langle *p, *(p{->}next) \rangle$ because p and $p{->}next$ both refer to different objects after this assignment but their alias relationship does not change.

4.2 Aliases_intro_by_call(*node*)

For each alias $\langle a, b \rangle$ in $bind_{call}(\emptyset)$, `make_true(`*entry*, $\langle a, b \rangle, \langle a, b \rangle$`)` where *entry* is the entry node corresponding to *call*.

4.3 Alias_at_call_implies(*call*,$\mathcal{AA}$,$\mathcal{PA}$)

A *may-hold* at a call node has effects on the corresponding entry and return nodes.

Effects on corresponding entry node (entry): For each alias $\langle a, b \rangle$ in $bind_{call}(\mathcal{PA})$, `make_true(`*entry*, $\langle a, b \rangle, \langle a, b \rangle$`)`.

Effects on corresponding return node (return): For simplicity assume that we are dealing with a programming language that has no local variables, and thus no formal parameters. We are interested in the relationship between *may-hold* at a return node and *may-hold* at its predecessors. Clearly, *may-hold* ($[(return, \mathcal{AA}), \langle a, b \rangle]$) is *true* if $\langle a, b \rangle$ holds at the corresponding *exit* node, conditional on $\mathcal{AA}'$ holding at its *entry* **and** $\mathcal{AA}'$, conditional on $\mathcal{AA}$, holds at the corresponding *call* node.

This situation can be generalized resulting in the equation in Figure 3. We have to introduce the functions *back-bind* and *back-bind'* which have the following definitions:[5]

back-bind$_{call}(\langle a, b \rangle)$ specifies the alias on any path to *call* that guarantees a is aliased to b after control flows to corresponding entry node.

back-bind'$_{call}(\langle a, non\_visible \rangle, o)$ specifies the alias on any path to *call* that guarantees a is aliased to the non-visible object name o after control flows to corresponding entry node.

These definitions imply:

back-bind$_{call}(\langle a, b \rangle) = \langle c, d \rangle$ iff $\langle a, b \rangle \in bind_{call}(\langle c, d \rangle)$

back-bind'$_{call}(\langle a, non\_visible \rangle, o) = \langle c, d \rangle$ iff

$\langle a, non\_visible \rangle \in bind_{call}(\langle c, d \rangle)$

and *non_visible* represents o.

We can now proceed to describe the effects of *may-hold* ($[(call, \mathcal{AA}), \mathcal{PA}]$) at the corresponding return. To do a case analysis, let $\mathcal{PA} = \langle a, b \rangle$:

1. If a and b are both not visible in the called procedure then the procedure invocation does not affect this alias. The desired action is obviously `make_true(`*return*$, \mathcal{AA}, \langle a, b \rangle$`)`.

2. a and b are both visible in the called procedure. We know *may-hold*($[(call, \mathcal{AA}), \mathcal{PA}]$) ($\mathcal{PA} \equiv$ *back-bind*$_{call}(\mathcal{AA}')$) and we consider its relationship to *Rule 2*. From *call* we can get *exit* and *return* of *Rule 2*. From $\mathcal{PA}$ we can get $\mathcal{AA}'$.[6] This leaves *Rule 2* with one free variable $\langle x, y \rangle$, so the obvious action for *may-hold*($[(call, \mathcal{AA}), \langle a, b \rangle]$) would be[7]:

```
For each (AA', _) in bind_call(⟨a, b⟩)
  for every possible ⟨x, y⟩:
    if may-hold([(exit, AA'), ⟨x, y⟩]) is true
      make_true(return, AA, ⟨x, y⟩)
```

However, this is not acceptable because it requires work to be done for every possible $\langle x, y \rangle$ even though most $\langle x, y \rangle$ are not necessary. Since in *Rule 2* we are performing a conjunction in which we know one half is true, instead of doing work for all $\langle x, y \rangle$, we would prefer to only do work for $\langle x, y \rangle$ such that *may-hold*($[(exit, \mathcal{AA}'), \langle x, y \rangle]$) is *true*. This can be done at the cost of maintaining an additional data structure[Lan92].

[4] "$p = \&x$" is handled similarly. Simply consider $q \equiv \&x$ and treat $*\&x \equiv x$.

