Pointer-induced Aliasing: A Problem Classification*

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

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

Abstract

Aliasing occurs at some program point during execution when two or more names exist for the same location. We have isolated various programming language mechanisms which create aliases. We have classified the complexity of the alias problem induced by each mechanism alone and in combination, as $\mathcal{NP}$-hard, complement $\mathcal{NP}$-hard, or polynomial ($\mathcal{P}$). We present our problem classification, give an overview of our proof that finding interprocedural aliases in the presence of single level pointers is in $\mathcal{P}$, and present a representative proof for the $\mathcal{NP}$-hard problems.

1 Introduction

Aliasing occurs at some program point during program execution when two or more names exist for the same location. The aliases of a name at a program point t are all other names that may refer to the same memory location on some execution path to t. While the calculation of aliases for FORTRAN is well understood [1, 5, 6, 18], if general pointers are added as a language construct, the problem of computing aliases becomes $\mathcal{NP}$-hard and no good approximation algorithms exist. Moreover, aliasing complicates most data flow analysis problems, and the absence of alias information can prevent many optimizations.

Our results show which aspects of alias problems are provably hard and need to be approximated. A clear understanding of what makes aliasing hard, lends insight into where information is lost in various approximations and whether that loss of information is justified. Also, an understanding of how easier alias problems can be handled precisely, can be useful as a framework for approximating harder alias problems.

Theoretical Complexity Results In this paper, we present the theoretical complexity of solving the *Intraprocedural May Alias, Intraprocedural Must Alias, Interprocedural May Alias,* and *Interprocedural Must Alias* problems in the presence of several programming language mechanisms that create aliases. The following mechanisms are considered alone and in combination: reference formal parameters, single level pointers, multiple level pointers (i.e., pointers whose dereferenced values are pointers), and pointers to structures containing single level pointers. Informally, our results show that multiple levels of indirection lead to $\mathcal{NP}$-hard or co-$\mathcal{NP}$-hard alias problems, whereas a single level of indirection introduces aliases that can be found in polynomial time.

Comparison to FORTRAN Aliasing In FORTRAN, the only dynamic method for creating aliases is through the use of reference formals. Solving for aliasing in the presence of pointers presents several additional complications. The most obvious difference is that with only reference formals, aliases that hold at invocation of a procedure, hold during the entire execution of the called procedure. However, when pointers are present, this is not the case, because pointer assignments can change aliases in the called procedure. In addition, with only reference formals, a call to a procedure cannot affect the aliases in the calling procedure, but if pointers are present, this is no longer true. Both these facts indicate that existing FORTRAN alias algorithms are not

*The research reported here was supported, in part, by the New Jersey Commission on Science and Technology, the CAIP Center's Industrial Members, NSF grant CCR-8920078, and Siemens Research Corporation.

extensible to handle pointers.

Related Work Weihl devised an approximation algorithm for finding aliases in the presence of pointers [21, 22]. Essentially relational in approach, his algorithm is very imprecise because it ignores control flow. Chow and Rudmik [3] also presented an algorithm for finding aliases in the presence of pointers; however, their algorithm is inadequate, because they treat interprocedural aliasing as an intraprocedural problem. Benjamin Cooper [4] has developed an algorithm which uses explicit path information in the form of *alias histories* to insure (for interprocedural paths) that a procedure returns to the call site that invoked it. This method seems infeasible for an implementation and is reminiscent of the work of Sharir and Pnueli [20].

A related area of research is the work done by the compiling community on conflict detection in recursive structures [2, 8, 9, 11, 15]. A *conflict* occurs between two statements when one statement writes a location and the other accesses (reads or writes) the same location, thus preventing the possibility of those statements being executed in parallel. Aliasing occurs *at a program point*, and conflicts, in general, occur *between two or more program points*. The research on conflict detection has largely been concerned with statically representing and determining the structure of dynamically allocated objects (e.g. trees, dags, lists). It has not addressed the interprocedural complications that result because calls must return to the call site which invoked them; we, conversely, concentrate on these interprocedural issues.

Overview Section 2 states relevant definitions and a summary of our theoretical results. Section 3 presents an overview of the Interprocedural May Alias proof by specifying a polynomial algorithm. Section 4 presents the proof that Intraprocedural May Alias in the presence of multiple level pointers is $\mathcal{NP}$-hard. Section 5 previews future work.

2 Alias Problem Classification

Program Representation We represent programs by *interprocedural control flow graphs*, (ICFGs) that are intuitively, the union of the control flow graphs (CFGs)[1] [7] for each procedure, with calls connected to the procedures they invoke. Formally, an ICFG is a triple ($\mathcal{N}$, $\mathcal{E}$, ρ) where: ρ is the entry node for *main*; $\mathcal{N}$ contains one node for each statement in the program, an *entry* and *exit* node for each procedure, a *call* and *return* node for each call site; and $\mathcal{E}$ contains all edges in the CFG for each procedure, with a slight modification of edges involving call sites. In the ICFG, a call site is split into a *call* and a *return* node. An intraprocedural edge into a call node represents execution flow into a call site, while an intraprocedural edge out of a return node represents flow from a call site. In addition to the intraprocedural edges, two interprocedural edges are added for each call site: one from the call node to the entry node of the invoked procedure, and one from the exit node of the procedure to the return node of the call site. See Figure 1 for an example of an ICFG.

