Interprocedural Reaching Definitions in the Presence of Single Level Pointers*

Hemant D. Pande
Siemens Corporate Research Inc
755 College Rd. East
Princeton, NJ 08540
pande@cadillac.siemens.com

William Landi
Siemens Corporate Research Inc
755 College Rd. East
Princeton, NJ 08540
landi@cadillac.siemens.com

Barbara G. Ryder
Department of Computer Science
Rutgers University
ryder@cs.rutgers.edu

October 13, 1992

Abstract

This paper describes the first algorithm that calculates Interprocedural Def-Use Associations in *C* software systems. Our algorithm accounts for program-point-specific pointer-induced aliases, although it is currently limited to programs using a single level of indirection. We prove the *NP*-hardness of the Interprocedural Reaching Definitions Problem and point out the approximation made by our polynomial-time algorithm. Initial empirical results are also presented.

*The research reported here was supported by Siemens Corporate Research and NSF grants CCR-8920078 and CCR-9023628 1/5.

1 Introduction

Currently, most software tools ignore program constructs that involve pointers because extant analysis techniques are too approximate. Determining data dependences more precisely in the presence of pointers facilitates the construction of *effective* debugging, testing and maintenance tools for C systems. Our Interprocedural Def-Use analysis is the first step in solving useful data flow problems with a high degree of precision for C programs. Our analysis deals with programs that only use one level of pointer indirection and shows promise of generalizing to multiple levels of indirection[1]. Given the broad range of software written in C and C^{++}, this represents initial work in the analysis of an important class of software systems. This paper describes the Def-Use analysis algorithm which is based on a polynomial-time Interprocedural Reaching Definitions calculation. We also report preliminary implementation results.

Previous research in alias analysis of C programs has proven the theoretical difficulty of precisely solving the aliasing problem for programs with multiple levels of indirection [23, 24]. This research initiated the idea of performing *conditional* alias analysis in the presence of pointers. It allows us to analyze the code in a procedure under certain input assumptions. We then combine the results of these conditional analyses for those assumptions that actually may occur during program execution. We can generalize the conditional analysis technique which answers the question: *"If there is a path to the entry of the procedure containing node n, on which condition P holds, can fact Q hold at n?"* In this paper we show how to apply this conditional analysis technique to obtain Interprocedural Reaching Definitions in C programs which use only a single level of indirection. We chose the Reaching Definitions problem for our study because of its direct relation to data dependences and therefore, to data flow testing methodology and slicing-based debugging techniques. Our polynomial-time algorithm is approximate, which is expected since we have also shown the *NP-hardness* of the problem we are solving. However, because the aliasing information we are using is specific to a program point (rather than the assumption that the same aliases hold throughout a procedure), we are confident that sufficient accuracy is obtained to insure the utility of the Def-Use information.

At this time, we are using conditional techniques to obtain an approximate analysis of aliasing in C programs with multiple levels of indirection [25, 22]. We are also investigating the extension of our Interprocedural Reaching Definitions algorithm to an approximate algorithm for analyzing programs with multiple levels of indirection. Initially, we have chosen to concentrate on programs with only a single level of indirection because our experience indicates that the major theoretical difficulties in solving problems for programs with multiple levels of indirection are also inherent for programs with a single level of indirection.

1.1 Applications

Def-Use information is necessary for a range of software development environment tools. Data flow information is crucial to data flow based testing systems [11, 14, 33, 32, 37]. The accuracy of the static Def-Use information determines the efficiency of the test case coverage. If imprecise information is an underestimate of the Def-Use Associations, it can lead to missed test paths; if

[1] By limiting programs to one level of indirection we mean, for example, that *int ** variables can occur but not *int *** variables, and we do not allow recursive structures (e.g. linked lists).

it is an overestimate of the Def-Use Associations it can lead to generating unnecessary test cases, resulting in more lengthy testing. If we obtain a *safe* approximation to Def-Use Associations solution, then only the latter situation can occur.

Debuggers based on static and dynamic slicing methods [1, 19, 21, 34, 41, 43] offer the promise of efficient on-line analyses of programs. The aim of slicing is to concentrate the programmer's attention on those parts of the program significant to the computation under current investigation; the more imprecise the static information, the less effectively the slicing method can prune away unrelated computations.

Experimental techniques for merging independently altered versions of programs [18, 44] are of great interest to the programming-in-the-large community. In this domain also, the precision of the data flow information greatly impacts the comparison of the program semantics before and after changes; imprecise information may lead to incorrect conclusions of semantic differences.

Software maintenance aids that compute and report the semantic change impact of evolving software systems [38, 39, 40], similarly rely on the accuracy of their static analyses to provide reasonably precise side effect information. Our previous work in providing efficient incremental semantic change impact analysis of *C* programs using interprocedural data flow analysis, has been hampered by the lack of precision in the alias analyses. Many spurious side effects are generated because of this imprecision [23].

1.2 Related Work

Recent emphasis in the static analysis community has been on expanding compile-time analyses to include interprocedural information [3, 7, 8, 14, 15, 18, 28, 29, 30]. The Fortran model of interprocedural communication has been successfully analyzed [3, 7, 8, 15, 29], although some analyses have yet to demonstrate their practicality. Callahan [3] and Harrold-Soffa [15] suggested factoring the aliases into the problem solution after the side effect analysis, as in previous work by Lomet [29]. Lomet's approach suggested that an approximation of side effects could be obtained by analyzing the procedure under different aliasing conditions and then combining them at some loss of precision. This is similar to our approach, except we are dealing with pointer-induced aliasing. By solving a conditional version of the data flow problem, we avoid some of the loss of precision that Lomet incurred.

Pointers in *C* are difficult to analyze because the address of an arbitrary variable can become the value of another variable, and therefore, can be transferred by assignment statements in a program. The Fortran model of aliasing fails for *C* in two ways: (i) aliases cannot be created intraprocedurally during execution of a Fortran procedure, (ii) aliases in the calling procedure in Fortran cannot be affected by activity in the called procedure. Both (i) and (ii) can occur in *C* programs. Most previous work in analyzing pointer-induced aliasing has been incomplete, impractical, or imprecise by design [5, 6, 9, 10, 12, 31, 42]. This would render corresponding Def-Use analysis imprecise as well. Thus, our work on Interprocedural Reaching Definitions in the presence of pointers, which builds on our work in pointer-induced aliasing, is qualitatively different from previous work.

There also has been some work in discerning the values of pointer variables associated with dynamic data structures such as lists and records. The context of this work has been the development of parallelizing compilers; it has concentrated on *conflict/dependence analysis*, which asks *"When can two names point in a dynamic structure to the same cell?"* [4, 13, 16, 17, 20, 27]. In aliasing we are interested in finding when there may be two names for the same cell *at the same time* during execution, while conflict detection seeks to find out when two names may point to the same cell *at different times* during execution.

1.3 Paper Overview

First, we provide definitions for our program representation and the concepts relevant to the Reaching Definitions problem. Second, we discuss the *may-hold* and *must-hold* aliasing problems. Third, we describe our Interprocedural Reaching Definitions algorithm, first without and then with aliasing effects. Fourth, we outline our Def-Use Associations algorithm. This description does not correspond to the actual implemented version of the algorithm, but serves to illustrate the underlying ideas. Fifth, we explain our demand driven implementation techniques and report our preliminary results. Finally, we summarize the contributions of this research.

2 Problem Specifications

2.1 Program Representation

A *control flow graph* (CFG) for a procedure consists of nodes that represent single-entry/single-exit regions of executable code and edges which represent possible execution branches between code regions. We represent a program with *an interprocedural control flow graph* (ICFG), which intuitively is the union of control flow graphs for the individual procedures comprising the program. Formally, an ICFG is a triple $(\mathcal{N}, \mathcal{E}, \rho)$ where $\mathcal{N}$ is the set of nodes, $\mathcal{E}$ is the set of directed edges connecting the nodes in $\mathcal{N}$, and ρ is the *entry* node for procedure `main`. $\mathcal{N}$ contains a node for each simple statement in the program, an *entry* and an *exit* for each procedure, and a *call* and a *return* node for each call site. An intraprocedural edge into a *call* node represents the execution flow into a call site, while an intraprocedural edge out of a *return* node represents control flow from the call site. An interprocedural edge joins each *call* node to the *entry* node of its corresponding called procedure, while an interprocedural edge joins each *exit* node to the corresponding *return* node. See Figure 1 for an example of a program and its ICFG[2].

2.2 Terminology

The following terminology will be used throughout this paper.

object: An object is a location accessible by a variable either directly or through a pointer indirection. We refer to the objects by *object names* like v and $*v$.

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

reaching definition: A definition $<n : a>$ of object a at node n *reaches* node m if there is a realizable path $nn_1n_2\ldots n_jm$ such that a is <u>not</u> redefined in any $\{n_i\}_{i=1}^{j}$. If a is a pointer variable we consider a definition of a to also be an implicit definition of $*a$ (i.e., the object name obtained by a single level indirection from a).

