INTERPROCEDURAL ALIASING IN THE PRESENCE OF POINTERS

BY WILLIAM ALEXANDER LANDI

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Barbara Gershon Ryder

and approved by

_____

_____

_____

_____

_____

New Brunswick, New Jersey

January, 1992

© 1992
William Alexander Landi
ALL RIGHTS RESERVED

ABSTRACT OF THE DISSERTATION

Interprocedural Aliasing in the Presence of Pointers

by William Alexander Landi, Ph.D.

Dissertation Director: Barbara Gershon Ryder

An *Alias* occurs at some program point during execution when two or more names exist for the same location. We've investigated the theoretical difficulty of determining the aliases of a program, developed an approximation algorithm for solving for aliases in C like languages, and explored the precision (i.e., closeness of our approximate solution to the actual solution) and time behavior of this algorithm.

Myers [Mye81] explored the theoretical difficulty of solving flow sensitive interprocedural data flow problems in the presence of aliasing. However, he did not make any claims about the difficulty of determining aliases. We isolate various programming language mechanisms that create aliases. The complexity of the alias problem (i.e., determining the aliases for a program) induced by each mechanism and their combinations is considered separately and categorized as $\mathcal{NP}$-hard, complement $\mathcal{NP}$-hard, or polynomial time ($\mathcal{P}$). We proved that if there are two (or more) levels of indirection, regardless of the alias mechanisms used, aliasing is $\mathcal{NP}$-hard. However, if only one level of indirection is possible, then the alias problem is in $\mathcal{P}$. In addition, we show the alias problem in the presence of some mechanisms to be $\mathcal{P}$-space complete.

Therefore, in a language which allows general purpose pointers, the problem of determining which aliases can occur during program execution is $\mathcal{NP}$-hard. We present an algorithm which safely approximates *Interprocedural May Alias* (i.e., we fail to report

an alias only if it never occurs) in the presence of pointers. We are able to show, relative to our definition of precision, that our algorithm is as precise as possible in the worst case. We also pinpoint the sources of imprecision in our algorithm and can empirically bound its precision. Our algorithm has been implemented in a prototype analysis tool for C programs, and we present a preliminary empirical investigation of algorithm speed and precision.

Acknowledgements

I thank Barbara Ryder and the PROLANGs community for providing an environment for doing productive research. I thank Siemens Corporate Research, Hemant Pande, Michael Platoff, and Michael Wagner for providing me with a front end for my implementation. I also thank Hemant Pande for helping me proofread various parts of this thesis. Finally, I thank my parents, family and friends for their support and encouragement and everyone I played volleyball and cards with for helping keep me sane.

Table of Contents

Abstract ii

Acknowledgements iv

List of Tables x

List of Figures xi

1. Introduction 1

2. Problem Representation 7

2.1. Interprocedural Control Flow Graph 7

2.2. Types 7

2.3. Object Names 9

2.4. Aliases 12

2.5. Some Terminology 13

3. Related Work 15

3.1. Theoretical Classification 15

3.2. Algorithms for Finding Aliases in Programs with Pointers 16

3.3. Conflict/Dependence Analysis 19

4. Classification of the Alias Problems 24

4.1. Reference Formals 26

4.2. Single Level Pointers 28

4.2.1. Intraprocedural May Alias 28

4.2.2. Intraprocedural Must Alias 31

4.2.3. Interprocedural May Alias 32

Computing Conditional May Alias information, assuming no local variables . . . 34
Modeling the effects of parameter bindings . . . 35
Computing Conditional May Alias information at a return node, factoring in local variables . . . 38
Computing Interprocedural May Alias using Conditional May Alias 40
4.2.4. Interprocedural Must Alias . . . 42
Conditional Must Alias . . . 43
Computing Interprocedural Must Alias using Conditional Must Alias . . . 48
4.3. Non-pointer Reference Formals and Single Level Pointers . . . 48
4.4. Multiple Level Pointers . . . 51
4.4.1. May Alias . . . 51
4.4.2. Must Alias . . . 53
4.5. Reference Formals and Single Level Pointers . . . 54
4.5.1. Interprocedural May Alias . . . 54
4.5.2. Interprocedural Must Alias . . . 54
4.6. Structures Containing Single Level Pointers . . . 57
4.6.1. May Alias . . . 57
4.6.2. Must Alias . . . 59
4.7. Alias Sets Associated with Program Paths . . . 60
4.7.1. Single Level Pointers . . . 60
4.7.2. Reference Formals . . . 62
4.8. $\mathcal{P}$-space-hard alias problems . . . 63

5. A Safe Approximate Algorithm for Interprocedural May Alias . . . 77
5.1. The may-hold Relation . . . 77
5.1.1. Conditional May Alias: Definition . . . 77
5.1.2. Representation . . . 79

5.1.3. Alias Consequences 80
5.1.4. Modeling Parameter Bindings 81
Computing bind($call$,$\emptyset$) 83
Computing bind($call$,$\langle x, y \rangle$) 84
5.2. Computing may-hold 88
5.2.1. Aliases_introduced_by_assignment($node$) 89
5.2.2. Aliases_introduced_by_call($node$) 89
5.2.3. Alias_at_call_implies($call$,$assumed_alias$,$possible_alias$) 90
5.2.4. Alias_at_exit_implies($exit$,$assumed_alias$,$possible_alias$) 98
5.2.5. Any_other_alias_implies($node$,$assumed_alias$,$possible_alias$) . . . 99
5.3. May Alias 105

6. Theoretical and Empirical Results 108
6.1. Theoretical Precision 108
6.2. Prototype 115
6.2.1. Optimizations 116
6.3. Empirical Comparison to Weihl's Algorithm 118
6.3.1. Number of Aliases 118
6.3.2. Time 118
6.4. May Alias Solution 119
6.4.1. Empirically Measured Solution Size 119
6.4.2. Measurement of Empirical Precision 124
6.5. May Alias Solution Size vs Conditional May Alias Size 128
6.6. Algorithm Time Performance 131
6.6.1. Theoretical Issues 131
6.6.2. Conclusions 132

7. Conclusions and Future Work 138
7.1. Summary of Thesis 138
7.2. Future Work 139

Appendix A. Dictionary of Functions 140

Appendix B. Intraprocedural Aliasing in the Presence of Single Level Pointers 144

B.1. Building the SPPAG 144

B.2. SPPAG based algorithm is polynomial time and precise for May Alias . 146

B.3. SPPAG based algorithm is polynomial time and precise for Must Alias . 152

Appendix C. Interprocedural Aliasing in the Presence of Single Level Pointers 158

C.1. Back-Bind 158

C.1.1. Definition 158

C.1.2. Proof of correctness 159

C.1.3. Computable in constant time 162

C.2. Bind 162

C.2.1. Definition 162

C.2.2. Proof of correctness 163

C.2.3. Computable in constant time 164

C.3. A Precise Polynomial Time Algorithm for Computing Interprocedural May Alias Sets in the Presence of Single Level Pointers 165

C.3.1. Terminology 165

C.3.2. Building the Pointer Alias Graph (PAG) 165

C.3.3. Algorithm for Computing Precise May Alias Sets in the Presence of Single Level Pointers 169

C.3.4. Proof of Lemma 4.2.1 171

C.3.5. Proof of Theorem 4.2.3 175

C.4. A Precise Polynomial Time Algorithm for Computing Interprocedural Must Alias Sets in the Presence of Single Level Pointers 189

C.4.1. Lattice of Conditional Must Alias Sets and Definitions 189

C.4.2. Algorithm for Computing Precise Conditional Must Alias Sets . 190

C.4.3. Algorithm for Computing Precise Must Alias Sets in the Presence of Single Level Pointers 192
C.4.4. Conditional Must Alias Sets 193
Relationships between $COND_{node}^{alias\text{-}pair}$ values 194
C.4.5. Proof of Correctness of Precise Conditional Must Alias Algorithm 200
C.4.6. Proof of Theorem 4.2.4 211
C.5. Replacing reference formals with pointers 214

Appendix D. Interprocedural May Alias Approximate Algorithm Pseudo Code 234
D.1. Approximate-when-both-non_visible 235
D.2. Alias_at_call_implies 237
D.3. Alias_at_exit_implies 239
D.4. Any_other_alias_implies 241

Appendix E. Safe Approximate Algorithm 244
E.1. Types of Approximation 244
References 251
Vita 255

List of Tables

4.1. Alias problem decomposition and classification 27

6.1. Number of Aliases: Comparison to Weihl 119

6.2. Time: Comparison to Weihl 120

6.3. Number of Aliases and Time: Comparison to Weihl 120

6.4. Size of our May Alias Solution ($k = 2$) 123

6.5. Size of our May Alias Solution 125

6.6. Precision of our May Alias Solution 129

6.7. Size of our May Alias Solution vs Size of *may-hold* 130

6.8. Number of Assumptions per Alias ($k = 2$) 133

6.9. Number of Assumptions per Alias 134

6.10. Timings for our Algorithm ($k = 2$) 135

6.11. Timings for our Algorithm 136

6.12. *may-hold*/sec 137

List of Figures

2.1. A C program and its ICFG 8

2.2. Syntax-directed definition for object names for a given program 10

2.3. Some examples of *apply_trans* 12

3.1. Bad case for Weihl's Algorithm 17

4.1. Polynomial time algorithm for **Intraprocedural May Alias** given one level pointers 31

4.2. Polynomial time algorithm for **Intraprocedural Must Alias** given one level pointers 32

4.3. *holds* at a return node (no local variables) 36

4.4. Calls affecting alias pairs involving non-visible object names 37

4.5. *holds*([(ICFG-node, assumed-alias), possible-alias]) for the program in Figure 2.1 41

4.6. Polytime algorithm for **Interprocedural May Alias** in the presence of single level pointers 42

4.7. *may-alias*(ICFG-node) for the program in Figure 2.1 43

4.8. Conditional Must Alias: an example 44

4.9. Lattice for possible values of $COND_{node}^{alias\text{-}pair}$ 45

4.10. *must-holds*(ICFG-node,possible-alias) for the program in Figure 2.1 . . 47

4.11. Polynomial time algorithm for **Interprocedural Must Alias** in the presence of single level pointers 49

4.12. *must-alias*(ICFG-node) for the program in Figure 2.1 49

4.13. 3-SAT solution iff [L3,$\langle *false, no \rangle$] in Intraprocedural May Alias 52

4.14. 3-SAT solution iff [L3,$\langle *false, no \rangle$] in Interprocedural May Alias 55

4.15. 3-SAT solution iff [L3,$\langle *false, no \rangle$] in Interprocedural May Alias 56

4.16. 3-SAT solution iff [L3,$\langle *(false.next), no\rangle$] in Intraprocedural May Alias 58

4.17. 3-SAT solution iff [L3,$\{\langle *c_i, true\rangle \mid 1 \leq i \leq n\}$] in Intraprocedural May Set Alias 61

4.18. 3-SAT solution iff [L3,$\{\langle c_i, true\rangle \mid 1 \leq i \leq n\}$] in Interprocedural May Set Alias 64

4.19. 3-SAT solution iff [L3,$\{\langle c_i, true\rangle \mid 1 \leq i \leq n\}$] in Interprocedural May Set Alias 65

4.20. Reduction of $\mathcal{L}$ to an alias problem 68

4.21. Justification of code for state $\mathtt{s}_j$ with 2 in-edges 72

5.1. Computing the Consequences of an Alias 81

5.2. Calls affecting alias pairs involving non-visible object names 83

5.3. Computing bind(*call*,$\emptyset$) 85

5.4. Support functions for computing bind(*call*,$\langle x, y\rangle$) 86

5.5. Computing bind(*call*,$\langle x, y\rangle$) 87

5.6. Computing may-hold 88

5.7. Aliases_introduced_by_assignment(*node*) 89

5.8. Aliases_introduced_by_call(*node*) 89

5.9. *holds* relation at return nodes 91

5.10. Creation of alias between two non-visible object names 95

5.11. *may-hold* representation for interesting stores of Figure 5.10 96

5.12. Implication of a known *may-hold* at an exit node 99

5.13. Action for *may-hold*($[(node, assumed_alias), \langle y, z\rangle]$) for successor, *succ* $\equiv$ "$p = q$", of *node* and is_prefix_with_deref(q,y) 103

5.14. Action for *may-hold*($[(node, assumed_alias), \langle p, v\rangle]$) for successor, *succ* = "$p = q$" of *node* 106

6.1. Program which yields very imprecise results 110

6.2. Description of programs in our empirical study 122

Chapter 1

Introduction

Programming language environments contain tools to improve the quality, efficiency, understandability, and usability of code. Tools such as optimizers, debuggers, testers, verifiers, and parallelizers, use *data flow analysis* [Hec77] to statically extract semantic information from programs so as to increase their efficacy. Alias information is important semantic information that can greatly affect the quality of optimized code and the utility of many other program languages tools.

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

Although the calculation of aliases for FORTRAN is well understood [Ban79, Coo85, CK89, Mye81], we show that general pointers added as a language construct cause the problem of computing aliases to become $\mathcal{NP}$-hard, a situation for which no good approximation algorithms exist. Moreover, aliases complicate most data flow analysis problems, and the absence of good alias information can prevent many optimizations. For example, consider the following example:

```
s0: i = 0;
    *p = 0;
    while (i < 2)
    {
s1:     x = 5;
s2:     *p = *p + x;
        i = i + 1;
    }
s3:
```

We would like to optimize the above code using code motion [ASU86] to:

```
s0: i = 0;
    *p = 0;
s1: x = 5;
    while (i < 2)
    {
s2:     *p = *p + x;
        i = i + 1;
    }
s3:
```

If *p is not aliased to x at s_0, both versions of the code will have x = 5 and *p = 10 at s_3. However, if *p is aliased to x at s_0, the original version would have x = *p = 10 at s_3, but the "optimized" version would have x = *p = 20 at s_3. Without alias information, this and many other optimizations can not be done.

Originally, we became interested in this problem while working on ISMM [Ryd89], an incremental data flow analyzer which solves the *modification side effect problem (MOD)* [Ban78, Ban79, Bur90, CK84, CK87b] for C programs and is designed to help programmers maintain large evolving software systems. MOD is an *interprocedural data flow problem*, meaning that we determine semantic information about programs across

procedure boundaries[1]. In MOD we are determining all the variables in the program whose value can change (i.e., can be modified) by execution of a statement, procedure invocation, or procedure. The MOD problem is traditionally decomposed into simpler problems, which, when combined, yield the MOD solution. ISMM is an incremental implementation of the standard FORTRAN decomposition for the MOD problem [CK84, CK87a, Bur90]. However, the difference between C and FORTRAN turned out to be a major theoretical difficulty. One of the major components of the MOD decomposition is determining aliases. The MOD solution is obtained by first ignoring alias effects and afterward factoring them to obtain the full solution. Unfortunately, the C alias problem has little in common with the FORTRAN alias problem, and it seems unlikely that this decomposition will be useful for C.

In FORTRAN, the only dynamic method for creating aliases is through the use of reference formals. Solving for aliases in the presence of pointers presents several additional complications. With 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, since aliases can be affected by pointer assignments in the called procedure. In addition, with reference formals, a call to a procedure cannot affect aliases in the calling procedure, but if pointers are present, this is no longer true. These facts suggest that existing FORTRAN alias algorithms are not extensible to handle pointers.

Existing Alias Algorithms Originally we tried using an extant alias technique [Wei80a] for finding aliases with pointers combined with the FORTRAN decomposition for solving MOD. Using this information resulted in a solution that safely claimed, for all practical purposes, every statement could modify every variable. This was the result of using an inappropriate decomposition for MOD for C and the fact that the extant aliasing technique [Wei80a] was too approximate (see Chapter 3). Since MOD and many other data flow solutions are extremely useful, especially in optimization, we decided to derive a better algorithm for determining aliases in C and reformulate MOD

[1] In contrast, an *intraprocedural data flow problem* [Hec77] is one that concerns semantic information within procedures and does not deal with procedure calls.

for the C paradigm. We are still looking at approaches for MOD; the rest of this thesis is devoted to the problem of solving for aliases.

Theoretical Complexity Results As a starting point, we examined the theoretical difficulty ($\mathcal{P}$ vs $\mathcal{NP}$-hard) of various aspects of the alias problem. Our results show which aspects of alias problems are provably hard and need to be approximated. A clear understanding of what makes the alias problem difficult lends insight into where information is lost by approximation and whether that loss of information is necessary. Also, an understanding of how the easier alias problems can be handled precisely can be useful as a framework for good approximations to harder alias problems.

In Chapter 4, 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 values are pointers), and 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 aliases introduced by a single level of indirection can be found in polynomial time.

An Alias Approximation Algorithm for C In Chapter 5 we present a detailed description of our algorithm for finding a safe approximation for Interprocedural May Alias in the presence of general pointers and call-by-value parameter passing. For efficiency, our algorithm is *demand driven* (i.e., we calculate information only for aliases that hold on some path) and must maintain data structures for quickly accessing certain kinds of information about the alias solution. For safety, our algorithm must account for various sources of approximation, and never miss an alias that occurs on some path.

The main ideas of our algorithm are highlighted in the text and the interested reader can find pseudo-code for the algorithm in the figures and Appendices. Our algorithm is program-point-specific and thus more precise than Weihl's algorithm [Wei80a]. We currently have a prototype implementation in C for analyzing C programs.

Precision[2] and Efficiency of Approximate Algorithm In Chapter 6 we give some theoretical bounds on the precision of our algorithm and for any other approximation algorithm. We show that for at least one definition of precision, no algorithm can be more precise than ours in the worst case. We are also able to pinpoint all sources of approximation and thus empirically bound the imprecision of our algorithm. Interestingly, all the sources are intraprocedural. That is not to say that there can be no interprocedural approximation, but that any possibly erroneous interprocedural alias that is included for safety depends on a possibly erroneous intraprocedural alias that was also included for safety[3]. In this chapter, we also discuss our implementation of the algorithm presented in Chapter 5 and give preliminary empirical results that, while not conclusive, are very encouraging.

The remainder of the thesis contains the supporting infrastructure. Chapter 1 (this chapter) contains the introduction. Chapter 2 contains our representation of the alias problem along with necessary definitions. Chapter 3 overviews other work in related areas. Chapter 7 summarizes the contents of the thesis and gives directions for future research. Appendix A is a glossary of functions used within the thesis, and the remainder of the Appendices are formal proofs of various theorems and lemmas.

Contributions The main contributions of this thesis are:

- A better understanding of the difficulty of detecting aliases. In particular, a realization that the multiple levels of indirection is what makes the alias problem difficult (Chapter 4, p. 24).

- Presentation of a precise[4] solution for an interprocedural flow sensitive problem (Interprocedural May Alias in the presence of single level pointers; Chapter 4.2.3). This algorithm is used by [PRL91].

- Formulation of the *Conditional May Alias* problem (Chapter 4.2.3, p. 32), which we feel maybe more useful than May Alias for data flow analysis in the presence of

[2] *Precision* is a measure of the erroneous information contained in the solution.

[3] However, it is possible for the intraprocedural alias to be in the precise solution and the interprocedural alias not to be.

[4] under the standard assumptions of data flow analysis

pointers. The work in [PRL91] indicates that this indeed was the case for *reaching definitions*, and we are currently investigating the suitability of Conditional May Alias information for MOD.

- Development of an approximation algorithm for Interprocedural May Alias in the presence of pointers (Chapter 5) that is provably as precise as possible in the worst case (Chapter 6.1). We empirically compared our approximate algorithm to Weihl's algorithm [Wei80a]. Not surprisingly, we found that our algorithm was slower but produced a more precise solution (Chapter 6.3).

- Isolation of sources of approximation for our algorithm (Lemma E.1.1 (p. 245)). This allows us to bound empirically the imprecision of our algorithm (Chapter 6.4.2) and, perhaps more importantly, gain a better understanding of where other algorithms must introduce imprecision.

Chapter 2

Problem Representation

2.1 Interprocedural Control Flow Graph

In this paper, we are dealing with C-like languages: imperative; sophisticated pointer usage and data structures; explicit function calls (without function variables). We allow arrays but simply treat them as aggregates.

We represent programs by *interprocedural control flow graphs* (ICFGs) that we originally presented in [LR91]. An ICFG is, intuitively, the union of the control flow graphs (CFGs)[5] [Hec77] 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 2.1 for an example of an ICFG.

2.2 Types

An object is a location that can store information (for example, variables). Objects in C have types, and we need to perform a handful of operations on types. They are all

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

```
int *q;
void A(f)
  int *f;
{
  q = f;
}
main()
{
  int p,*r;
  r = NULL;
  A(q);
  r = q;
  A(&p);
}
```

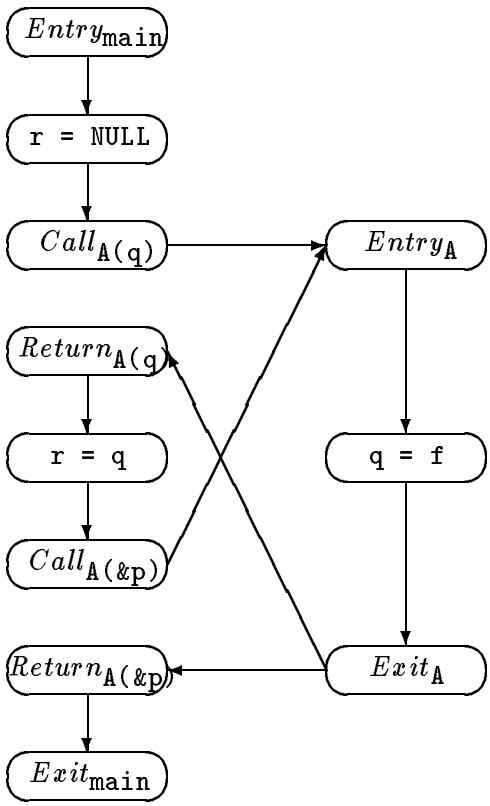


Figure 2.1: A C program and its ICFG

straightforward, but will be listed here to avoid confusion:

address_type(*type***)** returns the type of objects which can point to *type*.

can_deref(*type***)** is *true* iff *type* can be legally dereferenced without casting.

deref_type(*type***)** If *type* can be dereferenced, **deref_type** returns the type of the objects to which *type* may point.

field_type(*type*,*field***)** If *type* is a structure with *field*, **field_type** returns the type of *field*.

is_field_of(*field*,*type***)** is *true* iff *type* is a structure type and *field* is a legal field of *type*.

is_struct(*type***)** is *true* iff *type* is a structure.

2.3 Object Names

Because *objects* are locations that can store information, in (terminating) programs containing recursive data structures there are arbitrarily many potentially addressable objects. For example, in a program with a linked list *llist*, $llist(->next)^*$ are all possible names for distinct run-time objects. Thus, any practical alias algorithm will have to represent the set of all possible objects and the alias relationships between those objects with a (small) finite data structure. We use a solution that is roughly analogous to *k-limited* as defined by Jones and Muchnick[JM79]. Less naive schemes have be developed [CWZ90, HN89, HPR89, LH88], but we have yet to examine their suitability for our purposes.

Object names provide ways to refer to objects in a program. An object name is a variable and a (possibly empty) sequence of dereferences and field accesses. A syntax-directed definition [ASU86] for object names is in Figure 2.2. Because we have prohibited casting, we are interested only in object names which do not have type *error*.

If there are any recursively defined data structures (for example, linked lists) then the number of object names that do not have type *error* is infinite. However, we can not deal with an infinite number of object names in a finite amount of time, so we

Rule	Production	Semantic Rules
1	$OBJECT_1 \rightarrow$ $*(OBJECT_2)$	if can_deref($OBJECT_2$.type) then $OBJECT_1$.type := deref_type($OBJECT_2$.type) else $OBJECT_1$.type := *error*
2	$OBJECT_1 \rightarrow$ $(OBJECT_2)$.**field**†	if is_field_of(**field**,$OBJECT_2$.type) then $OBJECT_1$.type := field_type($OBJECT_2$.type,**field**) else $OBJECT_1$.type := *error*
3	$OBJECT \rightarrow$ **var**‡	$OBJECT$.type := type of **var**

† **field** is any declared field of any structure in the program.
‡ **var** is any variable declared in the program.

Figure 2.2: Syntax-directed definition for object names for a given program

will limit to some constant, k, the number of applications of the rule 1 of Figure 2.2. This will restrict the number of object names which do not have type *error* that can be generated to a finite number $[\mathcal{O}(number_vars * (max_number_fields_in_one_struct)^k)]$. This raises the issue of how to deal with object names with more than k dereferences. We do this rather simplistically by considering any object name with $l > k$ dereferences to be represented by the object name obtained by ignoring the last $l - k$ dereferences yielding a unique k-limited name. Thus, for $k = 1$, $p\text{->}f_1\text{->}f_2$ would be represented by $p\text{->}f_1$ (and not by $*p$). We will borrow Jones and Muchnick [JM79] terminology and call this *k-limiting*, even though they *k-limit* dynamic structures while we *k-limit object names*, because the two processes are analogous.

Unfortunately, despite being interested in aliases between object names, there is one construct which is not an object name but can affect alias information, the address operator (for example, $\&p$). Thus, we will define a class **object_name** which is either an *object name* or an $\&$(*object name*).

To state the algorithm concisely and accurately in pseudo-code, we need the following non-trivial functions for class object_name:

is_prefix(*object_name*$_1$, *object_name*$_2$**)** returns *true* iff *object_name*$_1$ can be transformed into *object_name*$_2$ by a (possibly empty) sequence of dereferences and field accesses (i.e. applications of rule 1 and rule 2 of Figure 2.2).

is_prefix_with_deref($object_name_1$, $object_name_2$**)** returns *true* iff $object_name_1$ can be transformed into $object_name_2$ by a sequence of dereferences and field accesses (i.e. applications of rule 1 and rule 2 of Figure 2.2) with AT LEAST ONE dereference.

apply_trans($object_name_1$,$object_name_2$,$object_name_3$**)** : $object_name_1$ and $object_name_3$ must have the same type and $is_prefix(object_name_1, object_name_2)$ must be *true*. The function applies to $object_name_3$ the sequence of dereferences and field accesses necessary to transform $object_name_1$ into $object_name_2$. It returns *true* iff any dereference occurs somewhere in the sequence. Some examples of *apply_trans* can be found in Figure 2.3.

We will also use the following, fairly straightforward functions:

amp_object_name(*object_name***)** is *&(object_name)*, if *object_name* does not start with an &, otherwise *error*.

deref(*object_name***)** removes the & if *object_name* starts with an & and otherwise returns $*(object_name)$. This corresponds to rule 1 of Figure 2.2.

field_access(*object_name,field_name***)** is *(object_name).field_name*, if *object_name* does not start with an &, otherwise error. This corresponds to rule 2 of Figure 2.2.

object_type(*object_name***)** is the type of data that can be stored in objects with *object_name*,[6] if *object_name* does not start with "&". If, for some *name*, $object_name = \&name$ then it is address_type(*name*).

is_k-limited(*object_name***)** is *true* iff object_type(*object_name*) can be dereferenced, *object_name* contains at least k dereferences, and *object_name* does not start with an address operator.

simple_object_name(*variable***)** is the object name which is just *variable*, i.e. rule 3 of Figure 2.2.

---

[6] This is the same as the value of the type attribute for *object_name* obtained from Figure 2.2.

$object_name_1$	$object_name_2$	initial $object_name_3$	final $object_name_3$	return value
p	$*p$	$\&q$	q	*true*
$p->next$	$p->next->next$	$\&q$	$q.next$	*true*
$*(p->next)$	$p->next->data$	q	$q.data$	*false*
$*(p->next)$	$p->next->data$	$*r$	$r->data$	*false*
$p->next$	$p->next->data$	r	$r->data$	*true*
$p->next$	$p->next->next$	$q.next$	$(q.next)->next$	*true*
$p->next$	$p->next->next$	$r->next$	$r->next->next$	*true*

Figure 2.3: Some examples of *apply_trans*

2.4 Aliases

An alias occurs when two or more object names refer to the same location at some point during program execution. As in [LR91] we will represent aliases by unordered pairs of object names (for example, $\langle v, *p \rangle$). The order is unimportant because the alias relation is symmetric. Since we have k-limited object_names, in order to represent aliases safely we must assume that an alias $\langle a, b_k \rangle$ with k-limited component b_k, represents not only the alias $\langle a, b_k \rangle$ but also any alias $\langle a, b'_k \rangle$ such that b_k can be transformed into b'_k by a sequence of dereferences and field accesses (i.e., is_prefix(b_k,b'_k) = $true$). Also an alias $\langle a_k, b_k \rangle$ with two k-limited components represents not only itself, but all aliases $\langle a'_k, b'_k \rangle$ such that a_k is a prefix of a'_k and b_k is a prefix of b'_k.

An alias $\langle a, b \rangle$ represents the fact that a and b refer to the same object. In C objects have types and since we are not allowing casting it would seem that the type of the object would be the type of both a and b. However, with k-limiting this is not necessarily true. Say there is an alias, at a program point, $\langle x, **y \rangle$ where x is type "int" and y is type "int **". If we are k-limiting with $k = 1$ this alias is represented by $\langle x, *y \rangle$. We use the function *alias_type* to determine the type of the object represented by an alias.

alias_type(*alias***)** is the type of the object which is referred to by both object names in *alias*. This function correctly accounts for k-limiting. For example, alias_type($\langle x, *y \rangle$) is "int" for $k = 1$ in the above example.