Definitions The following definitions will be used throughout the paper:

realizable: A path is **realizable** iff it is a path in the ICFG (CFG in intraprocedural problems) such that whenever a procedure on this path returns, it returns to the call site which invoked it. Basically, realizable means *potentially executable* under the common assumption of static analysis that all program paths are executable.

holds: Alias $<a,b>$ **holds** on the realizable path $\rho n_1 n_2 ... n_i$ iff a and b refer to the same location at program point n_i whenever the execution sequence defined by the path occurs. Note that aliases are symmetric, thus $<a,b>$ holds on a path iff $<b,a>$ also holds on that path.

Interprocedural May Alias: The precise solution for **Interprocedural May Alias** is

$$\left\{ [n, <a,b>] \;\middle|\; \begin{array}{l} \exists \text{ a realizable path, } \rho n_1 n_2 ... n_{i-1} n, \\ \text{in the ICFG on which } <a,b> \text{ holds} \end{array} \right\}$$

[1]Each node in our CFG is a source code statement.

```
int *q;

void A(f)
  int *f;
{
  q = f;
}

main( )
{
  int *q,*r;

  A(q);
  r = q;
  A(p);
}
```

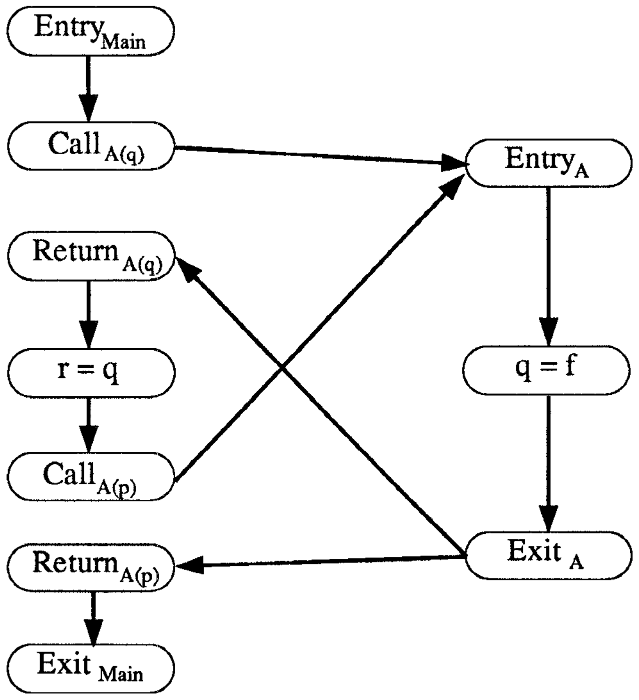

Figure 1: A program and its ICFG

Interprocedural Must Alias: The precise solution for **Interprocedural Must Alias** is

$$\left\{ [n, <a,b>] \;\middle|\; \begin{array}{l}\forall \text{ realizable paths, } \rho n_1 n_2 \ldots n_{i-1} n,\\ \quad \text{in the ICFG } <a,b> \text{ holds}\end{array} \right\}$$

visible: At a call site, an object name (e.g. $*x$) in 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 the called procedure.[2]

Problem Classification We have analyzed the theoretical difficulty of solving for aliases (assuming no procedure variables) in the presence of reference formals, single level pointers, multiple level pointers, and structures containing single level pointers. The results our analyses are shown in Table 1. Blanks in Table 1 correspond to problems which involve reference parameters and thus are inherently interprocedural. Surprisingly, there is no difference in problem difficulty between intraprocedural and interprocedural problems, at least in terms of $\mathcal{NP}$-hard vs $\mathcal{P}$. The salient property is the number of possible levels of indirection, regardless of the mechanism used to create the indirection. If only one level of indirection is possible, then aliasing can be precisely solved in a polynomial amount of time, but as soon as two levels are present, the problem becomes $\mathcal{NP}$-hard.

There must be at least two distinct approximations in any practical aliasing algorithm. In any program that contains recursive data structures, there are potentially an infinite number of objects which can have aliases. Any aliasing algorithm will have to represent all possible objects by a finite (polynomial) number of objects[3]. The type of representation and its precision are what distinguishes the different conflict detection methods.

There is a second source of approximation illustrated by the following scenario. Suppose there is an assignment $p = x$ at program point t, alias pair $<p,q>$ holds on

[2]The notion of visibility is needed to define precisely the aliases of objects in recursive procedures. We are assuming that all program variables have unique names.