[2]This representation of a program is similar to those of [15, 30], although we use it differently during analysis.

```
int   *f, *p, *r, *q;

A ( ) {
    if ( - ) {
        j : f = p;
        k : B ( );
    } else {
        l : f = q;
        m : B ( );
        n : r = q;
    }
}

B ( ) {
    h : q = f;
}
```

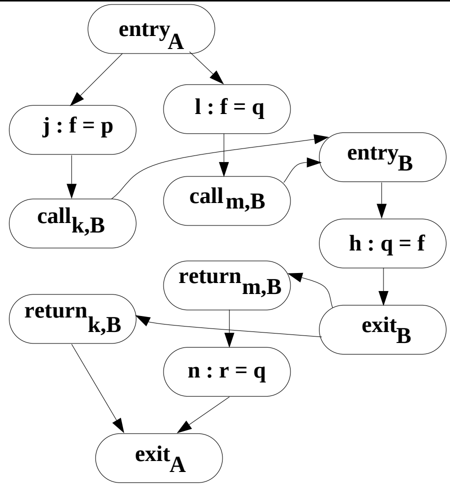


Figure 1: A program segment and its ICFG

def-use association: If a definition $<n : a>$ reaches a use of a at node m, then $\ll n, m \gg$ is said to be a Def-Use Association for a.

holds: Alias $<a, b>$ holds on the realizable path $\rho n_1 n_2 \ldots n_j n$ iff a and b are names for the same object after program point n, whenever the execution sequence defined by the path occurs[3].

may be aliased: a may be aliased to b at n iff there exists a realizable path $\rho n_1 n_2 \ldots n_i n$ on which $<a, b>$ holds. These aliases are defined at a program point, not just at procedure entry as in the Fortran analysis of [8]; thus, they are *program-point-specific aliases.*

must be aliased: a must be aliased to b at n iff for all realizable paths $\rho n_1 n_2 \ldots n_{i-1} n$, $<a, b>$ holds. These aliases are also program-point-specific.

may alias: The precise[4] solution for interprocedural may alias is
$\{[n, <a, b>] | \exists$ a realizable path, $\rho n_1 n_2 \ldots n_j n$, in the ICFG on which $<a, b>$ holds$\}$.

must alias: The precise solution for interprocedural must alias is
$\{[n, <a, b>] | \forall$ realizable path, $\rho n_1 n_2 \ldots n_j n$, in the ICFG on which $<a, b>$ holds$\}$.

visible: At a call site, an object name (e.g. $*x$) is visible in the called procedure iff the called procedure is in the scope of the object name *and* at run time the object refers to the same object in both the calling and called procedure. This means that 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.

[3] Aliases are symmetric (i.e., $<a, b>$ holds on a path iff $<b, a>$ also holds on the same path).

[4] We are using the usual data flow definition of precise which means "precise up to symbolic execution". In other words assuming all paths through the program are executable [2].

3 Theoretical Complexity of the Problem

Myers [30] showed that precise Interprocedural Reaching Definitions is *NP-complete* in the presence of aliasing. Landi and Ryder [23] proved the *NP-hardness* of the Intraprocedural Alias Problem in the presence of multiple level pointers. We show that precise solution of Intraprocedural Reaching Definitions in the presence of single level pointers is *NP-hard*. Our proof is by reduction of the 3-SAT problem and is a variation of similar proofs in [23, 26, 30].

Theorem 1 *In the presence of single level pointers, the problem of calculating precise Intraprocedural Reaching Definitions is NP-hard.*

We proceed by reducing the 3-SAT problem for $\bigwedge_{i=1}^{n}(l_{i,1} \vee l_{i,2} \vee l_{i,3})$ with the propositional variables $\{v_1, v_2, \ldots, v_m\}$ to the Reaching Definitions problem. A precise solution to the Reaching Definitions would yield a solution to the 3-SAT problem. The reduction is specified by the program in Figure 2. The program is polynomial in the size of the 3-SAT problem. We interpret $*v_i$ aliased to *true* as meaning the variable v_i is true. We explicitly assign the complement of the value of v_i to $\overline{v_i}$ in the reduction. This is consistent with the interpretation in propositional logic that whenever v_i has the *true*, $\overline{v_i}$ is false, and vice versa. Any path from L1 to L2 is a truth assignment to the variables. The propositional formula is represented between program points L3 and L4.

Suppose the formula is satisfiable, then there exists a path from L3 to L4 on which each $*l_{i,j}$ is aliased to *true*, which implies that all assignments on that path are in effect "`true = YES`". Thus the definition "`d: false = NO`" reaches L4.

Suppose the formula is unsatisfiable, then for every truth assignment there is at least one row where every $*l_{i,j}$ belonging to the row is aliased to *false*. In effect, we have an entire row of "`false = YES`" no matter which path is taken from L1 to L2. Thus "`d: false = NO`" does not reach L4.

Thus 3-SAT is polynomially reducible to Intraprocedural Reaching Definitions with single level pointers, and we have proved Theorem 1. □

There are some easy corollaries that follow this theorem. All the following have the problem in Theorem 1 as a subproblem.

Corollary 1 *In the presence of multiple level pointers, the problem of calculating precise Intraprocedural Reaching Definitions is NP-hard.*

Corollary 2 *In the presence of either single or multiple level pointers, the problem of calculating precise Interprocedural Reaching Definitions is NP-hard.*

Our polynomial-time algorithm for Interprocedural Reaching Definitions fails to yield precise results, but always yields a *safe* solution. A definition may be reported to reach a node although that definition is killed at an intermediate program point; however, no definition that reaches a node is missed. Thus, the algorithm calculates a conservative, *safe* solution.

4 May and Must Hold

To solve the Interprocedural Reaching Definitions problem, first we need to solve for the program-point-specific alias information. To solve for the pairs of object names which may be aliased, a two

```
    int *$v_1$,*$\overline{v_1}$,*$v_2$,*$\overline{v_2}$,..., *$v_m$,*$\overline{v_m}$;
    int true,false;
    const 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:
d:  false = NO;
L3:
    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;

L4:
```

† $l_{i,j}$ is not the string $l_{i,j}$, but the literal it represents (i.e. v_k or $\overline{v_k}$ for some k).

Figure 2: NP-hardness of the problem

```
int *p, *q, *r, *s;
A ()  {
        k : r = NULL;
        if (-)
                  m : r = p;
        else
                  l : r = q;
        n : ...;
}
```

Figure 3: A program segment

step approach is used. The first step answers the question: *"If there is a path to the entry node of the procedure containing n on which all aliases in the set $\mathcal{A}$ hold, then may* a *be aliased to* b *at n?"*. Intuitively we think of the answer to this question as defining *conditional aliasing*. The second step then uses these *conditional aliases* to solve for the actual aliases.

4.1 The *may-hold* Predicate

To represent the conditional may-alias information, we use the *may-hold* predicate[5]. *may-hold*($[n$, *assumed-alias*, *alias-pair*$]$) is *true* iff *alias-pair* holds on some realizable path from *entry*(n)[6] to n, assuming there is a realizable path from the program entry node ρ to *entry*(n) on which *assumed-alias* holds. For single level pointers, at most one alias pair is required to hold at the entry node to insure that *alias-pair* exists at the end of a particular realizable path from entry node to n [22]. Therefore, *assumed-alias* is either $\emptyset$ or a single alias pair. Note however, that for a particular *alias-pair*, different paths may require different assumptions at the entry. For example in Figure 3, assuming $<*p, *s>$ may hold at $entry_A$ implies that $<*r, *s>$ may hold at node n on path $[entry_A][m : r = p][n]$. On the other hand, assuming $<*q, *s>$ may hold at $entry_A$ implies that the same alias pair $<*r, *s>$ may hold at the same node n, but on the path $[entry_A][l : r = q][n]$. Stated formally, *may-hold* $([n, <*p, *s>, <*r, *s>])$ and *may-hold* $([n, <*q, *s>, <*r, *s>])$ are both *true*. Thus *may-hold* captures the aliasing effects on individual paths. No alias assumption is needed at the entry to insure that $<*r, *p>$ holds on some realizable path ending at the node m. Thus, *may-hold* $([m, \emptyset, <*r, *p>]) = true$.

4.2 The *must-hold* Function

To represent conditional must-alias information, we use the *must-hold* function. The conditional must-alias set for a node n of the ICFG and an *alias-pair* is the unique minimal set of pairs that must be aliased at the entry node of the procedure containing n which insures that *alias-pair* must

[5] This predicate appears as *holds* in [23, 24].

[6] *entry*(n) denotes the entry node of the procedure containing n, and $entry_A$ denotes the entry node of procedure A.

hold at n. If no such set exists, we say *must-hold*$(n, \textit{alias-pair})$ is $\perp$. This implies that *alias-pair* cannot be aliased on all paths to node n, regardless of what must-alias set exists at entry of the procedure containing n. By contrast, *must-hold*$(n, \textit{alias-pair}) = \emptyset$ implies that without requiring any assumptions at the entry node, *alias-pair* must be aliased at node n.

In Figure 3, *must-hold*$(n, <*r, *s>) = \{<*p, *s>, <*q, *s>\}$, since both pairs must be aliased at entry to insure $<*r, *s>$ must hold at n, no matter which path is taken. On the other hand, *must-hold*$(k, <*r, *s>) = \perp$, since no set at entry can insure that $<*r, *s>$ must be aliased immediately after node k is executed.

Our Interprocedural Reaching Definitions algorithm assumes the availability of *may-hold* and *must-hold* information. In the Reaching Definitions analysis for languages without any aliasing mechanisms, an assignment generates a definition of a single object name. In the presence of aliases, a definition is generated for each object name that may be aliased to the assigned object name. Thus, the *may-hold* predicate is responsible for the *generation* of reaching definitions. In the absence of aliases, an assignment to an object name kills the reaching definitions of the same object name. In the presence of aliases, the reaching definitions of all the object names that are must aliased to the assigned object name must be killed. Thus, the *must-hold* sets are responsible for killing of reaching definitions. Therefore, *may-hold* is necessary to obtain a safe Reaching Definitions solution, while *must-hold* provides a way of obtaining a more precise Reaching Definitions solution.

5 Algorithm for Interprocedural Reaching Definitions

In this section, we begin with the algorithm to calculate Interprocedural Reaching Definitions in the absence of pointers and call-by-reference formals, in effect, in the absence of mechanisms that cause aliasing. Later, we account for the aliasing effects of single level pointers and call-by-reference parameter passing. The conditional approach obviates the need for any special treatment to handle recursive procedures; when the conditional analysis assumes a condition to hold at the entry of a procedure, it is regardless of whether the condition holds due to a recursive or non-recursive call. Finally, we describe the algorithm to compute Def-Use Associations given the Reaching Definitions for each ICFG node. We present all these algorithms informally; a formal algorithm description can be found in Appendix B.

5.1 No Aliasing Mechanisms

First we ask, "Given that m, n are ICFG nodes, if the definition *assumed-rd* of an object name[7] reaches the entry of the procedure containing node n, can the definition $<m : a>$ reach node n?". The answer is called *Conditional Reaching Definitions* information. Second, we show how to use Conditional Reaching Definitions to obtain the Interprocedural Reaching Definitions sets (*rdtop*) at the top of every ICFG node. This use of conditional information enables us to restrict our analysis to realizable paths in the ICFG. Finally, we describe how to solve for Conditional Reaching Definitions.