[5] These functions recover the $\mathcal{PA}$ aliases from call-sites.

[6] $bind_{call}(\text{back-}bind_{call}(\mathcal{AA}')) = \mathcal{AA}'$.

[7] Strictly speaking this means if *may-hold*($[(exit, \emptyset), \langle x, y \rangle]$) was *true*, we make *may-hold*($[(return, \mathcal{AA}), \langle x, y \rangle]$) *true* for all possible $\mathcal{AA}$. However, in practice it is sufficient to only make *may-hold*($[(return, \emptyset), \langle x, y \rangle]$) *true*.

Rule 1 If x and y are both not visible in the called procedure:

$$may\text{-}hold([(return, \mathcal{AA}), \langle x, y\rangle]) = may\text{-}hold([(call, \mathcal{AA}), \langle x, y\rangle])$$

Rule 2 If x and y are both visible in the called procedure:

$$may\text{-}hold([(return, \mathcal{AA}), \langle x, y\rangle]) = may\text{-}hold([(exit, \emptyset), \langle x, y\rangle]) \vee \bigvee_{AA' \in ASSUMED^{\dagger}} \begin{pmatrix} may\text{-}hold([(exit, AA'), \langle x, y\rangle]) \wedge \\ may\text{-}hold([(call, \mathcal{AA}), back\text{-}bind_{call}(AA')]) \end{pmatrix}$$

Rule 3 If x is visible but y is not (the symmetric case is similar):

$$may\text{-}hold([(return, \mathcal{AA}), \langle x, y\rangle]) = \bigvee_{\langle o, non\_visible\rangle \in ASSUMED^{\dagger}} \begin{pmatrix} may\text{-}hold([(exit, \langle o, non\_visible\rangle), \langle x, non\_visible\rangle]) \wedge \\ may\text{-}hold([(call, \mathcal{AA}), back\text{-}bind'_{call}(\langle o, non\_visible\rangle, y)]) \end{pmatrix}$$

† $\mathcal{ASSUMED}$ is the set of all possible assumed aliases.

Figure 3: *may-hold* relation at return nodes

3. Assume a is not visible in the called procedure but b is.

   This corresponds to *Rule 3* in Figure 3 and is analogous to the case where a and b are both visible in the called procedure, except now we need to fill in the *non_visible* at *exit* with a. Thus we would get the following action for $may\text{-}hold([(call, \mathcal{AA}), \langle a, b\rangle])$:

   ```
   For each (AA', nv) in bind_call(⟨a,b⟩),
   for every possible ⟨x,y⟩:
   (assume x contains non_visible[8]):
      if may-hold([(exit, AA'), ⟨x,y⟩]) is true
      {  let x' = nv
         apply_trans(non_visible,x,x')
         make_true(return,AA,⟨x',y⟩ }
   ```

More Complex Effects on Return Nodes in Case 3: While the action described above is sufficient if only single level pointers are allowed, it is not sufficient in the general case. In general, it is possible to have an alias between two non_visible's. For example, the second call of P in Figure 1 creates the alias $\langle **l_1, *l_2\rangle$ at n_{10} even though neither l_1 nor l_2 are visible in P. Thus we must handle the case of creation in the called procedure of an alias between two non-visible object names. We will do this with a special case of *may-hold* with two assumed aliases:

$$may\text{-}hold([(exit, \begin{matrix}\langle o_1, non\_visible\rangle \\ \langle o_2, non\_visible\rangle\end{matrix}), \langle nv_1, nv_2\rangle])$$

This represents the fact that if o_1 is aliased to non-visible object name l_1 and o_2 is aliased to non-visible object name l_2 on a path to the entry of the procedure then, on some path to *exit*, nv_1 (where the non_visible portion represents l_1) is aliased to nv_2 (where the non_visible portion represents l_2). Thus,

$$may\text{-}hold([(n_{11}, \begin{matrix}\langle g_1, *non\_visible\rangle \\ \langle g_2, *non\_visible\rangle\end{matrix}), \langle *non\_visible, non\_visible\rangle])$$

represents the alias $\langle **l_1, *l_2\rangle$ at n_{11} because

- g_1 is aliased to $*non\_visible$ ($non\_visible \equiv l_1$) at n_9 when called from n_6 and
- g_2 is also aliased to $*non\_visible$ ($non\_visible \equiv l_2$) at n_9 when called from n_6.