2.5 Some Terminology

The following are definitions which will be used throughout the paper:

program point: We use **program point** to refer to the part of a program represented by an ICFG node.

realizable: A path is **realizable** iff it is a path in the ICFG[7] and whenever a procedure on this path returns, it returns to the call site which invoked it. By definition all intraprocedural paths are vacuously realizable.

holds: Alias $\langle a, b \rangle$ **holds** on the realizable path $\rho n_1 n_2 ... n_i$ iff a and b refer to the same location after execution of program point n_i whenever the execution sequence defined by the path occurs. Aliases are symmetric, that is $\langle a, b \rangle$ holds on a path iff $\langle b, a \rangle$ also holds on that path; we will not distinguish between $\langle a, b \rangle$ and $\langle b, a \rangle$.

Interprocedural May Alias: The precise[8] solution for **Interprocedural May Alias** is $\{[n, \langle a, b \rangle] \mid \exists$ a realizable path, $\rho n_1 n_2 ... n_{i-1} n$, in the ICFG on which $\langle a, b \rangle$ holds$\}$.

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

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

Intraprocedural Must Alias: The precise solution for **Intraprocedural Must Alias** is $\{[n, \langle a, b \rangle] \mid \forall$ paths, $\rho n_1 n_2 ... n_{i-1} n$, in the CFG $\langle a, b \rangle$ holds$\}$.

complement of Interprocedural Must Alias: The precise solution for **complement of Interprocedural Must Alias** is $\{[n, \langle a, b \rangle] \mid \exists$ a realizable path, $\rho n_1 n_2 ... n_{i-1} n$, in the ICFG on which $\langle a, b \rangle$ does not hold$\}$.

---

[7] CFG in the intraprocedural case.

[8] 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 [Ban79].

complement of Intraprocedural Must Alias: The precise solution for **complement of Intraprocedural Must Alias** is $\{[n, \langle a, b\rangle] \mid \exists$ a path, $\rho n_1 n_2 ... n_{i-1} n$, in the CFG on which $\langle a, b\rangle$ does not hold$\}$.

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

Chapter 3

Related Work

There are three main areas of work that are related to the topics in this thesis. The first deals with the theoretical classification of various portions of the alias problem as $\mathcal{NP}$-hard or $\mathcal{P}$. The second concerns other alias algorithms in the presence of pointers. The third is conflict/dependence detection which, while not identical to alias analysis and not directly addressed in this thesis, is similar enough to warrant comment.

3.1 Theoretical Classification

Myers [Mye81] proves that certain data flow problems in the presence of aliases are $\mathcal{NP}$-complete. He considers FORTRAN-like aliasing (i.e., aliasing that is the result of passing call-by-reference parameters); however, he did not classify the alias problem itself. All our $\mathcal{NP}$-hardness proofs are variations of Myers' proof. Larus [Lar89] also modified Myers' proof to prove that intraprocedural aliasing in the presence of structures is $\mathcal{NP}$-hard. Owing to their common origin, Larus' and our $\mathcal{NP}$-hardness proofs are similar.

Many algorithms have been developed to solve for aliasing in the presence of call-by-reference formals (i.e., aliasing in FORTRAN). Myers [Mye81] presents an algorithm for finding alias sets which is precise under the traditional assumptions of static analysis [Ban79]. If an alias set $\mathcal{S}$ is in Myers' solution at a program point p, then on some path to p, every (and only) the aliases represented in $\mathcal{S}$ exist at p. Since Myers' solution is a set of alias sets, its size can be exponential in the number of variables in the program. To avoid this problem, later algorithms, including our algorithm, compute alias pairs instead of alias sets, because the number of alias pairs is polynomial in the number of variables in the program. Some precise polynomial time algorithms for finding aliases

in FORTRAN programs are [Ban79, Bur90, Coo85, CK89]. [MR91] presents an incremental algorithm for finding aliases in the presence of call-by-reference parameters. This work is noteworthy because it uses a mechanism similar to our assumed alias in Chapter 4.2.3, but instead of assuming an alias at entry of a procedure, they assume an alias at the head node of a strongly connected component of the call graph.

3.2 Algorithms for Finding Aliases in Programs with Pointers

One of the first algorithms for finding aliases in the presence of pointers was developed by Weihl [Wei80a, Wei80b]. Weihl finds program aliases (aliases on some path in the program) rather than program point aliases (aliases on some path to a specific program point) and approximates object names using 1-limiting. His solution is fairly simple. For each possible type of pointer assignment he adds the aliases introduced by that assignment to a set (AFFECT). For example, for the statement "$p = q$" he would add the pair $\langle *p, *q \rangle$ to AFFECT. Once this has been done for each statement in the program, he then computes the alias solution as being $\text{AFFECT}^* \circ (\text{AFFECT}^*)^\top$.

Unfortunately, Weihl's algorithm is very imprecise. In fact on a simple program (straight line code: see Figure 3.1) which has $\mathcal{O}(n)$ aliases, where n is the number of variables in the program, Weihl reports $\mathcal{O}(n^2)$ aliases. We have solved the modification side effect problem on several C programs using Weihl's algorithm to determine the aliases. Our analysis generally reported that almost all variables could be modified. This was because Weihl's algorithm does not give information of sufficient quality to be useful for this problem. Some empirical observations on the size of the alias solutions produced by Weihl's algorithm can be found in Figure 6.1 in Chapter 6.

Chow and Rudmik [CR82] also presented an algorithm for finding aliases in the presence of single level pointers. Their approach is in some sense a generalization of Weihl's approach to computing program point aliases instead of program aliases. Like Weihl, they start out making a pass through each statement in the program and adding the direct affects to the initial alias solution. For example, for "$p = q$", $\langle *p, *q \rangle$ would be added to the alias solution at the program point "$p = q$". They also define how the alias solution at one node is related to the alias solution at its immediate predecessors.

```
int *temp;
int *a_1, *a_2, ..., *a_{n-1};

main ()
{
        temp = a_1;
        a_1 = temp;
        temp = a_2;
        a_2 = temp;
              ⋮
        temp = a_{n-1};
        a_{n-1} = temp;
}
```

Program has $2(n-1)$ *non-reflexive alias pairs* $(\bigcup_{1 \leq i \leq n-1} \{\langle *temp, *a_i \rangle, \langle *a_i, *temp \rangle\})$ *where* n *is the number of variables in the program.*

Weihl reports the set of all possible alias pairs, $n^2 - n$ *non-reflexive alias pairs.*

Figure 3.1: Bad case for Weihl's Algorithm

Their solution is simply the maximum fixed point of these rules. Chow and Rudmik's algorithm suffers because they treat interprocedural alias as an intraprocedural problem. That is, in their analysis they do not insure that, when a procedure returns, control passes to the call site which invoked it. In addition, they handle local variables incorrectly.

[ASU86] presents an intraprocedural algorithm for finding aliases in the presence of single level pointers which is similar to the intraprocedural portion of Chow and Rudmik's algorithm, except that they allow arrays treated as aggregates. It is also similar to the our intraprocedural algorithm presented in Chapter 4.2.

Coutant [Cou86] extended Weihl's work in a different direction. She kept his restriction of finding program aliases but relaxed his restriction to 1-limiting and added additional language constructs (for example, structures and arrays). She doesn't explicitly mention recursive data structures, but it doesn't appear that she can handle them unless she is doing some type of k-limiting for an arbitrary k. Coutant's algorithm follows the same general pattern as Weihl's; she computes initial information by a statement by statement pass through the program and then does a transitive closure. In addition to her other extensions to Weihl, Coutant also considers the use of *pragmas*, user provided information about aliasing, to improve the precision of her aliasing algorithm.

Benjamin Cooper [Coo89] 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. His alias histories are based on [Mye81], and consist of the last call site on the execution stack and a set of aliases that hold on the path to the entry of the procedure. This bears some resemblance to our assumed alias (Chapter 4.2.3) except that we do not maintain the top of the execution stack and we restrict our alias sets at entry to size one (or zero). Cooper's method seems time and space infeasible without restricting the size of the alias sets.

There also has been some work [Deu90, NPD87] in detecting aliases in higher order programming languages (i.e., languages where functions are treated like any other data type). [NPD87] only considers programs with single level dereferences and has the

added difficulty of tracking the binding of functions to names. Interprocedurally, they solve the alias problem for a procedure with the initial information induced by each call chain[9]. They fail to take advantage of using assumed alias pairs instead of initial alias sets. The problem addressed by [Deu90] is another order of magnitude complication over general aliasing. He allows closures (partially evaluated functions) and continuations (storing of runtime environment for later reuse). [Deu90] uses a very formal *abstract interpretation* [CC77] and is difficult to read.

3.3 Conflict/Dependence Analysis

A related area of research is the work done by the compiling community on dependence analysis and conflict detection in programs with recursive structures [CWZ90, Gua88, HA90, HN89, HPR89, JM82, LH88]. This is particularly important for parallelizing programs.

A *conflict* [LH88] occurs between two statements when one statement writes a location and the other accesses (reads or writes) the same location (*loc*), thus preventing the possibility of those statements being executed in parallel. A data dependence exists between two statements iff they conflict and there is an execution path from one program point to the other on which *loc* is not written. There are actually three types of data dependences [PW86, Wol89]; flow dependences, anti-dependences, and output dependences. The type of dependence is determined by the direction of the path and which endpoint(s) write *loc*.

Algorithms for detecting dependences (or conflicts) need more information than an alias calculation. In aliasing, we need only concern ourselves with the relationship between object names *at a program point*. In dependence analysis, in general, the relationship between object names at different program points must be maintained. Another fundamental difference is that alias pair information is not sufficient for parallelizing compilers, the target application for much of the dependence analysis community, because detailed information about the whole data structure (and not just isolated parts

[9] recursion requires a fixed point calculation

of the data structure) is needed to parallelize loops. Thus dependence analysis and alias analysis have different goals and restrictions placed on them.

We will concentrate on papers in the dependence analysis community that deal with recursive data structures. Jones and Muchnick [JM79] first analyzed how to track dynamic LISP-like structures; most research in this area is based in some way on the ideas in [JM79]. They restrict themselves to an intraprocedural domain, and use graphs to represent the structure of dynamic memory. Nodes in the graph represent locations in memory and (directed) edges represents the fact that one location refers to the other. Each program point will have associated with it a set of such graphs. G associated with statement s means that there is a path to s for which G correctly represents the dynamic store. Unfortunately, there are an infinite number of possible graphs. They introduce the notion of *k-limiting*, which only allows nodes that are at most k dereferences away from a labeled node. Summary nodes are used to represent the portions of the graph that must be collapsed to meet the requirements of k-limiting. k-limiting effectively reduces the number of graphs to a finite number. We have borrowed this notion from them and use it in our algorithm.

In [JM82] Jones and Muchnick show how to do interprocedural data flow analysis in the presence of recursive structures. They use *abstract interpretation* [CC77] to do this. They represent the possible execution states of a program by triples, $\langle q, e, t \rangle$ where:

- q is a program point
- e is an abstraction of the program store
- t is an abstraction of the runtime stack

They do this as a general framework and do not espouse any particular abstraction for the program store and the runtime stack. We have not coaxed our solution into their framework nor presented it as an abstract interpretation. Nevertheless, at a high level our alias algorithm can be viewed in their model as abstracting the program store with a set of (k-limited) alias pairs and abstracting the runtime stack with a single assumed alias pair (or with no alias assumption).

Larus and Hilfinger [LH88] developed an algorithm for conflict detection using [JM79]. They use an *alias graph* to represent the dynamic store. They extended the graphs in [JM79] to make alias graphs by adding unique labels to each node, thus making it easy to detect conflicts. Two statements conflict if one writes and the other reads (or writes) a node in their respective alias graphs with the same label. While the conflict detection becomes easy, the labeling processes is not trivial.

Unlike [LH88], Horwitz, Pfeiffer, and Reps [HPR89] are interested in intraprocedural detection of dependencies. They also do this by augmenting [JM79] graphs, but, whereas unique labels were added in [LH88], here each node is labeled with the allocation site that last assigned to the storage represented by the node. This allows easy dependence detection. [HPR89] also use abstract interpretation as a framework for presenting their solution. However, they extended [CC77] by adding a fourth semantics; the *instrumented semantics*. This was done to prove the validity of their abstraction. It is unclear whether this extension will have utility outside of their paper.

Hendren and Nicolau [HN89, HN90] take a different approach. They assume that something is known about the underlying data structure; for example, it is a linked list, tree, dag, etc... This assumption is necessary for their analysis, but is also verified by their algorithm. Also, instead of using graphs to represent the dynamic store, they use *path matrices* whose entries are *path expressions*.[10] The rows and columns of their path matrices are all the *handles* (variables) of the program. The path expression in row h_r and column h_c represents the relationship (if any) between h_r and h_c. They are restricted regular expressions[11].

[HN89, HN90] is an interprocedural algorithm and in the absence of recursion[12] is interprocedurally analogous to our assumed alias algorithm. However, they have what amounts to a assumed path matrix at the entry of the procedure and we would have what was basically an assumed individual entry of that matrix. This is a loose

[10] Of course these matrices can be viewed an adjacency matrices, and thus viewed as graphs with labeled edges.

[11] They need to be restricted so that equality of path expressions can be computed in time polynomial in the size of the expressions.

[12] and function calls within intraprocedural loops

analogy; they don't use have assumed path matrices, but rather propagate path matrices along execution paths, and like [NPD87] effectively analyze a procedure once for each calling chain which invokes it. However, if our idea of limiting the assumption to a single alias assumption is valid for their problem[13], then both these methods would produce the same solution, but ours would do so more efficiently. We should be able to reduce the amount of redundant calculation being done by not having to re-do the calculation for identical portions of different assumed path matrices. In the presence of recursion[14], [HN89, HN90] would use what amounted to an assumed condition (p_{in} in their nomenclature) which is the meet of all the conditions on entry to the procedure for that recursive invocation. For us, a recursive function call is treated no differently than an non-recursive call and we avoid the imprecision of meeting the various conditions that hold at entry to the recursive procedure.

Chase, Wegman, and Zadeck [CWZ90] use *storage shape graphs* to represent the dynamic store, but they change the notion of k-limiting. The original idea of k-limiting was simply to represent everything further than distance k from a labeled node by a summary node. However, if the data structure being represented was a linked list of even length, the even length information would be lost by such a representation. They propose a method of "limiting" graphs that would preserve such information. It seems possible to view [CWZ90] storage shape graph as some type of finite state automaton. We are not sure how this relates to [HN89, HN90]'s limited regular expressions which are used to represent path expressions. [CWZ90] also gives an algorithm for determining if an underlying data structure is a list or tree, and seems to agree with [HN89, HN90] that such information can be gainfully exploited.

Guarna [Gua88] uses trees to represent what we defined as object names. Where we defined object names by the set of names derivable from a grammar, Guarna uses the actual parse trees. He then redefines the traditional data dependence calculation [PW86] in terms of these trees. He does account for point-specific aliasing, but he does a transitive closure on his May Alias solution at each program point to insure safety.

---

[13] We have not as yet determined whether such an assumption would be viable for [HN89, HN90].

[14] and while loops

We think this is very imprecise, because, although aliasing on a path is a transitive relation, May Alias is not and the transitive closures will almost certainly introduce unnecessary imprecision.

Chapter 4

Classification of the Alias Problems

Problem Classification We have analyzed the theoretical difficulty of solving for aliases in the presence of reference formals, single level pointers, multiple level pointers, and structures containing single level pointers. We allow arrays but simply treat them as aggregates. The results of our analyses are shown in Table 4.1. Blanks in Table 4.1 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[15], 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 four distinct approximations in any practical alias algorithm. In any program that contains recursive data structures, there are a potentially 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[16]. The type of representation and its precision are what distinguishes the different conflict detection methods.

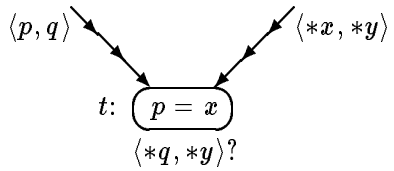
There is a second source of approximation illustrated by the following scenario. Suppose there is an assignment $p = x$ at program point t, alias pair $\langle p, q \rangle$ holds on some path[17] to an immediate predecessor of t and $\langle *x, *y \rangle$ also holds on some path to

[15]By a level of indirection, we mean using the value stored in a location as the address of another location.

[16]For example, *k-limited* as defined by Jones and Muchnick [JM79].

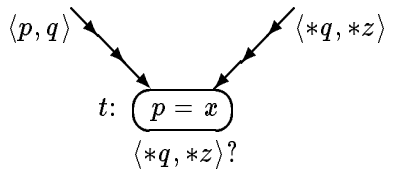
[17]Remember that *holds* is defined after execution of the last statement on the path.

an immediate predecessor of t. Does $\langle *q, *y \rangle$ hold on some path through t?



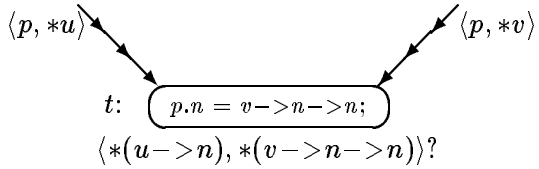
If both $\langle p, q \rangle$ and $\langle *x, *y \rangle$ occur on the **same** path, then $\langle *q, *y \rangle$ holds on that path extended by t; therefore, we must conclude this, even though it may not be true. Thus, to solve for alias pairs precisely, we need information about multiple alias 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 $\mathcal{O}(n)$ (the number of variables in a program) aliases hold (Theorem 4.7.1), some approximation must occur.

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



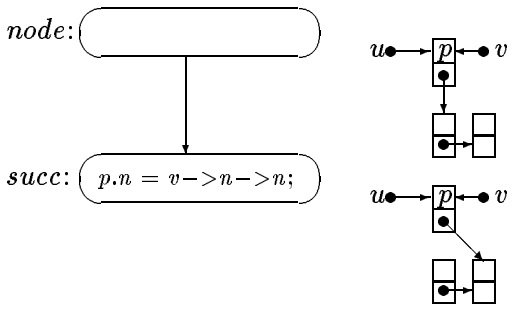
If on at least one path through *pred*, $\langle *q, *z \rangle$ holds and neither (or both) $\langle p, q \rangle$ nor $\langle p, z \rangle$ holds (on the same path), then $\langle *q, *z \rangle$ holds on that path extended by t. However, if on all paths through *pred* on which $\langle *q, *z \rangle$ holds, $\langle p, q \rangle$ also holds, then $\langle *q, *z \rangle$ does not necessarily hold on any path through t.

The fourth involves two distinct aliases of the LHS of an assignment.



Normally, $\langle *(u{->}n), *(v{->}n{->}n) \rangle$ should hold on a path through t because assigning $v{->}n{->}n$ to $p.n$ is also an assignment to $u{->}n$ on the path on which $\langle p, *u \rangle$ holds.

This, however, is not necessarily the case. If, for example, on the same path $\langle p, *v \rangle$ holds then $\langle *(u->n), *(v->n->n) \rangle$ does not necessarily hold:



This fourth kind of approximation is different from the other three in that it requires either recursive data structures or type casting to occur. Even in those cases, it seems likely that this approximation will rarely occur.

All $\mathcal{NP}$-hardness proofs are variations of proofs by Myers [Mye81]; a similar, although independently discovered proof for recursive structure aliasing (as indicated in Table 4.1) can be found in [Lar89]. All problems which are categorized as polynomial time are corollaries of proofs of Theorem 4.2.3 or Theorem 4.2.4. The rest of this chapter is a presentation of proofs of the theorems in Table 4.1.

4.1 Reference Formals

Theorem 4.1.1 *There exists a polynomial algorithm for determining precise Interprocedural* **May** *Alias sets in the presence of reference formals.*

Theorem 4.1.2 *There exists a polynomial algorithm for determining precise Interprocedural* **Must** *Alias sets in the presence of reference formals.*

The problem of determining Interprocedural May Alias sets in the presence of reference formals has been examined extensively. Most notable are the algorithms presented in [Ban79, Coo85, CK89, Mye81]. We do not present any formal proofs of Theorem 4.1.1 and Theorem 4.1.2, which follow directly from previous work. Both of these theorems are easy corollaries of Theorem 4.3.1 and Theorem 4.3.2.

Alias Mechanism	Intraprocedural May Alias	Intraprocedural Must Alias	Interprocedural May Alias	Interprocedural Must Alias
Reference Formals, No Pointers, No Structures	–	–	Polynomial [Ban79, Coo85] (Theorem 4.1.1)	Polynomial [Ban79, Coo85] (Theorem 4.1.2)
Single level pointers, No Ref. Formals, No Structures	Polynomial (Theorem 4.2.1)	Polynomial (Theorem 4.2.2)	Polynomial (Theorem 4.2.3)	Polynomial (Theorem 4.2.4)
Single level pointers, Reference Formals, No Pointer Reference Formals, No Structures	–	–	Polynomial (Theorem 4.3.1)	Polynomial (Theorem 4.3.2)
Mult. level pointers, No Ref. Formals, No Structures	$\mathcal{NP}$-hard (Theorem 4.4.1)	Complement is $\mathcal{NP}$-hard (Theorem 4.4.2)	$\mathcal{NP}$-hard (Corollary 4.4.1)	Complement is $\mathcal{NP}$-hard (Corollary 4.4.2)
Single level pointers, Pointer Reference Formals, No Structures	–	–	$\mathcal{NP}$-hard (Theorem 4.5.1)	Complement is $\mathcal{NP}$-hard (Theorem 4.5.2)
Single level pointers, Structures, No Ref. Formals	$\mathcal{NP}$-hard [Lar89] (Theorem 4.6.1)	Complement is $\mathcal{NP}$-hard (Theorem 4.6.2)	$\mathcal{NP}$-hard [Lar89] (Corollary 4.6.1)	Complement is $\mathcal{NP}$-hard (Corollary 4.6.2)

Table 4.1: Alias problem decomposition and classification

4.2 Single Level Pointers

Before we precede to our own results in this area, it should be mentioned that Chow and Rudmik [CR82] previously attempted to devise an algorithm for determining aliases in the presence of this mechanism. Their definition of "*precise Interprocedural May Alias*" is weaker than ours, in that we consider only realizable paths in the ICFG, while they consider all paths. Also their algorithm is incomplete.

4.2.1 Intraprocedural May Alias

The algorithm in this section is not the best way to solve for Intraprocedural May Alias in the presence of single level pointers. However, it is polynomial time and precise and is extensible to the Interprocedural May Alias problem. The algorithm below is explained in a manner to aid understanding; it is not an efficient implementation.

This algorithm was originally presented in [LR90]. However, the presentation in [LR90] contains the following implicit assumption that, while valid in practice, is not reasonable in the proofs contained in this chapter. Consider the statement, "$p = q$". In [LR90] we assumed that $\langle *p, *q \rangle$ holds on any path through "$p = q$", but this is only valid if q is not $NULL$. For example, if "$q = NULL$" was the only immediate predecessor of "$p = q$" then $\langle *p, *q \rangle$ would not hold on any path to "$p = q$". We consider the assumption that q will not be $NULL$ on some path to "$p = q$" reasonable in practice, and thus use it in our approximate algorithm (see Chapter 5). The algorithms in this chapter have been modified to avoid this assumption.

We use the reflexive aliases to keep track of non-$NULL$ pointers; i.e., $\langle *q, *q \rangle$ holds on a path P iff q is not $NULL$ after the execution sequence defined by P is taken. This is a very natural extension because $*q$ refers to the same location as $*q$ iff q contains some location and thus is non-$NULL$. This extension of holds implies that $[n, \langle *q, *q \rangle]$ is in *May Alias* iff on some path up to and including n, q is non-$NULL$. It also implies that $[n, \langle *q, *q \rangle]$ is in *Must Alias* iff on all paths to n, q is non-$NULL$. This means that the alias $\langle *q, *q \rangle$ in *May Alias* precisely captures the information about q's relation to $NULL$ that is needed to solve for *May Alias* and $\langle *q, *q \rangle$ in *Must Alias* precisely

captures the information about q's relation to $NULL$ needed for *Must Alias*. Note that $NULL$ has not been introduced into the class *object name* and can never occur in an alias pair. Thus the definitions in Chapter 2 are equally valid regardless of whether $NULL$ is explicitly considered or not.

Given the CFG=($\mathcal{N}$,$\mathcal{E}$,ρ) of the procedure to be analyzed, a **single procedure pointer alias graph** (SPPAG) is constructed. Every node in the SPPAG is of the form $[CFG\text{-}node, alias\text{-}pair]$; the fixed point value of $holds([CFG\text{-}node, alias\text{-}pair])$ is *true* iff there is a path from ρ to $CFG\text{-}node$[18] in the CFG on which *alias-pair* holds.[19] Edges in the SPPAG represent a dependence between the holds predicate at various SPPAG nodes. If $holds([n, \langle a, b\rangle])$ does not depend on any other $[CFG\text{-}node, alias\text{-}pair]$ pair then $[n, \langle a, b\rangle]$ has a single in-edge from ρ. Thus, if $holds([n, \langle a, b\rangle]) = holds([m_1, \langle a, b\rangle]) \vee holds([m_2, \langle a, b\rangle])$ then the edges $\ll[m_1, \langle a, b\rangle], [n, \langle a, b\rangle]\gg$ and $\ll[m_2, \langle a, b\rangle], [n, \langle a, b\rangle]\gg$ would be in the SPPAG. If $holds([n, \langle a, b\rangle]) = true$ regardless of any other *holds* value, then the edge $\ll\rho, [n, \langle a, b\rangle]\gg$ would be in the SPPAG.

Like Chow and Rudmik [CR82], we consider the relationship on an execution path between aliases that hold before a statement is executed and aliases that hold after it is executed. Consider any node $[n, \langle a, b\rangle]$[20] in the SPPAG. The following statements are true:

- if $a = b$ and a is not a dereferenced pointer variable

  Since a variable is aliased to itself on all paths, $holds([n, \langle a, a\rangle])$ must be true. All cases below assume that $a \neq b$ or a is a dereferenced pointer variable.

- if n is not an assignment to a pointer

  $holds([n, \langle a, b\rangle])$ is true iff there is a path to an immediate predecessor of n on which $\langle a, b\rangle$ holds.

- if n is "$p = q$" for p, q pointers

[18] Remember that a node in the CFG corresponds to a single statement.

[19] This implies that $holds(CFG\text{-}node, alias\text{-}pair)$ is calculated on exit from $CFG\text{-}node$.

[20] Remember that aliases are symmetric, thus a represents either *object name* in the alias and b is just the other name.

  - if $a = b = *p$ then $holds([n, \langle *p, *p \rangle])$ is true iff there is a path to an immediate predecessor of n on which $\langle *q, *q \rangle$ holds.
  - if a is $*p$ $[a \neq b]$ then $holds([n, \langle *p, b \rangle])$ is true iff there is a path to an immediate predecessor of n on which $\langle *q, b \rangle$ holds.
  - otherwise $holds([n, \langle a, b \rangle])$ is true iff there is a path to an immediate predecessor of n on which $\langle a, b \rangle$ holds.

- if n is "$p = \&v$"
  - if $a = b = *p$ then $holds([n, \langle *p, *p \rangle])$ is true
  - if a is $*p$ $[a \neq b]$ then $holds([n, \langle *p, b \rangle])$ is true iff there is a path to an immediate predecessor of n on which $\langle v, b \rangle$ holds.
  - otherwise $holds([n, \langle a, b \rangle])$ is true iff there is a path to an immediate predecessor of n on which $\langle a, b \rangle$ holds.

- if n is "$p = malloc()$"[21] or "$p = NULL$".
  - if $a = b = *p$ then $holds([n, \langle *p, *p \rangle])$ is $\begin{cases} true & \text{for "p = malloc()"} \\ false & \text{for "p = NULL"} \end{cases}$
  - if a is $*p$ $[a \neq b]$ then $holds([n, \langle *p, b \rangle])$ must be $false$ (i.e. $\langle *p, b \rangle$ does not hold on any path).
  - otherwise $holds([n, \langle a, b \rangle])$ is true iff there is a path to an immediate predecessor of n on which $\langle a, b \rangle$ holds.

The SPPAG, which is formally specified in Appendix B.1, is built to reflect the above rules. Our naive algorithm for computing Intraprocedural May Alias is given in Figure 4.1.

Theorem 4.2.1 *There exists a polynomial time algorithm for determining precise* **Intraprocedural May Alias** *sets in the presence of single level pointers.*

[21] We are mallocing a primitive type here, not a structure.

```
Build the SPPAG.
Initialize holds([n, ⟨a, b⟩]) to false for all [n, ⟨a, b⟩] in the SPPAG.
Calculate the fixed point of the holds relation.
```

Figure 4.1: Polynomial time algorithm for **Intraprocedural May Alias** given one level pointers

The claim is that the algorithm in Figure 4.1 is such an algorithm. It is polynomial because there are $\mathcal{O}(|\mathcal{N}| * v^2)$ nodes and edges in the SPPAG, where $\mathcal{N}$ is the node set of the CFG and v is the number of object names[22] which may have aliases, and in the fixed point calculation $holds(SPPAG\text{-}node)$ can change its value at most once on our $\{true, false\}$ lattice. The proof of precision is by induction on path length and induction on iteration of the fixed point calculation. A formal proof can be found in Appendix B.2.