[7] In the no aliasing situation definitions (and uses) of object names are just definitions (and uses) of variables. However, for consistency with later sections we will use the term object name here.

```
int     a, b;
P ()    {
        d1 : a = 2;
        m : Q ();
}
```

```
Q ()    {
        d2 : b = 3;
        n:...;
}
```

Figure 4: No aliasing mechanisms

5.1.1 Conditional Reaching Definitions

We define the *reaches* predicate to represent Conditional Reaching Definitions information at the bottom of each ICFG node. *reaches* ($[ICFG\text{-}node, assumed\text{-}rd, rd]$) is *true* iff assuming *assumed-rd* reaches the entry of the procedure containing *ICFG-node*, *rd* reaches the bottom of *ICFG-node*. It is easy to see that the value of *assumed-rd* is either $\emptyset$ or *rd* itself. *rd* can be generated during the execution of the procedure containing *ICFG-node* and subsequently reach *ICFG-node*. In this case, no assumptions are necessary at the entry and *assumed-rd* is $\emptyset$. For example in Figure 4, $reaches([n, \emptyset, <d2 : b>]) = true$, since the definition $<d2 : b>$ reaches node n without any assumptions at the entry of procedure Q. On the other hand, a definition *rd* may reach the entry of the procedure and be preserved on some realizable path from entry to *ICFG-node*, in which case *assumed-rd* = *rd*. For example in Figure 4, $reaches([n, <d1 : a>, <d1 : a>]) = true$, since the definition $<d1 : a>$ reaches node n contingent on the assumption that $<d1 : a>$ reaches the entry of procedure Q. It should also be noted that $reaches([n, \emptyset, <d1 : a>]) = false$.

5.1.2 *rdtop* from *reaches*

We now formulate a data flow problem on the ICFG to compute the Reaching Definitions set *rdtop* from *reaches* at every ICFG node. For each node n in the ICFG = ($\mathcal{N}$, $\mathcal{E}$, ρ), *rdtop* is calculated as follows:

- $rdtop(\rho) = \emptyset$
- if n is an entry node, then $rdtop(n) = \bigcup_{\ll l,n \gg \in \mathcal{E}}(rdtop(l))$
- if n is a return node, $rdtop(n) =$

$$\left\{ <m : a> \middle| \begin{array}{l} \big(reaches([n, \emptyset, <m : a>]) \bigvee \\ \qquad (<m : a> \in rdtop(entry(n)) \bigwedge reaches([n, <m : a>, <m : a>]))\big) \end{array} \right\}$$

- otherwise, $rdtop(n) =$

$$\left\{ <m : a> \middle| \begin{array}{l} \bigvee_{\ll l,n \gg \in \mathcal{E}} \big(reaches([l, \emptyset, <m : a>]) \bigvee \\ \qquad (<m : a> \in rdtop(entry(n)) \bigwedge reaches([l, <m : a>, <m : a>]))\big) \end{array} \right\}$$

Theorem 2 *The computation of rdtop given reaches is a polynomial-time fixed point calculation.*

Let $\mathcal{N}$ denote the set of ICFG nodes and v the number of object names defined in the program. For each ICFG node, the value of set *rdtop* can grow at most $\mathcal{O}(|\mathcal{N}| * v)$ times during the fixed point calculation. Since there are $\mathcal{O}(|\mathcal{N}|)$ ICFG edges, calculation of *rdtop* from *reaches* is a polynomial-time problem. □

5.1.3 Calculation of *reaches*

For each ICFG node, the value of *reaches* is calculated as follows:

Assignment node: Intraprocedurally, definition $<m : a>$ reaches the bottom of an ICFG node if it reaches any of its predecessors in the ICFG, unless the node itself re-defines a. An assignment node *generates* a reaching definition, to be *propagated* to its successors. Formally, consider $reaches([n, assumed\text{-}rd, <m : a>])$,

- if n is an assignment "$a = \ldots$"
  $reaches([n, assumed\text{-}rd, <m : a>]) = true$ if $n = m$
  $reaches([n, assumed\text{-}rd, <m : a>]) = false$ if $n \neq m$
- Otherwise
  $reaches([n, assumed\text{-}rd, <m : a>]) = true$ iff $reaches([l, assumed\text{-}rd, <m : a>])$ is *true* for some immediate predecessor l of n.

Entry node: The Conditional Reaching Definitions analysis is oblivious of what actually reaches the entry of the procedure during the course of execution. Thus at the entry site of a procedure the algorithm must make the important assumption: *reaches* $([entry, <m : a>, <m : a>]) =$ *true* for all possible definitions of object names visible across the procedure. Also, for each formal parameter f, *reaches* $([entry, \emptyset, <entry : f>]) =$ *true*.

Call/exit node: *call* and *exit* nodes simply collect the Reaching Definitions information. Thus:

$$reaches([call/exit, assumed\text{-}rd, <m : a>]) = \bigvee_{\ll l, call/exit \gg \in \mathcal{E}} reaches([l, assumed\text{-}rd, <m : a>])$$

Return node: The *return* nodes offer the most interesting case in the interprocedural analysis. Suppose we are interested in whether $reaches([return, assumed\text{-}rd, <m : a>])$ is *true*. The predicate is *true* if $<m : a>$ reaches the corresponding *exit*, either because $<m : a>$ was generated in the called procedure, **or** it reached the corresponding *call* **and** was preserved through the called procedure. Formally,

$$reaches([return, assumed\text{-}rd, <m : a>]) =$$

$$\left(\begin{array}{ll} \left(\begin{array}{l} reaches([exit, \emptyset, <m : a>]) \vee \\ reaches([call, assumed\text{-}rd, <m : a>]) \wedge reaches([exit, <m : a>, <m : a>]) \end{array} \right) & a \in visible(call) \\ reaches([call, assumed\text{-}rd, <m : a>]) & a \notin visible(call) \end{array} \right)$$

The algorithm to calculate *reaches* is as follows:

1. Construct the ICFG.

2. Initialize $reaches(n, assumed\text{-}rd, rd)$ to *false* for all nodes n and reaching definitions rd.

3. Calculate the fixed point of *reaches* using a standard data flow analysis algorithm.

Theorem 3 *The computation of reaches predicate is a polynomial-time (in the number of ICFG nodes) fixed point calculation.*

Let $\mathcal{N}$ denote the set of ICFG nodes and v the number of object names that get defined in the program. There are $\mathcal{O}(|\mathcal{N}|^2 * v)$ *reaches* predicates[8]. Since each definition of an object name is represented by an ICFG node, it is clear that $v < |\mathcal{N}|$. The calculation of each *reaches* takes time $\mathcal{O}(predecessors)$ of the node which is at most $\mathcal{O}(|\mathcal{N}|)$. In the fixed point calculation the value of a *reaches* changes from *false* to *true* at most once. Thus the computation of all *reaches* predicates is a polynomial-time calculation. □

We actually perform the *reaches* calculation in a demand driven fashion. That is, we calculate the *reaches* information, as and when we need it for an *assumed-rd* which actually reachs an procedure entry node. Thus, the implemented algorithm has cost approximately proportional to the size of the *reaches* solution.

5.2 Accounting for Aliases

Given the framework described in the previous section, now we include single level pointers in the analysis. Since C has only pass-by-value, we present only that, but we can handle pass-by-reference by a transformation[35]. The introduction of pointers results in aliases at various program points. Our analysis must account for the generation and killing of reaching definitions due to aliasing effects. We also need to model the aliasing effects of pointer parameter bindings for each call site. For this purpose, we use the function $back\text{-}bind_{call_P}$ for each call site $call_P$. $back\text{-}bind_{call_P}$ (*assumed-alias*) specifies which alias holding on any path $\rho \ldots [call_P]$ guarantees that *assumed-alias* holds on the path $\rho \ldots [call_P][entry_P]$. Our Reaching Definitions algorithm for this problem is polynomial but imprecise, as mentioned in Section 3.

The *reaches* predicate for this version of the algorithm has the following interpretation:

$$reaches([ICFG\text{-}node, (assumed\text{-}rd, assumed\text{-}alias), rd])$$

is *true* iff assuming *assumed-rd* reaches and *assumed-alias* holds at the entry of the procedure containing *ICFG-node*, then *rd* reaches the bottom of *ICFG-node*. We have already seen the significance of *assumed-rd* in the Conditional Reaching Definitions. The following discussion motivates the significance of *assumed-alias* in the analysis. We describe the calculation of *reaches* for each type of ICFG node, with examples from Figure 5.

Assignment node: At ICFG node $d1$, $*c$ and $*b$ may be aliased. Thus the definition of $*b$ may in effect be a definition of $*c$ too. While analyzing the procedure Q, we must know under what conditions $<*c, *b>$ may hold at node $d1$. We use the *may-hold* predicate for this purpose. By inspection, *may-hold* $([d1, <*a, *b>, <*c, *b>]) = true$. In other words, assuming $<*a, *b>$

[8] a definition is a node/object name pair ($\mathcal{O}(|\mathcal{N}| * v)$) and *reaches* is a node/definition pair.

```
int *a, *b, *c;
P ()  {                    Q ()  {                    R ()  {
        m : a = b;                 d0 : c = a;                l : Q ();
        n : Q ();                  d1 : *b = ...;       }
}                          {
```

Figure 5: With pointer aliases

holds at $entry_Q$, $<*c, *b>$ holds at $d1$. In effect, assuming $<*a, *b>$ holds at $entry_Q$, we can claim that the definition of $*b$ may also be a definition of $*c$ at $d1$[9]. As a result,

$$\textit{may-hold}([d1, <*a, *b>, <*c, *b>]) \Rightarrow \textit{reaches}([d1, (\emptyset, <*a, *b>), <d1 : *c>]) = \textit{true}$$

Note that the *assumed-rd* component of the *reaches* relation is $\emptyset$. Since the definition $<d1 : *c>$ is generated at node $d1$, we do not need any assumptions at the entry node for this definition to reach the node $d1$.