[3]For example, *k-limited* as defined by Jones and Muchnick [11].

Alias Mechanism	Intraprocedural May Alias	Intraprocedural Must Alias	Interprocedural May Alias	Interprocedural Must Alias
Reference Formals, No Pointers, No Structures	-	-	Polynomial[1, 5]	Polynomial[1, 5]
Single level pointers, No Reference Formals, No Structures	Polynomial	Polynomial	Polynomial	Polynomial
Single level pointers, Reference Formals, No Pointer Reference Formals, No Structures	-	-	Polynomial	Polynomial
Multiple level pointers, No Reference Formals, No Structures	$\mathcal{NP}$-hard	Complement is $\mathcal{NP}$-hard	$\mathcal{NP}$-hard	Complement is $\mathcal{NP}$-hard
Single level pointers, Pointer Reference Formals, No Structures	-	-	$\mathcal{NP}$-hard	Complement is $\mathcal{NP}$-hard
Single level pointers, Structures, No Reference Formals	$\mathcal{NP}$-hard[14]	Complement is $\mathcal{NP}$-hard	$\mathcal{NP}$-hard[14]	Complement is $\mathcal{NP}$-hard

Table 1: Alias problem decomposition and classification

some path to t and $<*x, *y>$ also holds on some path to t. If both $<p, q>$ and $<*x, *y>$ occur on the **same** path, then $<*q, *y>$ holds at t; therefore, to be safe we must conclude this, even though it may not be true. Thus, to solve for alias pairs precisely, we need information about multiple alias pairs on a path. Unfortunately, this property generalizes; that is, to determine precisely if there is a single path on which a set of i alias pairs hold, you need information about sets of more than i alias pairs. Since it is $\mathcal{NP}$-hard even in the presence of single level pointers to determine if there is an intraprocedural path on which a set of $\mathcal{O}(n)$ (n, the number of variables in a program) aliases hold [13], some approximation must occur.

All the $\mathcal{NP}$-hardness proofs are variations of proofs by Myers [18]; a similar, although independently discovered, proof for recursive structure aliasing (as indicated in Table 1) was developed by Larus [14]. All problems which are categorized as polynomial are corollaries of proofs that the Interprocedural May Alias and Interprocedural Must Alias problems in the presence of single level pointers are polynomially solvable (the proofs for these two problems are, surprisingly, fairly disparate). The key ideas used in the proof that the Interprocedural May Alias problem in the presence of single level pointers is in $\mathcal{P}$ are presented in Section 3. The proof that the Intraprocedural May Alias problem is $\mathcal{NP}$-hard is given in Section 4. This proof is representative of all those for $\mathcal{NP}$-hard problems. Other proofs are omitted but can be found in [13].

3 Interprocedural May Alias with Single Level Pointers

The main difficulty in solving Interprocedural May Alias is to determine how to restrict information propagation only to realizable paths. To accomplish this, we solve data flow problems for a procedure assuming an alias condition on entry; that is, we solve data flow conditionally based on some assumption at procedure entry. This is somewhat reminiscent of Lomet's approach to solving data flow problems under different aliasing conditions [16] and Marlowe's notion of a representative data flow problem within a region[17].

We use a two step algorithm to solve for aliases. In the first step, we solve for *conditional aliases*, that is, we answer the question "If there is a path to the entry node of the procedure containing n_i on which the alias set $\mathcal{A}$ holds, then may a be aliased to b at n_i?". In the second step, we use these *conditional aliases* to solve for the actual aliases.

This two step approach avoids the unrealizable path problem. In the first step, the edges from call nodes to entry nodes in the ICFG are ignored. Information is propagated from procedure entry nodes to exits; the calculation at a return node combines information from its corresponding call node and the called procedure's exit node. Thus, potential alias effects of the called procedure on the calling procedure are incorporated; however, the aliases introduced by the call itself are ignored. In the second step, aliases introduced are propagated through a call node to the corresponding entry node of the called procedure, ignoring edges from exit nodes to return nodes. Conceptually, this is analogous to propagation on the program call graph [7]. Related ideas for handling the unrealizable path problem were presented in [10].

This idea of using conditional aliases does not seem promising at first, as there are an exponential number of possible sets of aliases. But Lemma 3.1 insures that it is sufficient to consider sets $\mathcal{A}$ where $|\mathcal{A}| \leq 1$. If more than one level of indirection is possible, it is no longer precise to use $|\mathcal{A}| \leq 1$, but it is *safe* (i.e., our alias solution will be imprecise, but all actual alias pairs will be contained within the calculated solution).