□

4.2.2 Intraprocedural Must Alias

The Intraprocedural Must Alias algorithm (see Figure 4.2) is almost identical to the Intraprocedural May Alias algorithm. In the Intraprocedural May Alias algorithm, for each SPPAG node we specify an alias which must have held on some path to an immediate predecessor; by contrast in must alias the same alias must hold, not on one path but on all paths to immediate predecessors. This means that in the *holds* relation, we need to conjoin ($\bigwedge$), instead of disjoin ($\bigvee$), the predecessors of a node. Let $holds^{\wedge}$ be the *holds* relation defined by conjoining immediate predecessors. The only other difference between the may and must algorithms is that in the must alias algorithm, $holds^{\wedge}$ is initialized to *true*.

Theorem 4.2.2 *There exists a polynomial time algorithm for determining precise* **Intraprocedural Must Alias** *sets in the presence of single level pointers.*

The claim is that the algorithm is Figure 4.2 is such an algorithm. The proof is

[22]For programs with just single level pointers this is also the number of variables in the program.

```
Build the SPPAG.
Initialize holds^([n, ⟨a, b⟩]) to true for all [n, ⟨a, b⟩] in the SPPAG.
Calculate the fixed point of the holds^ relation.
```

Figure 4.2: Polynomial time algorithm for **Intraprocedural Must Alias** given one level pointers

similar to that for Theorem 4.2.1 and is formally proved in Appendix B.3.

□

4.2.3 Interprocedural May Alias

Interprocedural May Alias presents an additional complication. The obvious solution is to use the intraprocedural algorithm on the ICFG. This, unfortunately, introduces imprecision. The intraprocedural algorithm computes the MFP (maximum fixed point) for a distributive framework and thus computes the MOP (meet over all paths) solution [Hec77]. We cannot use this same approach in the interprocedural algorithm because not all paths in the ICFG are realizable; a procedure call must return to the site that invoked it. As seen by the example in Figure 2.1, ignoring this problem can result in imprecise alias computations; the alias $\langle *r, p \rangle$ cannot hold on any realizable path, but it does hold on the path $[Entry_{Main}]$ $[r = NULL]$ $[Call_{A(q)}]$ $[Entry_A]$ $[q = f]$ $[Exit_A]$ $[Return_{A(q)}]$ $[r = q]$ $[Call_{A(\&p)}]$ $[Entry_A]$ $[q = f]$ $[Exit_A]$ $[Return_{A(q)}]$ $[r = q]$.

One possible solution to this problem is to keep path information and use it to avoid paths which are not realizable. This concept of "alias histories" is used by [Coo89, SP81]. We have not chosen this approach. Instead, we use the idea of solving a data flow problem for a procedure assuming an alias condition on entry. This is reminiscent of Lomet's approach to solving data flow problems under different aliasing conditions [Lom77] and also to Marlowe's notion of a representative data flow problem [MR90].

The key idea in our solution to the unrealizable path problem is to devise a two step algorithm. In the first step, we solve for *Conditional May 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 the *Conditional May Aliases* solution to solve for the actual aliases.

This two step approach avoids alias propagation along unrealizable paths. 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 returns. Conceptually, this is analogous to propagation on the program call graph [Hec77].[23]

This idea of using Conditional May Aliases does not seem promising at first, as there are an exponential number of possible sets of aliases. But Lemma 4.2.1 insures that it is sufficient to consider sets $\mathcal{A}$ where $\mid \mathcal{A} \mid \leq 1$. If more than one level of indirection is possible, it is no longer precise to consider $\mid \mathcal{A} \mid \leq 1$, but it is *safe*.[24] Again, we present a naive version of the algorithm in a manner to aid understanding and facilitate proof of correctness; it is not an efficient implementation.

Lemma 4.2.1 *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 for the procedure containing* n_i *and the number of calls on the path* $n_1 n_2 ... n_{i-1}$ *equals the number of returns.),*

- *and for all possible alias pairs* $\langle a, b \rangle$*;*

  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* $\langle a, b \rangle$ *holds at* n_i

  then

[23] A similar two pass approach to construct interprocedural program slices was presented in [HRB88] and in [HS90].

[24] I.e., our alias solution will be imprecise but all actual alias pairs will be contained within the solution calculated.

either assuming no aliases at n_1 *and executing path* P *forces* $\langle a, b\rangle$ *to hold at* n_i

or

$\exists k(1 \leq k \leq m)$ *such that when assuming* **only** *the alias pair* A_k *at* n_1, *executing the path* P *forces* $\langle a, b\rangle$ *to hold at* n_i.

Lemma 4.2.1 basically states that all the interprocedural path information can be captured by a single (or no; i.e., an empty set of assumptions) alias assumption. The proof of the Lemma 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 is required. It can be found in Appendix C.3.4.

□

Computing Conditional May Alias information, assuming no local variables

Conditional May Aliases are computed by building a *pointer alias graph* (PAG).[25] Nodes in the alias graph are of the following form; $[(ICFG\text{-}node, assumed\text{-}alias), alias\text{-}pair]$ with *assumed-alias* being either $\emptyset$ or a single alias pair. The fixed point of

$$holds([(ICFG\text{-}node, assumed\text{-}alias), alias\text{-}pair])$$

is *true* iff *alias-pair* holds on some path from procedure entry to $ICFG$-*node*, assuming there is a path to the entry of the procedure containing $ICFG$-*node* on which *assumed-alias* holds. As in the SPPAG, edges in the PAG represent a dependence between the *holds* predicates of PAG nodes.

First let us consider the intraprocedural aspects of the PAG. Assume in the SPPAG that

$$holds([n, \langle a, b\rangle]) = (holds([m_1, \langle c, d\rangle]) \vee \ \ldots \ \vee holds([m_k, \langle c, d\rangle]))$$

It is obvious this relationship still holds in the PAG for a fixed *assumed-alias*, thus for all possible *assumed-alias*

[25] Again our explanation of the algorithm is illustrative, but not intended to be an efficient implementable algorithm.

$$\mathit{holds}([(n, \mathit{assumed\text{-}alias}), \langle a, b\rangle]) =$$

$$(\mathit{holds}([(m_1, \mathit{assumed\text{-}alias})), \langle c, d\rangle]) \vee \ \ldots \ \vee \mathit{holds}([(m_k, \mathit{assumed\text{-}alias}), \langle c, d\rangle]))$$

Thus an understanding of the SPPAG together with the realization that for an entry node, $\mathit{holds}([(\mathit{entry}, \langle a, b\rangle), \langle a, b\rangle])$ is always *true* should give an intuition into the intra-procedural aspects of the PAG.

The unrealizable path problem is avoided in the PAG by **not including edges** between call nodes and the entry nodes of the procedure invoked. By definition, in Conditional May Aliases, a call site gives no information about which Conditional May Aliases hold at the entry of the invoked procedure; that is, $\mathit{holds}([(\mathit{entry}, \mathcal{AA}), \langle a, b\rangle])$ is *true* if $(\mathcal{AA} = \langle a, b\rangle)$[26] or ($a = b$ and a is not a dereferenced pointer) and otherwise is false.

call and *exit* nodes simply collect alias information. Thus:

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

Programs without Parameters Now, for simplicity, assume that we are dealing with a programming language that has no local variables, and thus no parameters. Further, assume that we are interested in whether $\mathit{holds}([(\mathit{return}, \mathit{assumed\text{-}alias}), \langle a, b\rangle])$ is *true*. Clearly it is *true* if $\langle a, b\rangle$ holds at the exit node conditional on *assumed-alias'* holding at its entry **and** *assumed-alias'*, conditional on *assumed-alias*, holds at the call node (see Figure 4.3). Let $\mathcal{ASSUMED}$ be the set of all possible assumed aliases. *Holds* for a return node is defined as:

$$\mathit{holds}([(\mathit{return}, \mathit{assumed\text{-}alias}), \langle a, b\rangle]) = \mathit{holds}([(\mathit{exit}, \emptyset), \langle a, b\rangle]) \vee$$

$$\bigvee_{\mathcal{AA} \in \mathcal{ASSUMED}} (\ \mathit{holds}([(\mathit{exit}, \mathcal{AA}), \langle a, b\rangle]) \wedge \mathit{holds}([(\mathit{call}, \mathit{assumed\text{-}alias}), \mathcal{AA}])\ )$$

Modeling the effects of parameter bindings

We need to be able to model the effects of parameter bindings and will use $\mathit{back\text{-}bind}_{\mathit{call}_P}$ and $\mathit{back\text{-}bind}'_{\mathit{call}_P}$ at each call site for this. $\mathit{back\text{-}bind}_{\mathit{call}_P}(\mathit{assumed\text{-}alias})$ specifies the alias holding on any path $\rho ... [\mathit{call}_P]$ that guarantees *assumed-alias* holds on

[26] We will in places use $\mathcal{AA}$ to represent assumed alias sets ($\mathcal{A}$ from before) that are of size 0 or 1.

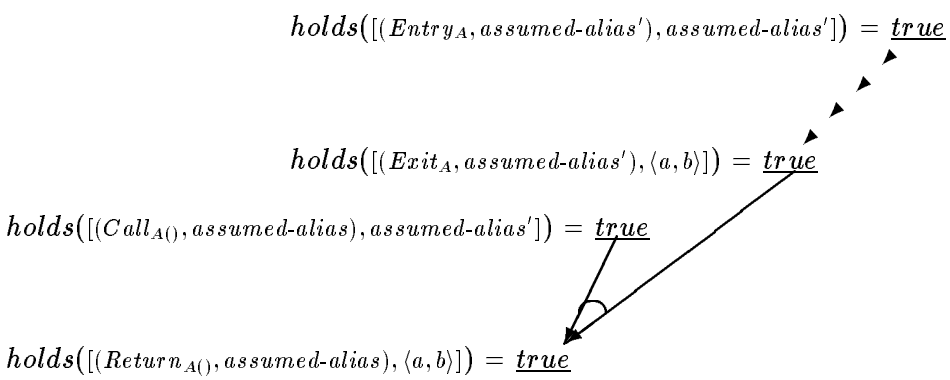


Figure 4.3: *holds* at a return node (no local variables)

$\rho...[call_P][entry_P]$. *back-bind* is formally specified in Appendix C.1, but we present an informal definition below.

Consider $back\text{-}bind_{call_P}(assumed\text{-}alias)$ for arbitrary $call_P$ and *assumed-alias*:

- $assumed\text{-}alias = \emptyset$

  This condition always holds.

- $assumed\text{-}alias = \langle c, d\rangle$ where both c and d are local to the called procedure

  Clearly if either c or d is not a dereferenced formal of the procedure, they can never be aliases on entry to the procedure. They will be aliased on entry if their corresponding actuals are aliased at the call. $back\text{-}bind_{call_P}(\langle c, d\rangle) = \langle *a_j, *a_k\rangle$[27] where a_j is the actual corresponding to c and a_k is the actual corresponding to d.

- $assumed\text{-}alias = \langle c, d\rangle$ where c is a local and d is not

  This alias will hold on entry if c is a dereferenced formal and its actual is aliased to d at the call. $back\text{-}bind_{call_P}(\langle c, d\rangle) = \langle *a_j, d\rangle$ where a_j is the actual corresponding to c.

- $assumed\text{-}alias = \langle c, d\rangle$ and neither c nor d are locals

  $\langle c, d\rangle$ holds at entry iff it held at the call node. $back\text{-}bind_{call_P}(\langle c, d\rangle) = \langle c, d\rangle$.

- otherwise

[27] We consider $*\&x \equiv x$

```
int *a,*b;
int y;

P()
{
    b=a
    a=&y;
}

main () {
    int x;

    a=&x;
    P();
}
```

Note: The call to P in *main* creates the alias pair $\langle *b, x \rangle$ and destroys the alias pair $\langle *a, x \rangle$.

Figure 4.4: Calls affecting alias pairs involving non-visible object names

assumed-alias can never hold on entry.

Unfortunately, such a simple modeling of parameter bindings is not sufficient. *A procedure call can both create and destroy, in the calling procedure, an alias involving an object name not visible in the called procedure.* For example, the call `P()` in Figure 4.4 creates the alias $\langle *b, x \rangle$ and destroys the alias $\langle *a, x \rangle$ at $\texttt{return}_{\texttt{P()}}$ where x is not visible in `P`. However, only references to the visible object name in an alias pair can affect whether that alias holds on a path (there can be no direct references to an object name which is not visible). Fortunately, a procedure has the *same effect* on all alias pairs which contain visible object name w and any non-visible object name. Thus for every object name w which may have aliases, we introduce the alias pair $\langle w, non_visible \rangle$, representing w aliased to a non-visible object name, into the set of possible alias pairs and the set of possible assumed aliases.

In addition to *back-bind*, we define *back-bind'*; $back\text{-}bind'_{call_P}(\langle a, non_visible \rangle, b)$ specifies the alias holding on any path $\rho...[call_P]$ that guarantees a will be aliased to

the non-visible object name b on $\rho...[call_P][entry_P]$. *back-bind'* is also formally specified in Appendix C.1, but an informal definition follows.

Consider $back\text{-}bind'_{call_P}(\langle a, non_visible\rangle, b)$ for arbitrary $call_P$, a, and b:

- a is not a local of the called procedure.

  $back\text{-}bind'_{call_P}(\langle a, non_visible\rangle, b) = \langle a, b\rangle$

- a is a dereferenced formal.

  $back\text{-}bind'_{call_P}(\langle a, non_visible\rangle, b) = \langle *a_j, b\rangle$ where $*a_j$ is the actual corresponding to a.

- otherwise

  $\langle a, non_visible\rangle$ can never hold on entry.

Computing Conditional May Alias information at a return node, factoring in local variables

Assume that we are interested in whether $\langle a, b\rangle$ holds on the realizable path P to node $return_Q$. We will use the following conventions:

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

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, Q:

  It is impossible for the called procedure to create or destroy this alias pair, thus

$\langle a, b\rangle$ holds on P_{return_Q} iff it holds on P_{call_Q}. Thus

$\mathit{holds}([(return, assumed\text{-}alias), \langle a, b\rangle]) = \mathit{holds}([(call, assumed\text{-}alias), \langle a, b\rangle])$

- a and b are both visible in the called procedure, Q:

  If $\langle a, b\rangle$ holds on P_{return_Q}, it must also hold on P_{exit_Q}. By Lemma 4.2.1, either no aliases need hold on P_{entry_Q} for $\langle a, b\rangle$ to hold on P_{exit_Q} or there exists a $\langle c, d\rangle$ such that only $\langle c, d\rangle$ must hold on P_{entry_Q} for $\langle a, b\rangle$ to hold on P_{exit_Q}. In the first case, $\mathit{holds}([(exit, \emptyset), \langle a, b\rangle])$ will be true (by definition of *holds*). In the second case, $\mathit{holds}([(exit, \langle c, d\rangle), \langle a, b\rangle])$ must be true and $\mathit{back\text{-}bind}_{call_Q}(\langle c, d\rangle)$ must hold on P_{call_Q}. Again by Lemma 4.2.1 there is an *assumed-alias* (either $\emptyset$ or a single alias pair) which must hold on P_{entry_R} and by definition of *holds*, $\mathit{holds}([(call_Q, assumed\text{-}alias), \mathit{back\text{-}bind}_{call_Q}(\langle c, d\rangle)])$ is *true*. Thus:

  $\mathit{holds}([(return, assumed\text{-}alias), \langle a, b\rangle]) =$

  $$\mathit{holds}([(exit, \emptyset), \langle a, b\rangle]) \vee \bigvee_{\mathcal{AA} \in \mathcal{ASSUMED}} \left(\begin{array}{l} \mathit{holds}([(exit, \mathcal{AA}), \langle a, b\rangle]) \wedge \\ \mathit{holds}([(call, assumed\text{-}alias), \mathit{back\text{-}bind}_{call}(\mathcal{AA})]) \end{array} \right)$$

- Assume that a is visible but b is not.

  The $\langle a, b\rangle$ holds on P_{return_Q} iff $\langle a, non_visible\rangle$ holds on P_{exit_Q} and b is the non-visible object name "*non_visible*". By Lemma 4.2.1 there is a single $\langle a', non_visible\rangle$ which must hold on path P_{entry_Q} and thus $\mathit{back\text{-}bind}'_{call_Q}(\langle a', non_visible\rangle, b)$ must hold on path P_{call_Q}. By Lemma 4.2.1 there is an *assumed-alias* (either $\emptyset$ or a single alias pair) which must hold on P_{entry_R} and by definition of *holds*, $\mathit{holds}([(call_Q, assumed\text{-}alias), \mathit{back\text{-}bind}'_{call_Q}(\langle a', non_visible\rangle, b)])$ is *true*. Thus:

  $\mathit{holds}([(return, assumed\text{-}alias), \langle a, b\rangle]) = \bigvee_{\langle a', non_visible\rangle \in \mathcal{ASSUMED}}$

  $$\left(\begin{array}{l} \mathit{holds}([(exit, \langle a', non_visible\rangle), \langle a, non_visible\rangle]) \wedge \\ \mathit{holds}([(call, assumed\text{-}alias), \mathit{back\text{-}bind}'_{call}(\langle a', non_visible\rangle, b)]) \end{array} \right)$$

A formal specification for the PAG and *holds* relation can be found in Appendix C.3. The fixed point of *holds* for the program in Figure 2.1 is in Figure 4.5. The assumed-aliases $\langle *r, *r\rangle$, $\langle *q, *r\rangle$, and $\langle *r, non_visible\rangle$ are not included in *main*'s table because

each involves at least one local and thus these aliases could never hold on a path to $ENTRY_{main}$. Assumed alias $\langle *q, *q \rangle$ is omitted for space consideration from *main*'s and A's table; it is a reflexive assumption that never occurs. A **true*** entry means the value of $holds([(\text{ICFG-node}, \text{assumed-alias}), \text{possible-alias}])$ is *true* **and** assumed-alias does hold on some path to the entry of the procedure containing ICFG-node. These are the only values that an efficient version of the algorithm would have to compute.

Computing Interprocedural May Alias using Conditional May Alias

Once we have computed Conditional May Aliases, Interprocedural May Alias information can be computed by a simple data flow problem on the ICFG. As in Conditional May Alias we have to be able to model the effects of parameter bindings. We have to know which aliases hold at an *entry* node given that a certain alias holds at the *call* node. We will use the $bind_{call}$ functions to model parameter effects. $bind_{call}(\mathcal{A})$ is the set of aliases which hold on the path $\rho n_1 ... n_{i-2}[call][entry_P]$ if $\mathcal{A}$ holds on $\rho n_1 ... n_{i-2}[call]$. $bind_{call}$ is formally specified in Appendix C.2, but may be thought of as:

$$bind_{call}(\mathcal{A}) = \left\{ \langle c, d \rangle \;\middle|\; \begin{array}{l} [back\text{-}bind_{call}(\langle c, d \rangle) = \emptyset] \text{ or} \\ [(\exists\, \langle a, b \rangle \in \mathcal{A})\ (back\text{-}bind_{call}(\langle c, d \rangle) = \langle a, b \rangle)] \end{array} \right\}$$

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

- $may\text{-}alias(\rho) = \emptyset$
- if n is an entry node then $may\text{-}alias(n) = \bigcup_{\ll m,n \gg \in \mathcal{E}} \left(bind_m(may\text{-}alias(m)) \right)$
- otherwise, $may\text{-}alias(n) =$

$$\left\{ \langle a, b \rangle \;\middle|\; \begin{array}{l} \text{neither } a \text{ nor } b \text{ is } non_visible \text{ and} \\ [holds([(n, \emptyset), \langle a, b \rangle]) = true] \bigvee \\ [\left(\exists \mathcal{AA} \in may\text{-}alias(entry(n))\right)\ holds([(n, \mathcal{AA}), \langle a, b \rangle]) = true] \end{array} \right\}$$

Theorem 4.2.3 *There exists a polynomial time algorithm for determining precise* **Interprocedural May Alias** *sets in the presence of single level pointers.*

ICFG-node	assumed-alias	possible-alias						
		$\langle p, *q \rangle$	$\langle p, *r \rangle$	$\langle *q, *r \rangle$	$\langle *q, nv^{\dagger} \rangle$	$\langle *r, nv \rangle$	$\langle *q, *q \rangle$	$\langle *r, *r \rangle$
$ENTRY_{main}$	$\emptyset$	$false$	$false$	$false$	$false$	$false$	$false$	$false$
$r = NULL$	$\emptyset$	$false$	$false$	$false$	$false$	$false$	$false$	$false$
$CALL_{A(q)}$	$\emptyset$	$false$	$false$	$false$	$false$	$false$	$false$	$false$
$RETURN_{A(q)}$	$\emptyset$	$false$	$false$	$false$	$false$	$false$	$false$	$false$
$r = q$	$\emptyset$	$false$	$false$	$false$	$false$	$false$	$false$	$false$
$CALL_{A(\&p)}$	$\emptyset$	$false$	$false$	$false$	$false$	$false$	$false$	$false$
$RETURN_{A(\&p)}$	$\emptyset$	**true***	$false$	$false$	$false$	$false$	**true***	$false$
$EXIT_{main}$	$\emptyset$	**true***	$false$	$false$	$false$	$false$	**true***	$false$
$ENTRY_{main}$	$\langle *q, nv \rangle$	$false$	$false$	$false$	true	$false$	$false$	$false$
$r = NULL$	$\langle *q, nv \rangle$	$false$	$false$	$false$	true	$false$	$false$	$false$
$CALL_{A(q)}$	$\langle *q, nv \rangle$	$false$	$false$	$false$	true	$false$	$false$	$false$
$RETURN_{A(q)}$	$\langle *q, nv \rangle$	$false$	$false$	$false$	true	$false$	$false$	$false$
$r = q$	$\langle *q, nv \rangle$	$false$	$false$	$false$	true	true	$false$	$false$
$CALL_{A(\&p)}$	$\langle *q, nv \rangle$	$false$	$false$	$false$	true	true	$false$	$false$
$RETURN_{A(\&p)}$	$\langle *q, nv \rangle$	true	$false$	$false$	$false$	true	true	$false$
$EXIT_{main}$	$\langle *q, nv \rangle$	true	$false$	$false$	$false$	true	true	$false$

ICFG-node	assumed-alias	possible-alias				
		$\langle *q, *f \rangle$	$\langle *q, nv \rangle$	$\langle *f, nv \rangle$	$\langle *q, *q \rangle$	$\langle *f, *f \rangle$
$ENTRY_A$	$\emptyset$	$false$	$false$	$false$	$false$	$false$
$q = f$	$\emptyset$	$false$	$false$	$false$	$false$	$false$
$EXIT_A$	$\emptyset$	$false$	$false$	$false$	$false$	$false$
$ENTRY_A$	$\langle *f, *f \rangle$	$false$	$false$	$false$	$false$	**true***
$q = f$	$\langle *f, *f \rangle$	**true***	$false$	$false$	**true***	**true***
$EXIT_A$	$\langle *f, *f \rangle$	**true***	$false$	$false$	**true***	**true***
$ENTRY_A$	$\langle *q, *f \rangle$	**true***	$false$	$false$	$false$	$false$
$q = f$	$\langle *q, *f \rangle$	$false$	$false$	$false$	$false$	$false$
$EXIT_A$	$\langle *q, *f \rangle$	$false$	$false$	$false$	$false$	$false$
$ENTRY_A$	$\langle *q, nv \rangle$	$false$	**true***	$false$	$false$	$false$
$q = f$	$\langle *q, nv \rangle$	$false$	$false$	$false$	$false$	$false$
$EXIT_A$	$\langle *q, nv \rangle$	$false$	$false$	$false$	$false$	$false$
$ENTRY_A$	$\langle *f, nv \rangle$	$false$	$false$	**true***	$false$	$false$
$q = f$	$\langle *f, nv \rangle$	$false$	**true***	**true***	$false$	$false$
$EXIT_A$	$\langle *f, nv \rangle$	$false$	**true***	**true***	$false$	$false$

† $nv \equiv non_visible$.

Figure 4.5: *holds*([(ICFG-node, assumed-alias), possible-alias]) for the program in Figure 2.1

```
Build the PAG.
Initialize holds([(n,AA),PA]) to false for all [(n,AA),PA] ∈ PAG.
Calculate the fixed point of holds.
Initialize may-alias(n) to ∅ for all n in the ICFG.
Calculate the fixed point of may-alias.
```

Figure 4.6: Polytime algorithm for **Interprocedural May Alias** in the presence of single level pointers

The claim is that the algorithm is Figure 4.6 is such an algorithm. We claim that the *holds* calculation can be computed in polynomial time. 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 Lemma 4.2.1[28]. The formal proof of Theorem 4.2.3 is by induction on path length and by induction on number of iterations of the fixed point calculation. This is formally proved in Appendix C.3.5.

□

Figure 4.7 gives the fixed point of *may-alias* for the program in Figure 2.1.

4.2.4 Interprocedural Must Alias

As with Interprocedural May Alias, we will solve the Interprocedural Must Alias problem using a two step algorithm. In the first step we solve for Conditional Must Aliases and in the second step we use the Conditional Must Alias information to calculate the must alias sets. However, the nature of Conditional Must Alias has little in common with Conditional May Alias.

[28] We directly prove that the MFP of our equations is the same as the precise May Alias solution, thus we do not need to prove distributivity of our function space to show MFP ≡ MOP [Kil73].

ICFG-node	*may-alias*(ICFG-node)
$ENTRY_{main}$	$\emptyset$
$r = NULL$	$\emptyset$
$CALL_{A(q)}$	$\emptyset$
$RETURN_{A(q)}$	$\emptyset$
$r = q$	$\emptyset$
$CALL_{A(\&p)}$	$\emptyset$
$RETURN_{A(\&p)}$	$\{\langle *p, *q\rangle, \langle *q, *q\rangle\}$
$EXIT_{main}$	$\{\langle *p, *q\rangle, \langle *q, *q\rangle\}$
$ENTRY_A$	$\{\langle *f, *f\rangle\}$
$q = f$	$\{\langle *q, *f\rangle, \langle *f, *f\rangle, \langle *q, *q\rangle\}$
$EXIT_A$	$\{\langle *q, *f\rangle, \langle *f, *f\rangle, \langle *q, *q\rangle\}$

Figure 4.7: *may-alias*(ICFG-node) for the program in Figure 2.1

Conditional Must Alias

Intuitively, Conditional Must Alias for a *node* of the ICFG and an *alias-pair* is the unique minimal set of assumed must aliases at the entry node of the procedure containing *node* which insures that *alias-pair* must hold on all paths to *node*. If no such set exists, Conditional Must Alias is *false*. For example, in Figure 4.8, to insure that $\langle *\mathtt{p}, \mathtt{x}\rangle$ holds on all paths to n_1 both $\langle *\mathtt{q}, \mathtt{x}\rangle$ and $\langle *\mathtt{r}, \mathtt{x}\rangle$ must be assumed at $entry_{\mathtt{p}}$. We will use $COND^{alias\text{-}pair}_{node}$ to refer the precise Conditional Must Alias information.

$COND^{alias\text{-}pair}_{node}$ always has a value. If, on some path to *node*, *alias-pair* doesn't hold then $COND^{alias\text{-}pair}_{node}$ by definition is *false*. Now, consider the case when *alias-pair* holds on all paths, P_i, to *node*. By Lemma 4.2.1 (p. 34) there is a unique $\mathcal{AA}_i$ which is either $\emptyset$ (i.e., no assumption) or a single assumed alias, which holds on P_i only up to (and including) the entry node of the procedure containing *node*, that guarantees *alias-pair* holds on P_i. The unique minimal set of assumed must aliases at the entry node of the procedure containing *node* which insures that *alias-pair* must hold on all paths to *node* is clearly the union of all such $\mathcal{AA}_i$ (i.e., the union of assumptions over all individual paths).

The possible values for $COND^{alias\text{-}pair}_{node}$ are *false* and any subset of the set of all

```
int x, *p, *q, *r;
...
P()
{
   if (x == 0)
      p = q;
   else p = r;
n1 :
}
```

On all paths to n_1 $\langle *\mathtt{p}, \mathtt{x}\rangle$ holds iff both
$\langle *\mathtt{q}, \mathtt{x}\rangle$ and $\langle *\mathtt{r}, \mathtt{x}\rangle$ hold on all paths to $entry_{\mathtt{P}}$.

Figure 4.8: Conditional Must Alias: an example

possible aliases ($\mathcal{POSSIBLE}$-$\mathcal{ALIASES}$). These values when ordered by the restrictiveness of the Conditional Must Aliases form a lattice. *false* is most restrictive since it means that no conditions will force *alias-pair* to hold at *node*. Thus, we are just lifting the superset lattice and adding a new bottom (*false*). That is $a \sqsubseteq b$ iff a is *false* or $a \supseteq b$[29]. The meet ($\sqcap$) of two elements of the lattice is *false*, if either element is *false* and is the (set) union of the two elements otherwise. The lattice is formally specified in Figure 4.9 and Appendix C.4.1.