Entry node: As in Section 5.1.3, the analysis must consider a definition to reach the entry node of a procedure if the definition is *assumed* to reach the node. For example, $\textit{reaches}([entry_Q, (<m : a>, \emptyset), <m : a>]) = \textit{true}$. This enables us to determine whether a definition, if it reached the entry node of a procedure, would reach the exit node of the procedure. Note that the *assumed-alias* component is $\emptyset$. Once a definition is assumed to reach the entry node, the *assumed-alias* component plays no part in deciding whether the definition is preserved through the procedure. A definition of an object a is killed at a node if the node assigns to an object which *must be aliased* to a, irrespective of the *assumed-alias* which may have generated the definition[10]. Also, for each formal parameter (and its one level indirection in case the formal is a pointer), a definition is generated at the entry. Details can be found in Appendix B.

The formulation described in the two cases above has a nice property that *assumed-rd* and *assumed-alias* are never both non-$\emptyset$. This property limits the number of *reaches* relations by eliminating a multiplicative effect which would arise if relations with both non-$\emptyset$ components occurred in the analysis.

Call/Exit node: These nodes simply collect the Reaching Definitions information. Thus:

$$\textit{reaches}([call/exit, (\textit{assumed-rd}, \textit{assumed-alias}), <m : a>]) = \bigvee_{\ll l, call/exit \gg \in \mathcal{E}} \textit{reaches}([l, (\textit{assumed-rd}, \textit{assumed-alias}), <m : a>])$$

[9]Remember that we are interested in the *reaches* information at the bottom of a node. Thus, the information *at the node* $d1$ also reflects the effects of executing $d1$ itself.

[10]In Section 5.4, we will use the *must-hold* function to further improve the precision of the Reaching Definitions calculation, but *assumed-alias* will continue to play no role in killing a definition.

Return node: $reaches([return, (assumed\text{-}rd, assumed\text{-}alias), rd])$ is *true* iff any of the following situations exists:

1. rd was generated during the execution of the called procedure without any assumptions at entry and reached the exit. For example,

$$reaches([exit_Q, (\emptyset, \emptyset), <d1 : *b>]) \Rightarrow reaches([return_{n,Q}, (\emptyset, \emptyset), <d1 : *b>])$$

2. rd reached the call site (and thus the entry of the called procedure) and is preserved through the called procedure to reach its exit. For example, the definition $<m : a>$ reaches the call site $call_{n,Q}$ and is preserved through Q to reach $exit_Q$. As a result, we have

$$\begin{aligned} reaches([call_{n,Q}, (\emptyset, \emptyset), <m : a>]) \quad &\wedge \quad reaches([exit_Q, (<m : a>, \emptyset), <m : a>]) \\ &\Rightarrow \quad reaches([return_{n,Q}, (\emptyset, \emptyset), <m : a>]) \end{aligned}$$

3. rd was generated during the execution of the called procedure due to an alias present at the call site (and thus at the entry of the called procedure) and reached the exit. For example, $<*a, *b>$ may hold at the call site $call_{n,Q}$. As we saw, the definition $<d1 : *c>$ is generated at $d1$ due to this alias at $entry_Q$. This definition propagates to the exit of procedure Q. Thus,

$$\begin{aligned} reaches([exit_Q, (\emptyset, <*a, *b>), <d1 : *c>]) \quad &\wedge \quad may\text{-}hold([call_{n,Q}, \emptyset, <*a, *b>]) \\ &\Rightarrow \quad reaches([return_{n,Q}, (\emptyset, \emptyset), <d1 : *c>]) \end{aligned}$$

Let $ALIAS$ be the set of all possible assumed alias pairs in the program. The following formula to calculate the value of *reaches* for a *return* node accounts for the three situations described above. For simplicity, we assume rd represents the definition of an object name visible in the called procedure (see Appendix B for further details).

$$reaches([return, (assumed\text{-}rd, assumed\text{-}alias), rd]) =$$

$$\begin{array}{ll} 1. & reaches([exit, (\emptyset, \emptyset), rd]) \vee \\ 2. & \left(\begin{array}{l} reaches([call, (assumed\text{-}rd, assumed\text{-}alias), rd]) \wedge \\ reaches([exit, (rd, \emptyset), rd]) \end{array} \right) \\ 3. & \bigvee_{AA' \in ALIAS} \left(\begin{array}{l} reaches([exit, (\emptyset, AA'), rd]) \wedge \\ may\text{-}hold([call, assumed\text{-}alias, back\text{-}bind_{call}(AA')]) \end{array} \right) \end{array}$$

5.3 Algorithm Complexity

Theorem 4 *The algorithm to calculate* reaches *is polynomial in the number of ICFG nodes and object names.*

Let $\mathcal{N}$ denote the set of ICFG nodes and v the number of object names. Aliases are pairs of object names; thus there are $\mathcal{O}(v^2)$ of them. Definitions are node/object name pairs, so there are $\mathcal{O}(|\mathcal{N}| * v)$ of them. *reaches* is a quadruple [*node*, (*assumed-rd*, *assumed-alias*), *rd*] with the restriction that *assumed-rd* is either $\emptyset$ or rd; thus there are $\mathcal{O}(|\mathcal{N}|^2 * v^3)$ *reaches* predicates. For each *reaches* calculation at a *return* node we may do $\mathcal{O}(v^2)$ work because there may be as many as v^2 *assumed-alias* values. The work at an arbitrary ICFG node is bounded by the work at a return node. Therefore the total cost of processing for all ICFG nodes, is bounded above by $\mathcal{O}(|\mathcal{N}|^2 * v^5)$ since, in the fixed point calculation, the value of a *reaches* changes from *false* to *true* at most once. Thus the computation of *reaches* predicate is a polynomial-time calculation. □

As mentioned in Section 5.1.3, we perform the *reaches* calculation in a demand driven fashion. Thus, the implemented algorithm has cost approximately proportional to the size of the *reaches* solution.

5.4 Using *must-hold* for Further Precision

In Section 5.2, we proposed the use of must-alias information to kill a Reaching Definition. To increase the accuracy of the Reaching Definition calculation, we introduce *must-hold* function in the *reaches* calculation. The new *reaches* has the form

$$reaches([ICFG\text{-}node, (assumed\text{-}rd, assumed\text{-}alias, assumed\text{-}must\text{-}alias), rd])$$

and is *true* iff *rd* reaches *ICFG-node*, with the assumption that *assumed-rd* reaches, *assumed-alias* holds and *assumed-must-alias* is the must-alias set at the entry node of the procedure containing *ICFG-node*. The values of *assumed-must-alias* are restricted to the set of alias pairs that must be imposed at the entry node of the procedure due to the *must-alias* set at some corresponding call node. Each call to the procedure has a must-alias set associated with it; $bind_{call}(must\text{-}alias(call))$ provides the *assumed-must-alias* holding at the entry of the called procedure when the procedure is called from the call site *call*. For each entry node of a procedure, the number of distinct *assumed-must-alias* sets is at most equal to the number of calls to the procedure[11].

We describe the role of *must-hold* in killing a Reaching Definition at an assignment node. A complete and formal description of this final version of the algorithm appears in Appendix B. In Figure 5, the call site $call_{n,Q}$ imposes the must-alias set $\{<\!*a, *b\!>\}$ at $entry_Q$. As a result, $reaches([d0, (\emptyset, \emptyset, \{<\!*a, *b\!>\}), <\!d0 : *c\!>]) = true$ represents the generation of $<\!d0 : *c\!>$ with respect to this call. On the other hand, the call site $call_{l,Q}$ does not impose any non-trivial must-alias set[12]. So $reaches([d0, (\emptyset, \emptyset, \emptyset), <\!d0 : *c\!>]) = true$ corresponds to this call. At node $d1$, $must\text{-}hold(d1, <\!*b, *c\!>) = \{<\!*a, *b\!>\}$. For the assignment "$d1 : *b = \ldots$" to kill $<\!d0 : *c\!>$, $\{<\!*a, *b\!>\}$ must be a subset of the *assumed-must-alias* set at $entry_Q$. Accordingly, $reaches([d1, (\emptyset, \emptyset, \{<\!*a, *b\!>\}), <\!d0 : *c\!>]) = false$. On the other hand, $reaches([d1, (\emptyset, \emptyset, \emptyset), <\!d0 : *c\!>]) = true$ because in the absence of *must-alias* information, $<\!d0 : *c\!>$ is not killed by statement $d1$. Definition $<\!d0 : *c\!>$ reaches $call_{l,Q}$ but not $call_{n,Q}$ because the *must-alias* set $\{<\!*a, *b\!>\}$ exists at $call_{n,Q}$ and not at $call_{l,Q}$.

5.5 *rdtop* from *reaches*

We formulate a data flow problem on the ICFG to compute the Reaching Definitions set *rdtop* from *reaches* at every ICFG node. The algorithm is a generalization of the algorithm described in Section 5.1.2. Given *bind* and *may-alias*, the algorithm is polynomial by Theorem 2.

[11] Two different call sites can impose the same must-alias sets at the entry.

[12] For brevity, we do not mention the trivial *must-alias* pairs like $<\!*a, *a\!>$.