Lemma 3.1 *If pointer usage is restricted to single level pointers then*

- *for all realizable paths $P = n_1 n_2 ... n_i$ (where n_1 is the entry node of the procedure containing n_i and the number of calls on the path $n_1, n_2, ..., n_i$ equals the number of returns),*
- *and for all possible alias pairs $<a, b>$;*

If
all alias pairs in the set $\mathcal{A} = \{A_1, A_2, ..., A_m\}$ holding at n_1 and the execution of path P implies that $<a, b>$ holds at n_i
then
either assuming no aliases at n_1 and executing path P forces $<a, b>$ to hold at n_i
or
$\exists k (1 \leq k \leq m)$ such that when assuming **only** *the alias A_k at n_1, executing the path P forces $<a, b>$ to hold at n_i.*

The proof of Lemma 3.1 is by induction on $|P|$, the basis is trivially true and the induction step is an easy, but messy, case analysis on possible n_i[12]. □

We use the *holds* relation to represent conditional alias information.

holds([(*ICFG-node*, *assumed-alias-pair*), *alias-pair*])

is *true* iff *alias-pair* holds on some path to *ICFG-node*, assuming there is a path to the entry of the procedure containing *ICFG-node* on which *assumed-alias-pair* holds. By Lemma 3.1, *assumed-alias-pair* is either a single alias pair or $\emptyset$, where $\emptyset$ represents assuming no aliases on procedure entry.

3.1 Computing May Alias using Conditional May Alias

Given the *holds* relation, Interprocedural May Alias information can be computed by a simple data flow problem on the ICFG. We will use the function $bind_{call}$ to model the effects of parameter bindings at each call site. $bind_{call}(\mathcal{A})$ is the set of aliases which hold on the path $\rho n_1 ... n_{i-2}[call][entry_P]$ if $\mathcal{A}$ holds on $\rho n_1 ... n_{i-2}[call]$. $bind_{call}(\mathcal{A})$ is formally defined in Figure 2.

Given *holds* and the *bind* functions, for any node n in the ICFG=$(\mathcal{N}, \mathcal{E}, \rho)$, *may-alias*($n$) can be defined as follows:

- *may-alias*(ρ) = $\emptyset$
- if n is an entry node then *may-alias*(n) =

$$\bigcup_{\ll m,n \gg \in \mathcal{E}} \left(bind_m(may\text{-}alias(m)) \right)$$

- otherwise, *may-alias*(n) =

$$\left\{ <a,b> \;\middle|\; \begin{array}{l} [holds([(n,\emptyset), <a,b>]) = true] \bigvee \\ [\left(\exists <c,d> \in may\text{-}alias(entry(n))\right) \\ \quad holds([(n, <c,d>), <a,b>]) = true] \end{array} \right\}$$

Theorem 3.1 *There exists a polynomial algorithm for determining precise* **Interprocedural May Alias** *sets in the presence of single level pointers.*

$$bind_{call}(\mathcal{A}) = bind'_{call}(\emptyset) \cup \bigcup_{<a,b> \in \mathcal{A}}(bind'_{call}(<a,b>))$$

$$bind'_{call}(\emptyset) = \left(\begin{array}{l} \left\{ <*f_i, *f_j> \;\middle|\; \begin{array}{l} f_i \text{ and } f_j \text{ are pointer formals} \\ \text{with actuals } a_i \text{ and } a_j \text{ respectively, and } a_i = a_j \end{array} \right\} \cup \\ \left\{ <*f_i, *a_i> \;\middle|\; \begin{array}{l} f_i \text{ is a pointer formal with actual } a_i, \\ \text{and } a_i \text{ is visible in the called procedure} \end{array} \right\} \cup \\ \left\{ <*a_i, *f_i> \;\middle|\; \begin{array}{l} f_i \text{ is a pointer formal with actual } a_i, \\ \text{and } a_i \text{ is visible in the called procedure} \end{array} \right\} \end{array} \right)$$

$$bind'_{call}(<a,b>) = \left(\begin{array}{l} \{ <a,b> \mid \text{if } a \text{ and } b \text{ are visible in the called procedure } \} \cup \\ \left\{ <a, *f_i> \;\middle|\; \begin{array}{l} \text{if } a \text{ is visible in the called procedure, } f_i \text{ is} \\ \text{a pointer formal with actual } a_i, \text{ and } *a_i = b \end{array} \right\} \cup \\ \left\{ <*f_i, b> \;\middle|\; \begin{array}{l} \text{if } b \text{ is visible in the called procedure, } f_i \text{ is} \\ \text{a pointer formal with actual } a_i, \text{ and } *a_i = a \end{array} \right\} \cup \\ \left\{ <*f_i, *f_j> \;\middle|\; \begin{array}{l} \text{if } f_i \text{ and } f_j \text{ are pointer formals with corresponding} \\ \text{actuals } a_i \text{ and } a_j, \text{ and } *a_i = a \text{ and } *a_j = b \end{array} \right\} \end{array} \right)$$