The action for Case 3 given above must be modified to handle the special case where *may-holds* $([(call, \mathcal{AA}), \mathcal{PA}])$ implies one of the two assumed aliases needed for an alias with two non_visibles. Details can be found in [Lan92].

4.4 Alias_at_exit_implies($exit$,$\mathcal{AA}$,$\mathcal{PA}$)

An exit node can have any number of successors, however they are all return nodes. This function encodes the rules for return nodes in Figure 3 with the additional case of aliases between two non-visible object names. The encoding is analogous to that for `Alias_at_call_implies` and we omit a formal description of this routine.

[8] For example, $x = *non\_visible$.

4.5 Any_other_alias_implies($node$,$\mathcal{AA}$,$\mathcal{PA}$)

The implications of *may-hold*($[(node, \mathcal{AA}), \mathcal{PA}]$) depend on its successors and must be considered separately for each successor. Since we have examined the cases where *node* is a call or exit node, the successors must be either a call, an exit, or a statement in the program.

Successor (succ) is a call node, exit node, or a program statement which is not an assignment to a pointer: These nodes simply collect *may-hold* information from their parents. When *succ* is of one of these types, the action for *may-hold*($[(node, \mathcal{AA}), \mathcal{PA}]$) is simply `make_true`($succ$,$\mathcal{AA}$,$\mathcal{PA}$).

Successor (succ) is an assignment to a pointer: This case encompasses the major intraprocedural affects of pointers on aliasing. The effects of *may-hold* ($[(node, \mathcal{AA}), \mathcal{PA}]$) depend on the relationship of the object names in $\mathcal{PA}$ and the object names involved in the pointer assignment. In the following discussion we will consider *succ* to the be statement "$p = q$", where p and q are arbitrary object names of pointer type. What follows is a case analysis; the algorithm applies all suitable cases. The cases are: 1. Does the assignment preserve the alias? 2. What are the effects of an alias of $*q$? 3. What are the effects of an alias of p?

1. $\mathcal{PA} = \langle y, z \rangle$ where p is a prefix of neither y nor z. (preserves the alias)

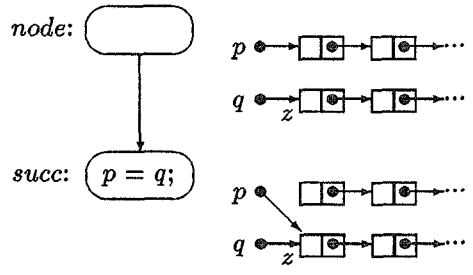
   In all cases, y and z point to the same object after the assignment as before since only p changes its value and the assignment has no effects on $\langle y, z \rangle$. The action in this case is simply `make_true`($succ$, $\mathcal{AA}$,$\mathcal{PA}$). This is clearly safe, but it can also be approximate.

2. $\mathcal{PA} = \langle y, z \rangle$ where **is_prefix_with_deref**(q,y)[9] (effects of an alias of $*q$).

   There are 3 different cases that need to be handled (the first two are mutually exclusive, but either can occur in conjunction with the third). They are: (i) *not* **is_prefix**(p,z), (ii) **is_prefix**(p,z), and (iii) the interaction of $\langle *q, z \rangle$ with other known aliases.

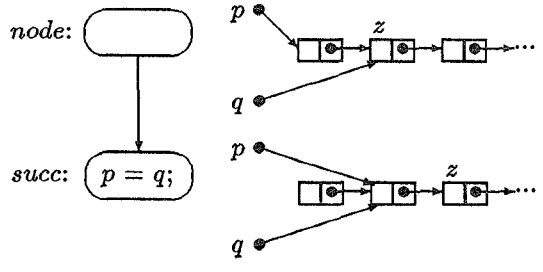
   In general, the effects of the assignment on this alias depend on whether or not **is_prefix**(p,z) is true. The two types of effects are characterized by the following examples:

   Case 2.i: $y \equiv *q$



   In case 2.i, *may-hold*($[(node, \mathcal{AA}), \langle *q, z \rangle]$) implies *may-hold*($[(succ, \mathcal{AA}), \langle *q, z \rangle]$) and *may-hold*($[(succ, \mathcal{AA}), \langle *p, z \rangle]$).