We will need to model the effects of procedure calls; as in Interprocedural May Alias we use the *back-bind* relation (see Section 4.2.3 and Appendix C.1). The basis of our Conditional Must Alias algorithm will be the relationship between the various $COND_{node}^{alias\text{-}pair}$ values. Conditional Must Alias information will be computed by a maximal fixed point calculation of the *must-holds* relation:

To simplify the formulas below,

- Let *must-holds*$(n, false) = false$ for all nodes n in the ICFG.
- Let *must-holds*$(n, \langle o, o\rangle) = \emptyset$ for all nodes n in the ICFG, all possible non-dereferenced pointer object names o.

[29] If b is $false$, $a \sqsubseteq b$ iff a is $false$ also.

Let $\mathcal{POSSIBLE}$-$\mathcal{ALIASES}$ be the set of all possible aliases.
Define the lattice $\mathcal{L} = (\mathcal{S}, \sqsubseteq, \sqcap, \bot, \top)$ as:

- $\mathcal{S} = powerset(\mathcal{POSSIBLE}\text{-}\mathcal{ALIASES}) \cup false$
- $a \sqsubseteq b$ iff $(a = false)$ or $(a \supseteq b)$
  NOTE: If b is $false$, $a \sqsubseteq b$ iff a is $false$ also.
- $a \sqcap b = \begin{cases} false & \text{if } (a = false) \text{ or } (b = false) \\ a \cup b & \text{otherwise} \end{cases}$
- $\top = \emptyset$
- $\bot = false$

Figure 4.9: Lattice for possible values of $COND_{node}^{alias\text{-}pair}$

Define the relation *must-holds*$(n, \langle a, b \rangle)$, n a node of the ICFG=($\mathcal{N}$,$\mathcal{E}$,ρ) and $\langle a, b \rangle \in$ $\mathcal{POSSIBLE}$-$\mathcal{ALIASES}$ ($a \neq b$ or a is a dereferenced pointer), as follows:

- n is an entry node

  Clearly $\{\langle a, b \rangle\}$ is the unique minimal set of aliases which must hold at entry.

- n is a return node ($return_Q$)

  Assume that we are interested in whether $\langle a, b \rangle$ holds on all realizable paths P to $return_Q$. Where, as on page 38:

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

  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:

  - Both a and b are visible in the procedure which was called.

$\langle a, b\rangle$ must hold on all P_{return_Q} if $\langle a, b\rangle$ must hold on all P_{exit_Q} whenever Q is invoked by call site of $return_Q$. $\langle a, b\rangle$ must hold on all P_{exit_Q} if all aliases in $\mathit{must\text{-}holds}(exit_Q, \langle a, b\rangle)$ hold on all P_{entry_Q}. This is true if for each $\langle c, d\rangle$ in $\mathit{must\text{-}holds}(exit_Q, \langle a, b\rangle)$, $\mathit{back\text{-}bind}_{call_Q}(\langle c, d\rangle)$ must hold on all P_{call_Q}. This must be the case if each alias in

$$\prod_{\langle c,d\rangle \in \mathit{must\text{-}holds}(exit_Q, \langle a,b\rangle)} \left(\mathit{must\text{-}holds}(call_Q, \mathit{back\text{-}bind}_{call_Q}(\langle c, d\rangle)) \right)$$

holds on all P_{entry_R}.

– Either a or b but not both is visible in the procedure which was called.

$\langle a, b\rangle$ must hold on all P_{return_Q} if $\langle a, \mathit{non_visible}\rangle$ must hold on all P_{exit_Q} and "$\mathit{non_visible}$" is the non-visible object b. $\langle a, \mathit{non_visible}\rangle$ must hold on all P_{exit_Q} if all aliases in $\mathit{must\text{-}holds}(exit_Q, \langle a, \mathit{non_visible}\rangle)$ hold on all P_{entry_Q}. This is true if for each $\langle c, \mathit{non_visible}\rangle$ in $\mathit{must\text{-}holds}(exit_Q, \langle a, \mathit{non_visible}\rangle)$, $\mathit{back\text{-}bind}'_{call_Q}(\langle c, \mathit{non_visible}\rangle, b)$ must hold on all P_{call_Q}. This must be the case if each alias in

$$\prod_{\substack{\langle c, \mathit{non_visible}\rangle \in \\ \mathit{must\text{-}holds}(exit_Q, \langle a, \mathit{non_visible}\rangle)}} \left(\mathit{must\text{-}holds}(call_Q, \mathit{back\text{-}bind}'_{call_Q}(\langle c, \mathit{non_visible}\rangle, b)) \right)$$

holds on all P_{entry_R}.

– Both a and b are not visible in the procedure which was called.

$\langle a, b\rangle$ must hold on all P_{return_Q} iff $\langle a, b\rangle$ must hold on all P_{call_Q}.

- otherwise (n is neither an entry nor a return node)

From Interprocedural May Alias we know that there is a unique $\langle c, d\rangle$ such that for any path $\rho n_1 n_2 ... n_i$ ($n = n_i$) in the ICFG, $\langle a, b\rangle$ holds at n_i iff $\langle c, d\rangle$ holds on the path $\rho n_1 ... n_{i-1}$ (see Section 4.2.3 and Section 4.2.1). Thus $\mathit{must\text{-}holds}(n, \langle a, b\rangle)$ is the union of the Conditional Must Alias sets for $\langle c, d\rangle$ at all immediate predecessors, if the sets exist and *otherwise* is $false$. Intuitively this is true because if $\langle a, b\rangle$ must hold at n, $\langle c, d\rangle$ must hold at all immediate predecessors of n.

must-holds is formally specified in Appendix C.4.2. The fixed point of *must-holds* for the program in Figure 2.1 is in Figure 4.10.

ICFG-node	**possible-alias**			
	$\langle p, *q \rangle$	$\langle p, *r \rangle$	$\langle *q, *r \rangle$	$\langle *q, nv^{\dagger} \rangle$
$ENTRY_{main}$	$\{\langle p, *q \rangle\}$	$\{\langle p, *r \rangle\}$	$\{\langle *q, *r \rangle\}$	$\{\langle *q, nv \rangle\}$
$r = NULL$	$\{\langle p, *q \rangle\}$	$false$	$false$	$\{\langle *q, nv \rangle\}$
$CALL_{A(q)}$	$\{\langle p, *q \rangle\}$	$false$	$false$	$\{\langle *q, nv \rangle\}$
$RETURN_{A(q)}$	$\{\langle p, *q \rangle\}$	$false$	$false$	$\{\langle *q, nv \rangle\}$
$r = q$	$\{\langle p, *q \rangle\}$	$\{\langle p, *q \rangle\}$	$\{\langle *q, *q \rangle\}$	$\{\langle *q, nv \rangle\}$
$CALL_{A(\&p)}$	$\{\langle p, *q \rangle\}$	$\{\langle p, *q \rangle\}$	$\{\langle *q, *q \rangle\}$	$\{\langle *q, nv \rangle\}$
$RETURN_{A(\&p)}$	$\emptyset^{\ddagger}$	$\{\langle p, *q \rangle\}$	$\{\langle p, *q \rangle\}^{\dagger\dagger}$	$\{\langle p, nv \rangle\}$
$EXIT_{main}$	$\emptyset$	$\{\langle p, *q \rangle\}$	$\{\langle p, *q \rangle\}$	$\{\langle p, nv \rangle\}$

ICFG-node	**possible-alias**		
	$\langle *r, nv \rangle$	$\langle *q, *q \rangle$	$\langle *r, *r \rangle$
$ENTRY_{main}$	$\{\langle *r, nv \rangle\}$	$\{\langle *q, *q \rangle\}$	$\{\langle *r, *r \rangle\}$
$r = NULL$	$false$	$\{\langle *q, *q \rangle\}$	$false$
$CALL_{A(q)}$	$false$	$\{\langle *q, *q \rangle\}$	$false$
$RETURN_{A(q)}$	$false$	$\{\langle *q, *q \rangle\}$	$false$
$r = q$	$\{\langle *q, nv \rangle\}$	$\{\langle *q, *q \rangle\}$	$\{\langle *q, *q \rangle\}$
$CALL_{A(\&p)}$	$\{\langle *q, nv \rangle\}$	$\{\langle *q, *q \rangle\}$	$\{\langle *q, *q \rangle\}$
$RETURN_{A(\&p)}$	$\{\langle *q, nv \rangle\}$	$\emptyset$	$\{\langle *q, *q \rangle\}$
$EXIT_{main}$	$\{\langle *q, nv \rangle\}$	$\emptyset$	$\{\langle *q, *q \rangle\}$

ICFG-node	**possible-alias**				
	$\langle *q, *f \rangle$	$\langle *q, nv \rangle$	$\langle *f, nv \rangle$	$\langle *q, *q \rangle$	$\langle *f, *f \rangle$
$ENTRY_A$	$\{\langle *q, *f \rangle\}$	$\{\langle *q, nv \rangle\}$	$\{\langle *f, nv \rangle\}$	$\{\langle *q, *q \rangle\}$	$\{\langle *f, *f \rangle\}$
$q = f$	$\{\langle *f, *f \rangle\}$	$\{\langle *f, nv \rangle\}$	$\{\langle *f, nv \rangle\}$	$\{\langle *f, *f \rangle\}$	$\{\langle *f, *f \rangle\}$
$EXIT_A$	$\{\langle *f, *f \rangle\}$	$\{\langle *f, nv \rangle\}$	$\{\langle *f, nv \rangle\}$	$\{\langle *f, *f \rangle\}$	$\{\langle *f, *f \rangle\}$

† $nv \equiv non_visible$.

‡ because $must\text{-}holds(EXIT_A, \langle *q, nv \rangle) = \langle *f, nv \rangle$,
$back\text{-}bind'_{CALL_{A(\&p)}}(\langle *f, nv \rangle, p) = \langle p, p \rangle$,
and $must\text{-}holds(CALL_{A(\&p)}, \langle p, p \rangle) = \emptyset$.

†† because $must\text{-}holds(EXIT_A, \langle *q, nv \rangle) = \langle *f, nv \rangle$,
$back\text{-}bind'_{CALL_{A(\&p)}}(\langle *f, nv \rangle, *r) = \langle p, *r \rangle$,
and $must\text{-}holds(CALL_{A(\&p)}, \langle p, *r \rangle) = \langle p, *q \rangle$.

Figure 4.10: *must-holds*(ICFG-node,possible-alias) for the program in Figure 2.1

Computing Interprocedural Must Alias using Conditional Must Alias

Interprocedural Must Alias information can be computed by a simple data flow problem on the ICFG when given Conditional Must Alias information. As in Interprocedural May Alias we have to be able to model the effects of parameter bindings. We will again use the *bind* relation (see Section 4.2.3 and Appendix C.2).

Given *must-holds* and the *bind* functions, for any node n in the ICFG, *must-alias*(n) can be defined as follows:

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

  $$\textit{must-alias}(n)^{30} = \bigcap_{\ll m,n \gg \in \mathcal{E}} \left\{ \langle a, b\rangle \;\middle|\; \begin{array}{l} \langle a, b\rangle \in \textit{bind}_m(\textit{must-alias}(m)) \text{ and} \\ \text{neither } a \text{ nor } b \text{ contain } \textit{non_visible} \end{array} \right\}$$

- otherwise,

  $$\textit{must-alias}(n) = \{\langle a, b\rangle \mid \textit{must-holds}(n, \langle a, b\rangle) \subseteq'^{31} \textit{must-alias}(\textit{entry}(n))\}$$

Theorem 4.2.4 *There exists a polynomial time algorithm for determining precise* **Interprocedural Must** *Alias sets in the presence of single level pointers.*

The claim is that the algorithm in Figure 4.11 is such an algorithm. This is formally proved in Appendix C.4.6. Both fixed point calculations in Figure 4.11 converge in a polynomial amount of time since both lattices are of height polynomial in the size of the program[32] and both function spaces are monotonic.

□

Figure 4.12 gives the fixed point of *must-alias* for the program in Figure 2.1.

4.3 Non-pointer Reference Formals and Single Level Pointers

A program with these mechanisms can easily be transformed into a program with only single level pointers by the following rules:

---

[30] An alias has to hold on all calls.

[31] $(X \subseteq' Y) = \begin{cases} \textit{false} & \text{if } X = \textit{false} \\ X \subseteq Y & \textit{otherwise} \end{cases}$

[32] Both lattices have height $\mathcal{O}(v^2)$, where v is the number of variables in the program.

For all n in the ICFG, all $\langle a, b\rangle \in \mathcal{POSSIBLE}$-$\mathcal{ALIASES}$
    initialize *must-holds*$(n, \langle a, b\rangle)$ to $\emptyset$.
Calculate the fixed point of *must-holds*.
For all n in the ICFG, initialize *must-alias*(n) to
    $\mathcal{POSSIBLE}$-$\mathcal{ALIASES}$.
Calculate the fixed point of *must-alias*.

Figure 4.11: Polynomial time algorithm for **Interprocedural Must Alias** in the presence of single level pointers

ICFG-node	*must-alias*(ICFG-node)
$ENTRY_{main}$	$\emptyset$
$r = NULL$	$\emptyset$
$CALL_{A(q)}$	$\emptyset$
$RETURN_{A(q)}$	$\emptyset$
$r = q$	$\emptyset$
$CALL_{A(\&p)}$	$\emptyset$
$RETURN_{A(\&p)}$	$\{\langle p, *q\rangle, \langle *q, *q\rangle\}$
$EXIT_{main}$	$\{\langle p, *q\rangle, \langle *q, *q\rangle\}$
$ENTRY_A$	$\emptyset$
$q = f$	$\emptyset$
$EXIT_A$	$\emptyset$

Figure 4.12: *must-alias*(ICFG-node) for the program in Figure 2.1

- For every procedure P, for every reference formal r in P
  - Replace the declaration "ref[33] $< type > r$" with "$< type > *r$".
  - Replace each "$\&r$" in the program with "r".
  - Replace each "r" in the program with "$*r$", except where r is used as an actual for a reference formal. Do no transformation in the latter case.
- For every call C, for every actual a for a reference formal
  - Replace "a" in C with "$\&a$", except when a is itself a reference formal.
- All formals are pass-by-value.

This transformation is essentially the inverse of *refizing* as developed by Carroll [Car88].

Lemma 4.3.1 *For all realizable paths $P = \rho n_1 n_2 ... n_i$ in the ICFG for the original program, $\langle x, y \rangle$ holds on P iff*

- *case: x and y are reference formals*
  $\langle *x, *y \rangle$ holds on the same path in the transformed program.
- *case: x is a reference formal but y is not*
  $\langle *x, y \rangle$ holds on the same path in the transformed program.
- *case: neither x nor y are reference formals*
  $\langle x, y \rangle$ holds on the same path in the transformed program.

The proof is by induction on path length and is formally proved in Appendix C.5.

□

Theorem 4.3.1 *There exists a polynomial time algorithm for determining precise* **Interprocedural May Alias sets** *in the presence of non-pointer reference formals and single level pointers.*

[33] **ref** indicates reference formal

Follows from Lemma 4.3.1 and Theorem 4.2.3.

□

Theorem 4.3.2 *There exists a polynomial time algorithm for determining precise* **Interprocedural Must Alias sets** *in the presence of non-pointer reference formals and single level pointers.*

Follows from Lemma 4.3.1 and Theorem 4.2.4.

□

4.4 Multiple Level Pointers

So far we have presented precise polynomial time algorithms for determining aliases in the presence of various mechanisms. This is not possible for the remainder of the mechanisms in this chapter. We present proofs, in the presence of all subsequent mechanisms, that determining aliases is $\mathcal{NP}$-hard (or co-$\mathcal{NP}$-hard).

4.4.1 May Alias

Theorem 4.4.1 *In the presence of two-level pointers, the problem of determining precise* **Intraprocedural May Alias** *sets is $\mathcal{NP}$-hard.*

Corollary 4.4.1 *In the presence of two-level pointers, the problem of determining precise* **Interprocedural May Alias** *sets is $\mathcal{NP}$-hard.*[34]

The proof of Theorem 4.4.1 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 propositional variables $\{v_1, v_2, ..., v_m\}$. The reduction is specified by the program in Figure 4.13, 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. As will be seen later, paths through the code between `L1` and `L2` will represent truth assignments to the propositional variables. The truth assignment for a particular path will be encoded in the aliasing relationship of certain variables in the

[34]The intraprocedural version of this problem (Theorem 4.4.1) is a subproblem of the interprocedural version.

```
    int **v1,**v̄1,**v2,**v̄2,..., **vm,**v̄m;
    int *true,*false;
    int yes,no;

    /* A path through this section of code
        corresponds to a truth assignment */
L1:
    if (-) {v1 = &true; v̄1 = &false}
       else {v1 = &false; v̄1 = &true}
    if (-) {v2 = &true; v̄2 = &false}
       else {v2 = &false; v̄2 = &true}
                    .  .  .
    if (-) {vm = &true; v̄m = &false}
       else {vm = &false; v̄m = &true}

L2: false = &no;

    /* The code below will break the ⟨*false, no⟩ alias iff the
        truth assignment from above makes the formula false */

    if (-) *l1,1† = &yes else if (-) *l1,2 = &yes else *l1,3 = &yes;
    if (-) *l2,1 = &yes else if (-) *l2,2 = &yes else *l2,3 = &yes;
                          .  .  .
    if (-) *ln,1 = &yes else if (-) *ln,2 = &yes else *ln,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 4.13: 3-SAT solution iff [L3,$\langle *false, no\rangle$] in Intraprocedural May Alias

program. Paths between L2 and L3 then encode in the alias relationships whether or not the truth assignment resultant from path taken to L2 satisfies $\bigwedge_{i=1}^{n}(l_{i,1} \vee l_{i,2} \vee l_{i,3})$.

If we interpret $*v_i$ aliased to *true* as meaning the variable v_i is true, then any path from L1 to L2 can be interpreted as a truth assignment. Further, every truth assignment corresponds to some such path. This idea of using aliases to represent a truth assignment is from Myers [Mye81]; Larus [Lar89] independently developed a similar proof for aliasing in dynamic structures.

Now consider the paths from L2 to L3. If the truth assignment for the particular path from L1 to L2 satisfies the formula then every clause has a literal which is true.

Pick the path from L2 which for each clause assigns $\&yes$ to its true literal. Thus each assignment corresponds to $true = \&yes$ and [L3,$\langle *false, no \rangle$] must be in Intraprocedural May Alias. However if the formula is unsatisfiable then every truth assignment has a clause, say $(l_{i,1} \vee l_{i,2} \vee l_{i,3})$, with 3 false literals. This means $*l_{i,1}$, $*l_{i,2}$, and $*l_{i,3}$ are all aliased to $false$. Because every path from L2 to L3 must go through the statement

if (-) $*l_{i,1}$ = &yes else if (-) $*l_{i,2}$ = &yes else $*l_{i,3}$ = &yes;

alias $\langle *false, yes \rangle$ must hold on all paths to L3 and thus $\langle *false, no \rangle$ never does. Thus 3-SAT is polynomial reducible to Intraprocedural May Alias with two-level pointers and Theorem 4.4.1 holds.

□

4.4.2 Must Alias

Theorem 4.4.2 *In the presence of two-level pointers, the complement of the problem for determining precise* **Intraprocedural Must Alias** *sets is $\mathcal{NP}$-hard.*

Corollary 4.4.2 *In the presence of two-level pointers, the complement of the problem for determining precise* **Interprocedural Must Alias** *sets is $\mathcal{NP}$-hard.*

The proof of Theorem 4.4.2 is by the same reduction from 3-SAT as Theorem 4.4.1 (see Figure 4.13). However, the claim now is that the 3-SAT formula is satisfiable iff [L3,$\langle *false, yes \rangle$] is in the complement of Intraprocedural Must Alias. By proof of Theorem 4.4.1 if the 3-SAT formula is satisfiable then there is a path to L3 on which $\langle *false, no \rangle$ holds, and thus [L3,$\langle *false, yes \rangle$] is not in Intraprocedural Must Alias. If the 3-SAT formula is unsatisfiable then on every path to L3, $\langle *false, yes \rangle$ holds and thus [L3,$\langle *false, yes \rangle$] is in Intraprocedural Must Alias.

□

4.5 Reference Formals and Single Level Pointers

4.5.1 Interprocedural May Alias

Theorem 4.5.1 *In the presence of reference formals and single level pointers, the problem of determining precise* **Interprocedural May Alias** *sets is $\mathcal{NP}$-hard.*

The proof of Theorem 4.5.1 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 $\{v_1, v_2, ..., v_m\}$. It is conceptually identical to the proof of Theorem 4.4.1. The reduction is specified by the program in Figure 4.14[35] 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 intraprocedural paths are executable.

The following is essentially a reiteration of the proof of Theorem 4.4.1.

If we interpret v_i aliased to *true* as meaning the variable v_i is true then any path from L1 to L2 can be interpreted as a truth assignment, and every truth assignment corresponds to some such path.

Now consider the paths from L2 to L3. If the truth assignment for the particular path from L1 to L2 satisfies the formula then every clause has a literal which is true. Pick the path from L2 which for each clause assigns $\&yes$ to its true literal. Thus each assignment corresponds to $true = \&yes$ and [L3,$\langle *false, no\rangle$] must be in Interprocedural May Alias. However if the formula is unsatisfiable then every truth assignment has a clause, say $(l_{i,1} \vee l_{i,2} \vee l_{i,3})$, with 3 false literals. This means $l_{i,1}$, $l_{i,2}$, and $l_{i,3}$ are all aliased to $false$. Because every path from L2 to L3 must go through the statement

if (-) $l_{i,1}$ = &yes else if (-) $l_{i,2}$ = &yes else $l_{i,3}$ = &yes;

alias $\langle *false, yes\rangle$ must hold on all paths to L3 and thus $\langle *false, no\rangle$ never does.

□

4.5.2 Interprocedural Must Alias

Theorem 4.5.2 *In the presence of reference formals and single level pointers, the complement of the problem for determining precise* **Interprocedural Must Alias** *sets is*

[35] Figure 4.15 is an alternate reduction that works for C scope rules.

```
        int *true, *false, yes, no;

        /* A path through this section of code
            corresponds to a truth assignment */
        var_1(v_1, v̄_1)
             ref† int *v_1, *v̄_1;
        {
             if (-) var_2(true,false) else var_2(false,true);
        }
        var_2(v_2, v̄_2)
             ref int *v_2, *v̄_2;
        {
             if (-) var_3(true,false) else var_3(false,true)
        }
                                . . .
        var_m(v_m, v̄_m)
             ref int *v_m, *v̄_m;
        {
             partb()
        }

        /* The code below will break the ⟨*false, no⟩ alias iff
            the truth assignment from above makes the formula false */
        partb() {
L2:          false = &no;
             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:      }

        main() {
L1:          if (-) var_1(true,false) else var_1(false,true);
        }
```

Note: In order for proof to work *partb*() must be in the scope of $v_1, v_2, ..., v_m$, $\overline{v_1}, ...,$ and $\overline{v_m}$. Thus with Pascal-like scoping rules these procedures would have to be nested.

† **ref** indicates reference formal

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

Figure 4.14: 3-SAT solution iff [L3,$\langle *false, no \rangle$] in Interprocedural May Alias

```
    int *true, *false, yes, no;

    /* A path through this section of code
        corresponds to a truth assignment */
    var_1(v_1, v̄_1)
        ref† int *v_1, *v̄_1;
    {
        if (-) var_2(v_1, v̄_1,true,false) else var_2(v_1, v̄_1,false,true);
    }
    var_2(v_1, v̄_1, v_2, v̄_2)
        ref int *v_1, *v̄_1, *v_2, *v̄_2;
    {
        if (-) var_3(v_1, v̄_1, v_2, v̄_2,true,false)
             else var_3(v_1, v̄_1, v_2, v̄_2,false,true)
    }
                              .  .  .
    var_m(v_1, v̄_1, v_2, v̄_2, ..., v_m, v̄_m)
        ref int *v_1, *v̄_1, *v_2, *v̄_2, ...  *v_m, *v̄_m;
    {
        partb(v_1, v̄_1, v_2, v̄_2, ..., v_m, v̄_m)
    }

    /* The code below will break the ⟨*false, no⟩ alias iff
        the truth assignment from above makes the formula false */
    partb(v_1, v̄_1, v_2, v̄_2, ..., v_m, v̄_m)
        ref int *v_1, *v̄_1, *v_2, *v̄_2, ...  *v_m, *v̄_m;
    {
L2:     false = &no;
        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: }

    main() {
L1:     if (-) var_1(true,false) else var_1(false,true);
    }
```

† **ref** indicates reference formal

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

Figure 4.15: 3-SAT solution iff [L3,⟨$*false, no$⟩] in Interprocedural May Alias

$\mathcal{NP}$-hard.

The proof of Theorem 4.5.2 is by the same reduction from 3-SAT as Theorem 4.5.1 (see Figure 4.14). However, the claim now is that the 3-SAT formula is satisfiable iff [L3,$\langle *false, yes \rangle$] is in the complement of Interprocedural Must Alias. By proof of Theorem 4.5.1 if the 3-SAT formula is satisfiable then there is a path to L3 on which $\langle *false, no \rangle$ holds, and thus [L3,$\langle *false, yes \rangle$] is not in Interprocedural Must Alias. If the 3-SAT formula is unsatisfiable then on every path to L4, $\langle *false, yes \rangle$ holds and thus [L3,$\langle *false, yes \rangle$] is in Interprocedural Must Alias.

□

4.6 Structures Containing Single Level Pointers

4.6.1 May Alias

Theorem 4.6.1 *In the presence of structures containing single level pointers, the problem of determining precise* **Intraprocedural May Alias** *sets is $\mathcal{NP}$-hard.*

Corollary 4.6.1 *In the presence of structures containing single level pointers, the problem of determining precise* **Interprocedural May Alias** *sets is $\mathcal{NP}$-hard.*

The proof of Theorem 4.6.1 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 $\{v_1, v_2, ..., v_m\}$. Larus [Lar89] presents an alternate proof of this theorem. Our proof is conceptually identical to the proof of Theorem 4.4.1. The reduction is specified by the program in Figure 4.16 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.

The following is essentially a reiteration of the proof of Theorem 4.4.1.

If we interpret $*(v_i.next)$ aliased to *true* as meaning the variable v_i is true then any path from L1 to L2 can be interpreted as a truth assignment, and every truth assignment corresponds to some such path.

Now consider the paths from L2 to L3. If the truth assignment for the particular path from L1 to L2 satisfies the formula then every clause has a literal which is true.

```
    struct list {
         int value;
         struct list *next;
    }

    struct list v_1, v̄_1 , v_2, v̄_2, ...  , v_m, v̄_m;
    struct list true, false;
    struct list yes, no;

    /* A path through this section of code
        corresponds to a truth assignment */
L1:
    if (-) {v_1.next = &true; v̄_1.next = &false}
        else {v_1.next = &false; v̄_1.next = &true}
    if (-) {v_2.next = &true; v̄_2.next = &false}
        else {v_2.next = &false; v̄_2.next = &true}
                    .   .   .
    if (-) {v_m.next = &true; v̄_m.next = &false}
        else {v_m.next = &false; v̄_m.next = &true}

L2: false.next = &no;

    /* The code below will break the ⟨*(false.next), no⟩ alias iff
        the truth assignment from above makes the formula false */

    if (-) *(l_{1,1}.next).next† = &yes
        else if (-) *(l_{1,2}.next).next = &yes
        else *(l_{1,3}.next).next = &yes;

    if (-) *(l_{2,1}.next).next = &yes
        else if (-) *(l_{2,2}.next).next = &yes
        else *(l_{2,3}.next).next = &yes;
                          .   .   .
    if (-) *(l_{n,1}.next).next = &yes
        else if (-) *(l_{n,2}.next).next = &yes
        else *(l_{n,3}.next).next = &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 4.16: 3-SAT solution iff [L3,$\langle *(false.next), no\rangle$] in Intraprocedural May Alias

Pick the path from L2 which for each clause assigns $\&yes$ to its true literal. Thus each assignment corresponds to $true.next = \&yes$ and [L3,$\langle *(false.next), no\rangle$] must be in Intraprocedural May Alias. However if the formula is unsatisfiable then every truth assignment has a clause, say $(l_{i,1} \vee l_{i,2} \vee l_{i,3})$, with 3 false literals. This means $*(l_{i,1}.next)$, $*(l_{i,2}.next)$, and $*(l_{i,3}.next)$ are all aliased to $false$. Because every path from L2 to L3 must go through the statement

if (-) $*(l_{i,1}.next).next$ = &yes

else if (-) $*(l_{i,2}.next).next$ = &yes

else $*(l_{i,3}.next).next$ = &yes;

alias $\langle *(false.next), yes\rangle$ must hold on all paths to L3 and thus $\langle *(false.next), no\rangle$ never does.