Figure 2: Specification of $bind_{call}(\mathcal{A})$

The claim is that calculating the fixed point of *may-alias* is such an algorithm. We can prove that the *holds* calculation can be computed in polynomial time (Lemma 3.2 in Section 3.2). Therefore, the fixed point calculation for Interprocedural May Alias takes polynomial time because, for each node in the ICFG, *may-alias* can change its value at most $\mathcal{O}(v^2)$ times, where v is the number of variables in the program. The precision of our algorithm stems from Lemmas 3.1 and 3.2. The formal proof is by induction on path length and by induction on number of iterations of the fixed point calculation[12]. □

3.2 Computing Conditional May Alias

We handle the intraprocedural aspects of Interprocedural May Alias similarly to Chow and Rudmik [3]. On an execution path, we consider the relationship between aliases that hold before a statement is executed and aliases that hold after it is executed. Considering $holds([(n,\mathcal{AA}),<a,b>])$, the following statements are true:

- if n is not an assignment to a pointer
  $holds([(n,\mathcal{AA}),<a,b>])$ is true iff for some immediate predecessor m of n, $holds([(m,\mathcal{AA}),<a,b>])$ is true.
- if n is "p = q" {or "p = &v"} for p and q pointers
  - if a is $*p$ then $holds([(n,\mathcal{AA}),<*p,b>])$ is true iff for some immediate predecessor m of n, $holds([(m,\mathcal{AA}),<*q,b>])$ {$holds([(m,\mathcal{AA}),<v,b>])$ for "p = &v"} is true.
  - if b is $*p$ then $holds([(n,\mathcal{AA}),<a,*p>])$ is true iff for some immediate predecessor m of n, $holds([(m,\mathcal{AA}),<a,*q>])$ {$holds([(m,\mathcal{AA}),<a,v>])$ for "p = &v"} is true.
  - otherwise,
    $holds([(n,\mathcal{AA}),<a,b>])$ is true iff for some immediate predecessor m of n, $holds([(m,\mathcal{AA}),<a,b>])$ is true.
- if n is "p = malloc()" or "p = null". (We are mallocing a primitive type here, not a structure.)

- if a or b is $*p$ then $holds([(n, \mathcal{AA}), <a,b>])$ must be $false$ ($<a,b>$ does not hold on any path).
- otherwise, $holds([(n, \mathcal{AA}), <a,b>])$ is true iff for some immediate predecessor m of n, $holds([(m, \mathcal{AA}), <a,b>])$ is true.

Recall that the main problem in computing precise Interprocedural May Alias is insuring that only realizable paths are considered. However, this is easily handled by our conditional alias formulation. $holds([(entry, \mathcal{AA}), <a,b>])$ is $true$ if ($\mathcal{AA} = <a,b>$) or ($a = b$) and otherwise is $false$. *call* and *exit* nodes simply collect alias information. Thus:

$$holds([(call/exit, \mathcal{AA}), <a,b>]) = \bigvee_{\ll m, call/exit \gg \in \mathcal{E}} \left(holds([(m, \mathcal{AA}), <a,b>]) \right)$$

Now, 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 whether $holds([(return, assumed\text{-}alias), <a,b>])$ is $true$ (see Figure 3). Clearly it is $true$ if $<a,b>$ holds at the corresponding *exit* node, conditional on *assumed-alias'* holding at its *entry* **and** *assumed-alias'*, conditional on *assumed-alias*, holds at the corresponding *call* node. Let $\mathcal{ASSUMED}$ be the set of all possible assumed aliases in the program. *Holds* for a return node is defined as:

$$holds([(return, assumed\text{-}alias), <a,b>]) = holds([(exit, \emptyset), <a,b>]) \vee \bigvee_{\mathcal{AA} \in \mathcal{ASSUMED}} \begin{pmatrix} holds([(exit, \mathcal{AA}), <a,b>]) \wedge \\ holds([(call, assumed\text{-}alias), \mathcal{AA}]) \end{pmatrix}$$

Factoring in local variables Unfortunately, adding local variables complicates matters considerably. A procedure call can both create and destroy an alias in the *calling* procedure, involving a non-visible object in the called procedure. For example, the second call of `A` in Figure 1 creates the alias $<*q, *p>$ and destroys the alias $<*q, *r>$ at $\mathtt{return}_{A(p)}$ even though $*p$ and $*r$ are not visible in `A`. However, only references to a visible object in an alias pair can affect whether the alias holds on a path (i.e., there can be no direct references to a non-visible object). Fortunately, a procedure has the same effect on all alias pairs which contain visible object w and any non-visible object. Thus for every object w, which may have aliases, we introduce the alias pair $<w, \cdot>$, representing w aliased to a non-visible object, into the set of possible alias pairs and the set of possible assumed aliases.