   Case 2.ii: $y \equiv *q$, $z \equiv *(p\text{->}next)$



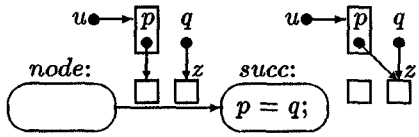
   In case 2.ii, *may-hold*($[(node, \mathcal{AA}), \langle *q, *(p\text{->}next) \rangle]$) gives no information about the aliasing that occurs at *succ*[10]. Thus the action when *may-hold*($[(node, \mathcal{AA}), \langle y, z \rangle]$) is *true* for $succ$ = "$p = q$" where is_prefix_with_deref(q,y) would appear to be:

```
if !is_prefix(p,z)
{ apply_trans(q,y,p')
  make_true(succ,AA,⟨p',z⟩) }
```

[9] **is_prefix_with_deref**(q,y) is the same as **is_prefix**(q,y) except that y must have at least one more deference than q.

[10] The only alias that holds at *succ* in case 2.ii, is $\langle *p, *q \rangle$ which holds regardless of the alias situation at *node*.

This can miss some aliases when p is not a prefix of z. Consider the following case 2.iii:

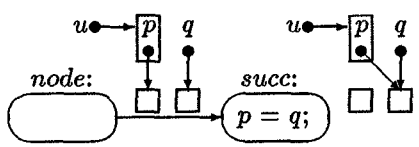


The problem is that the existence of the alias $\langle **u, z\rangle$ at *succ* does not necessarily follow from a single alias at *node*. Instead, $\langle **u, z\rangle$ can hold on $[entry_{main}]...[node]$ $[succ]$ if both $\langle *u, p\rangle$ and $\langle z, *q\rangle$ hold on the path $[entry_{main}]...[node]$. Unfortunately, we do not keep any information about pairs of aliases holding on the same path. Whenever we have *may-hold*$([(node, \mathcal{AA}), \langle *u, p\rangle])$ and *may-hold*$([(node, \mathcal{AA}'), \langle z, *q\rangle])$ we have to assume *may-hold*$([(\text{"}p = q\text{"}, \mathcal{AA}), \langle **u, z\rangle])$ in order for our solution to be safe. We must extend the above action to account for this situation. One notable feature of this extension which deserves mention is how to handle the case when $\mathcal{AA} \neq \mathcal{AA}'$ when both occur on the same path. We do not allow multiple assumptions, so we must safely approximate this situation. Clearly both assumptions are individually necessary and either can be safely chosen. In general, if one assumption contains *non_visible*, then use that one (so that we remember how to instantiate *non_visible*); otherwise use either.

3. $\mathcal{PA} = \langle p, v\rangle$ (effects of an alias of p)

   Again there are three cases to consider; (i) simple effects of $\langle p, v\rangle$, (ii) secondary effects of $\langle p, v\rangle$, and (iii) interaction of $\langle p, v\rangle$ with other known aliases.

   The effects of "$p = q$" on $\langle p, *u\rangle$ (i.e., $v \equiv *u$) are characterized by:



   Case 3.i: *may-hold*$([(node, \mathcal{AA}), \langle p, *u\rangle])$ implies *may-hold*$([(succ, \mathcal{AA}), \langle p, *u\rangle])$ and, unless u or p is a prefix of q[11], *may-hold*$([(succ, \mathcal{AA}), \langle *q, **u\rangle])$.

   Case 3.ii: An alias $\langle *p, o\rangle$ at *node* is, in general, implicitly killed. However, in the case where $\langle p, *u\rangle$ holds on some path to *node*, $\langle *p, **u\rangle$ will not be killed by the assignment "$p = q$" and we have to account for this.

   Case 3.iii: The only other effect of *may-hold*$([(node, \mathcal{AA}), \langle p, *u\rangle])$ comes from handling the other half of case 2.iii and is handled in the same way.