Let *caller-list* (n) be the set of call nodes corresponding to *entry* (n). For each node n in the ICFG = $(\mathcal{N}, \mathcal{E}, \rho)$, *rdtop* is calculated as follows:

- $rdtop(\rho) = \emptyset$
- if n is an entry node, then $rdtop(n) = \bigcup_{\ll l,n \gg \in \mathcal{E}} (rdtop(l))$
- if n is a return node, $rdtop(n) =$

$$\left\{ rd \;\middle|\; \begin{array}{c} \exists k \in caller\text{-}list(n)[\mathcal{AMA} = bind_k(must\text{-}alias(k))] \wedge \\ \left(\left(\begin{array}{l} reaches([n, (\emptyset, \mathcal{AA}, \mathcal{AMA}), rd]) \\ \wedge \;\; \mathcal{AA} \in (may\text{-}alias(entry(n)) \cup \{\emptyset\}) \end{array} \right) \vee \left(\begin{array}{l} reaches([n, (rd, \emptyset, \mathcal{AMA}), rd]) \\ \wedge \;\; rd \in rdtop(entry(n)) \end{array} \right) \right) \end{array} \right\}$$

- otherwise, $rdtop(n) =$

$$\left\{ rd \;\middle|\; \bigvee_{\ll l,n \gg} \left(\begin{array}{c} \exists k \in caller\text{-}list(l)[\mathcal{AMA} = bind_k(must\text{-}alias(k))] \wedge \\ \left(\left(\begin{array}{l} reaches([l, (\emptyset, \mathcal{AA}, \mathcal{AMA}), rd]) \\ \wedge \;\; \mathcal{AA} \in (may\text{-}alias(entry(l)) \cup \{\emptyset\}) \end{array} \right) \vee \left(\begin{array}{l} reaches([l, (rd, \emptyset, \mathcal{AMA}), rd]) \\ \wedge \;\; rd \in rdtop(entry(l)) \end{array} \right) \right) \end{array} \right) \right\}$$

5.6 Algorithm for Def-Use Associations

Once the Interprocedural Reaching Definitions are obtained in the form of *rdtop* set at each ICFG node, it is straightforward to compute the Def-Use Associations. If a node n uses an object a (before possibly defining a), and there is a definition $<m : a>$ reaching the top of node n, then we establish a Def-Use Association $\ll m, n \gg$ for a. This algorithm is clearly polynomial in the number of nodes in the ICFG.

6 Present Status

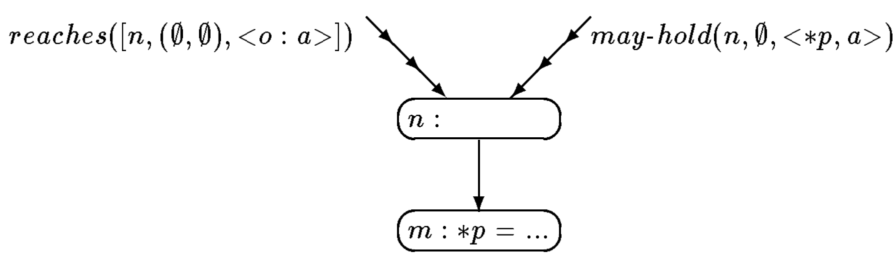
For the ease of explanation and understanding we presented our algorithm in this paper as a two phase process. First we described the calculation of the *reaches* predicate with all theoretically possible assumptions each the entry node. Then we described the calculation of *rdtop* sets which picks out only those *reaches* predicates that have realizable assumptions at an entry node; a *realizable assumption* results from a realizable path from the entry of main to an entry node on which the assumption holds. For example in Figure 5, $reaches([entry_R, (<m : a>, \emptyset), <m : a>])$ is *true* by definition, but there exists no realizable path on which $<m : a>$ reaches $entry_R$. An implementation strictly as described would be prohibitively inefficient, but alternative implementations need not be, as we demonstrate below.

We have constructed a prototype implementation of our algorithm to observe its performance on C programs. We do not use *must-hold* information as there are still theoretical difficulties that must be surmounted before *must-hold* can be practically implemented. We are using the PTT system from Siemens [36] for our C parser. We have aimed to eliminate the effect of purely hypothetical assumptions, such as $<m : a>$ at $entry_R$ in Figure 5, and to calculate the *reaches* solution in time proportional to the number of *true reaches* predicates. A similar situation exists in efficiently finding aliases; we have applied our aliasing algorithm implementation techniques [25] to the *reaches* calculation.

Basically, in our implementation we propagate information forward from each ICFG node to its successors. At an assignment node, we initialize the appropriate *reaches* predicates to *true* depending on the *may-hold* predicates at the node. We then propagate the *reaches* values along the realizable paths. For example, suppose $reaches([\textit{call-node}, (\emptyset, \emptyset), rd])$ is *true* at a call site denoted as *call-node*. As a result, $reaches([\textit{entry-node}, (rd, \emptyset), rd])$ is *true* capturing the fact there exists a realizable path on which *rd* reaches *entry-node*; this is in contrast to our theoretical algorithm statement in which we simply assume a possible reaching definition at entry. With this approach, our implementation never processes any unrealizable assumptions. As a result the implementation takes time proportional to the size of *reaches* solution.

To obtain a lower bound on the empirical precision of our Reaching Definitions solution we use methods similar to those in [25]. We can show that there is only one source of approximation in our algorithm [35] which is illustrated by the following scenario. Assume that:

- m is the assignment "$*p = ...$"
- n is an immediate predecessor of m in the ICFG.
- We know that the definition $<o : a>$ reaches the bottom of n on some path (e.g., $reaches([n, (\emptyset, \emptyset), <o : a>])$ is $true$).
- We also know that $<*p, a>$ holds on some path to the bottom of n[13] (e.g., $\textit{may-hold}(n, \emptyset, <*p, a>)$ is $true$).



Our algorithm as described would conclude that $reaches([m, (\emptyset, \emptyset), <o : a>])$ is $true$, but is that correct? If definition $<o : a>$ reaches n on some path and $<*p, a>$ does not hold on that path then definition $<o : a>$ reaches the bottom of m. However, if for every path on which definition $<o : a>$ reaches n, $<*p, a>$ holds, then $<o : a>$ does not reach the bottom of m and our algorithm is being imprecise by saying $reaches([m, (\emptyset, \emptyset), <o : a>])$ is $true$. We have designed our implementation to keep track of these possibly erroneous reporting of $reaches$[14]. From this we compute *%precision* which is the percentage of the solution generated which is definitely not erroneous. Thus *%precision* is a lower bound on the precision of our solution because it assumes *every* assumption made by our algorithm for safety was incorrect.

[13] and thus the top of node m

[14] A reporting of a *reaches* being *true* is counted as possibly erroneous if it is the result of some form of the above case, or if it depends on some possibly erroneous *reaches*.

Program Name	Lines	Nodes	Alias Calc. Time	Rds Calc. Time	Rds Per Node	Def-Use Calc. Time	Def-Use Assocs.	%precision
parser	731	1274	1.02 sec	4.59 sec	33.53	1.04 sec	604	100.00
fixoutput	400	622	0.51 sec	1.81 sec	28.7	0.37 sec	362	87.32
jacobi	172	361	0.33 sec	0.69 sec	29.9	0.66 sec	302	95.01
SOR	177	356	0.27 sec	0.69 sec	28.7	0.62 sec	270	88.37
crypt	396	385	0.11 sec	1.39 sec	52.8	0.63 sec	466	93.53
strings	185	389	0.17 sec	0.41 sec	8.5	0.15 sec	250	100.00
random	366	240	0.30 sec	0.44 sec	25.8	0.20 sec	218	93.65
atol-etc	100	147	0.05 sec	0.11 sec	5.12	0.04 sec	105	97.47
bcmp-etc	50	44	0.01 sec	0.02 sec	2.5	0.01 sec	24	100.00
pattern	99	170	0.06 sec	0.25 sec	20.8	0.18 sec	104	97.31
weather	132	124	0.06 sec	0.15 sec	12.4	0.07 sec	110	95.26

Table 1: Preliminary Results

Presently our prototype implementation obtains the Reaching Definitions and Def-Use Associations for code with at most a single level of indirection. Finding *C* programs that use only single level pointers has been extremely difficult, so our test suite is limited. We present our preliminary implementation results in Table 1. The programs in the first group were written as programming assignments for graduate courses at Rutgers University. The second group is chosen from the source code for *C* library functions. The third group is from the test suite for TACTIC [33], which is currently using the Def-Use Associations information to perform data-flow based test coverage of *C* programs. The timings for Alias and Reaching Definitions calculation on these programs are promising. Additional experiments are needed to confirm that our techniques remain practical for larger programs. The precision of our algorithm appears to be good; at least 87% of the Reaching Definitions reported by our algorithm would be present in the precise solution[15]. Our algorithm qualifies as the first highly precise technique to calculate Interprocedural Reaching Definitions in the presence of single level pointers.

7 Conclusions

We have presented a polynomial-time technique to find Interprocedural Reaching Definitions and the Def-Use Associations in *C* programs with only a single level of indirection; this is the first algorithm which accounts for pointer-induced aliases in *C* systems. We have demonstrated the theoretical difficulty of obtaining a precise solution for Interprocedural Reaching Definitions, the basis for Interprocedural Def-Use information. We are empirically testing the viability and precision of our algorithm, and have reported preliminary results. Our research marks an important milestone

[15] under the common assumptions of static analysis [2].

in the practical static analysis of C programs, producing information useful for debugging, testing and maintenance tools. In the future, we plan to extend our algorithm to handle multiple level pointers and recursive structures, as well as to increase the efficiency of the algorithm. We also plan to apply these ideas to other compile-time analyses (e.g., program slicing).

8 Acknowledgments

We thank Michael Platoff, Michael Wagner and Thomas Ostrand for discussions and suggestions on this work.

References

[1] H. Agrawal and J.R. Horgan. Dynamic program slicing. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, June 1990. also available as SIGPLAN Notices, vol 25, no 6, June 1990.

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

[3] 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.

[4] 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.

[5] 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.

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

[7] K. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 247–258, June 1984. SIGPLAN Notices, Vol 19, No 6.

[8] 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.

[9] 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.

[10] 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.

[11] P.G. Frankl and E.J. Weyuker. A data flow testing tool. In *Proceedings of Softfair II*. IEEE, December 1985.

[12] 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.

[13] W. L. Harrison III and Z. Ammarguellat. Parcel and Miprac: parallelizers for symbolic and numeric programs. In *Proceedings of International Workshop on Compilers for Parallel Computers*, pages 329–346. Ecole des Mines de Paris - CAI, UPMC - Laboratoire MASI, December 1990. Paris, France.

[14] M. J. Harrold and M. L. Soffa. Interprocedural data flow testing. In *Proceedings of the Third Testing, Analysis, and Verification Symposium*, pages 158–167, December 1989.

[15] M. J. Harrold and M. L. Soffa. Computation of interprocedural definition and use dependencies. In *Proceedings of the 1990 International Conference on Computer Languages*, pages 297–306, 1990.

[16] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[17] 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.

[18] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[19] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 35–46, June 1988.

[20] 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.

[21] Bogdan Korel and Janusz Laski. Dynamic slicing of computer programs. *Journal of Systems Software*, 13:187–195, 1990.