As in Interprocedural May Alias, we need to be able to model the effects of parameter bindings. However, we now have a different perspective and will use the functions $back\text{-}bind_{call_P}$ and $back\text{-}bind'_{call_P}$ for each call site to model parameter bindings. $back\text{-}bind_{call_P}(assumed\text{-}alias)$ specifies the alias holding on any path $\rho...[call_P]$ that guarantees *assumed-alias* holds on $\rho...[call_P][entry_P]$. $back\text{-}bind'_{call_P}(<a, \cdot>, o)$ specifies the alias holding on any path $\rho...[call_P]$ that guarantees a will be aliased to the non-visible object o on $\rho...[call_P][entry_P]$. *back-bind* and *back-bind'* are formally defined in Figure 4.

Assume that we are interested in whether $<a,b>$ holds on the realizable path P_{return_Q} to node $return_Q$. We will use the following conventions:

$$\begin{aligned} P_{return_Q} &= \rho n_1 ... n_i [entry_R] m_1 ... m_j [call_Q][entry_Q] o_1 ... \\ &\quad o_k [exit_Q][return_Q] \\ P_{exit_Q} &= \rho n_1 ... n_i [entry_R] m_1 ... m_j [call_Q][entry_Q] o_1 ... \\ &\quad o_k [exit_Q] \\ P_{entry_Q} &= \rho n_1 ... n_i [entry_R] m_1 ... m_j [call_Q][entry_Q] \\ P_{call_Q} &= \rho n_1 ... n_i [entry_R] m_1 ... m_j [call_Q] \\ P_{entry_R} &= \rho n_1 ... n_i [entry_R] \end{aligned}$$

where $return_Q$ is in procedure R, both $m_1...m_j$ and $o_1...o_k$ are realizable paths with the same number of calls as returns, and $entry_Q$, $exit_Q$, $call_Q$, and $return_Q$ are the entry, exit, call, and return nodes, respectively, associated with the call. Consider the following cases:

- a and b are both *not* visible in the called procedure; It is impossible for the called procedure to create or destroy this alias pair, thus $<a,b>$ holds on P_{return_Q} iff it holds on P_{call_Q}. This implies *Rule 1* of Figure 5.

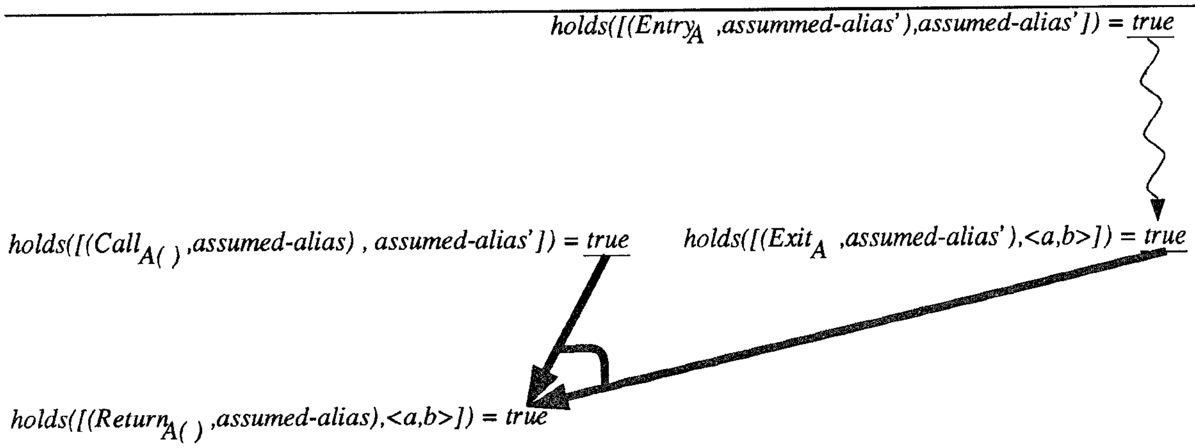
holds([(Entry_A ,assummed-alias'),assumed-alias']) = true
holds([(Call_A( ) ,assumed-alias) , assumed-alias']) = true
holds([(Exit_A ,assumed-alias'),<a,b>]) = true
holds([(Return_A( ) ,assumed-alias),<a,b>]) = true

Figure 3: *holds* at a return node (no local variables)

- a and b are both visible in the called procedure;

  If $<a,b>$ holds on P_{return_Q}, it must also hold on P_{exit_Q}. By Lemma 3.1, either no aliases need hold on P_{entry_Q} for $<a,b>$ to hold on P_{exit_Q} or there exists a $<c,d>$ such that only $<c,d>$ must hold on P_{entry_Q} for $<a,b>$ to hold on P_{exit_Q}. In the first case, $holds([(exit,\emptyset),<a,b>])$ will be *true* (by definition of *holds*). In the second case, $holds([(exit,<c,d>),<a,b>])$ must be *true* and $back\text{-}bind_{call_Q}(<c,d>)$ must hold on P_{call_Q}. Again by Lemma 3.1 there is an *assumed-alias* (either $\emptyset$ or a single alias pair) which must hold on P_{entry_R} and by definition of *holds*, $holds([(call_Q, assumed\text{-}alias), back\text{-}bind_{call_Q}(<c,d>)])$ is *true.* Thus *holds* obeys *Rule 2* of Figure 5.