5 Empirical Results

This section starts with a theoretical examination of the worst case precision of our algorithm. We next discuss our implementation of the algorithm and empirically compare our solution to Weihl's solution. We then report empirical data on algorithm precision.

Precision Throughout this paper we have used k-limiting to deal with infinite sets of object names, and we now apply that notion to precision.

$$limit_k(solution) = \left\{ (node, \langle a', b'\rangle) \;\middle|\; \begin{array}{l} (node, \langle a, b\rangle) \in solution, a' \text{ is} \\ \text{the k-limited representation} \\ \text{for } a, \text{ and } b' \text{ is the k-limited} \\ \text{representation for } b \end{array} \right\}$$

We use the following definition for the precision of a safe algorithm A when analyzing program P:

$$precision_k(A, P) = \frac{|limit_k(\{A\text{'s solution for } P\})|}{|limit_k(\{\text{precise solution for } P\})|}$$

Consider the program *all-or-none* in Figure 4 for any given n. This program has the unfortunate property that if no aliases hold before it is executed then the precise solution under the common assumptions of static

[11] when u or p is a prefix of q, we do not want to create $\langle *u, *q\rangle$ for the same reason we do not want to create $\langle *p, *(p{-}{>}next)\rangle$ for "$p = p{-}{>}next$".

```
while (-)
{ #for all k, 1 ≤ k ≤ n:
    if (-)
    { v_k = b;
      b = NULL;
    }
  #end for all
  if (-)
  { b = d;
    d = NULL;
  }
}
```

Figure 4: Program *all-or-none(n)*; n is a parameter determining the size of the program

analysis has $n + 1$ program point aliases. However, if *just* the alias $\langle *b, *d \rangle$ holds before program *all-or-none* then, for all i and j $(1 \leq i, j \leq n)$, $*v_i$ is aliased to $*v_j$ on some path to all program points ($\mathcal{O}(n^3)$ of them). Any approximate algorithm can erroneously produce the alias $\langle *b, *d \rangle$. Thus when *all-or-none(n)* follows the creation of the erroneous $\langle *b, *d \rangle$ alias, the algorithm will report $\mathcal{O}(n^3)$ aliases[12] even though there are only $\Omega(n)$ aliases. For our algorithm, using the the precision measurement $precision_k$ this is the worst case.

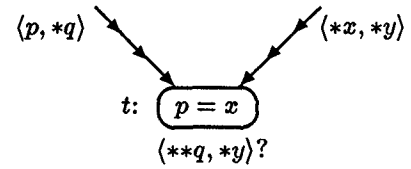
Prototype Our prototype implementation, written in C, finds aliases for a reduced version of C that excludes: union types, nested structures, casting, pointers to functions, and exception handling. The first three of these omissions are not theoretically difficult to handle, but complicate the implementation. The other two require more theoretical examination. We do allow arrays and pointer arithmetic; however, we deal with these on a very simple and naive level and treat them as aggregates. For building ICFGs, we were fortunate to have access to *ptt*, a program developed by Siemens Research Corporation.

Empirical Comparison to Weihl's Algorithm Unfortunately, we can not directly compare our alias solution to Weihl's solution because we find program-point specific aliases and Weihl does not. Therefore let us define *program-aliases* as:

$$\{\langle a, b \rangle \,|(\exists n)\text{a ICFG node and } \langle a, b \rangle \in \mathit{may\text{-}alias}(n)\}$$

In Table 1 we compare our solution versus Weihl's solution. As expected Weihl's algorithm reports more program aliases than our algorithm. The timing for Weihl's algorithm is approximate because in our implementation of his algorithm, we were only able to time the second stage of his calculation (the transitive closure part) and not the total time taken. On average Weihl reported 30.7 times as many aliases and the timings are more or less comparable. The numbers reported here are based on an improved implementation of our algorithm. *make* was included in the original data set, but the new implementation does not yet handle it.

Measurement of Empirical Precision There are four distinct approximations in our alias algorithm (this is proved in [Lan92]). Our first source of approximation, k-limiting, is discussed in Section 2. A second source of approximation is illustrated by the following scenario. Suppose there is an assignment $p = x$ at program point t, alias pair $\langle p, *q \rangle$ holds on some path[13] to an immediate predecessor of t and $\langle *x, *y \rangle$ also holds on some path to an immediate predecessor of t. Does $\langle **q, *y \rangle$ hold on some path to t?