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

[23] 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.

[24] 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.

[25] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992.

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

[27] 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.

[28] Sying-Syang Liu and Abu-Bakr Taha. Interprocedural definition-use dependency analysis for recursive procedures. Technical Report SERC-TR-42-F, Software Engineering Research Center, University of Florida, Gainesville, Florida, March 1990.

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

[30] 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.

[31] 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.

[32] T. J. Ostrand and E. Weyuker. Data flow based test adequecy analysis for languages with pointers. In *Proceedings of the 1991 Symposium on Software Testing, Analysis and Verification (TAV4)*, October 1991. Victoria, B.C., Canada.

[33] Thomas J. Ostrand. Data-flow testing with pointers and function calls. In *Proceedings of the Pacific Northwest Software Quality Conference*, October 1990.

[34] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, May 1984.

[35] H. D. Pande. *Static Analysis in the Presence of Pointers.* PhD thesis, Rutgers University, 1993. in preparation.

[36] Michael Platoff, Michael Wagner, and Joseph Camaratta. An integrated program representation and toolkit for the maintenanc e of C programs. In *Proceedings of the Conference on Software Maintenance.* IEEE Computer Society Press, October 1991.

[37] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

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

[39] B. G. Ryder and M. D. Carroll. An incremental algorithm for software analysis. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 171–179, December 1986.

[40] B. G. Ryder, W. Landi, and H. Pande. Profiling an incremental data flow analysis algorithm. *IEEE Transactions on Software Engineering*, 16(2):129–140, February 1990.

[41] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 107–119, June 1991.

[42] 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.

[43] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[44] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accomodates semantics preserving transformations. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 133–143, December 1990. also available as SIGSOFT Notes, vol 15, no 6, December 1990.

Appendix A: Example

Sample Program:

```
int *p;
int a,b;
R(f)
int *f;
{
n1:   *f = 1;
}
```

```
main()
{
n2:   a = 0;
n3:   b = 0;
n4:   if (-) {
n5:          p = &a;
n6:          R(p);
      } else {
n7:          p = &b;
n8:          R(p);
      }
}
```

Summary of *may-hold*
(unless otherwise noted *may-hold*([node,assumed-alias,possible-alias]) = $false$):

$\textit{may-hold}([node, \emptyset, <x, x>]) = \textit{true}$ for $node \in \{entry_R, n_1, exit_R, entry_{main}, n_2, n_3, n_4, n_5, call_{n_6}, return_{n_6}, n_7, call_{n_8}, return_{n_8}, exit_{main}\}$ and $x \in \{a, b, *p, *f\}$

$\textit{may-hold}([entry_R,\text{alias},\text{alias}]) = \textit{may-hold}([[n_1 : \ *\text{f} = 1],\text{alias},\text{alias}]) = \textit{may-hold}([exit_R,\text{alias},\text{alias}]) = \textit{true}$
for alias $\in \{<*\text{p}, \text{a}>, <*\text{p}, \text{b}>, <*\text{f}, \text{a}>, <*\text{f}, \text{b}>, <*\text{f}, *\text{p}>\}$

$\textit{may-hold}([[n_5 : \text{p} = \&\text{a}],\emptyset, <*\text{p}, \text{a}>]) = \textit{true}$

$\textit{may-hold}([call_{n_6},\emptyset, <*\text{p}, \text{a}>]) = \textit{true}$

$\textit{may-hold}([return_{n_6},\emptyset, <*\text{p}, \text{a}>]) = \textit{true}$

$\textit{may-hold}([[n_7 : \text{p} = \&\text{b}],\emptyset, <*\text{p}, \text{b}>]) = \textit{true}$

$\textit{may-hold}([call_{n_8},\emptyset, <*\text{p}, \text{b}>]) = \textit{true}$

$\textit{may-hold}([return_{n8},\emptyset, <*\text{p}, \text{b}>]) = \textit{true}$

$\textit{may-hold}([exit_{main},\emptyset,<*\text{p}, \text{b}>]) = \textit{true}$

$\textit{may-hold}([exit_{main},\emptyset,<*\text{p}, \text{a}>]) = \textit{true}$

Summary of *reaches*:

$$\text{Let } DEFS = \left\{ \begin{array}{l} <entry_R : \text{f}>, <entry_R : *\text{f}>, <n_1 : \text{a}>, <n_1 : \text{b}>, <n_1 : *\text{p}>, <n_1 : *\text{f}>, \\ <n_2 : \text{a}>, <n_2 : *\text{p}>, <n_3 : \text{b}>, <n_3 : *\text{p}>, <n_5 : \text{p}>, <n_5 : *\text{p}>, <n_5 : \text{a}>, \\ <n_5 : \text{b}>, <n_7 : \text{p}>, <n_7 : *\text{p}>, <n_7 : \text{a}>, <n_7 : \text{b}> \end{array} \right\}$$

Let $ALIAS = \{\emptyset, <*\text{p}, \text{a}>, <*\text{p}, \text{b}>, <*\text{f}, \text{a}>, <*\text{f}, \text{b}>, <*\text{p}, *\text{f}>\}$

We present the *reaches* information in a tabular fashion. The first column represents the *reaches* relations for each ICFG node, and the second column represents the corresponding equations used to calculate their fixed point values. For an assignment node there are three calculations shown. The first describes which definitions are generated at the node, the second describes which definitions are killed if they reach the node, and the third describes which definitions are propagated through this node. The *reaches* behavior for an *exit* and *call* node depends directly upon that of its predecessors. The *reaches* calculation of a *return* node follows the equations given in Section 5.2. *reaches* is *true* at an *entry* node for only two types of definitions: those definitions that we assume reach the *entry* nodes, and implicit definitions of the formal parameters of the corresponding procedure.

$reaches([entry_R,(\emptyset,\mathcal{A}),\mathcal{R}])$ $\mathcal{R} \in \{<entry_R : \mathrm{f}>, <entry_R : *\mathrm{f}>\}$ $\mathcal{A} \in ALIAS$	$true$
$reaches([entry_R,(\emptyset,\mathcal{A}),\mathcal{R}])$ $\mathcal{R} \in DEFS - \{<entry_R : \mathrm{f}>, <entry_R : *\mathrm{f}>\}$ $\mathcal{A} \in ALIAS$	$false$
$reaches([entry_R,(\mathcal{AR},\emptyset),\mathcal{R}])$ $\mathcal{AR},\mathcal{R} \in DEFS$	$true$ if $\mathcal{AR} = \mathcal{R}$ $false$ otherwise
$reaches([[n_1 : *\mathrm{f} = 1],(\mathcal{AR},\mathcal{A}), <n_1 : x>])$ $\mathcal{AR} \in DEFS, \mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$ $x \in \{a, b, *\mathrm{p}, *\mathrm{f}\}$	*may-hold* $([[n_1 : *\mathrm{f} = 1], \mathcal{A}, <*f, x>])$
$reaches([[n_1 : *\mathrm{f} = 1],(\mathcal{AR},\mathcal{A}), <entry_R : *\mathrm{f}>])$ $\mathcal{AR} \in DEFS, \mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$	$false$
$reaches([[n_1 : *\mathrm{f} = 1],(\mathcal{AR},\emptyset), \mathcal{R}])$ $\mathcal{R} \in DEFS - \{<entry_R : *\mathrm{f}>, <n_1 : *\mathrm{f}>\}$ $\mathcal{AR} \in DEFS \cup \{\emptyset\}$	$reaches([entry_R,(\mathcal{AR},\emptyset), \mathcal{R}])$
$reaches([exit_R,(\mathcal{AR},\mathcal{A}),\mathcal{R}])$ $\mathcal{AR} \in (DEFS \cup \{\emptyset\}), \mathcal{R} \in DEFS$ $\mathcal{A} \in ALIAS$, where $\mathcal{AR} = \emptyset$ or $\mathcal{A} = \emptyset$	$reaches([[n_1 : *\mathrm{f} = 1],(\mathcal{AR},\mathcal{A}), \mathcal{R}])$
$reaches([entry_{main},(\mathcal{AR},\mathcal{A}),\mathcal{R}])$ $\mathcal{AR} \in (DEFS \cup \{\emptyset\}),\mathcal{R} \in DEFS$ $\mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ or $\mathcal{A} = \emptyset$	$true$ if $\mathcal{AR} = \mathcal{R}$ $false$ otherwise
$reaches([[n_2 : \mathrm{a} = 0],(\mathcal{AR},\mathcal{A}), <n_2 : \mathrm{x}>])$ $\mathcal{AR} \in DEFS, \mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$ $x \in \{a, b, *\mathrm{p}, *\mathrm{f}\}$	*may-hold* $([[n_2 : \mathrm{a} = 0], \mathcal{A}, <a, x>])$
$reaches([[n_2 : \mathrm{a} = 0],(\mathcal{AR},\mathcal{A}), <node : \mathrm{a}>])$ $\mathcal{AR} \in DEFS, \mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$ $node \in \{n_1, n_5, n_7\}$	$false$
$reaches([[n_2 : \mathrm{a} = 0],(\mathcal{AR},\emptyset), \mathcal{R}])$ $\mathcal{R} \in DEFS - \{<node : \mathrm{a}> \mid node = n_1, n_2, n_5, \text{or } n_7\}$ $\mathcal{AR} \in DEFS \cup \{\emptyset\}$	$reaches([entry_{main},(\mathcal{AR},\emptyset), \mathcal{R}])$