□

4.6.2 Must Alias

Theorem 4.6.2 *In the presence of structures containing single level pointers, the complement of the problem for determining precise* **Intraprocedural Must Alias** *sets is $\mathcal{NP}$-hard.*

Corollary 4.6.2 *In the presence of structures containing single level pointers, the complement of the problem for determining precise* **Interprocedural Must Alias** *sets is $\mathcal{NP}$-hard.*

The proof of Theorem 4.6.2 is by the same reduction from 3-SAT as Theorem 4.6.1 (see Figure 4.16). However, the claim now is that the 3-SAT formula is satisfiable iff [L3,$\langle *(false.next), yes\rangle$] is in the complement of Intraprocedural Must Alias. By proof of Theorem 4.6.1 if the 3-SAT formula is satisfiable then there is a path to L3 on which $\langle *(false.next), no\rangle$ holds, and thus [L3,$\langle *(false.next), yes\rangle$] is not in Intraprocedural Must Alias. If the 3-SAT formula is unsatisfiable then on every path to L3, $\langle *(false.next), yes\rangle$ holds and thus [L3,$\langle *(false.next), yes\rangle$] is in Intraprocedural Must Alias.

□

4.7 Alias Sets Associated with Program Paths

If, instead of determining whether one alias pair holds on some path to a program point, we try to determine whether a set of alias pairs holds on some path to a program point, then the alias problem introduced by any of the previously considered sets of mechanisms is $\mathcal{NP}$-hard.

holds: A set of alias pairs $\mathcal{A}$ **holds** on the realizable path $\rho n_1 n_2 ... n_i$ iff for each $\langle a, b \rangle \in \mathcal{A}$, a and b refer to the same location at program point n_i whenever the execution sequence defined by the path occurs.

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

$\{[n, \mathcal{A}] \mid \exists$ a realizable path, $\rho n_1 n_2 ... n_{i-1} n$, in the ICFG on which $\mathcal{A}$ holds$\}$.

Intraprocedural May Set Alias: The precise solution for **Intraprocedural May Set Alias** is

$\{[n, \mathcal{A}] \mid \exists$ a path, $\rho n_1 n_2 ... n_{i-1} n$, in the CFG on which $\mathcal{A}$ holds$\}$.

4.7.1 Single Level Pointers

Theorem 4.7.1 *In the presence of* **single level pointers**, *the problem of determining precise* **Intraprocedural May Set Alias** *is $\mathcal{NP}$-hard.*

The proof of Theorem 4.7.1 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 4.17 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.

If we interpret $*v_i$ aliased to *true* as meaning the variable v_i is true, then any path from L1 to L2 can be interpreted as a truth assignment. Further, every truth assignment corresponds to some such path.

Now consider the paths from L2 to L3. If the truth assignment for the particular path from L1 to L2 satisfies the formula, then every clause has a literal which is true. Pick the path from L2 for which the i^{th} clause assigns its true literal to c_i. Thus each

```
    int *v_1,*v̄_1,*v_2,*v̄_2,..., *v_m,*v̄_m;
    int *c_1, *c_2, ...  ,*c_n;
    int true,false;

    /* A path through this section of code
        corresponds to a truth assignment */
L1:
    if (-) {v_1 = &true; v̄_1 = &false}
        else {v_1 = &false; v̄_1 = &true}
    if (-) {v_2 = &true; v̄_2 = &false}
        else {v_2 = &false; v̄_2 = &true}
                    .  .  .
    if (-) {v_m = &true; v̄_m = &false}
        else {v_m = &false; v̄_m = &true}

L2:

    /* The code below will create a { ⟨*c_i, true⟩ | 1 ≤ i ≤ n } alias
       iff the truth assignment from above makes the formula true*/

    if (-) c_1 = l_1,1† else if (-) c_1 = l_1,2 else c_1 = l_1,3;
    if (-) c_2 = l_2,1 else if (-) c_2 = l_2,2 else c_2 = l_2,3;
                              .  .  .
    if (-) c_n = l_n,1 else if (-) c_n = l_n,2 else c_n = l_n,3;

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 4.17: 3-SAT solution iff [L3,$\{\langle *c_i, true\rangle \mid 1 \leq i \leq n\}$] in Intraprocedural May Set Alias

assignment creates the alias $\langle *c_i, true \rangle$ and [L3,$\left\{ \langle *c_i, true \rangle \mid 1 \leq i \leq n \right\}$] must be in Intraprocedural May Set Alias. However if the formula is unsatisfiable then every truth assignment has a clause, say $(l_{j,1} \vee l_{j,2} \vee l_{j,3})$, with 3 false literals. This means $*l_{j,1}$, $*l_{j,2}$, and $*l_{j,3}$ are all aliased to $false$. Because every path from L2 to L3 must go through the statement

if (-) $c_j = l_{j,1}$ else if (-) $c_j = l_{j,2}$ else $c_j = l_{i,3}$;

the alias $\left\{ \langle *c_i, true \rangle \mid 1 \leq i \leq n \right\}$ never holds on a path to L3. Thus 3-SAT is polynomial reducible to Intraprocedural May Set Alias with single level pointers and Theorem 4.7.1 holds.

□

The following are easy corollaries of Theorem 4.7.1:

Corollary 4.7.1 *In the presence of* **single level pointers** *the problem of determining precise* **Interprocedural May Set Alias** *is $\mathcal{NP}$-hard.*

Corollary 4.7.2 *In the presence of* **multiple level pointers** *the problem of determining precise* **Intraprocedural May Set Alias** *is $\mathcal{NP}$-hard.*

Corollary 4.7.3 *In the presence of* **multiple level pointers** *the problem of determining precise* **Interprocedural May Set Alias** *is $\mathcal{NP}$-hard.*

Corollary 4.7.4 *In the presence of* **structures containing single level pointers** *the problem of determining precise* **Intraprocedural May Set Alias** *is $\mathcal{NP}$-hard.*

Corollary 4.7.5 *In the presence of* **structures containing single level pointers** *the problem of determining precise* **Interprocedural May Set Alias** *is $\mathcal{NP}$-hard.*

4.7.2 Reference Formals

Theorem 4.7.2 *In the presence of* **reference formals**, *the problem of determining precise* **Interprocedural May Set Alias** *is $\mathcal{NP}$-hard.*

The proof of Theorem 4.7.2 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 $\{v_1, v_2, ..., v_m\}$. It is conceptually identical to the proof of Theorem 4.7.1. The reduction is specified by the program in Figure 4.18[36] 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.

If we interpret v_i aliased to $true$ as meaning the variable v_i is true, then any path from L1 to L2 can be interpreted as a truth assignment. Further, every truth assignment corresponds to some such path.

Now consider the paths from L2 to L3. If the truth assignment for the particular path from L1 to L2 satisfies the formula then every clause has a literal which is true. Pick the path from L2 for which the $clause_{i-1}$ passes the true literal of the i^{th} clause to $clause_i$. This call corresponds to passing $true$ to c_i and [L3,$\left\{ \langle c_i, true \rangle \;\middle|\; 1 \leq i \leq n \right\}$] must be in Interprocedural May Set Alias. However if the formula is unsatisfiable then every truth assignment has a clause, say $(l_{j,1} \vee l_{j,2} \vee l_{j,3})$, with 3 false literals. This means $l_{j,1}$, $l_{j,2}$, and $l_{j,3}$ are all aliased to $false$. Because every path from L2 to L3 must go through $clause_{j-1}$ the alias $\left\{ \langle c_i, true \rangle \;\middle|\; 1 \leq i \leq n \right\}$ never holds on a path to L3. Thus 3-SAT is polynomial reducible to Interprocedural May Set Alias with reference formals and Theorem 4.7.2 holds.

□

4.8 $\mathcal{P}$-space-hard alias problems

In this Chapter we have proved May and Must Alias in the presence of various language constructs to be $\mathcal{NP}$-hard (see Table 4.1, p. 27). A natural question to ask is whether these problems are $\mathcal{NP}$-complete or not. Unless $\mathcal{P}$-space = $\mathcal{NP}$-time, the answer is no. $\mathcal{P}$-space is the set of all languages accepted by polynomial space bounded deterministic Turing machines (DTMs), and $\mathcal{NP}$-space is the set of all languages accepted by polynomial space bounded nondeterministic Turing machines (NDTMs). $\mathcal{NP}$-space $\equiv$ $\mathcal{P}$-space ([AHU74], p. 395). In this section we show that the $\mathcal{NP}$-hard problems

[36]Figure 4.19 is an alternate reduction that works for C scope rules.

```
        int true,false;

        /* A path through this section of code corresponds to a truth assignment */
        var_1(v_1, v̄_1)
              ref† int v_1, v̄_1;
        {
              if (-) var_2(true,false) else var_2(false,true);
        }
        var_2(v_2, v̄_2)
              ref int v_2, v̄_2;
        {
              if (-) var_3(true,false) else var_3(false,true)
        }
                                        . . .
        var_m(v_m, v̄_m)
              ref int v_m, v̄_m;
        {
              clause_0()
        }

        /* The code below will create a { ⟨c_i, true⟩ | 1 ≤ i ≤ n } alias iff
           the truth assignment from above makes the formula true */
        clause_0()
        {
L2:           if (-) clause_1(l_{1,1}‡) else if (-) clause_1(l_{1,2}) else clause_1(l_{1,3})
        }
        clause_1(c_1)
              ref int c_1;
        {
              if (-) clause_2(l_{2,1}) else if (-) clause_2(l_{2,2}) else clause_2(l_{2,3})
        }
        clause_2(c_2)
              ref int c_2;
        {
              if (-) clause_3(l_{3,1}) else if (-) clause_3(l_{3,2}) else clause_3(l_{3,3})
        }
                                        . . .
        clause_n(c_n)
              ref int c_n;
        {
L3:     }
        main() {
L1:           if (-) var_1(true,false) else var_1(false,true);
        }
```

Note: In order for proof to work $clause_0()$ and all $clause_k(c_k)[1 \leq k \leq n]$ must be in the scope of $v_1, v_2, ..., v_m, \overline{v_1}, ...,$ and $\overline{v_m}$. Also $clause_n(c_n)$ must be in the scope of $c_1, c_2, ...,$ and c_n.
Thus with Pascal-like scoping rules these procedures would have to be nested.

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

Figure 4.18: 3-SAT solution iff [L3,$\{\langle c_i, true\rangle \mid 1 \leq i \leq n\}$] in Interprocedural May Set Alias

```
        int true,false;

        /* A path through this section of code corresponds to a truth assignment */
        var_1(v_1, v̄_1)
              ref† int v_1, v̄_1;
        {
              if (-) var_2(v_1, v̄_1,true,false) else var_2(v_1, v̄_1,false,true);
        }
        var_2(v_1, v̄_1, v_2, v̄_2)
              ref int v_1, v̄_1, v_2, v̄_2;
        {
              if (-) var_3(v_1, v̄_1, v_2, v̄_2,true,false) else var_3(v_1, v̄_1, v_2, v̄_2,false,true)
        }
                                        . . .
        var_m(v_1, v̄_1, v_2, v̄_2, ..., v_m, v̄_m)
              ref int v_1, v̄_1, v_2, v̄_2, ..., v_m, v̄_m;
        {
              clause_0(v_1, v̄_1, v_2, v̄_2, ..., v_m, v̄_m)
        }

        /* The code below will create a { ⟨c_i, true⟩ | 1 ≤ i ≤ n } alias iff
           the truth assignment from above makes the formula true */
        clause_0(v_1, ..., v̄_m)
              ref int v_1, ..., v̄_m;
        {
L2:           if (-) clause_1(v_1, ..., v̄_m, l_1,1‡) else if (-) clause_1(v_1, ..., v̄_m, l_1,2)
                  else clause_1(v_1, ..., v̄_m, l_1,3)
        }
        clause_1(v_1, ..., v̄_m, c_1)
              ref int v_1, ..., v̄_m, c_1;
        {
              if (-) clause_2(v_1, ..., v̄_m, c_1, l_2,1) else if (-) clause_2(v_1, ..., v̄_m, c_1, l_2,2)
                  else clause_2(v_1, ..., v̄_m, c_1, l_2,3)
        }
        clause_2(v_1, ..., v̄_m, c_1, c_2)
              ref int v_1, ..., v̄_m, c_1, c_2;
        {
              if (-) clause_3(v_1, ..., v̄_m, c_1, c_2, l_3,1) else if (-) clause_3(v_1, ..., v̄_m, c_1, c_2, l_3,2)
                  else clause_3(v_1, ..., v̄_m, c_1, c_2, l_3,3)
        }
                                        . . .
        clause_n(v_1, ..., v̄_m, c_1, c_2, ..., c_n)
              ref int v_1, ..., v̄_m, c_1, c_2, ..., c_n;
        {
L3:     }
        main() {
L1:           if (-) var_1(true,false) else var_1(false,true);
        }
```

† **ref** indicates reference formal

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

Figure 4.19: 3-SAT solution iff [L3,$\{\langle c_i, true\rangle \mid 1 \leq i \leq n\}$] in Interprocedural May Set Alias

from Table 4.1 are at least $\mathcal{P}$-space hard, if four levels of indirection are possible. We show that some are $\mathcal{P}$-space complete, but for many of these problems, it is open as to whether they are complete for $\mathcal{P}$-space or not.

Theorem 4.8.1 *In the presence of multiple level pointers with at least four levels of indirection, Intraprocedural May Alias is complete for $\mathcal{P}$-space under the common assumptions of static analysis.*

Theorem 4.8.2 *In the presence of multiple level pointers with at least four levels of indirection, Intraprocedural Must Alias is complete for $\mathcal{P}$-space under the common assumptions of static analysis.*

We will prove Theorem 4.8.1 and Theorem 4.8.2 concurrently. Since $\mathcal{P}$-space is defined on deterministic Turing machines (DTMs), $\mathcal{P}$-space is closed under complementation. Thus for Theorem 4.8.2 we will show that the complement of Intraprocedural Must Alias is complete for $\mathcal{P}$-space.

First we show that both problems are in $\mathcal{P}$-space. Consider the following polynomial space bounded NDTM M: Given a Control Flow Graph (CFG) G with start node ρ, M starts with the path ρ and nondeterministically generates a legal path through G. With each new node of G on the path, M computes the aliases on the new path given the aliases on the old path. To do this, all that M needs to remember about the path is the last node on the path and the aliases at the end of the path. For any program with a less than k (some constant) dereferences, there are only a polynomial number of possible alias pairs and thus M only needs a polynomial amount of storage. For any given node/alias pair, $[node, \langle a, b\rangle]$, $[node, \langle a, b\rangle]$ is in Intraprocedural May Alias iff M can generate some path ending in $node$ for which $\langle a, b\rangle$ is in the alias solution. $[node, \langle a, b\rangle]$ is in the complement of Intraprocedural Must Alias iff M can generate some path ending in $node$ and $\langle a, b\rangle$ is not in the alias solution. Since there are only a polynomial number of such pairs, by asking if $[node, \langle a, b\rangle]$ is in Intraprocedural May Alias and the complement of Intraprocedural Must Alias we generate the full solutions to both of these problems. Thus Intraprocedural May Alias and Intraprocedural Must Alias (because $\mathcal{P}$-space is closed under complementation) are both in $\mathcal{P}$-space.

To show that these problems are $\mathcal{P}$-space hard is more complicated. Our proof uses the fact that a certain language is known to be $\mathcal{P}$-space complete and we will reduce that language to our problems. If R is a regular expression, let $L(R)$ denote the language R represents. Also let $\mathcal{L}$ be the set of regular expressions, R, such that $\overline{L(R)} \neq \emptyset$. $\mathcal{L}$ is complete for $\mathcal{P}$-space [AHU74]. We will reduce $\mathcal{L}$ to various alias problems. We will need the following lemma throughout this section.

Lemma 4.8.1 *Given a regular expression R with n symbols and operators, we can construct a NFA $M = (S,\Sigma,\Delta,s_0,F)$ in time $\mathcal{O}(n)$ such that M accepts $L(R)$, $|S| \leq 2 * n$, for all $s \in S$ in-degree(s) $\leq$ 2, and for all $\sigma \in \Sigma$ (i.e. $\sigma \neq \epsilon$) if $(s_i,\sigma,s_j) \in \Delta$ then in-degree(s_j) = 1.*

Theorem 9.2 (pp. 322-323) if [AHU74] is identical to Lemma 4.8.1 except we have the additional claim that

$$\text{for all } \sigma \in \Sigma \text{ if } (s_i,\sigma,s_j) \in \Delta \text{ then in-degree}(s_j) = 1$$

An examination of the NFA construction in [AHU74] will verify that the above claim is true.

□

We now reduce $\mathcal{L}$ to Intraprocedural May Alias and the complement of Intraprocedural Must Alias. Given a regular expression R with n symbols and operators over alphabet σ ($|\Sigma| \leq n$) build an NFA $M = (S,\Sigma,\Delta,s_0,F)$ meeting the requirements of Lemma 4.8.1. The program in Figure 4.20 simulates M.

The key ideas are to use paths through the program in Figure 4.20 to represent various strings over Σ and, given a particular path P through the program which represents string w, to use the alias information to capture all possible states of M which M could achieve given input w. In this way, for any given path P which represents some string w, we can determine if w is in $L(R)$ or not, and, since May Alias requires the existence of an alias on some path, we can also determine whether there exists some path representing a string which is not in $L(R)$. This last part is true iff R is in $\mathcal{L}$. The size of this program is dominated by the ϵ-`closure()` macro which is of size $\mathcal{O}(|S| * |\Delta|)$ = $\mathcal{O}(n^2)$. It can be generated from R in time polynomial with respect to n.

```
/* Given regular expression R, M = (S,Σ,Δ,s0,F) is the NFA of Lemma 4.8.1. */
     #define macro ε-closure()
     #repeat physically |S| times
      if (-)                                          }
        {*sj = *si; sj = si;}                         }   once for each
      else *si = *sj;                                 }   (si,ε,sj) ∈ Δ such that
      not_in_state = &no;                             }   in-degree(sj) = 1
      if (-)                                          }
        {*sj = *si; *sk = *si; sj = si;}              }   once for each
      else if (-)                                     }   (si,ε,sj) ∈ Δ and
        {*sj = *sk; *si = *sk; sj = sk;}              }   (sk,ε,sj) ∈ Δ such that
      else {*si = *sj; *sk = *sj;}                    }   si ≠ sk and
      not_in_state = &no;                             }   in-degree(sj) = 2
     #end repeat
     #end macro

     int ****nc;
     int ***σ;                                        once for each σ ∈ Σ
     int **s,**s';                                    once for each s ∈ S
     int *in_state,*not_in_state;
     int yes,no,**temp;

     in_state = &yes; not_in_state = &no;
     σ = &temp;                                       once for each σ ∈ Σ
     s = &not_in_state;                               once for each s ∈ S
     s0 = &in_state;
     ε-closure()
l1:  while (-) {
      if (-) nc = &σ1;                                }
      else if (-) nc = &σ2;                           }   Let Σ = {σ1,...,σm}
      ...                                             }
      else nc = &σm;                                  }
      s' = &not_in_state;                             once for each s ∈ S
      σ = &s'j;                                       }
      **nc = si;                                      }   once for each
      σ = &temp;                                      }   (si,σ,sj) ∈ Δ (σ ∈ Σ)
      s = s';                                         once for each s ∈ S
l4:   ε-closure()
     }
l2:
     *sf = &no;                                       once for each sf ∈ F
l3:
```

Figure 4.20: Reduction of $\mathcal{L}$ to an alias problem

The following variables are declared in Figure 4.20.

```
int ****nc;
int ***σ;                    once for each σ ∈ Σ
int **s,**s';                once for each s ∈ S
int *in_state,*not_in_state;
int yes,no,**temp;
```

In the program each letter $\sigma \in \Sigma$ has a variable (called σ). Some string will be generated by the `while` loop; each pass through the loop adds another character to the string. `*nc` is always aliased to the next character of the string. For any path P to $\mathtt{l}_2$ where $\mathtt{l}_1$ has been executed at least $m+1$ times, let $string_m = c_1 c_2 ... c_m$ where $c_i \in \Sigma$ $(1 \leq i \leq m)$ is the character referred to by `nc` on the i^{th} iteration of the `while` loop on path P. $string_m$ has no representation in the program, but it refers to the initial substring of the input that M has read so far.

There are two variables for each state `s` in `S` called `s` and `s'`. (Ideally we would like to have had `*s` aliased to `in_state` iff $(\mathtt{s}_0, string) \vdash_M^* (\mathtt{s}, \epsilon)$, and thus simulate M. This wasn't possible, but we came up with a sufficient compromise.) We use the alias $\langle$`*in_state, yes`$\rangle$ holding on a path P to represent the fact that P is a valid simulation of M. If some path P up to l_1 is a valid simulation of M (i.e., `*in_state` is aliased to `yes` on P) then `*s` is aliased to `in_state` iff $(\mathtt{s}_0, string_m) \vdash_M^* (\mathtt{s}, \epsilon)$, where l_1 appears $m+1$ times on P. All `s'`, as well as `temp`, are just temporary work areas.

We now present some Lemmas about the program in Figure 4.20.

Lemma 4.8.2 *For all non-trivial paths P for the program in Figure 4.20,* $\langle$`*in_state, yes`$\rangle$ *holds on P iff* $\langle$`*in_state, no`$\rangle$ *does not.*

There are only two "`int`" variables in the program; `yes` and `no`. Since `in_state` is type "`int *`", it has to refer to one or the other or nothing at all. An examination of the code makes it obvious that `in_state` must always refer to something.

□

Lemma 4.8.3 *For any path P up to (but not including) the code for* ϵ`-closure()`[37], *let* $S_P = \left\{ s \;\middle|\; \langle \mathtt{*s, in_state} \rangle \text{ holds on } P \right\}$.

[37] That is ϵ`-closure()` is the very next thing to be executed.

- *If* $\langle *\texttt{in_state}, \texttt{yes}\rangle$ *holds on* P *then there is a path* P' *through* ϵ-`closure()` *such that* $\langle *\texttt{in_state}, \texttt{yes}\rangle$ *holds on* $P \circ P'$
- *For all paths* P' *through* ϵ-`closure()`, *if* $\langle *\texttt{in_state}, \texttt{yes}\rangle$ *holds on* $P \circ P'$ *then for all* s_i $\langle *\texttt{s}_i, \texttt{in_state}\rangle$ *iff* $s_i \in \left\{ s' \;\middle|\; \text{for some } s \in S_P,\ (s,\epsilon) \vdash_M^* (s',\epsilon) \right\}$.

Consider $(\texttt{s}_i, \epsilon, \texttt{s}_j) \in \Delta$ where in-degree($\texttt{s}_i$) = 1. The following code represents adding $\texttt{s}_i$ to the ϵ-`closure()` if $\texttt{s}_j$ is in it:

```
if (-)
  {*sj = *si; sj = si;}
else *si = *sj;
not_in_state = &no;
```

Besides adding $\texttt{s}_j$ to the ϵ-`closure()` if $\texttt{s}_i$ is in it, the above code needs to insure that if $\texttt{s}_i$ was in the ϵ-`closure()` before this code, it is also in the ϵ-`closure()` after the code. There are four possible relevant alias conditions before this code is executed (remember $\langle *\texttt{not_in_state}, \texttt{no}\rangle$ holds on all paths[38]).

aliases before executed	`then` path	`else` path
$\langle *\texttt{s}_j, \texttt{not_in_state}\rangle$ $\langle *\texttt{s}_i, \texttt{not_in_state}\rangle$	$\langle *\texttt{s}_j, \texttt{not_in_state}\rangle$	$\langle *\texttt{s}_j, \texttt{not_in_state}\rangle$
$\langle *\texttt{s}_j, \texttt{not_in_state}\rangle$ $\langle *\texttt{s}_i, \texttt{in_state}\rangle$	$\langle *\texttt{s}_j, \texttt{in_state}\rangle$	$\langle *\texttt{s}_j, \texttt{not_in_state}\rangle$ $\langle *\texttt{in_state}, \texttt{no}\rangle$
$\langle *\texttt{s}_j, \texttt{in_state}\rangle$ $\langle *\texttt{s}_i, \texttt{not_in_state}\rangle$	$\langle *\texttt{s}_j, \texttt{not_in_state}\rangle$ $\langle *\texttt{in_state}, \texttt{no}\rangle$	$\langle *\texttt{s}_j, \texttt{in_state}\rangle$
$\langle *\texttt{s}_j, \texttt{in_state}\rangle$ $\langle *\texttt{s}_i, \texttt{in_state}\rangle$	$\langle *\texttt{s}_j, \texttt{in_state}\rangle$	$\langle *\texttt{s}_j, \texttt{in_state}\rangle$

To achieve the desired effect, $*\texttt{s}_j$ should be aliased to `not_in_state` after execution of this code iff $*\texttt{s}_j$ and $*\texttt{s}_i$ are aliased to `not_in_state` before its execution. In all cases, at least one branch through this code meets the above requirements and any path that doesn't, destroys the alias $\langle *\texttt{in_state}, \texttt{yes}\rangle$ by creating $\langle *\texttt{in_state}, \texttt{no}\rangle$. The code for states with two in-edges can be justified similarly. Figure 4.21 gives the same analysis as above except, for space reasons, we use the abbreviations listed at the top of the figure. In the figure we list the relevant alias situations before execution of the code:

[38] with possible local exceptions

```
if (-)
  {*s_j = *s_i; *s_k = *s_i; s_j = s_i;}
else if (-)
  {*s_j = *s_k; *s_i = *s_k; s_j = s_k;}
else {*s_i = *s_j; *s_k = *s_j;}
not_in_state = &no;
```

for some $(s_i,\epsilon,s_j) \in \Delta$ and $(s_k,\epsilon,s_j) \in \Delta$ such that $s_i \neq s_k$ and in-degree(s_j) = 2

We then specify the desired alias situation after execution of the above code as well as the alias situation achieved by each path through the code. One particular abbreviation deserves comment, *error* represents the fact that $\langle *\texttt{in_state}, \texttt{no}\rangle$ holds on paths indicated by Figure 4.21. When $\langle *\texttt{in_state}, \texttt{no}\rangle$ holds on a path, $\langle *\texttt{in_state}, \texttt{yes}\rangle$ does not (in_state can not have two different values on the same path) and such paths are not a valid simulation of M.

Thus if $\langle *\texttt{in_state}, \texttt{yes}\rangle$ holds on P then on some path P'_i through the i^{th} copy of

```
if (-)
  {*s_j = *s_i; s_j = s_i;}
else *s_i = *s_j;
not_in_state = &no;
```

once for each $(s_i,\epsilon,s_j) \in \Delta$ such that in-degree(s_j) = 1

```
if (-)
  {*s_j = *s_i; *s_k = *s_i; s_j = s_i;}
else if (-)
  {*s_j = *s_k; *s_i = *s_k; s_j = s_k;}
else {*s_i = *s_j; *s_k = *s_j;}
not_in_state = &no;
```

once for each $(s_i,\epsilon,s_j) \in \Delta$ and $(s_k,\epsilon,s_j) \in \Delta$ such that $s_i \neq s_k$ and in-degree(s_j) = 2

in ϵ-closure() (by simple induction on i):

- $\langle *\texttt{in_state}, \texttt{yes}\rangle$ holds on $P \circ P'_i$.
- $\langle *\texttt{s}, \texttt{in_state}\rangle$ holds on $P \circ P'_i$ if $(s_z,\epsilon) \vdash_M^k (s,\epsilon)$ for some $0 \leq k \leq i$ and $*s_z$ aliased to in_state immediately before the execution of ϵ-closure().
- $\langle *\texttt{s}, \texttt{in_state}\rangle$ holds on $P \circ P'_i$ only if $(s_z,\epsilon) \vdash_M^* (s,\epsilon)$ for some $*s_z$ aliased to in_state immediately before the execution of ϵ-closure().

In addition for all paths for which the above is not true, *in_state is aliased to no and not yes. Clearly, $(s_z,\epsilon) \vdash_M^* (s,\epsilon)$ iff $(s_z,\epsilon) \vdash_M^i (s,\epsilon)$ for some i, $0 \leq i \leq |S|$, because otherwise we have a cycle involving just ϵ-transitions and we don't need to consider paths with ϵ-cyles. Therefore, the code for ϵ-closure() satisfies Lemma 4.8.3.

□

Let $(\mathtt{s}_i, \epsilon, \mathtt{s}_j) \in \Delta$, $(\mathtt{s}_k, \epsilon, \mathtt{s}_j) \in \Delta$, and $\mathtt{s}_i \neq \mathtt{s}_k$
Let $\mathtt{s}_k$-in $\equiv \langle *\mathtt{s}_k, \mathtt{in_state} \rangle$ for $k = 1$, 2, or j
Let $\mathtt{s}_k$-not-in $\equiv \langle *\mathtt{s}_k, \mathtt{not_in_state} \rangle$ for $k = 1$, 2, or j
Let error $\equiv \langle *\mathtt{in_state}, \mathtt{no} \rangle$ and thus breaking $\langle *\mathtt{in_state}, \mathtt{yes} \rangle$

aliases before executed	desired $\mathtt{s}_j$ alias after execution	`then` path	`else-then` path	`else-else` path
$\mathtt{s}_j$-not-in $\mathtt{s}_i$-not-in $\mathtt{s}_k$-not-in	$\mathtt{s}_j$-not-in	$\mathtt{s}_j$-not-in	$\mathtt{s}_j$-not-in	$\mathtt{s}_j$-not-in
$\mathtt{s}_j$-not-in $\mathtt{s}_i$-not-in $\mathtt{s}_k$-in	$\mathtt{s}_j$-in	$\mathtt{s}_j$-not-in error	$\mathtt{s}_j$-in	$\mathtt{s}_j$-not-in error
$\mathtt{s}_j$-not-in $\mathtt{s}_i$-in $\mathtt{s}_k$-not-in	$\mathtt{s}_j$-in	$\mathtt{s}_j$-in	$\mathtt{s}_j$-not-in error	$\mathtt{s}_j$-not-in error
$\mathtt{s}_j$-not-in $\mathtt{s}_i$-in $\mathtt{s}_k$-in	$\mathtt{s}_j$-in	$\mathtt{s}_j$-in	$\mathtt{s}_j$-in	$\mathtt{s}_j$-not-in error
$\mathtt{s}_j$-in $\mathtt{s}_i$-not-in $\mathtt{s}_k$-not-in	$\mathtt{s}_j$-in	$\mathtt{s}_j$-not-in error	$\mathtt{s}_j$-not-in error	$\mathtt{s}_j$-in
$\mathtt{s}_j$-in $\mathtt{s}_i$-not-in $\mathtt{s}_k$-in	$\mathtt{s}_j$-in	$\mathtt{s}_j$-not-in error	$\mathtt{s}_j$-in	$\mathtt{s}_j$-in
$\mathtt{s}_j$-in $\mathtt{s}_i$-in $\mathtt{s}_k$-not-in	$\mathtt{s}_j$-in	$\mathtt{s}_j$-in	$\mathtt{s}_j$-not-in error	$\mathtt{s}_j$-in
$\mathtt{s}_j$-in $\mathtt{s}_i$-in $\mathtt{s}_k$-in	$\mathtt{s}_j$-in	$\mathtt{s}_j$-in	$\mathtt{s}_j$-in	$\mathtt{s}_j$-in

Figure 4.21: Justification of code for state $\mathtt{s}_j$ with 2 in-edges

Lemma 4.8.4 *For all $w \in \Sigma^*$ there exists a path P through the program in Figure 4.20 to* `l2` *on which* `l1` *appears m ($m = |w| + 1$) times, $string_{m-1} = w$, and $\langle$*`*in_state, yes`$\rangle$ *holds on P.*

Quick inspection of the code shows that the only places where a variable with type "`int *`" can be directly or indirectly assigned are initialization, after `l2`, and in ϵ`-closure()`. Since `in_state` is of type "`int *`", the only place where the alias $\langle$`*in_state, yes`$\rangle$ can be destroyed (before `l2` is executed) is in ϵ`-closure()`. Thus by Lemma 4.8.3, Lemma 4.8.4 holds.

□

Lemma 4.8.5 *For all paths P through the program in Figure 4.20 to* `l1` *on which* `l1` *appears m times, if $\langle$*`*in_state, yes`$\rangle$ *holds on P then $\langle$*`*s, in_state`$\rangle$ *holds on P iff $(s_0, string_{m-1}) \vdash_M^* (s, \epsilon)$.*

Consider the meaning of a path $P_m = P_{m-1} \circ$ ... `l1` through the program in Figure 4.20 to `l1` where `l1` occurs m times and if $m > 0$ then `l1` occurs $m - 1$ times on P_{m-1} which also ends at `l1`. Proof of Lemma 4.8.5 is by induction on m.

Basis:

The beginning of the initialization phase is:

```
in_state = &yes;
not_in_state = &no;
σ = &temp;            once for each σ ∈ Σ
```

The alias $\langle$`*in_state, yes`$\rangle$ represents the fact that the path is thus far a legal simulation of M. The alias $\langle$`*not_in_state, no`$\rangle$ is an invariant. There are actually a few very local places where $\langle$`*not_in_state, yes`$\rangle$, but when this happens the invariant alias is immediately restored. The assignment "`σ = &temp;`" simply points all σ to some unimportant location.

The rest of the initialization phase is:

```
s = &not_in_state;   once for each s ∈ S
s0 = &in_state;
ε-closure()
```

This code states that initially M is in state s_0 (by creating the alias $\langle *s_0, \text{in_state}\rangle$). By Lemma 4.8.3, depending on which path is executed, ϵ-`closure()` either destroys the alias $\langle$`*in_state, yes`$\rangle$ or creates the alias $\langle *s, \text{in_state}\rangle$ for each state `s` such that $(s_0,\epsilon) \vdash_M^* (s,\epsilon)$. Fortunately, on at least one path the latter will be true and Lemma 4.8.5 holds.

Induction Hypothesis: Lemma 4.8.5 holds for P where l_1 occurs $m-1$ times.

Induction Step:

As stated before each path through the `while` loop represents some string. If $\langle$`*in_state, yes`$\rangle$ does not hold on P_m then the lemma follows. Assume that it does hold on P_m. It must also hold on P_{m-1} because this alias can ONLY be created during initialization. By induction, we get FACT$_1$: $(s_0,\text{string}_{m-2}) \vdash_M^* (s,\epsilon)$ iff $\langle *s, \text{in_state}\rangle$ holds on P_{m-1}.

The code:

$$\left.\begin{array}{l}\text{if (-) nc} = \&\sigma_1;\\ \text{else if (-) nc} = \&\sigma_2;\\ \ldots\\ \text{else nc} = \&\sigma_m;\end{array}\right\} \quad \text{Let } \Sigma = \{\sigma_1,\ldots,\sigma_m\}$$

simply generates the next character of the string being considered. FACT$_2$: The code

$$\begin{array}{ll}s' = \&\text{not_in_state}; & \text{once for each } s \in S\\ \left.\begin{array}{l}\sigma = \&s_j';\\ **\text{nc} = s_i;\\ \sigma = \&\text{temp};\end{array}\right\} & \begin{array}{l}\text{once for each}\\ (s_i,\sigma,s_j) \in \Delta\ (\sigma \in \Sigma)\\ \text{unique for } s_j \text{ by Lemma 4.8.1}\end{array}\end{array}$$

simulates the effects of all $(s_i,\sigma,s_j) \in \Delta\ (\sigma \in \Sigma)$. It does this by aliasing $*s_j'$ (for $(s_i,\sigma,s_j) \in \Delta$) to `in_state` iff $*s_i$ is aliased to `in_state` and the next character is σ (i.e., `*nc` is aliased to σ). Consider "`**nc = `s_i`;`"; if `*nc` is aliased to σ then `**nc` will be aliased to s_j' otherwise `**nc` will be aliased to `temp`. "`**nc = `s_i`;`" can therefore only change the value of s_j' iff `*nc` is aliased to σ. Assuming `*nc` is aliased to σ, "`**nc = `s_i`;`" aliases $*s_j'$ to `in_state` iff $*s_i$ is aliased to `in_state`.

FACT$_1$ and FACT$_2$ imply $(s_0,\text{string}_{m-1}) \vdash_M^* (s,nc_{m-1}) \vdash_M (s',\epsilon)$ iff $\langle$`*s', in_state`$\rangle$ holds on P_m up to the last l_4. The end of the loop is

```
        s = s';          once for each s ∈ S
l4:     ε-closure()
```

and thus given Lemma 4.8.3, Lemma 4.8.5 holds for P_m.

□

We now can finish the proof of Theorem 4.8.1 and Theorem 4.8.2.

Claim 1: $R \in \mathcal{L}$ iff $[l_3, \langle *\texttt{in_state}, \texttt{yes}\rangle]$ is in the precise Intraprocedural May Alias solution.

Claim 2: $R \in \mathcal{L}$ iff $[l_3, \langle *\texttt{in_state}, \texttt{no}\rangle]$ is in the complement of the precise Intraprocedural Must Alias solution.

- Assume $R \in \mathcal{L}$.

  There must exist some $w \in \Sigma^*$ such that $w \notin \mathrm{L}(R)$. By Lemma 4.8.4 there is a path $P = \rho...l_1 l_2$ on which $\texttt{l}_1$ is executed m times, $string_{m-1} = w$, and $\langle *\texttt{in_state}, \texttt{yes}\rangle$ holds on P. By Lemma 4.8.5 (because the same aliases that hold on $P = \rho...l_1 l_2$ also hold on $\rho...l_1$), for all states $\texttt{s}$ $\langle *\texttt{s}, \texttt{in_state}\rangle$ holds on P iff $(\mathrm{s}_0, string_{m-1}) \vdash_M^* (\mathrm{s}, \epsilon)$. Since $w \notin \mathrm{L}(R)$, all $*\texttt{s}$ aliased to $\texttt{in_state}$ are not in F (i.e., no such s is an accepting state) and thus the code after l_2 can not change the value of $\texttt{in_state}$ and $\langle *\texttt{in_state}, \texttt{yes}\rangle$ holds on P extended to $\texttt{l}_3$. Therefore, $[\texttt{l}_3, \langle *\texttt{in_state}, \texttt{yes}\rangle]$ is in May Alias and $[\texttt{l}_3, \langle *\texttt{in_state}, \texttt{no}\rangle]$ is in the complement of Must Alias.

- Assume $R \notin \mathcal{L}$.

  Assume for the program in Figure 4.20 there was a path $P = P_{l_1} \circ l_2...l_3$ ($P_{l_1} = \rho...l_1$) on which $\langle *\texttt{in_state}, \texttt{yes}\rangle$ holds and $\texttt{l}_1$ appears m times. Since the code after $\texttt{l}_2$ can only assign &no (and yes can never be an alias of no), $\langle *\texttt{in_state}, \texttt{yes}\rangle$ also holds on P_{l_1}. By Lemma 4.8.5, for all states $\texttt{s}$ $\langle *\texttt{s}, \texttt{in_state}\rangle$ holds on P_{l_1} iff $(\mathrm{s}_0, string_{m-1}) \vdash_M^* (\mathrm{s}, \epsilon)$. Since $w \in \mathrm{L}(R)$, for some $\mathrm{s}_f \in \mathrm{F}$ $\langle *\mathrm{s}_f, \texttt{in_state}\rangle$ must hold on P_{l_1}. The code between $\texttt{l}_2$ and $\texttt{l}_1$ must assign $*\mathrm{s}_f$ and therefore $\texttt{in_state}$ the value &no and $\langle *\texttt{in_state}, \texttt{yes}\rangle$ can not hold on P. Contradiction. Thus on all paths P $\langle *\texttt{in_state}, \texttt{yes}\rangle$ does not hold and by Lemma 4.8.2 $\langle *\texttt{in_state}, \texttt{no}\rangle$ does. Therefore, $[\texttt{l}_3, \langle *\texttt{in_state}, \texttt{yes}\rangle]$ is not in May Alias and $[\texttt{l}_3, \langle *\texttt{in_state}, \texttt{no}\rangle]$ is not in the complement of Must Alias.

□

Some easy corollaries to Theorem 4.8.1 and Theorem 4.8.2 are:

Corollary 4.8.1 *In the presence of multiple level pointers with at least four levels of indirection, both Interprocedural May Alias and Interprocedural Must Alias are $\mathcal{P}$-space-hard under the common assumptions of static analysis.*

Corollary 4.8.2 *In the presence of recursive data types, both Intraprocedural May Alias and Intraprocedural Must Alias are $\mathcal{P}$-space-hard under the common assumptions of static analysis.*

Corollary 4.8.3 *In the presence of recursive data types, both Interprocedural May Alias and Interprocedural Must Alias are $\mathcal{P}$-space-hard under the common assumptions of static analysis.*

Chapter 5

A Safe Approximate Algorithm for Interprocedural May Alias

5.1 The may-hold Relation

5.1.1 Conditional May Alias: Definition

In Chapter 4.2.3, we present a two step algorithm for finding Interprocedural May Alias on programs that have single level pointers as the only mechanism for creating aliases. The first step of this algorithm is to compute **Conditional May Alias** which is defined as the answer to the question: *If there is a path to the entry node of the procedure containing n_i on which every alias in the set $\mathcal{AA}$ holds, then may object name a be aliased to object name b on some path to n_i?* We represent the answer to this question by the boolean relation $holds([\langle n_i, \mathcal{AA}\rangle, \langle a, b\rangle])$. Fortunately, it is only necessary to consider $\mathcal{AA}$ (sets of aliases) with cardinality less than or equal to one (Lemma 4.2.1 (p. 34)). In the second step, we use Conditional May Alias to solve for May Alias using a simple fixed point calculation. This algorithm is precise under the standard assumptions of data flow analysis (i.e., up to symbolic execution).

We will use the single level pointer algorithm for determining May Alias as the basis of a safe Interprocedural May Alias algorithm for programs with general pointers. There are two major obstacles to this goal. The first obstacle is one of precision and safety. Once the restriction to single level pointers is lifted, the assumption that it is only necessary to consider sets of assumed aliases ($\mathcal{AA}$) of size less than or equal one is no longer valid. However, if some alias $\mathcal{PA}$ depends on an $\mathcal{AA}$ with multiple assumed aliases, then all of them are sufficient and any single assumption necessary. Thus any single assumption can safely be considered as the only assumption. Thus, we

will continue to deal with sets of assumed aliases of size one or less, but this will now be an approximation.

The second obstacle is one of practicality. As the algorithm is described in Chapter 4.2.3, we must compute $holds([(node, \mathcal{AA}), \mathcal{PA}])$ for every possible triple: node in the ICFG, $\mathcal{AA}$, and $\mathcal{PA}$. So for a program with $\mathcal{O}(n)$ ICFG nodes and $\mathcal{O}(v)$ variables (and thus $\mathcal{O}(v^2)$ different possible $\mathcal{AA}$ and $\mathcal{O}(v^2)$ possible $\mathcal{PA}$), this means a minimum of $\mathcal{O}(n * v^4)$ work for even just the single level pointer case[39].

To insure efficiency, we will present our algorithm differently than in Chapter 4.2.3. We will calculate *holds* in a demand driven fashion so that we will only concern ourselves with $holds([(node, \mathcal{AA}), \mathcal{PA}])$ which have the value $true$. Since most of the holds relation will have the value $false$, this will improve the average time complexity of our algorithm immensely, although unfortunately, it does not help the worst case complexity.

We also improve efficiency by changing our definition of Conditional May Alias from:

> If there is a path to the entry node of the procedure containing n_i on which the assumed alias $\mathcal{AA}$ holds, then there is a path to n_i on which $\langle a, b \rangle$ holds.

to:

> If there is a path to the entry node of the procedure containing n_i on which every alias in the set $\mathcal{AA}$ holds, then there is a path to n_i on which $\langle a, b \rangle$ holds **and there is a path to the entry node of the procedure containing n_i on which the assumed alias set $\mathcal{AA}$ holds.**

Since we have changed the definition of Conditional May Alias, we will change our notation: $may\text{-}hold([(n_i, \mathcal{AA}), \langle a, b \rangle])$ will be used to represent the answer to the new Conditional May Alias question. Formally:

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

[39] Actually, when you consider the work necessary at return nodes the complexity is $\mathcal{O}(n * v^6)$

We have also introduced a minor amount of imprecision in the algorithm in order to increase efficiency and understanding[40]. Consider the statement, "$p = q$". We assume that $\langle *p, *q \rangle$ holds on any path through "$p = q$" but this is only valid if q is not $NULL$[41]. For example, if "$q = NULL$" was the only immediate predecessor of "$p = q$" then $\langle *p, *q \rangle$ would not hold on any path to "$p = q$". We consider the assumption that q will be non-$NULL$ on some path to "$p = q$" reasonable in practice, and thus use it in our approximate algorithm. In general, if the existence of an alias depends on the fact that some object name is non-$NULL$ on some path, we assume, without verification, that it is.

5.1.2 Representation

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

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

One solution to this problem is to use a constant time initialization array. We do not do this because, even for single level pointers, this would require at least $\mathcal{O}(n * v^4)$ space.[42]

The solution we choose is to do dynamic hashing. We use the dynamic hashing scheme presented in [KS86] (except we did not implement table shrinking as, for our purposes, this is never needed) giving us constant time operations in the average case. We implement the needed operations as follows:

- Set *may-hold*($[(node, \mathcal{AA}), \mathcal{PA}]$) to $false$ for all possible *node*,$\mathcal{AA}$, and $\mathcal{PA}$.
  Simply initialize the hash table, which takes constant time.

---

[40] This "optimization" can be removed from the algorithm without great difficulty.

[41] This C constant NULL.

[42] n is the number of nodes in the ICFG and v is the number of variables in the program.

- Find the value of *may-hold*($[(node, \mathcal{AA}), \mathcal{PA}]$) for a given *node*,$\mathcal{AA}$, and $\mathcal{PA}$.
  Look up (*node*,$\mathcal{AA}$,$\mathcal{PA}$) in the hash table; if it is there, then return the value associated with it; otherwise return $false$.

- Set the value of *may-hold*($[(node, \mathcal{AA}), \mathcal{PA}]$) for a given *node*,$\mathcal{AA}$, and $\mathcal{PA}$.
  If (*node*,$\mathcal{AA}$,$\mathcal{PA}$) is in the hash table, then simply change the value associated with it; otherwise add it.

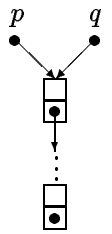
Note that *may-hold*($[(node, \emptyset), \langle a, b\rangle]$) implies *may-hold*($[(node, assumed_alias), \langle a, b\rangle]$) for every *assumed_alias*. Thus if we want to know the value of *may-hold*($[(node, assumed_alias), \langle a, b\rangle]$) (and $assumed_alias \neq \emptyset$) we first look up (*node*, $\emptyset$, $\langle a, b\rangle$). If its associated value is there return *true*, otherwise we return the value associated with (*node*,*assumed_alias*,$\langle a, b\rangle$), unless that's not in the table, in which case we return $false$.

5.1.3 Alias Consequences

Consider an alias between $*p$ and $*q$ where p and q are declared by

struct list {int data; struct list *next} $*p$,$*q$;

This alias yields the following situation:



Thus the alias between $*p$ and $*q$ implies the existence of other aliases; for example, between p->$next$ and q->$next$ and between p->$next$->$next$ and q->$next$->$next$. The existence of this last alias is based on the assumption that p->$next$ and q->$next$ are not $NULL$. The *consequences* of an alias $\langle *p, *q\rangle$ is defined as the set of all other aliases which are implied by $\langle *p, *q\rangle$. Alias consequences can be computed by a simple recursive procedure (see Figure 5.1). This procedure will terminate on aliases with recursive types because of k-limiting.

```
SET alias_consequences(⟨a,b⟩)          /* a and b must be the same type */
{
  answer = {⟨a,b⟩}
  if (can_deref(alias_type(⟨a,b⟩)))
    answer = answer ∪ alias_consequences(⟨deref(a),deref(b)⟩)
  if (is_struct(alias_type(⟨a,b⟩)))
    for each field, field, of alias_type(⟨a,b⟩)
      answer = answer ∪
        alias_consequences(⟨field_access(a,field),field_access(b,field)⟩)
}
```

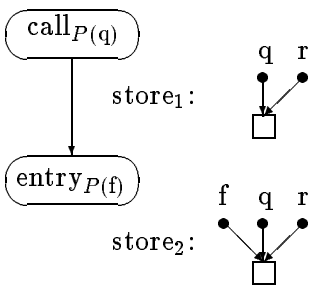
Figure 5.1: Computing the Consequences of an Alias

Assumption 5.1.1 (NULL) *When q is the RHS of an assignment or the actual of a procedure call, we assume for all a such that is_prefix(q,a) and a has less than k dereferences that a is not NULL on some path through some immediate predecessor.*

In the following sections, we present a detailed description of the main ideas in our interprocedural algorithm for computing *may-hold*, including pseudo-code descriptions. First, we present mechanisms for modeling the affects of parameter bindings on aliases. Second, we present an algorithm for computing the *may-hold* relation.

5.1.4 Modeling Parameter Bindings

In order to do interprocedural analysis, we will have to be able to model the affects of parameter bindings on aliases. We will do this with a function **bind(***call,assumed-alias***)**. Intuitively, bind(*call*,$\emptyset$) will be all the aliases on entry of a called procedure that must exist because of parameter bindings, while bind(*call*,$\langle a,b\rangle$) will be the set of aliases at entry of a called procedure whose existence is implied by a being aliased to b at *call*. Consider the following example (where q and r are global to P and q,r, and f are all type "int $*$"):



$\text{bind}(\text{call}_{P(q)},\emptyset) = \{\langle *q, *f\rangle\}$

$\text{bind}(\text{call}_{P(q)},\langle *q, *r\rangle) = \{\langle *q, *r\rangle, \langle *f, *r\rangle\}$

Unfortunately, this definition is not sufficient because sometimes we have to deal with object names that are not visible (Chapter 2.5). A procedure call can both create and destroy an alias in the *calling* procedure, involving an object name not visible in the called procedure. For example, the call `P()` in Figure 5.2 creates the alias $\langle *b, x\rangle$ and destroys the alias $\langle *a, x\rangle$ at $\texttt{return}_{\texttt{P()}}$ where x is not visible in `P`. However, only references to the visible object name in an alias pair can affect whether the alias holds on a path (i.e., there can be no direct references to an object name which is not visible). Fortunately, a procedure has the same effect on all alias pairs which contain visible object name w and any non-visible object name. To be able to correctly account for non-visible object names, we will need one object name for each type to represent all non-visible object names of that type; we call it *non_visible(type)*[43]. *non_visible(type)* is considered an ordinary object name, that is, it can be dereferenced or field accessed if the typing rules permit it. Although in an implementation explicit type information is necessary, in this paper we just use *non_visible* and leave the the type implicit.

This leads to a change in the **bind** function. For reasons that will become apparent later, if any of the aliases in the bind set involve *non_visible* we also want to know which object name in the calling procedure corresponds to *non_visible*. Consider the following example (where q is global to P but r is not and q,r, and f are all type "int *"):

[43] In [LR91] we referred to this by "·".

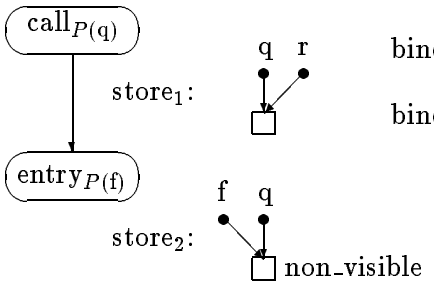
```
int *a,*b;
int y;

P()
{
    b=a
    a=&y;
}
main () {
    int x;

    a=&x;
    P();
}
```

Note: The call to P in *main* creates the alias pair $\langle *b, x \rangle$ and destroys the alias pair $\langle *a, x \rangle$.

Figure 5.2: Calls affecting alias pairs involving non-visible object names



$$\text{bind}(\text{call}_{P(\text{q})},\emptyset) = \{\langle *\text{q}, *\text{f}\rangle\}$$

$$\text{bind}(\text{call}_{P(\text{q})},\langle *\text{q}, *\text{r}\rangle) = \left\{ \begin{array}{l} (\langle *\text{q}, \text{non_visible}\rangle, *\text{r}), \\ (\langle *\text{f}, \text{non_visible}\rangle, *\text{r}) \end{array} \right\}$$

The occurrence of $(\langle *\text{q}, \text{non_visible}\rangle, *\text{r})$ in $\text{bind}(\text{call}_{P(\text{q})},\langle *\text{q}, *\text{r}\rangle)$ represents the fact that $*$q is aliased to some non-visible object name at the entry of the called procedure P, and that, in this case, the non-visible object name is $*$r.

Computing bind(*call*,∅)

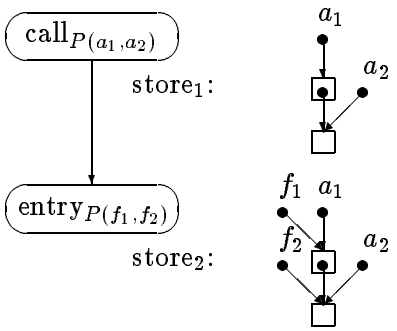
Thus, there are two ways to aliases can be implied by parameter bindings. The first alias corresponds to a simple formal to actual pairing. For example, if P is a function with formal "f" of type "int $*$" and *call* is an invocation of P with actual "a" then $\langle *f, *a\rangle$[44] is in bind(*call*,∅). The second occurs if two distinct formals are passed two

[44] $(\langle *f, \text{non_visible}\rangle, *a)$ if "a" is not visible in P.

actuals where one actual is a prefix of the other. For example, if P is a function with two formals "f_1" (type "int **") and "f_2" (type "int *") and *call* is P(a,*a) then $\langle **f_1, *f_2 \rangle$ is in bind(*call*,$\emptyset$). The algorithm for computing bind(*call*,$\emptyset$) is straightforward and is in Figure 5.3.

Computing bind(*call*,$\langle x, y \rangle$)

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



1^{st} way: $\langle **a_1, *a_2 \rangle$ at call $\Rightarrow$ $\langle **a_1, *a_2 \rangle$ at entry

2^{nd} way: $\langle **a_1, *a_2 \rangle$ at call $\Rightarrow$ $\langle *f_2, **a_1 \rangle$ at entry

3^{rd} way: $\langle **a_1, *a_2 \rangle$ at call $\Rightarrow$ $\langle **f_1, *f_2 \rangle$ at entry

In this example, since $*a_2$ is aliased to $**a_1$ at call$_{P(a_1,a_2)}$, $*f_2$ is aliased to $**a_1$. This example can be generalized to the second way an alias at a call site can imply an alias at the entry of a procedure. Whenever an actual has an alias to an object name, its corresponding formal picks up an alias to that object name or a non_visible, if the object name is not visible in the called procedure. Also in the example, since $*a_2$ is aliased to $**a_1$ at call$_{P(a_1,a_2)}$, $*f_2$ is aliased to $**f_1$ at entry$_{P(f_1,f_2)}$. This is typical of the third case; when two actuals are aliased (not necessarily directly) at a call site, the corresponding formals are aliased on entry to the called procedure. The algorithm for computing bind(*call*,$\langle x, y \rangle$) is a straightforward encoding of these three cases. It can be found in Figure 5.5.

```
implied_by_binding(call, f, a)
{ if a is visible in the called procedure
    return alias_consequences(⟨deref(f), deref(a)⟩)
  else return
    {(alias, a) | alias ∈ alias_consequences(⟨deref(f), deref(non_visible)⟩)}
}
```

```
bind(call, ∅)
{ bind = ∅
  /* Alias effects of each parameter binding */
  for each formal (f) actual (a) pair:
  { if (can_deref(object_type(f)))
      bind = bind ∪ implied_by_binding(f, a)
    if (is_struct(object_type(f)))
      for each field (field) of object_type(f) which can be dereferenced:
        bind = bind ∪
          implied_by_binding(field_access(f, field), field_access(a, field))
  }
  /* Alias effects of pairs of parameter bindings */
  for each formal (f_i) actual (a_i) pair:
    for each formal (f_j) actual (a_j) pair (j ≠ i):
      if is_prefix(a_i, a_j)
      { let† form_i = f_i
        apply_trans(a_i, a_j, form_i)
        if (can_deref(object_type(f_j)))
          bind = bind ∪ alias_consequences(⟨deref(f_j), deref(form_i)⟩)
        if (is_struct(object_type(f_j)))
          for each field (field) of object_type(f_j) which can be derefed:
            bind = bind ∪
                 alias_consequences(⟨ deref(field_access(f_j, field)),
                                      deref(field_access(form_i, field)) ⟩)
      }
  return bind
}
```

† ''let a = b'' puts a copy of b in a.

Figure 5.3: Computing bind($call$,$\emptyset$)

```
same_object(call, name1, name2)
/* Implications of name1 and name2 being the same object */
{ if (name1 and name2 is visible in the called procedure)
    return alias_consequences(⟨name1, name2⟩)
  if ( name1 or name2 (assume name1) are not visible
       in the called procedure but the other is )
    return {(alias, name1) | alias ∈ alias_consequences(⟨name2, non_visible⟩)}
}
```

```
same_value(call, name1, name2)
/* Implications of name1 and name2 referring to the same object */
  if (can_deref(alias_type(⟨name1, name2⟩)))
    return same_object(call, deref(name1), deref(name2))
  if is_struct(alias_type(⟨name1, name2⟩))
  { val = ∅
    for each field (field) of type alias_type(⟨name1, name2⟩) which
    can be dereferenced
      val = val ∪ same_object( call, deref(field_access(name1, field)),
                               deref(field_access(name2, field)) )
    return val
  }
}
```

Figure 5.4: Support functions for computing bind($call$,$\langle x, y \rangle$)

```
/* Uses functions defined in Figure 5.4 */
bind(call,⟨x,y⟩)
{ bind = ∅
  if both x and y are visible in the called procedure
    bind = bind ∪ {⟨x,y⟩}
  if x or y (assume x) is visible in the called
  procedure but the other is not
    bind = bind ∪ {(⟨x,non_visible⟩,y)}

  for each formal (f_i) actual (a_i) pair:
    if can_deref(object_type(f_i)) or
        is_struct(object_type(f_i))
      if for x or y (assume x) is_prefix(a_i,x)
      { let† form_i = f_i
        if (apply_trans(a_i,x,form_i))‡
          bind = bind ∪ same_object(call,form_i,y)
        else bind = bind ∪ same_value(call,form_i,y)
      }

  for each formal (f_i) actual (a_i) pair:
    for each formal (f_j) actual (a_j) pair:
      if is_prefix(a_i,x) and is_prefix(a_j,y)
      { let form_i = f_i; let form_j = f_j
        if (apply_trans(a_i,x,form_i) and
            apply_trans(a_j,y,form_j))††
          bind = bind ∪ same_object(call,form_i,form_j)
        else bind = bind ∪ same_value(call,form_i,form_j)
      }
}
```

1st way in Chapter 5.1.4

2nd way in Chapter 5.1.4

3rd way in Chapter 5.1.4

† "let $a = b$" puts a copy of b in a.

‡ When at least one dereference occurs $form_i$ and y must be aliases, otherwise they just contain the same value.

†† Both $form_i$ and $form_j$ must be dereferenced at least once for them to be aliases. If either $form_i$ or $form_j$ was not dereferenced, they contain the same value.

Figure 5.5: Computing bind($call$,$\langle x, y\rangle$)

```
find_aliases()
{ worklist = ∅
  /* Alias Introduction */
  for each node (node) in the ICFG
    if node is an assignment to a pointer
      aliases_introduced_by_assignment(node)
    if node is a call node
      aliases_introduced_by_call(node)

  /* Implied Aliases */
  while worklist is not empty
  { remove node,assumed_alias,possible_alias from worklist
    if node is a call node
      aliases_at_call_implies(node,assumed_alias,possible_alias)
    else if node is an exit node
      alias_at_exit_implies(node,assumed_alias,possible_alias)
    else any_other_alias_implies(node,assumed_alias,possible_alias)
  }
}
```

Figure 5.6: Computing may-hold

5.2 Computing may-hold

The algorithm for computing *may-hold* is, at a high level, simple. First, we find all the *may-hold* relations which are trivially true, for example, *may-hold*$([(\text{“}p = q\text{”}, \emptyset), \langle *p, *q \rangle])$ is true.[45] Once we have this initial set, we compute the set of all true *may-hold*s using a worklist algorithm. The algorithm at this level of abstraction is presented in Figure 5.6.

The remainder of this section gives detailed descriptions of this algorithm. In Chapter 5.2.2 to Chapter 5.2.4 we address the interprocedural aspects of *may-hold* which primarily entail insuring that only aliases on realizable paths are considered. In Chapter 5.2.1 and Chapter 5.2.5 we cover the intraprocedural aspects of *may-hold*.

[45] The fact that *may-hold*$([(\text{“}p = q\text{”}, \emptyset), \langle *p, *q \rangle])$ is true is based on an implicit assumption, that q is not NULL on all paths to this node. We could reformulate the algorithm so that this assumption was not necessary, but have not since it would make the algorithm more inefficient. The current formulation is safe and reasonable.