- a is visible but b is not (the symmetric case is similar);

  The alias $<a,b>$ holds on P_{return_Q} iff $<a,\cdot>$ holds on P_{exit_Q} and b is the non-visible object "$\cdot$". By Lemma 3.1 there is a single $<a',\cdot>$ which must hold on P_{entry_Q} and thus $back\text{-}bind'_{call_Q}(<a',\cdot>,b)$ must hold on P_{call_Q}. By Lemma 3.1 there is an *assumed-alias* (either $\emptyset$ or a single alias pair) which must hold on P_{entry_R} and by definition of *holds*, $holds([(call_Q, assumed\text{-}alias)), back\text{-}bind'_{call_Q}(<a',\cdot>,b)])$ is *true.* This justifies *Rule 3* of Figure 5.

Lemma 3.2 *In the presence of single level pointers, holds can be precisely computed in polynomial time.*

The computation of *holds* is a simple fixed point computation (initial value of *holds* at nodes is *false*). The calculation takes time polynomial in the size of the program because each $holds([(node,\mathcal{AA}),\mathcal{PA}])$ can only change its value once and there is only a polynomial number of such triples (polynomial in the size of the ICFG and the number of variables in the program). The proof of precision is by induction on number of iterations of the fixed point algorithm and induction on path length[12]. □

4 Intraprocedural May Alias with Multiple Level Pointers

Theorem 4.1 *In the presence of two level pointers, the problem of determining precise* **Intraprocedural May Alias** *sets is $\mathcal{NP}$-hard.*

The proof of Theorem 4.1 is by reduction from the 3-SAT problem for $\bigwedge_{i=1}^{n}(l_{i,1} \vee l_{i,2} \vee l_{i,3})$ with variables $\{v_1, v_2, ..., v_m\}$. The reduction is specified by the program in Figure 6 which is polynomial in the size of the 3-SAT problem. The conditionals are not specified in the program since we are assuming that all paths are executable. □

$back\text{-}bind_{call_p}(\emptyset) = \emptyset$

$$back\text{-}bind_{call_p}(<a,b>) = \begin{cases} <a,b> & \text{if } a \text{ and } b \text{ are both } \textbf{visible} \text{ in the procedure called by } call_p \\ <a,*a_j> & \text{if } a \text{ is visible in the procedure called by } call_p\text{, and } b \text{ is the dereferenced formal } *f_j \text{ with corresponding actual } a_j \\ <*a_j,b> & \text{if } b \text{ is visible in the procedure called by } call_p\text{, and } a \text{ is the dereferenced formal } *f_j \text{ with corresponding actual } a_j \\ <*a_j,*a_k> & \text{if } a \text{ is the dereferenced formal } *f_j \text{ with corresponding actual } a_j\text{,and } b \text{ is the dereferenced formal } *f_k \text{ with corresponding actual } a_k \\ false & \text{otherwise (either } a \text{ or } b \text{ is a local [non-parameter] object)} \end{cases}$$

$$back\text{-}bind'_{call_p}(<a,\cdot>,o) = \begin{cases} <a,o> & \text{if } a \text{ is } \textbf{visible} \text{ in the procedure called by } call_p \\ <*a_j,o> & \text{if } a \text{ is the dereferenced formal } *f_j \text{ with corresponding actual } a_j \\ false & \text{otherwise (} a \text{ is a local [non-parameter] object)} \end{cases}$$

Figure 4: Specification of $back\text{-}bind_{call_p}(assumed\text{-}alias)$ and $back\text{-}bind_{call_p}(<a,\cdot>,o)$

- *Rule 1* If a and b are both not visible in the called procedure:
  $holds([(return, assumed\text{-}alias), <a,b>]) = holds([(call, assumed\text{-}alias), <a,b>])$

- *Rule 2* If a and b are both visible in the called procedure:
  $holds([(return, assumed\text{-}alias), <a,b>]) =$
  $$holds([(exit,\emptyset),<a,b>]) \vee \bigvee_{AA \in \mathcal{ASSUMED}} \begin{pmatrix} holds([(exit,AA),<a,b>]) \wedge \\ holds([(call, assumed\text{-}alias), back\text{-}bind_{call}(AA)]) \end{pmatrix}$$

- *Rule 3* If a is visible but b is not (the symmetric case is similar):
  $$holds([(return, assumed\text{-}alias),<a,b>]) = \bigvee_{<o,\cdot> \in \mathcal{ASSUMED}} \begin{pmatrix} holds([(exit,<o,\cdot>),<a,\cdot>]) \wedge \\ holds([(call, assumed\text{-}alias), back\text{-}bind'_{call}(<o,\cdot>,b)]) \end{pmatrix}$$