If both $\langle p, *q \rangle$ and $\langle *x, *y \rangle$ occur on the **same** path, then $\langle **q, *y \rangle$ holds on that path extended by t; our algorithm safely concludes this, even though it may not be true.

[12] the $\mathcal{O}(n^3)$ from *all-or-none(n)* plus some constant number form the program that erroneously generates $\langle *b, *d \rangle$

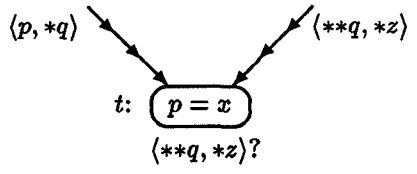
[13] Remember that *may-hold* is defined after execution of the last statement on the path.

Program	Lines	Weihl		*program-aliases*		Weihl / *program-aliases*	*program-aliases* / Weihl
		Number	Time	Number	Time	Number	Time
ul	523	4,851	3s	349	26s	13.8	8.7
pokerd	1,354	62,225	84s	352	4s	176.7	0.1
compress	1,488	6,316	4s	341	2s	18.5	0.5
loader	1,522	39,059	36s	496	7s	78.7	0.2
learn	1,642	61,845	46s	883	27s	70.0	0.6
ed	1,772	1,796	6s	1,455	42s	1.2	1.4
diff	1,793	44,366	58s	1,444	43s	30.4	0.7
tbl	2,545	4,401	10s	1,065	85s	4.1	8.5
lex	3,315	9,490	18s	1,240	50s	7.6	2.7

$program\text{-}aliases = \{\langle a, b\rangle \mid (\exists n) n$ is a node of the ICFG and $\langle a, b\rangle \in may\text{-}alias(n)\}$

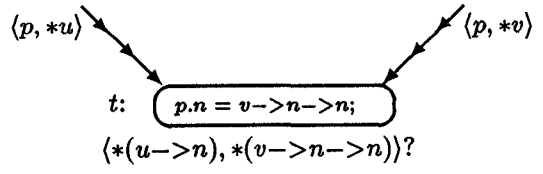
Table 1: Comparison to Weihl

The third source of approximation is similar to the second. Consider the assignment $p = x$ at program point t. Suppose $\langle p, *q\rangle$ holds on some path to an immediate predecessor of t and $\langle **q, *z\rangle$ holds on some path to an immediate predecessor of t. Does $\langle **q, *z\rangle$ hold on some path to t?

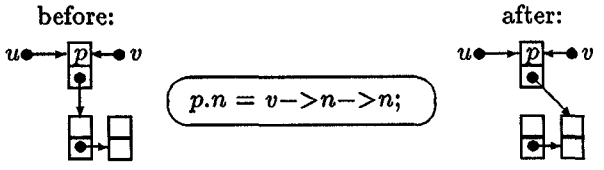


If on at least one path to an immediate predecessor of t $\langle **q, *z\rangle$ holds, and neither $\langle p, *q\rangle$ nor $\langle p, z\rangle$ does, then $\langle **q, *z\rangle$ holds on that path extended by t. However, if on all those paths $\langle **q, *z\rangle$ and $\langle p, *q\rangle$ both hold, then $\langle **q, *z\rangle$ does not necessarily hold on any path to t. Here, our algorithm safely assumes that $\langle **q, *z\rangle$ holds on some path to t.

The fourth approximation results from two distinct aliases of the LHS of an assignment.