$reaches([[n_3 : \mathrm{b} = 0], (\mathcal{AR}, \mathcal{A}), <n_3 : \mathrm{x}>])$ $\mathcal{AR} \in DEFS, \mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$ $x \in \{a, b, *\mathrm{p}, *\mathrm{f}\}$	*may-hold* $([[n_3 : \mathrm{b} = 0], \mathcal{A}, <b, x>])$
$reaches([[n_3 : \mathrm{b} = 0], (\mathcal{AR}, \mathcal{A}), <node : \mathrm{b}>])$ $\mathcal{AR} \in DEFS, \mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$ $node \in \{n_1, n_5, n_7\}$	$false$
$reaches([[n_3 : \mathrm{b} = 0], (\mathcal{AR}, \mathcal{A}), \mathcal{R}])$ $\mathcal{R} \in DEFS - \{<node : \mathrm{b}> \mid node = n_1, n_3, n_5, \text{or } n_7\}$ $\mathcal{AR} \in DEFS \cup \{\emptyset\}, \mathcal{A} \in ALIAS$	$reaches([[n_2 : \mathrm{a} = 0], (\mathcal{AR}, \mathcal{A}), \mathcal{R}])$
$reaches([[n_4\text{: if (-)}], (\mathcal{AR}, \mathcal{A}), \mathcal{R}])$ $\mathcal{AR} \in (DEFS \cup \{\emptyset\}), \mathcal{R} \in DEFS$ $\mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ or $\mathcal{A} = \emptyset$	$reaches([[n_3 : \mathrm{b} = 0], (\mathcal{AR}, \mathcal{A}), \mathcal{R}])$
$reaches([[n_5 : \mathrm{p} = \&\mathrm{a}], (\mathcal{AR}, \mathcal{A}), \mathcal{R}])$ $\mathcal{R} \in \{<n_5 : \mathrm{p}>, <n_5 : *\mathrm{p}>\}$ $\mathcal{AR} \in DEFS, \mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$	$true$
$reaches([[n_5 : \mathrm{p} = \&\mathrm{a}], (\mathcal{AR}, \mathcal{A}), \mathcal{R}])$ $\mathcal{R} \in \left\{ <node : \mathrm{x}> \middle\vert \begin{array}{l} node = n_1, n_2, n_3, \text{or } n_7 \\ \text{and x } = \text{p or } *\text{p} \end{array} \right\}$ $\mathcal{AR} \in DEFS, \mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$	$false$
$reaches([[n_5 : \mathrm{p} = \&\mathrm{a}], (\mathcal{AR}, \mathcal{A}), \mathcal{R}])$ $\mathcal{R} \in DEFS - \left\{ <node : \mathrm{x}> \middle\vert \begin{array}{l} node = n_1, n_2, n_3, \text{or } n_7 \\ \text{and x } = \text{p or } *\text{p} \end{array} \right\}$ $\mathcal{AR} \in DEFS \cup \{\emptyset\}$ $\mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$	$reaches([[n_4\text{: if (-)}], (\mathcal{AR}, \mathcal{A}), \mathcal{R}])$
$reaches([call_{n_6}, (\mathcal{AR}, \mathcal{A}), \mathcal{R}])$ $\mathcal{R} \in DEFS$, $\mathcal{AR} \in DEFS \cup \{\emptyset\}$ $\mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$	$reaches([[n_5 : \mathrm{p} = \&\mathrm{a}], (\mathcal{AR}, \mathcal{A}), \mathcal{R}])$
$reaches([return_{n_6}, (\mathcal{AR}, \mathcal{A}), \mathcal{R}])$ $\mathcal{R} \in DEFS$, $\mathcal{AR} \in DEFS \cup \{\emptyset\}$ $\mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$	$reaches([exit_R, (\emptyset, \emptyset), \mathcal{R}]) \vee$ $\left(\begin{array}{l} reaches([call_{n_6}, (\mathcal{AR}, \mathcal{A}), \mathcal{R}]) \wedge \\ reaches([exit_R, (\mathcal{R}, \emptyset), \mathcal{R}]) \end{array} \right) \vee$ $\bigvee_{\mathcal{A}' \in ALIAS} \left(\begin{array}{l} reaches([exit_R, (\emptyset, \mathcal{A}'), \mathcal{R}]) \wedge \\ may\text{-}hold([call_{n_6}, \mathcal{A}, back\text{-}bind(\mathcal{A}')]) \end{array} \right)$

$reaches([[n_7 : \mathrm{p} = \&\mathrm{b}],(\mathcal{AR},\mathcal{A}),\mathcal{R}])$ $\mathcal{R} \in \{<n_7 : \mathrm{p}>, <n_7 : *\mathrm{p}>\}$ $\mathcal{AR} \in DEFS, \mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$	*true*
$reaches([[n_7 : \mathrm{p} = \&\mathrm{b}],(\mathcal{AR},\mathcal{A}),\mathcal{R}])$ $\mathcal{R} \in \left\{ <node : \mathrm{x}> \;\middle|\; \begin{array}{l} node = n_1, n_2, n_3, \text{or } n_5 \\ \text{and x } = \text{p or } *\text{p} \end{array} \right\}$ $\mathcal{AR} \in DEFS, \mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$	*false*
$reaches([[n_7 : \mathrm{p} = \&\mathrm{b}],(\mathcal{AR},\mathcal{A}),\mathcal{R}])$ $\mathcal{R} \in DEFS - \left\{ <node : \mathrm{x}> \;\middle|\; \begin{array}{l} node = n_1, n_2, n_3, \text{or } n_5 \\ \text{and x } = \text{p or } *\text{p} \end{array} \right\}$ $\mathcal{AR} \in DEFS \cup \{\emptyset\}$ $\mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$	$reaches([[n_4\text{: if (-)}],(\mathcal{AR},\mathcal{A}),\mathcal{R}])$
$reaches([call_{n_8},(\mathcal{AR},\mathcal{A}),\mathcal{R}])$ $\mathcal{R} \in DEFS$, $\mathcal{AR} \in DEFS \cup \{\emptyset\}$ $\mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$	$reaches([[n_7 : \mathrm{p} = \&\mathrm{b}],(\mathcal{AR},\mathcal{A}),\mathcal{R}])$
$reaches([return_{n_8},(\mathcal{AR},\mathcal{A}),\mathcal{R}])$ $\mathcal{R} \in DEFS$, $\mathcal{AR} \in DEFS \cup \{\emptyset\}$ $\mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$	$reaches([exit_R,(\emptyset,\emptyset),\mathcal{R}]) \vee$ $\begin{pmatrix} reaches([call_{n_8},(\mathcal{AR},\mathcal{A}),\mathcal{R}]) \wedge \\ reaches([exit_R,(\mathcal{R},\emptyset),\mathcal{R}]) \end{pmatrix} \vee$ $\bigvee_{\mathcal{A}' \in ALIAS} \begin{pmatrix} reaches([exit_R,(\emptyset,\mathcal{A}'),\mathcal{R}]) \wedge \\ may\text{-}hold([call_{n_8},\mathcal{A},back\text{-}bind(\mathcal{A}')]) \end{pmatrix}$
$reaches([exit_{main},(\mathcal{AR},\mathcal{A}),\mathcal{R}])$ $\mathcal{R} \in DEFS$, $\mathcal{AR} \in DEFS \cup \{\emptyset\}$ $\mathcal{A} \in ALIAS$ where $\mathcal{AR} = \emptyset$ *or* $\mathcal{A} = \emptyset$	$reaches([return_{n_6},(\mathcal{AR},\mathcal{A}),\mathcal{R}]) \vee$ $reaches([return_{n_8},(\mathcal{AR},\mathcal{A}),\mathcal{R}])$

Appendix B: A Polynomial Algorithm for Computing Interprocedural Reaching Definitions in the Presence of Single Level Pointers

Let

- $\mathcal{O} = \left\{ *p \;\middle|\; p \text{ is a pointer variable in the program} \right\} \bigcup$
  $\left\{ v \;\middle|\; v \text{ is a non-pointer variable in the program} \right\} \bigcup \{\cdot\}$.
  $\mathcal{O}$ is the set of all object names in the program which may have aliases.
  "$\cdot$" represents object names which are not visible.
- $\mathcal{POSSIBLE\text{-}ALIASES} = (\mathcal{O} \times \mathcal{O}) - \{<\cdot,\cdot>\}$.
- $\mathcal{ASSUMED\text{-}ALIASES} = \mathcal{POSSIBLE\text{-}ALIASES} \cup \{\emptyset\}$.
- $\mathcal{OBJECTS} = \mathcal{O} \bigcup \left\{ p \;\middle|\; p \text{ is a pointer variable in the program} \right\}$.
- ICFG $= (\mathcal{N}, \mathcal{E}, \rho)$, the Interprocedural Control Flow Graph as defined in Section 2.1.
- $\mathcal{DEFS} = (\mathcal{N} \times \mathcal{OBJECTS})$.

- *exit* (n) be the corresponding exit node for return node n.
- *call* (n) be the corresponding call node for return node n.
- *entry* (n) be the entry node of the procedure containing node n.
- *caller-list* (n) be the set of call nodes corresponding to *entry* (n).
- $\textit{back-bind}_{call}(\textit{alias-pair})$ specify the alias holding at the *call* site which forces *alias-pair* to hold at the entry of the called procedure.
- $\textit{back-bind}'_{call_P}(<a, \cdot>, b)$ specify the alias holding on any path $\rho \ldots [call_P]$ that guarantees a will be aliased to the non-visible object name b on $\rho \ldots [call_P][entry_P]$.
- $\textit{bind}_{call_P}(\textit{alias-set})$ specify for all paths $\rho n_1 \ldots n_i [call_P][entry_P]$ which aliases hold assuming all aliases in *alias-set* hold on $\rho n_1 \ldots n_i [call_P]$.

Construct IRDG $= (\mathcal{N}', \mathcal{E}', \rho')$ and $\textit{reaches} : \mathcal{N}' \rightarrow \{true, false\}$ where

the *reaches* predicate has the same interpretation as in Section 5.4,

$$\mathcal{N}' = \left\{ [node, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd] \;\middle|\; \begin{array}{l} node \in \mathcal{N},\ rd \in \mathcal{DEFS},\ \mathcal{ARD} \in \{rd, \emptyset\}, \\ \mathcal{AA} \in \mathcal{ASSUMED\text{-}ALIASES}, (\mathcal{ARD} = \emptyset \vee \mathcal{AA} = \emptyset), \\ \mathcal{AMA} \in \left\{ A \;\middle|\; \begin{array}{c} (\exists m \in \mathcal{N})\ (m \in \textit{caller-list}\ (node) \\ \wedge\ A = \textit{bind}_m(\textit{must-alias}\ (m))) \end{array} \right\} \end{array} \right\}$$

ρ' = special entry node of the IRDG.