Figure 5: *holds* relation at return nodes

```
        int **$v_1$,**$v_2$,...,**$v_m$;
        int **$\overline{v_1}$,**$\overline{v_2}$,...,**$\overline{v_m}$;
        int *true,*false;
        int yes,no;

        /* A path through this section of
           code corresponds to a truth
           assignment */
L1:
        if (-) {$v_1$ = &true; $\overline{v_1}$ = &false}
            else {$v_1$ = &false; $\overline{v_1}$ = &true}
        if (-) {$v_2$ = &true; $\overline{v_2}$ = &false}
            else {$v_2$ = &false; $\overline{v_2}$ = &true}
                          . . .
        if (-) {$v_m$ = &true; $\overline{v_m}$ = &false}
            else {$v_m$ = &false; $\overline{v_m}$ = &true}

L2: false = &no;

        /* The code below will break the
            <*false, no> alias before
            reaching L3 iff the truth
            assignment from above makes the
            formula false */
        if (-) *$l_{1,1}$† = &yes
            else if (-) *$l_{1,2}$ = &yes
            else *$l_{1,3}$ = &yes;
        if (-) *$l_{2,1}$ = &yes
            else if (-) *$l_{2,2}$ = &yes
            else *$l_{2,3}$ = &yes;
                      . . .
        if (-) *$l_{n,1}$ = &yes
            else if (-) *$l_{n,2}$ = &yes
            else *$l_{n,3}$ = &yes;
L3:
```
† $l_{i,j}$ is not the string $l_{i,j}$, but the literal it represents (i.e. v_k or $\overline{v_k}$ for some k).

Figure 6: 3-SAT solution iff [L3,<*$false$, no>] in Intraprocedural May Alias

5 Future Work

Naively calculating the fixed point yields a polynomial time $\mathcal{O}(n * v^6)$ algorithm where n is the number of ICFG nodes and v is the number of objects that may have aliases; thus, our theoretical algorithm is not practically viable. However, by intelligent calculation of the fixed point, we can reduce the complexity to $\mathcal{O}(|\text{size of conditional alias solution}|^2/n)$ in the worst case. We expect a solution of conditional alias to be sparse in comparison to the possible solution space and have hopes for an $\mathcal{O}(|\text{size of alias solution}|)$ average case complexity.

Our goal is to develop a good alias approximation algorithm for general pointer usage in C. Currently, we are working on a practical approximation algorithm for Interprocedural May Alias in the presence of multiple level pointers. Our algorithm design combines ideas from the proof in Section 3, with modifications aimed at acceptable performance and precision. We plan to implement this algorithm and empirically test its performance on actual C programs. We intend to both theoretically and empirically examine the amount of imprecision of our approximations (i.e., the percentage of spurious aliases reported).

A validation of the usefulness of our algorithm will be its incorporation in various data flow analyses, including interprocedural modification side effect analysis for C systems. Specifically, we intend to use our algorithm in a new version of *ISMM*, our semantic change impact analyzer for C [19].

We also plan to include the analysis work on recursive data structures in our framework for pointer aliasing.

6 Conclusions

We have presented a classification of pointer-induced aliasing problems including Intraprocedural May Alias, Intraprocedural Must Alias, Interprocedural May Alias and Interprocedural Must Alias. Each problem has been defined by the programming language constructs that cause aliasing alone and in combination: reference formals, single level pointers, multiple level pointers and structures with embedded pointers. We have categorized the theoretical complexity of these problems, noting that the presence of multiple levels of indirection leads to $\mathcal{NP}$-hard and co-$\mathcal{NP}$-hard problems. An explanation of a polynomial time Interprocedural May Alias algorithm has been given. Current work focuses on design and implementation of a practical, approximate interprocedural aliasing algorithm based, in part, on the ideas in these complexity results.

Acknowledgments: We thank Hemant Pande for his help in improving this presentation.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 35–46, July 1988. SIGPLAN NOTICES, Vol. 23, No. 7.

[11] N. Jones and S. Muchnick. A survey of data flow analysis techniques. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice Hall, 1979.

[12] W. Landi. *Interprocedural Aliasing in the Presence of Pointers*. PhD thesis, Rutgers University, 1991. in preparation.

[13] W. Landi and B. G. Ryder. Aliasing with and without pointers: A problem taxonomy. Center for Computer Aids for Industrial Productivity Technical Report CAIP-TR-125, Rutgers University, September 1990.

[14] J. R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, University of California Berkeley, May 1989.

[15] 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.

[16] D. Lomet. Data flow analysis in the presence of procedure calls. *Journal of Research and Development*, 21(6):559–571, November 1977.

[17] T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–196, January 1990.

[18] 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.

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

[20] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[21] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. Master's thesis, M.I.T., June 1980.

[22] 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.