Normally, $\langle *(u{->}n), *(v{->}n{->}n)\rangle$ should hold on a path to t because assigning $v{->}n{->}n$ to $p.n$ is also an assignment to $u{->}n$ on paths on which $\langle p, *u\rangle$ holds. This, however, is not necessarily the case. If, for example, on the same path $\langle p, *v\rangle$ holds then $\langle *(u{->}n), *(v{->}n{->}n)\rangle$ does not necessarily hold:

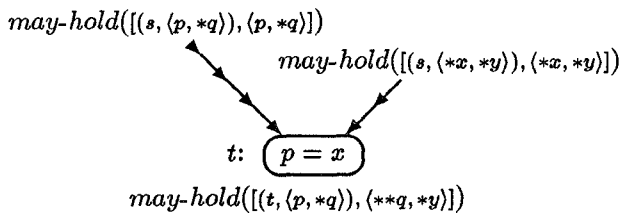


We've modified our algorithm to count the last three approximations. An alias is counted as possibly imprecise if it is the result of any of the last three types of the approximations or depends on the existence of another alias that is counted as possibly imprecise. In [Lan92], we prove that the four cases given above are the only *base* sources of imprecision in a variant our algorithm.[14]

There is one other source of approximation that is not a base source of approximation. Consider the fol-

[14]The algorithm presented here has the implicit assumption that the RHS of an assignment is not NULL on some path to the assignment, which can lead to imprecision. We consider this assumption reasonable, and have not counted this imprecision.

lowing (where s is some immediate predecessor of t):



This is the second source of approximation mentioned above. Note that since *may-hold* at t depends on two assumptions we arbitrary chose one. Assume that the *may-hold* at t reaches the exit of the procedure so that we get *may-hold*$([(exit, \langle p, *q\rangle), \langle **q, *y\rangle])$. It is possible possible that the alias $\langle **q, *y\rangle$ does indeed hold on some path to t (and some path to *exit*) and hence is not imprecise. However, we will propagate $\langle **q, *y\rangle$ to a *return* if only the alias $\langle p, *q\rangle$ holds at the corresponding *call*. This can be inaccurate even though the original *may-hold* at t is accurate. Notice, however, that any such inaccuracies will be counted, because the *may-hold* at t will be counted as possibly inaccurate and since the *may-hold* at exit depends (indirectly) on *may-hold* at t, it too will be counted as possibly inaccurate.

Define $\%YES_k(P)$ for *may-hold* computed from P as in Figure 5. Given that there are only the aforementioned four types of approximations, we claim that $L \subseteq limit_k$(precise solution). Thus $\%YES_k(P) \leq 100 * (1/precision_k(landi, P))$ and can be used to bound precision.[15]

Table 2 presents empirical precision results for 18 C programs with the k-limit constant equal to three[16]. This suite contains all the programs that we compared to Weihl's algorithm in Section 5. These programs came from a pool of available C programs that was collected for an ongoing empirical study of the structure of C programs [RP88]. The sample is by no means large enough to draw general conclusions about algorithm behavior, but it is large enough to indicate that our algorithm performs well over a limited domain of C programs.

[15] *landi* refers to our approximate algorithm.

[16] In [Lan92] we've also examined these programs for k= 1 to 4.

6 Conclusions

Our development of an approximate algorithm for solving for Interprocedural May Alias has been promising. It justified the use of Conditional May Alias as a method for the interprocedural aspects of the alias problem. While it is not precise in the presence of arbitrary pointers, it is safe, erring conservatively. We showed that for at least one definition of precision, in the worst case no algorithm can be more precise than our algorithm. Our empirical investigations yielded encouraging precision results and showed great improvement over the extant technique [Wei80]. Our idea of using a conditional version of aliasing to solve for aliases has been extended to the Interprocedural Reaching Definitions Problem in C [PRL91].

Acknowledgments We thank our colleagues at Siemens Corporate Research, Hemant Pande, Michael Platoff, and Michael Wagner, for their assistance with *ptt*. We also thank Siemens for allowing us to use their optimized implementation of our algorithm. The timings for Siemens implementation are on average 13 times faster than our initial ones. Rita Altucher, Bruce Ladendrof, and William Landi[17] are primarily responsible for the improvements.

References

[Ban79] J. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 29–41, January 1979.

[Bar78] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, 1978.

[Cal88] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 47–56, June 1988.