For each type of ICFG node, the edges comprising $\mathcal{E}'$ and the predicate *reaches* are specified as follows.
For all $node \in \mathcal{N}$, for all $\mathcal{AA} \in \mathcal{ASSUMED\text{-}ALIASES}$ and $rd \in \mathcal{DEFS}$:
If *node* is:

entry node *A definition reaches an entry node iff we assume it does, or the object name defined is a formal of the procedure.*
For each $\mathcal{AMA} = \textit{bind}_n(\textit{must-alias}\ (n))$ where $n \in \textit{caller-list}\ (node)$,

- Add $\ll\rho', [node, (rd, \emptyset, \mathcal{AMA}), rd]\gg$ to $\mathcal{E}'$.
- For each formal x of the procedure, add $\ll\rho', [node, (\emptyset, \emptyset, \mathcal{AMA}), <node : x>]\gg$ to $\mathcal{E}'$.
- For each pointer formal x of the procedure, add $\ll\rho', [node, (\emptyset, \emptyset, \mathcal{AMA}), <node : *x>]\gg$ to $\mathcal{E}'$.
- $\textit{reaches}([node, (rd, \emptyset, \mathcal{AMA}), rd]) = true$.
- For each formal x of the procedure, $\textit{reaches}([node, (\emptyset, \emptyset, \mathcal{AMA}), <node : x>]) = true$.
- For each pointer formal x of the procedure, $\textit{reaches}([node, (\emptyset, \emptyset, \mathcal{AMA}), <node : *x>]) = true$.

call node *A definition reaches a call node iff it reaches before the node under identical assumptions.*
For each $\mathcal{AMA} = \textit{bind}_n(\textit{must-alias}\ (n))$ where $n \in \textit{caller-list}\ (node)$,

- For every $\ll m, node \gg \in \mathcal{E}$, add
$\ll [m, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd], [node, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd] \gg$ to $\mathcal{E}'$.
- $reaches([node, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd]) =$
$$\bigvee_{\ll m,node \gg \in \mathcal{E}} (reaches\ ([m, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd])).$$

exit node *A definition reaches an exit node iff it reaches before the node under identical assumptions.*
For each $\mathcal{AMA} = bind_n(must\text{-}alias\ (n))$ where $n \in caller\text{-}list\ (node)$,

- For every $\ll m, node \gg \in \mathcal{E}$, add
$\ll [m, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd], [node, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd] \gg$ to $\mathcal{E}'$.
- $reaches([node, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd]) =$
$$\bigvee_{\ll m,node \gg \in \mathcal{E}} (reaches\ ([m, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd])).$$

return node
Let $\mathcal{MA}$ denote $bind_{call\ (node)}(must\text{-}alias\ (call\ (node)))$.
For each $\mathcal{AMA} = bind_n(must\text{-}alias\ (n))$ where $n \in caller\text{-}list\ (node)$,

- If $rd = <d : g>$ where g is a global object name.
  - Add $\ll [exit\ (node), (\emptyset, \emptyset, \mathcal{MA}), rd], [node, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd] \gg$ to $\mathcal{E}'$.
  - Add $\ll [exit\ (node), (rd, \emptyset, \mathcal{MA}), rd], [node, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd] \gg$ and $\ll [call\ (node), (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd], [node, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd] \gg$ to $\mathcal{E}'$.
  - For each $<c, d> \in \mathcal{POSSIBLE}\text{-}\mathcal{ALIASES}$ ($c \neq$ "·" and $d \neq$ "·"), if $back\text{-}bind_{call\ (node)}(<c, d>) = false$ add nothing to $\mathcal{E}'$, otherwise add $\ll [exit\ (node), (\emptyset, <c, d>, \mathcal{MA}), rd], [node, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd] \gg$ to $\mathcal{E}'$.
  - Calculate $reaches([node, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd])$ using the following formula. (Note: $c \neq$ "·" and $d \neq$ "·")

$reaches([node, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd]) =$

$$reaches([exit\ (node), (\emptyset, \emptyset, \mathcal{MA}), rd]) \vee$$
$$(reaches([exit\ (node), (rd, \emptyset, \mathcal{MA}), rd]) \wedge reaches([call\ (node), (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd]))$$
$$\bigvee_{<c,d> \in \mathcal{POSSIBLE}\text{-}\mathcal{ALIASES}}$$

$$\begin{pmatrix} false & back\text{-}bind_{call(node)}(<c,d>) = false \\ reaches([exit\ (node), (\emptyset, <c,d>, \mathcal{MA}), rd]) & back\text{-}bind_{call(node)}(<c,d>) = \emptyset \\ \begin{pmatrix} reaches([exit\ (node), (\emptyset, <c,d>, \mathcal{MA}), rd]) \wedge \\ may\text{-}hold([call\ (node), \mathcal{AA}, back\text{-}bind_{call\ (node)}(<c,d>)]) \end{pmatrix} & otherwise \end{pmatrix}$$

- If $rd = <d : l>$ where l is "·" or a local object name of the calling procedure.
  - Add $\ll [exit\ (node), (<d : \cdot>, \emptyset, \mathcal{MA}), <d : \cdot>], [node, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd] \gg$ and $\ll [call\ (node), (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd], [node, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd] \gg$ to $\mathcal{E}'$.

- For each $<c,\cdot> \in \mathcal{POSSIBLE\text{-}ALIASES}$, if $\textit{back-bind}'_{call\ (node)}(<c,\cdot>, l) = \textit{false}$, add nothing to $\mathcal{E}'$, otherwise add $\ll[\textit{exit}\ (\textit{node}), (\emptyset, <c,\cdot>, \mathcal{MA}), <d:\cdot>], [\textit{node}, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd]\gg$ to $\mathcal{E}'$.
- Calculate $\textit{reaches}([\textit{node}, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd])$ using the following formula.

$$\textit{reaches}([\textit{node}, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd]) =$$

$$\left(\begin{array}{l} \textit{reaches}([\textit{exit}\ (\textit{node}), (<d:\cdot>, \emptyset, \mathcal{MA}), <d:\cdot>]) \wedge \\ \qquad \textit{reaches}([\textit{call}\ (\textit{node}), (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd]) \end{array} \right)$$
$$\bigvee_{<c,\cdot> \in \mathcal{POSSIBLE\text{-}ALIASES}}$$
$$\left(\begin{array}{ll} \textit{false} & \textit{back-bind}'_{call\ (node)}(<c,\cdot>, l) = \textit{false} \\ \textit{reaches}([\textit{exit}\ (\textit{node}), (\emptyset, <c,\cdot>, \mathcal{MA}), <d:\cdot>]) & \textit{back-bind}'_{call\ (node)}(<c,\cdot>, l) = \emptyset \\ \left(\begin{array}{l} \textit{reaches}([\textit{exit}\ (\textit{node}), (\emptyset, <c,\cdot>, \mathcal{MA}), <d:\cdot>]) \wedge \\ \textit{may-hold}([\textit{call}\ (\textit{node}), \mathcal{AA}, \textit{back-bind}'_{call\ (node)}(<c,\cdot>, l)]) \end{array} \right) & \textit{otherwise} \end{array} \right)$$

pointer assignment “p = …” *p cannot have aliases. A definition of p is also considered a definition of *p.*
Let $rd = <m:b>$.
For each $\mathcal{AMA} = \textit{bind}_n(\textit{must-alias}\ (n))$ where $n \in \textit{caller-list}\ (\textit{node})$,

- $\forall \ll n, \textit{node} \gg \in \mathcal{E}$, if b is neither p nor $*p$, add $\ll[n, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), <m:b>], [\textit{node}, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), <m:b>]\gg$ to $\mathcal{E}'$.
- if $m = \textit{node}$, and b is either p or $*p$, add $\ll\rho', [\textit{node}, (\emptyset, \emptyset, \mathcal{AMA}), <m:b>]\gg$ to $\mathcal{E}'$.
- Calculate $\textit{reaches}([\textit{node}, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), <m:b>])$ using the following formula.

$$\textit{reaches}([\textit{node}, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), <m:b>]) =$$

$$\left(\begin{array}{ll} \bigvee_{\ll n, node \gg \in \mathcal{E}} \textit{reaches}([n, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), <m:b>]) & b \neq p \wedge b \neq *p \\ \textit{true} & \left(\begin{array}{l} m = \textit{node} \wedge (b = p \vee b = *p) \\ \wedge(\mathcal{ARD} = \mathcal{AA} = \emptyset) \end{array} \right) \\ \textit{false} & \textit{otherwise} \end{array} \right)$$

non-pointer assignment “a = …”
Let $rd = <m:b>$.
For each $\mathcal{AMA} = \textit{bind}_n(\textit{must-alias}\ (n))$ where $n \in \textit{caller-list}\ (\textit{node})$,

- $\forall \ll n, \textit{node} \gg \in \mathcal{E}$, add $\ll[n, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), <m:b>], [\textit{node}, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), <m:b>]\gg$ to $\mathcal{E}'$.
- if $m = \textit{node}$, add $\ll\rho', [\textit{node}, (\emptyset, \mathcal{AA}, \mathcal{AMA}), <m:b>]\gg$ to $\mathcal{E}'$.
- $\textit{reaches}([\textit{node}, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), <m:b>]) = \bigvee_{\ll n, node \gg \in \mathcal{E}}$ $(\textit{reaches}([n, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), <m:b>]) \wedge \textit{must-hold}(n, <a,b>) \not\subseteq \mathcal{AMA})$

- Also, if $m = node$
  $reaches([node, (\emptyset, \mathcal{AA}, \mathcal{AMA}), <m : b>]) = may\text{-}hold(n, \mathcal{AA}, <a, b>)$

otherwise (non-assignment nodes like "a == b")
For each $\mathcal{AMA} = bind_n(must\text{-}alias\ (n))$ where $n \in caller\text{-}list\ (node)$,

- $\forall \ll n, node \gg \in \mathcal{E}$, add $\ll [n, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd], [node, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd] \gg$ to $\mathcal{E}'$.
- $reaches([node, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd]) =$
  $\bigvee_{\ll n, node \gg \in \mathcal{E}} reaches([n, (\mathcal{ARD}, \mathcal{AA}, \mathcal{AMA}), rd])$