```
Aliases_introduced_by_assignment(node)
/* Let node be the pointer assignment ``p = q'' */
{ if !is_prefix(p,q)
    for each ⟨u,v⟩ in alias_consequences(⟨deref(p),deref(q)⟩)
    { set may-hold([(node,∅),⟨u,v⟩]) to true
      add (node,∅,⟨u,v⟩) to the worklist
    }
}
```

Figure 5.7: Aliases_introduced_by_assignment(*node*)

```
Aliases_introduced_by_call(node)
{ let entry be the entry of the called procedure
  for each ⟨u,v⟩ in bind(node,∅)
  { set may-hold([(entry,⟨u,v⟩),⟨u,v⟩]) to true
    add (entry,⟨u,v⟩,⟨u,v⟩) to the worklist
  }
}
```

Figure 5.8: Aliases_introduced_by_call(*node*)

5.2.1 Aliases_introduced_by_assignment(*node*)

This routine is very simple. Let *node* be the pointer assignment "$p = q$". Clearly, *may-hold*$([(node, \emptyset), \langle \text{deref}(p), \text{deref}(q)\rangle])$ and all *may-hold*$([(node, \emptyset), \langle u, v\rangle])$ such that $\langle u, v\rangle$ in alias_consequences$\langle \text{deref}(p), \text{deref}(q)\rangle$ are true unless p is a prefix of q. For example, the assignment "$p = p{->}next$" does not create an alias $\langle *p, *(p{->}next)\rangle$; both p and $p{->}next$ refer to different objects after this assignment but their alias relationship does not change. Figure 5.7 has the code for this function.

5.2.2 Aliases_introduced_by_call(*node*)

Given an implementation of the bind function (Chapter 5.1.4) this routine is also very simple (see Figure 5.8).

5.2.3 Alias_at_call_implies(*call*,*assumed_alias*,*possible_alias*)

A call node effectively has two successor nodes. The first is the entry node of the procedure it invokes which is explicitly represented by an edge in the ICFG. The second is the return node which corresponds to the same call site which, although not explicitly represented in the ICFG by an edge, must be known in order for the aliasing algorithm to function correctly (i.e., so that it only considers realizable paths in the ICFG). *may-hold*($[(call, assumed_alias), possible_alias]$) has effects on both its corresponding entry and return nodes.

Effects on corresponding entry node (*entry*) While *holds* as defined in Chapter 4.2.3 had the nice property that the *holds* relations which were true at a call node did not affect the *hold* relations at the entry of the called procedure, this is not true for *may-hold* since it requires the existence of a path to the entry node with certain characteristics whereas *holds* simply assumed the existence of such a path. The effects are very simple, for each alias $\langle a, b\rangle$ in bind(*call*, *possible_alias*), if *may-hold*($[(entry, \langle a, b\rangle), \langle a, b\rangle]$) is *false*[46], we set *may-hold*($[(entry, \langle a, b\rangle), \langle a, b\rangle]$) to *true* and add (*entry*, $\langle a, b\rangle$, $\langle a, b\rangle$) to the worklist.

Effects on corresponding return node (*return*) Before we can go into the details of how to handle the effects on *return*, we have to introduce the functions *back-bind* and *back-bind'* which have the following definitions:

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

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

These definitions imply that *back-bind*$_{call}(\langle a, b\rangle) = \langle c, d\rangle$ iff $\langle a, b\rangle \in$ *bind*$_{call}(\langle c, d\rangle)$ and *back-bind'*$_{call}(\langle a, non_visible\rangle, o) = \langle c, d\rangle$ iff $\langle a, non_visible\rangle \in$ *bind*$_{call}(\langle c, d\rangle)$ where the non_visible is the non-visible object name o.

[46] In other words, (*entry*, $\langle a, b\rangle$, $\langle a, b\rangle$) has not yet been added to the worklist.

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

$$\mathit{holds}([(\mathit{return}, \mathit{assumed_alias}), \langle x, y\rangle]) = \mathit{holds}([(\mathit{call}, \mathit{assumed_alias}), \langle x, y\rangle])$$

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

$$\begin{aligned}&\mathit{holds}([(\mathit{return}, \mathit{assumed_alias}), \langle x, y\rangle]) = \\ &\qquad \mathit{holds}([(\mathit{exit}, \emptyset), \langle x, y\rangle]) \vee \\ &\qquad \bigvee_{AA \in \mathcal{ASSUMED}} \begin{pmatrix} \mathit{holds}([(\mathit{exit}, AA), \langle x, y\rangle]) \wedge \\ \mathit{holds}([(\mathit{call}, \mathit{assumed_alias}), \mathit{back\text{-}bind}_{\mathit{call}}(AA)]) \end{pmatrix}\end{aligned}$$

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

$$\begin{aligned}&\mathit{holds}([(\mathit{return}, \mathit{assumed_alias}), \langle x, y\rangle]) = \\ &\qquad \bigvee_{\langle o, \mathit{non_visible}\rangle \in \mathcal{ASSUMED}} \begin{pmatrix} \mathit{holds}([(\mathit{exit}, \langle o, \mathit{non_visible}\rangle), \langle x, \mathit{non_visible}\rangle]) \wedge \\ \mathit{holds}\left(\begin{bmatrix} (\mathit{call}, \mathit{assumed_alias}), \\ \mathit{back\text{-}bind}'_{\mathit{call}}(\langle o, \mathit{non_visible}\rangle, y) \end{bmatrix}\right) \end{pmatrix}\end{aligned}$$

Figure 5.9: *holds* relation at return nodes

Figure 5.9 (which are the rules for a return node found in Chapter 4.2.3) contains the equations for *holds* at a return node (for an explanation of why those rules are valid see Chapter 4.2.3). *may-hold* is a simple encoding of these rules with a few minor modifications. The *holds* rules are still valid for *may-hold* because *may-hold* is basically *holds* with the added restriction that the *assumed alias* must hold on some path the the entry node of the procedure of interest. Since, in the equation for holds, each *holds* at *exit* with assumption not $\emptyset$ is paired as follows:

$$\mathit{holds}([(\mathit{exit}, \mathcal{AA}), \langle a, b\rangle]) \wedge \mathit{holds}([(\mathit{call}, \mathit{assumed_alias}), \mathit{back\text{-}bind}_{\mathit{call}}(\mathcal{AA})])$$

The only way that $\mathit{holds}([(\mathit{exit}, \mathcal{AA}), \langle a, b\rangle])$ can contribute to the value at a return node is if $\mathit{holds}([(\mathit{call}, \mathit{assumed_alias}), \mathit{back\text{-}bind}_{\mathit{call}}(\mathcal{AA})])$ is true and a simple inductive argument (on path length) will suffice to show that this means there must be a path to the entry of the procedure of *exit* on which the alias $\mathcal{AA}$ holds. Thus this is also a valid definition for *may-hold*.

We will consider the implications of *may-hold*$([(call, assumed_alias), possible_alias])$ on *may-hold* at *return*. Let *possible_alias* $= \langle a, b\rangle$, we need to do a case analysis:

1. If a and b are both not visible in the called procedure

   This corresponds to *Rule 1* in Figure 5.9, and thus the desired action is obviously: if *may-hold*$([(return, assumed_alias), \langle a, b\rangle])$ is not *true* then set it to *true* and add (*return*, *assumed_alias*, $\langle a, b\rangle$) to the worklist.

2. a and b are both visible in the called procedure

   This corresponds to *Rule 2* in Figure 5.9 which is[47]:

may-hold$([(return, assumed_alias), \langle x, y\rangle]) =$ *may-hold*$([(exit, \emptyset), \langle x, y\rangle]) \vee$

$\bigvee_{AA}$ (*may-hold*$([(exit, AA), \langle x, y\rangle])$ $\wedge$ *may-hold*$([(call, assumed_alias), back\text{-}bind_{call}(AA)])$)

We know *may-hold*$([(call, assumed_alias), \langle a, b\rangle])$.

The $\longrightarrow$ above denotes implication, that is *call* $\longrightarrow$ *exit* means that given *call* we can determine *exit*. Since we know *may-hold*$([(call, assumed_alias), back\text{-}bind_{call}(AA)])$ specifies *Rule 2* with only one free variable $\langle x, y\rangle$, so the obvious action for *may-hold*$([(call, assumed_alias), \langle a, b\rangle])$ would be:

```
For each AA in bind_call(⟨a,b⟩)
  for every possible ⟨x,y⟩:
    if may-hold([(exit,AA),⟨x,y⟩]) is true
      if may-hold([(return,assumed_alias),⟨x,y⟩]) is false then
      { set may-hold([(return,assumed_alias),⟨x,y⟩]) to true
        add (return,assumed_alias,⟨x,y⟩) to the worklist.
      }
```

This, however, is not acceptable because it requires work to be done for every possible $\langle x, y\rangle$ even though most $\langle x, y\rangle$ are not necessary. Since we are performing

[47] Strictly speaking this means if *may-hold*$([(exit, \emptyset), \langle x, y\rangle])$ was *true*, we should make *may-hold*$([(return, assumed_alias), \langle x, y\rangle])$ *true* for all possible *assumed_alias*. However, this is not necessary and it is sufficient to only make *may-hold*$([(return, \emptyset), \langle x, y\rangle])$ *true*.

a conjunction in which we know one half is true, instead of doing work for all $\langle x, y\rangle$, we would prefer to only do work for $\langle x, y\rangle$ such that *may-hold*($[(exit, \mathcal{AA}), \langle x, y\rangle]$). We will do this by keeping a set *true_under_assumption* (TUA) for each (exit node,assumed alias) pair:

true_under_assumption($exit$,$\mathcal{AA}$) =

$$\{\langle x, y\rangle \mid \textit{may-hold}([(exit, \mathcal{AA}), \langle x, y\rangle]) \text{ is } true\}$$

Given TUA, we can easily only consider the interesting $\langle x, y\rangle$. By using the trick we used for *may-hold* (Chapter 5.1.2) we can initialize all TUA to $\emptyset$ in constant time. Whenever we set *may-hold*($[(exit', \mathcal{AA}'), \langle x', y'\rangle]$) to *true* we also add $\langle x', y'\rangle$ to TUA($exit', \mathcal{AA}'$). This will work iff *may-hold*($[(exit, \mathcal{AA}), \langle x, y\rangle]$) becomes *true* before we need it when processing *may-hold*($[(call, assumed_alias), \langle a, b\rangle]$))[48]. However, if they become true in the reverse order, this will not work. Fortunately, the other case (i.e., when we process the exit node after the corresponding call node) will be captured in alias_at_exit_implies (Chapter 5.2.4).

3. a or b (assume a) is not visible in the called procedure (but b is)

   This corresponds to *Rule 3* in Figure 5.9 and should be analogous to the case where a and b are both visible in the called procedure, except now we need to fill in the *non_visible* at $exit$ with a. Thus we would get the following action for *may-hold*($[(call, assumed_alias), \langle a, b\rangle]$):

   ```
   For each (AA, nv) in bind_call(⟨a,b⟩), for every ⟨x,y⟩ on TUA(exit, AA)
   (assume x contains non_visible[49]):
   {  let x' = nv; apply_trans(non_visible,x,x')
      if may-hold([(return, assumed_alias), ⟨x',y⟩]) is false then
      {  set may-hold([(return, assumed_alias), ⟨x',y⟩]) to true
         add (return, assumed_alias, ⟨x',y⟩) to the worklist.
      }
   ```

[48] This may be the case when, for another call site of the same procedure, the assumption $\mathcal{AA}$ was produced at the entry of the called procedure at an earlier iteration of the alias algorithm.

}

More Complex Effects on Return Nodes[50]: While this is sufficient if only single level pointers are allowed, it is not sufficient in the general case. In general, it is possible to have an alias between two non_visible's (see Figure 5.10). Thus we must handle the case of creation in the called procedure of an alias between two non-visible object names. We will do this with a special case of *may-hold* with two assumed aliases:

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

This represents the fact that if o_1 is aliased to non-visible object name l_1 and o_2 is aliased to non-visible object name l_2 on a path to the entry of the procedure then, on some path to *exit*, nv_1 (where the non_visible portion represents l_1) is aliased to nv_2 (where the non_visible portion represents l_2). In Figure 5.11, we show how *may-hold* can be used to represent the stores of Figure 5.10. At this point the reader is not expected to be able to understand how these *may-hold*s are derived (they are indeed obtainable from our algorithm), but the figure is simply to show how *may-hold* can represent stores with aliases between non-visible object names. With this encoding scheme we can derive $\textit{may-hold}([(n_8, \emptyset), \langle *l_1, l_2\rangle])$ in Figure 5.11 because the following are *true*:

- $\textit{may-hold}([(n_7, \begin{matrix}\langle *g_1, non_visible\rangle \\ \langle *g_2, non_visible\rangle\end{matrix}), \langle *non_visible, non_visible\rangle])$
- $\textit{may-hold}([(n_4, \emptyset), \langle *g_1, l_1\rangle])$ and $\langle *g_1, l_1\rangle = \textit{back-bind}'_{n_4}(\langle *g_1, non_visible\rangle, l_1)$
- $\textit{may-hold}([(n_4, \emptyset), \langle *g_2, l_2\rangle])$ and $\langle *g_2, l_2\rangle = \textit{back-bind}'_{n_4}(\langle *g_2, non_visible\rangle, l_2)$

Fortunately, we only need this type of *may-hold* at exit nodes[51]. In order to handle this, we change TUA slightly. For $\textit{may-hold}([(exit, \begin{matrix}\langle o_1, non_visible\rangle \\ \langle o_2, non_visible\rangle\end{matrix}), \langle nv_1, nv_2\rangle])$

[49] For example, $x = *$non_visible.

[50] We are still considering the case where a is not visible in the called procedure but b is.

[51] However, in Figure 5.11, we included an alias of this type at n_6, which is not an exit node, so that the reader could more easily follow the example.

```
int **g1, *g2;
P()
{
  *g1 = g2;
}
main ()
{
  int *l1, l2;
  g1 = &l1;
  g2 = &l2;
  P();
}
```

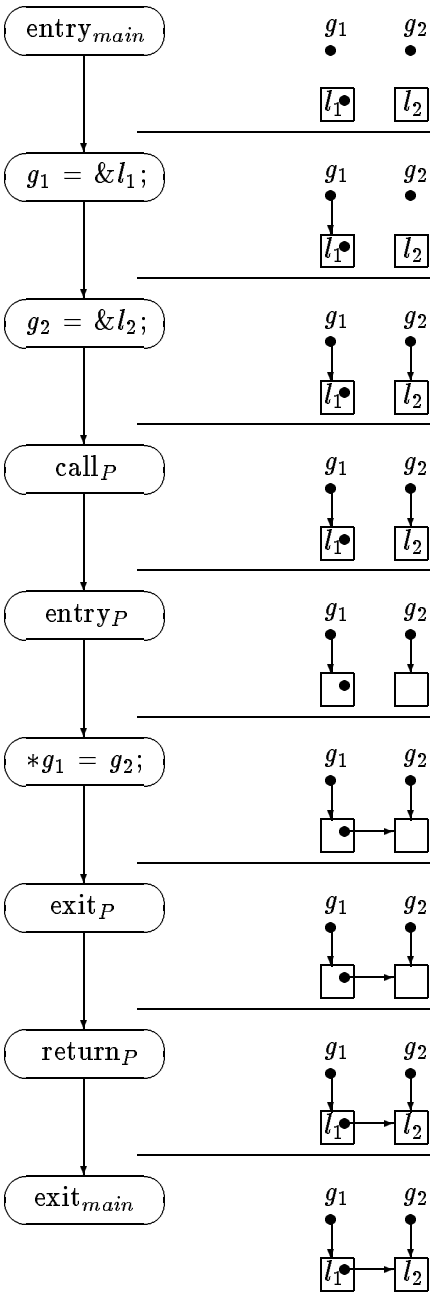
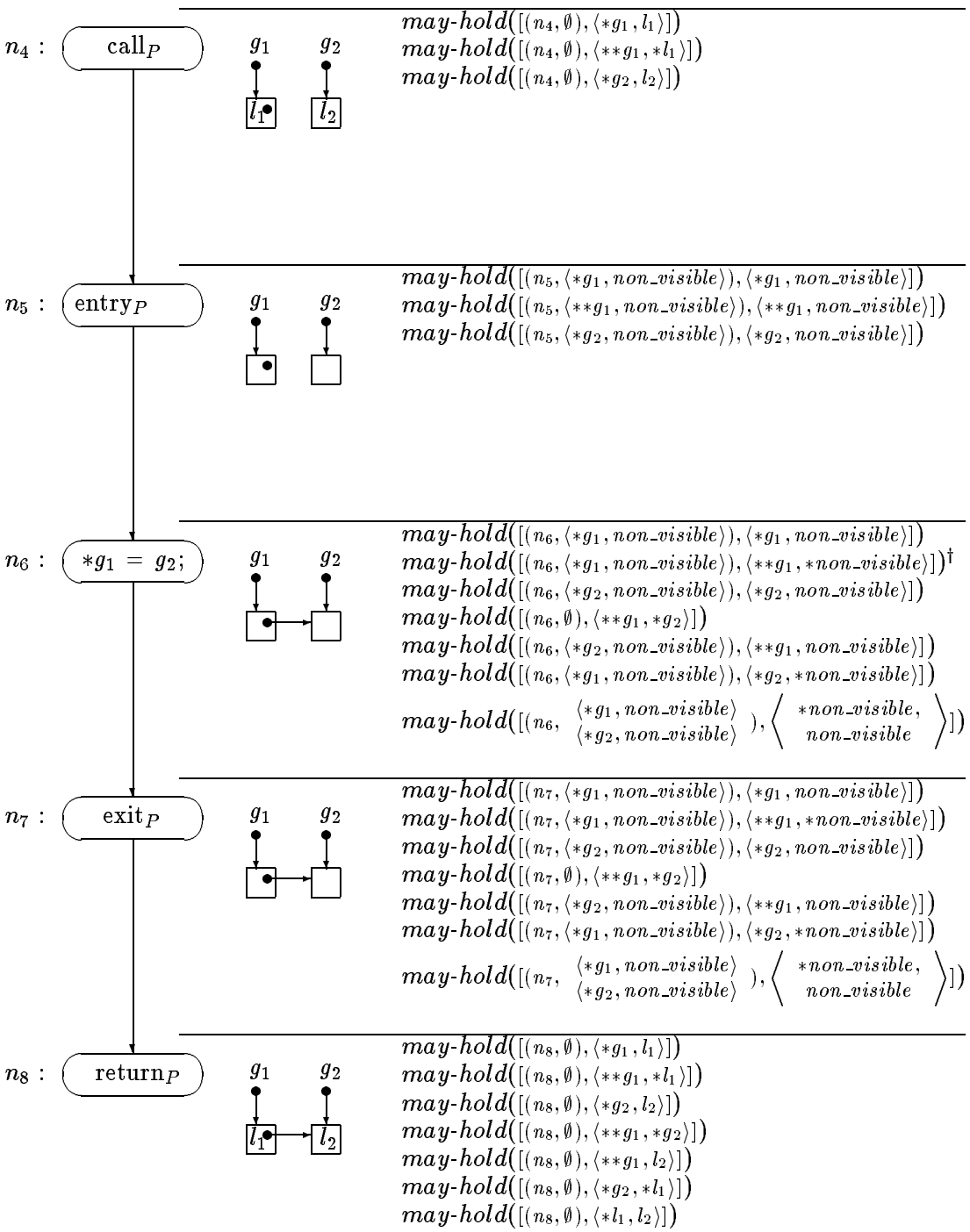


Figure 5.10: Creation of alias between two non-visible object names



† This is an implementation quirk.
$may\text{-}hold([(n_6, \langle **g_1, non_visible \rangle), \langle **g_1, non_visible \rangle])$ is not *true* because $*g_1$ is redefined at n_6. $may\text{-}hold([(n_6, \langle *g_1, non_visible \rangle), \langle **g_1, *non_visible \rangle])$ is *true* by Figure 5.14 because $\langle **g_1, *non_visible \rangle$ is in alias_consequences($\langle *g_1, non_visible \rangle$).

Figure 5.11: *may-hold* representation for interesting stores of Figure 5.10

we would add ($\langle nv_1, nv_2\rangle$,$o_2$), $\langle o_2, non_visible\rangle$ being the *other condition* necessary for $\langle nv_1, nv_2\rangle$, to TUA($exit$,$\langle o_1$, non_visible$\rangle$) and ($\langle nv_1, nv_2\rangle$,$o_1$) to TUA($exit$,$\langle o_2$, non_visible$\rangle$). To handle *may-hold*($[(call, assumed_alias), \langle a, b\rangle]$) correctly we need the following function:

approximate-when-both-non_visible (call-node, $\langle nv_1,\ nv_2\rangle$,
non_visible-name1-is, non_visible-name2-is, name1-AA, name2-AA) substitutes non_visible-name1-is for non_visible in nv_1, substitutes non_visible-name2-is for non_visible in nv_2, and uses the assumed aliases name1-AA and name2-AA to establish a safe assumed alias condition at procedure entry.

This function is specified in Appendix D.1, but is not discussed here because its implementation is not theoretically interesting. Finally, the action for *may-hold*($[(call, assumed_alias), \langle a, b\rangle]$) is:

```
For each (𝒜𝒜, nv) in bind_call(⟨a, b⟩)
for every ⟨x, y⟩ or (⟨x, y⟩,other_condition) on TUA(exit, 𝒜𝒜)
  if both x and y contain non_visible
    for each (𝒜𝒜, nv2) such that may-hold([(call,𝒜𝒜), alias]) and
    (⟨non_visible,other_condition⟩, nv2) in bind_call(alias)
      approximate-when-both-non_visible(call,⟨x, y⟩,nv,nv2,
        assumed_alias,𝒜𝒜)
  else (assume x contains non_visible):
    let x' = nv; apply_trans(non_visible,x,x')
    if may-hold([(return, assumed_alias), ⟨x', y⟩]) is false then
      set may-hold([(return, assumed_alias), ⟨x', y⟩]) to true
      add (return,assumed_alias,⟨x', y⟩) to the worklist.
```

In order to make this action efficient, we need to be able to find all:

```
(𝒜𝒜, nv2) such that for some alias, may-hold([(call,𝒜𝒜), alias])
and (⟨non_visible,other_condition⟩, nv2) in bind_call(alias)
```

in time linear to the number of such $(\mathcal{AA}, nv_2)$. We will do this with a data structure back_bind_true (BBT) which is defined as follows:

back_bind_true($call$,$assumed_alias$) =

$$\begin{cases} \left\{ (\mathcal{AA}, nv) \;\middle|\; \begin{array}{l} (\exists alias)\ \textit{may-hold}([(call, \mathcal{AA}), alias]) \text{ and} \\ (assumed_alias, nv) \in \text{bind}_{call}(alias) \end{array} \right\} & \text{if } \mathcal{AA} \text{ has an object name with non_visible in it} \\ \left\{ \mathcal{AA} \;\middle|\; \begin{array}{l} (\exists alias)\ \textit{may-hold}([(call, \mathcal{AA}), alias]) \\ \text{and } assumed_alias \in \text{bind}_{call}(alias) \end{array} \right\} & \text{otherwise} \end{cases}$$

In the preceding discussion we ignored the issue of scope of variables. Care must be taken to insure that aliases only appear at program points that are in the scope of the object names. This requires a simple scope check before setting values of *may-hold* to true. We will continue to omit this check in our code. Appendix D.2 contains the final algorithm for computing Alias_at_call_implies(*node*, *assumed_alias*, *possible_alias*).

5.2.4 Alias_at_exit_implies($exit$,$assumed_alias$,$possible_alias$)

An exit node can have any number of successors; however, they are all return nodes. This function simply encodes the rules for return nodes in Figure 5.9 with the additional case of aliases between two non-visible object names. The encoding is analogous to that for Alias_at_call_implies. Given that we know *may-hold*$([(exit, \mathcal{AA}), \langle x, y \rangle])$ is true, specifies the rule in Figure 5.9 leaving only one free variable, *assumed_alias* (see Figure 5.12). Since Alias_at_exit_implies presents no problems that weren't already addressed by Alias_at_call_implies, we omit an informal description of this routine. Pseudocode for it can be found in Appendix D.3.

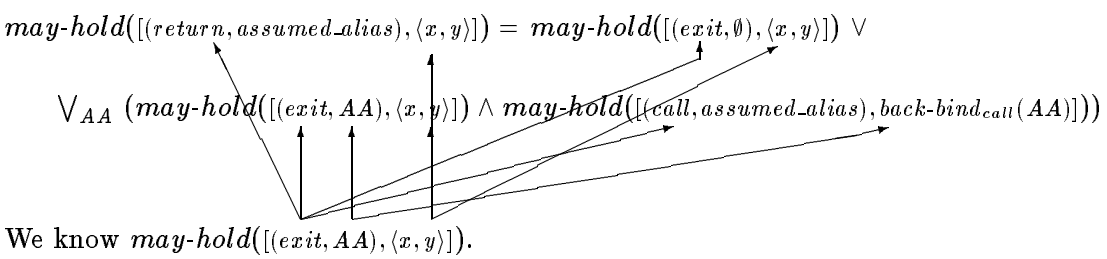


Figure 5.12: Implication of a known *may-hold* at an exit node

5.2.5 Any_other_alias_implies(*node*,*assumed_alias*,*possible_alias*)

The implications of *may-hold*([(*node*, *assumed_alias*), *possible_alias*]) depends strongly on its successors. These implications must be considered separately for each successor. Since we have already examined the cases where *node* is a call or exit node, the successors of *node* must be either a call, an exit, or a statement in the program (i.e., it is not possible for any successors to be an entry or return node).

Successor is a call node, exit node, or a program statement which is not an assignment to a pointer: These nodes simply collect *may-hold* information from their parents. An alias on a path to any of these nodes holds only if it held on the same path up to its parent. Thus the action for *may-hold*([(*node*, *assumed_alias*), *possible_alias*]) is simple in the case where its successor, *succ*, is of one of these types.