[17] while working for Siemens

$$\%YES_k(P) = 100 * \left(\frac{\left| L = \left\{ (node, \mathcal{PA}) \;\middle|\; \begin{array}{l}(\exists \mathcal{AA}) may\text{-}hold([(node, \mathcal{AA}), \mathcal{PA}]) = \text{YES} \\ \text{and not directly or indirectly the result} \\ \text{of a type 2, 3 or 4 approximation}\end{array} \right\} \right|}{\left| \{ (node, \mathcal{PA}) \mid (\exists \mathcal{AA}) may\text{-}hold([(node, \mathcal{AA}), \mathcal{PA}]) = \text{YES} \} \right|} \right)$$

Figure 5: Definition of $\%YES_k(P)$

Program	ICFG Nodes	May Aliases	$\%YES_k$ $(Program)$	Time
allroots	407	257	100†	1s
fixoutput	615	1,937	100†	1s
diffh	647	8,046	100†	1s
poker	896	3330	100†	2s
ul	1,625	101,273	100†	26s
lex315	1,204	5,163	100†	2s
loader	1,596	119,259	78†	24s
compress	1,914	8,656	67†	2s
tp	1,710	96,098	100†	9s

Program	ICFG Nodes	May Aliases	$\%YES_k$ $(Program)$	Time
pokerd	1,936	54,819	45	7s
learn	2,781	179,844	98†	27s
ed	3,299	127,502	100†	41s
assembler	3,631	1,260,582	10†	396s
diff	3,926	89,056	88†	40s
simulator	5,305	241,621	98†	31s
football	5,910	232,913	100†	23s
tbl	5,960	400,464	100†	80s
lex	6,792	420,268	96†	44s

†$\%YES_k(Program) \leq 100 * (1/precision_k(landi, Program))$

Table 2: Precision of our May Alias Solution ($k = 3$)

[CK89] K. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–59, January 1989.

[Coo85] K. Cooper. Analyzing aliases of reference formal parameters. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 281–290, January 1985.

[Coo89] B. G. Cooper. Ambitious data flow analysis of procedural programs. Master's thesis, University of Minnesota, May 1989.

[Cou86] D. S. Coutant. Retargetable high-level alias analysis. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 110–118, January 1986.

[CR82] A. Chow and A. Rudmik. The design of a data flow analyzer. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 106–113, June 1982.

[CWZ90] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990. SIGPLAN Notices, Vol 25, No 6.

[Deu90] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 157–168, January 1990.

[Deu92] Alain Deutsch. A storeless mode of aliasing and its abstractions using finite representation of right-regular equivalence relations. In *Proceedings of the IEEE 1992 Conference on Computer Languages*, April 1992.

[Gua88] C. A. Guarna. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proceedings of the International Conference on Parallel Processing*, pages 212–220, 1988.

[HA90] W.L. Harrison III and Z. Ammarguellat. Parcel and miprac: Parallelizers for symbolic and numeric programs. In *International Workshop on Compilers for Parallel Computers*, December 1990.

[HN89] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 49–56, August 1989.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 28–40, June 1989.

[JM79] N. Jones and S. Muchnick. Flow analysis and optimization of lisp-like structures. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice Hall, 1979.

[JM82] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 66–74, January 1982.

[KS86] H. Korth and A. Silberschatz. *Database System Concepts*. McGraw-Hill, New York, NY, 1986.

[Lan92] W. Landi. *Interprocedural Aliasing in the Presence of Pointers*. PhD thesis, Rutgers University, January 1992. LCSR-TR-174.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the*

SIGPLAN '88 Conference on Programming Language Design and Implementation, pages 21–34, July 1988. SIGPLAN NOTICES, Vol. 23, No. 7.

[LR91] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, January 1991.

[Mye81] E. M. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 219–230, January 1981.

[NPD87] A. Neirynck, P. Panangaden, and A. Demers. Computation of aliases and support sets. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 274–283, January 1987.

[PRL91] H.D. Pande, B. G. Ryder, and W. Landi. Interprocedural def-use associations in c programs. In *Proceedings of the Fifth Testing, Analysis, and Verification Symposium*, October 1991.

[RP88] B. G. Ryder and H. Pande. The interprocedural structure of c programs: An empirical study. Laboratory for Computer Science Research Technical Report LCSR-TR-99, Department of Computer Science, Rutgers University, February 1988.

[Ryd89] B. G. Ryder. Ismm: Incremental software maintenance manager. In *Proceedings of the IEEE Computer Society Conference on Software Maintenance*, pages 142–164, October 1989.

[Wei80] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, January 1980.