```
if may-hold([(succ, assumed_alias), possible_alias]) is false
    set may-hold([(succ, assumed_alias), possible_alias]) to true
    add (succ, assumed_alias, possible_alias) to the worklist
```

Successor (*succ*) is an assignment to a pointer: This case encompasses the major intraprocedural affects of pointers on aliasing. The effects of *may-hold*([(*node*, *assumed_alias*), *possible_alias*]) depends on the relationship of the object names in *possible_alias* and the object names involved in the pointer assignment. In the following discussion we will consider *succ* to the be statement "$p = q$", where p and q are arbitrary object names of pointer type (not necessarily simple variable names). What follows is a case analysis. The effects of *possible_alias* is the application of all

suitable cases. The cases are (let $possible_alias = \langle y, z \rangle$):

1. Does the assignment preserve the alias?

   This is true when p is a prefix of neither y nor z.

2. What are the effects of an alias of $*q$?

   This case is applicable when is_prefix_with_deref(q,y).

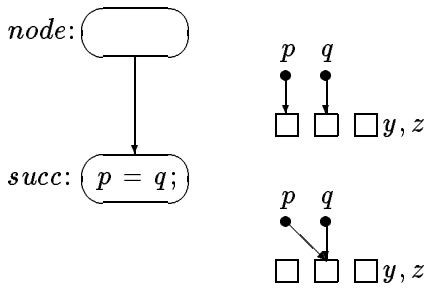
3. What are the effects of an alias of p?

   This case is applicable when $y = p$.

Below we do a detailed examination of these three cases.

1. $possible_alias = \langle y, z \rangle$ where p is a prefix of neither y nor z.

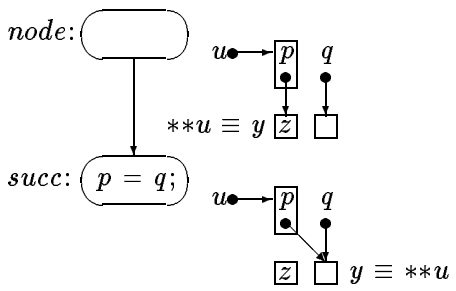
   In all cases, y and z point to the same object after the assignment as before since only p changes it's value. For example, when y and z are distinct object names from p and q.



   Clearly the assignment has no effects on $\langle y, z \rangle$, thus the action in this case is simply:

```
if may-hold([(succ, assumed_alias), possible_alias]) is false
   set may-hold([(succ, assumed_alias), possible_alias]) to true
   add (succ, assumed_alias, possible_alias) to the worklist
```

   This is clearly safe, but it could also be approximate. Consider the following situation:



Thus $\langle **u, z\rangle$ is killed by "$p = q$" because $\langle *u, p\rangle$ occurs at the same time. For safety we always assume that there is some path on which $\langle **u, z\rangle$ holds but $\langle *u, p\rangle$ does not. This is the third source of approximation from Chapter 4.

2. *possible_alias* = $\langle y, z\rangle$ where is_prefix_with_deref(q,y).

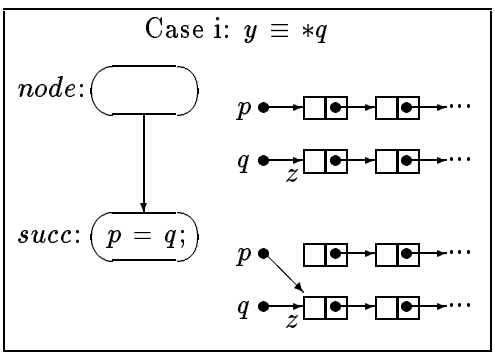
   There a 3 different cases that need to be handled (the first two are mutually exclusive, but either can occur in conjunction with the third). They are:

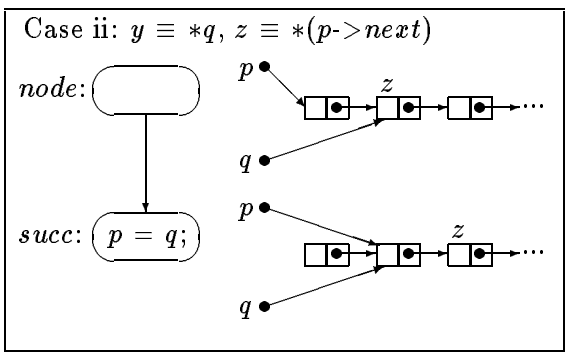
   i. *not* is_prefix(p,z)

   ii. is_prefix(p,z)

   iii. Interaction of $\langle *q, z\rangle$ with other known aliases (second type of approximation from Chapter 4).

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



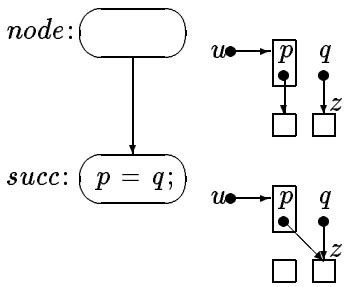


In case 2.i, *may-hold*($[(node, assumed_alias), \langle *q, z\rangle]$) implies *may-hold*($[(succ, assumed_alias), \langle *q, z\rangle]$) and *may-hold*($[(succ, assumed_alias), \langle *p, z\rangle]$). While in

case 2.ii, *may-hold*($[(node, assumed_alias), \langle *q, *(p\text{->}next)\rangle]$) gives no information about the aliasing that occurs at *succ*[52]. Thus it would seem that the action when *may-hold*($[(node, assumed_alias), \langle y, z\rangle]$) is *true* for *succ* = "$p = q$" where is_prefix_with_deref(q,y) would be:

```
if !is_prefix(p,z)
/*is_prefix(p,z) ⇒ case 2.ii; !is_prefix(p,z) ⇒ case 2.i*/
{ let p' = p
  apply_trans(q,y,p')
  if may-hold([(succ, assumed_alias), ⟨p', z⟩]) is false
  { set may-hold([(succ, assumed_alias), ⟨p', z⟩]) to true
    add (succ, assumed_alias, ⟨p', z⟩) to the worklist
  }
}
```

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



The problem is that the existence of the alias $\langle **u, z\rangle$ at *succ* does not necessarily follow from a single alias at *node*. Instead, $\langle **u, z\rangle$ can hold on $[entry_{main}]$...$[node]$ $[succ]$ if both $\langle *u, p\rangle$ and $\langle z, *q\rangle$ must hold on the path $[entry_{main}]$...$[node]$. Unfortunately, we do not keep any information about pairs of aliases holding on the same path. Thus whenever we have *may-hold*($[(node, \mathcal{AA}), \langle *u, p\rangle]$) and *may-hold*($[(node, \mathcal{AA}), \langle z, *q\rangle]$) we have to assume *may-hold*($[(\text{“}p = q\text{”}, \mathcal{AA}), \langle **u, z\rangle]$)

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

```
if !is_prefix(p,z)
/* is_prefix(p,z) ⇒ case 2.ii; !is_prefix(p,z) ⇒ case 2.i */
{ let p' = p
  apply_trans(q,y,p')
  if may-hold([(succ, assumed_alias), ⟨p', z⟩]) is false
  { set may-hold([(succ, assumed_alias), ⟨p', z⟩]) to true
    add (succ, assumed_alias, ⟨p', z⟩) to the worklist
    for each (𝒜𝒜, ⟨p, v⟩) such that may-hold([(node, 𝒜𝒜), ⟨p, v⟩]) is true
    /* Case 2.iii:  v ≡ *u and y ≡ *q in text */
      if v contains non_visible or !is_prefix(v,z)
      /*  In order to apply case 2.iii, we need ⟨y, z⟩ and ⟨p, v⟩ to hold on
          the same path.  However, if is_prefix(v,z) is true this is not
          possible.  For example, let z ≡ v->next.  On any path to node
          on which ⟨p, v⟩ holds, v will be redefined at succ and thus the
          alias ⟨y, z(≡ v->next)⟩ cannot hold at the same time.  If
          is_prefix(v,z) but v contained non_visible, we cannot safely
          assume that the non_visible in both represent the same non-
          visible object name.  */
      { let v' = v
        apply_trans(q,y,v')
        safely_make_alias(succ,v',𝒜𝒜,z,assumed_alias)
      }
  }
}
```

Figure 5.13: Action for *may-hold*([(*node*, *assumed_alias*), ⟨*y*, *z*⟩]) for successor, *succ* ≡ "$p = q$", of *node* and is_prefix_with_deref(q,y)

in order for our solution to be safe. Thus we must extend the action for *may-hold*([("$p = q$", *assumed_alias*), ⟨*y*, *z*⟩]) where is_prefix_with_deref(q,y) to account for this situation. The new action appears in Figure 5.13.

There are two notable features of Figure 5.13 which deserve explanation. First is the function **safely_make_alias** which is defined as follows:

safely_make_alias(*node*, on_1, $\mathcal{AA}_1$, on_2, $\mathcal{AA}_2$**)** When safely_make_alias is invoked we know that $\langle on_1, on_2 \rangle$ holds on a path to *node*. However, we also know that two assumptions $\mathcal{AA}_1$ and $\mathcal{AA}_2$ are necessary. We do not allow multiple assumptions, so this routine safely approximates this situation. If both assumptions contain non_visible then this will be the special case of an

alias in which both components contain non_visible (as discussed on p. 94). Otherwise both assumptions are individually necessary and either can be safely chosen. If one assumption contains non_visible, then use that one; otherwise use either (i.e., only use the assumption $\emptyset$ if both are $\emptyset$).

Second, a data structure must be maintained so that

`for each` $(\mathcal{AA}, \langle p, v\rangle)$ `such that` *may-hold*$([(node, \mathcal{AA}), \langle p, v\rangle])$ `is` *true*

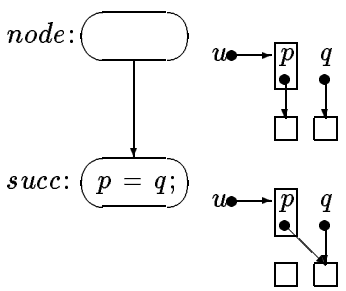
can be implemented efficiently, similarly to BBT and TUA. As with those two, this action will only work correctly if *may-hold*$([(node, \mathcal{AA}), \langle p, v\rangle])$ becomes true before
may-hold$([(node, \mathcal{AA}), \langle y, z\rangle])$ where is_prefix_with_deref(q,y), but as with BBT and TUA, another action will handle the case where they become true in the opposite order (see 3 below).

3. *possible_alias* $= \langle p, v\rangle$

Again there are three cases to consider:

i. Simple effects that must be done for ALL $\langle p, v\rangle$.
ii. Secondary effects that also must be done for ALL $\langle p, v\rangle$.
iii. Interaction of $\langle p, v\rangle$ with other known aliases (2nd type of approximation from Chapter 4).

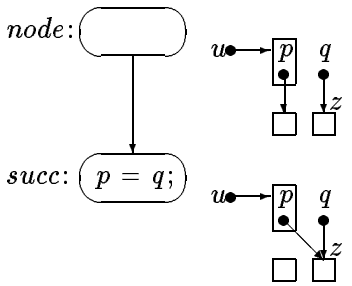
The effects of "$p = q$" on $\langle p, v\rangle$ are characterized by the following example in which v is $*u$:



Case 3.i: *may-hold*($[(node, assumed_alias), \langle p, *u\rangle]$) implies
may-hold($[(succ, assumed_alias), \langle p, *u\rangle]$) and, unless u or p is a prefix of q[53], *may-hold*($[(succ, assumed_alias), \langle *q, **u\rangle]$).

Case 3.ii: An alias $\langle *p, o\rangle$ at *node* is, in general, implicitly killed (even though it hasn't been explicitly mentioned, an examination of our actions will show that the alias $\langle *p, o\rangle$ at *node* will not have any effects at *succ*). However, in the case where $\langle p, *u\rangle$ holds on some path to *node*, $\langle *p, **u\rangle$ will not be killed by the assignment "$p = q$" and we have to account for this.

Case 3.iii: The only other effect of *may-hold*($[(node, assumed_alias), \langle p, *u\rangle]$) comes from handling the other half of case 2.iii. I.e., when
may-hold($[(node, assumed_alias), \langle *q, z\rangle]$) becomes *true* first in:



which was presented in the last case. The action for this is presented in Figure 5.14. Analogous to other cases, a data structure must be maintained to implement

> `for each` $(\langle y, z\rangle, \mathcal{AA})$ `such that` *may-hold*($[(node, \mathcal{AA}), \langle y, z\rangle]$) `and` `is_prefix_with_deref(`$q, y$`) and !is_prefix(`$p, z$`)`

efficiently.

5.3 May Alias

Computing May Alias given *may-hold* is extremely simple:

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

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

```
for each ⟨a,b⟩ in alias_consequences(⟨p,v⟩) /* Case ii */ or
  (provided !is_prefix(v,q) and !is_prefix(p,q))
  in alias_consequences(⟨*v,*q⟩) /* Case i */
{ if may-hold([(succ,assumed_alias),⟨a,b⟩]) is false
  { set may-hold([(succ,assumed_alias),⟨a,b⟩]) to true
    add (succ,assumed_alias,⟨a,b⟩) to the worklist
  }
}
for each (AA,⟨y,z⟩) such that may-hold([(node,AA),⟨y,z⟩]) and
      is_prefix_with_deref(q,y) and !is_prefix(p,z)

/*  is_prefix_with_deref(q,y) signals Case iii and we must account for it.
    However, if is_prefix(p,z) then we are saying we have to account for
    cases equivalent to, for example, v = p->next(≡ z).  In this case,
    we do not need to do anything about the relationship between
    v->next and p->next since v and p are both 'shifted' by ->next.  */

                                                  /* Rules out effects of
                                                     cases equivalent to,
  if v contains non_visible or !is_prefix(v,z)       for example,
                                                     v = v->next(≡ z) */
  { let v' = v
    apply_trans(q,y,v')
    safely_make_alias(succ,v',assumed_alias,z,AA)
  }
```

Figure 5.14: Action for *may-hold*($[(node, assumed_alias), \langle p, v\rangle]$) for successor, *succ* = "$p = q$" of *node*

This can clearly be computed in time linear in the size of the *may-hold* solution. In fact, given a clever data structure for *may-hold*, no additional computation (nor space) is needed to get the solution for *may-alias*.

Chapter 6

Theoretical and Empirical Results

Ideally we would like to be able to make theoretical claims about the speed and precision of our algorithm. However, in the worst case, our algorithm is neither very precise nor fast. Fortunately, the worst case assumptions are fairly contrived, and we have hopes for good average case precision and speed. Programs are too complex and subtle to allow us to do average case theoretical analysis. Thus, we have built a prototype implementation of our algorithm, to observe its performance.

This chapter starts with a theoretical examination of the worst case precision of our algorithm. We next discuss our implementation of the algorithm and empirically compare our solution to Weihl's solution. We then report empirical data on size of the Interprocedural May Alias and Conditional May Alias solutions, their timings, and the precision of our Interprocedural May Alias solution.

6.1 Theoretical Precision

A possible definition for the *precision* of a safe algorithm A when analyzing program P would be:

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

For algorithms (like ours) that use k-limiting and programs that have object names with more than k dereferences, this definition does not work. In our algorithm (*landi*[54]), an alias $\langle a_k, b\rangle$ with a k-limited object name a_k represents not just $\langle a_k, b\rangle$, but also all aliases $\langle a', b\rangle$ such that a_k is the k-limited representation of a'. Thus, to use the above

[54] In places we will use *landi* to refer to the algorithm presented in Chapter 5.

definition we would have to expand a k-limited alias to all the possible aliases it could represent. By this criteria, any program with recursive data structures is likely to have solution sizes (both precise and from algorithm A) which are infinite.

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

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

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

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

For programs where all object names in the precise solution have k (the k-limiting constant) or fewer dereferences, then $precision_k(A, P) = precision(A, P)$.

In the worst case, our algorithm can be very imprecise. Figure 6.1 shows how to construct a worst case example for our approximation algorithm; for any arbitrary n, Figure 6.1 is a program which has $\Theta(n)$ aliases for which our algorithm will find $\Theta(n^3)$ program point aliases[55]. We can pad the end of the program in Figure 6.1 with any number of statements which cannot affect aliasing. Each additional statement adds $|\mathcal{S}| = \Theta(n^2)$ aliases to our solution and none to the precise solution. Such a padded program (of size $m = \Omega(n)$, including additional statements) would have $\Theta(n^3)$ /* from Figure 6.1 */ + $(m - \Theta(n)) * \Theta(n^2)$ /* $\Theta(n^2)$ for each added statement */ = $\Theta(m * n^2)$ aliases reported with only $\Theta(n)$ aliases in the precise solution, thus $precision($ *landi*, Figure 6.1) is $\Theta(m * n)$. We claim that this is the worst case for our algorithm (Lemma 6.1.1) and that no other approximation algorithm can have a better worst case (Lemma 6.1.2). This worst case program is very contrived; preliminary empirical measurements of precision of our algorithm are very promising (see Chapter 6.4.2).

[55] Why this is true is explained below in the proof of Lemma 6.1.2.

	node	Precise May Alias at *node*	*may-alias*(*node*)
	`int **a,*b,*c,*d;`	$\begin{pmatrix}\text{omitting reflexive}\\ \text{aliases}\end{pmatrix}$	
	`int *v`$_i$; $(1 \le i \le n)$		
	`/* - Part I - */`		
	`a = malloc();`	$\emptyset$	$\emptyset$
	`b = malloc();`	$\emptyset$	$\emptyset$
	`c = malloc();`	$\emptyset$	$\emptyset$
	`d = malloc();`	$\emptyset$	$\emptyset$
	`if (-)`	$\emptyset$	$\emptyset$
	`  a = &b;`	$\{\langle *\mathtt{a}, \mathtt{b}\rangle\}$	$\{\langle *\mathtt{a}, \mathtt{b}\rangle, \langle **\mathtt{a}, *\mathtt{b}\rangle\}$
	`else c = d;`	$\{\langle *\mathtt{c}, *\mathtt{d}\rangle\}$	$\{\langle *\mathtt{c}, *\mathtt{d}\rangle\}$
	`*a = c;`	$\left\{\begin{array}{c}\langle *\mathtt{a}, \mathtt{b}\rangle, \langle **\mathtt{a}, *\mathtt{b}\rangle,\\ \langle *\mathtt{c}, *\mathtt{d}\rangle, \langle **\mathtt{a}, *\mathtt{c}\rangle,\\ \langle **\mathtt{a}, *\mathtt{d}\rangle, \langle *\mathtt{c}, *\mathtt{b}\rangle\end{array}\right\}$	$\left\{\begin{array}{c}\langle *\mathtt{a}, \mathtt{b}\rangle, \langle **\mathtt{a}, *\mathtt{b}\rangle,\\ \langle *\mathtt{c}, *\mathtt{d}\rangle, \langle **\mathtt{a}, *\mathtt{c}\rangle,\\ \langle **\mathtt{a}, *\mathtt{d}\rangle, \langle *\mathtt{c}, *\mathtt{b}\rangle,\\ \langle *\mathtt{b}, *\mathtt{d}\rangle\end{array}\right\}$
	`a = NULL;`	$\{\langle *\mathtt{c}, *\mathtt{d}\rangle, \langle *\mathtt{c}, *\mathtt{b}\rangle\}$	$\left\{\begin{array}{c}\langle *\mathtt{c}, *\mathtt{d}\rangle, \langle *\mathtt{c}, *\mathtt{b}\rangle,\\ \langle *\mathtt{b}, *\mathtt{d}\rangle\end{array}\right\}$
	`c = NULL;`	$\emptyset$	$\{\langle *\mathtt{b}, *\mathtt{d}\rangle\}$
	`/* - Part II - */`		
W_e:	`while (-) {`	$\emptyset$	$\mathcal{S} = \{\langle *\mathtt{b}, *\mathtt{d}\rangle\} \cup$ $\{\langle *\mathtt{v}_i, *\mathtt{b}\rangle \mid 1 \le i \le n\} \cup$ $\{\langle *\mathtt{v}_i, *\mathtt{d}\rangle \mid 1 \le i \le n\} \cup$ $\left\{\langle *\mathtt{v}_i, *\mathtt{v}_j\rangle \;\middle|\; \begin{array}{c}1 \le i,j \le n\\ \text{and } i \ne j\end{array}\right\}$
	for all k, $1 \le k \le n$:		
C_a^k:	`  if (-)`	$\emptyset$	$\mathcal{S}$
C_b^k:	`  {` `  v`$_k$ `= b;`	$\{\langle *\mathtt{v}_k, *\mathtt{b}\rangle\}$	$\mathcal{S}$
C_c^k:	`  b = NULL;` `  }`	$\emptyset$	$\{\langle *\mathtt{v}_i, *\mathtt{d}\rangle \mid 1 \le i \le n\} \cup$ $\left\{\langle *\mathtt{v}_i, *\mathtt{v}_j\rangle \;\middle|\; \begin{array}{c}1 \le i,j \le n\\ \text{and } i \ne j\end{array}\right\}$
	/* end for all */		
D_a:	`  if (-)`	$\emptyset$	$\mathcal{S}$
D_b:	`  {` `  b = d;`	$\{\langle *\mathtt{b}, *\mathtt{d}\rangle\}$	$\mathcal{S}$
D_c:	`  d = NULL;` `  }`	$\emptyset$	$\{\langle *\mathtt{v}_i, *\mathtt{b}\rangle \mid 1 \le i \le n\} \cup$ $\left\{\langle *\mathtt{v}_i, *\mathtt{v}_j\rangle \;\middle|\; \begin{array}{c}1 \le i,j \le n\\ \text{and } i \ne j\end{array}\right\}$
W_x:	`}`	$\emptyset$	$\mathcal{S}$

Number of program point aliases in the precise solution of May Alias: $n + 11$
Number of aliases in *may-alias*: $3n^3 + 7n^2 + 6n + 18$

Figure 6.1: Program which yields very imprecise results

Lemma 6.1.1 *For any program P of size m with n k-limited object names which are involved in some non-reflexive alias (in the precise solution), precision$_k$(landi, P) is $\mathcal{O}(m * n)$. I.e., our algorithm reports $\mathcal{O}(m * n^2)$ aliases and there are $\Omega(n)$ k-limited alias pairs in the precise solution.*

1. There are $\Omega(n)$ alias pairs in the precise solution.

   An alias is pair of object names. Therefore, if n object names have non-reflexive aliases, there are at least $n/2$ alias pairs.

2. If there are n_1 distinct object names which are involved in some alias pair reported by our algorithm, then our algorithm reports reports $\mathcal{O}(m * n_1^2)$ point specific aliases.

   In the worst possible case everything is reported aliased to everything else ($n_1 *$ ($n_1 - 1$) non-reflexive alias pairs) at every program point (m).

3. $n_1 = n$

   Consider programs for which Assumption 5.1.1 (NULL, p. 81) holds. We will show that if we report an object name in an alias, then it is in some alias in the precise solution (and thus $n_1 = n$). To do this we only need to consider aliases that contain object names that may not yet have appeared in another alias. This is sufficient because during program execution there has to be a first time that an object name o appears in an alias. Trivially for that first time, o can not have appeared in any other alias. There are only three places in our algorithm where an alias involving an object name that has not appeared in another alias can be created:

   - For statements $p = q$ we create all the aliases in alias_consequences($\langle *p, *q \rangle$) which by our assumption must be aliases in the precise solution.
   - For statements $p = \&v$ we create all the aliases in alias_consequences($\langle *p, v \rangle$) which must be in the precise solution by our assumption
   - When pointer formal f is passed actual a we create all the aliases in alias_consequences($\langle *f, *a \rangle$) which are in the precise solution by our assumption.

Unfortunately, if Assumption 5.1.1 (NULL, p. 81) doesn't hold for a program then n_1 does not equal n. However, it is easy to remove that assumption from our algorithm[56], in which case we could easily prove that $n_1 = n$.

4. Therefore, our algorithm reports at most $\mathcal{O}(m * n^2)$ aliases.

□

Lemma 6.1.2 *For any alias approximation algorithm A, there exists a program P of size m with n k-limited object names which are involved in some non-reflexive alias (in the precise solution), such that precision(A, P) is* $\Omega(m * n)$. *I.e., A reports* $\Omega(m * n^2)$ *aliases even though there are only* $\mathcal{O}(n)$ *aliases in the precise solution.*

The proof of this stems from an understanding of just why Figure 6.1 is the worst case for our algorithm. Consider the code in `Part II` of the figure. If no aliases hold on entry to the loop the first time, it is easy to see that no aliases ever hold when the loop terminates and only $\Theta(n)$ aliases hold anyplace in the loop. However, when the alias $\langle *\mathtt{b}, *\mathtt{d} \rangle$ holds on entry to the loop the first time then

$$\mathcal{S} = \{\langle *\mathtt{b}, *\mathtt{d} \rangle\} \cup \{\langle *\mathtt{v}_i, *\mathtt{b} \rangle \,|\, 1 \leq i \leq n\} \cup \{\langle *\mathtt{v}_i, *\mathtt{d} \rangle \,|\, 1 \leq i \leq n\} \cup \left\{ \langle *\mathtt{v}_i, *\mathtt{v}_j \rangle \;\middle|\; \begin{array}{c} 1 \leq i, j \leq n \\ \text{and } i \neq j \end{array} \right\}$$

is the precise alias solution at every program point in `Part II` except at "`b = NULL`" where $\mathcal{S}$ holds except `b` is not aliased to anything and at "`d = NULL`" where $\mathcal{S}$ holds except `d` is not aliased to anything. The proof of this follows:

- $\langle *\mathtt{b}, *\mathtt{d} \rangle$ holds on path $P_{node}^{\langle *\mathtt{b}, *\mathtt{d} \rangle}$ to:
  - W_e: $P_{W_e}^{\langle *\mathtt{b}, *\mathtt{d} \rangle} = W_e$
  - C_a^l $(1 \leq l \leq n)$: $P_{C_a^l}^{\langle *\mathtt{b}, *\mathtt{d} \rangle} = W_e C_a^1 C_a^2 ... C_a^l$
  - C_b^l $(1 \leq l \leq n)$: $P_{C_b^l}^{\langle *\mathtt{b}, *\mathtt{d} \rangle} = P_{C_a^l}^{\langle *\mathtt{b}, *\mathtt{d} \rangle} \circ C_b^l$
  - C_c^l $(1 \leq l \leq n)$: There is no such $P_{C_c^l}^{\langle *\mathtt{b}, *\mathtt{d} \rangle}$.

[56] but such a version of the algorithm would be much less efficient.

- D_a: $P_{D_a}^{\langle *\mathtt{b},*\mathtt{d}\rangle} = P_{C_a^n}^{\langle *\mathtt{b},*\mathtt{d}\rangle} \circ D_a$
- D_b: $P_{D_b}^{\langle *\mathtt{b},*\mathtt{d}\rangle} = P_{D_a}^{\langle *\mathtt{b},*\mathtt{d}\rangle} \circ D_b$
- D_c: There is no such $P_{D_c}^{\langle *\mathtt{b},*\mathtt{d}\rangle}$.
- W_x: $P_{W_x}^{\langle *\mathtt{b},*\mathtt{d}\rangle} = P_{D_a}^{\langle *\mathtt{b},*\mathtt{d}\rangle} \circ W_x$

- $\langle *\mathtt{v}_i, *\mathtt{d}\rangle$ holds on path $P_{node}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle}$ to:
  - C_b^i: $P_{C_b^i}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} = P_{C_a^i}^{\langle *\mathtt{b},*\mathtt{d}\rangle} \circ C_b^i$
  - C_c^i: $P_{C_c^i}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} = P_{C_b^i}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} \circ C_c^i$
  - D_a: $P_{D_a}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} = P_{C_c^i}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} \circ C_a^{i+1}C_a^{i+2}...C_a^n D_a$
  - D_b: $P_{D_b}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} = P_{D_a}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} \circ D_b$
  - D_c: There is no such path $P_{D_c}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle}$.
  - W_x: $P_{W_x}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} = P_{D_a}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} \circ W_x$
  - W_e: $P_{W_e}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} = P_{W_x}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} \circ W_e$
  - C_a^l $(1 \le l \le n)$: $P_{C_a^l}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} = P_{W_e}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} \circ C_a^1 C_a^2 ... C_a^l$
  - C_b^l $(1 \le l \le n$ and $i \ne l)$: $P_{C_b^l}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} = P_{C_a^l}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} \circ C_b^l$
  - C_c^l $(1 \le l \le n$ and $i \ne l)$: $P_{C_c^l}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} = P_{C_b^l}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} \circ C_c^l$

- $\langle *\mathtt{v}_i, *\mathtt{b}\rangle$ holds on path $P_{node}^{\langle *\mathtt{v}_i,*\mathtt{b}\rangle}$ to:
  - D_b: $P_{D_b}^{\langle *\mathtt{v}_i,*\mathtt{b}\rangle} = P_{D_a}^{\langle *\mathtt{v}_i,*\mathtt{d}\rangle} \circ D_b$
  - D_c: $P_{D_c}^{\langle *\mathtt{v}_i,*\mathtt{b}\rangle} = P_{D_b}^{\langle *\mathtt{v}_i,*\mathtt{b}\rangle} \circ D_c$
  - W_x: $P_{W_x}^{\langle *\mathtt{v}_i,*\mathtt{b}\rangle} = P_{D_c}^{\langle *\mathtt{v}_i,*\mathtt{b}\rangle} \circ W_x$
  - W_e: $P_{W_e}^{\langle *\mathtt{v}_i,*\mathtt{b}\rangle} = P_{W_x}^{\langle *\mathtt{v}_i,*\mathtt{b}\rangle} \circ W_e$
  - C_a^l $(1 \le l \le n)$: $P_{C_a^l}^{\langle *\mathtt{v}_i,*\mathtt{b}\rangle} = P_{W_e}^{\langle *\mathtt{v}_i,*\mathtt{b}\rangle} \circ C_a^1 C_a^2 ... C_a^l$
  - C_b^l $(1 \le l \le n)$: $P_{C_b^l}^{\langle *\mathtt{v}_i,*\mathtt{b}\rangle} = P_{C_a^l}^{\langle *\mathtt{v}_i,*\mathtt{b}\rangle} \circ C_b^l$
  - C_c^l $(1 \le l \le n)$: There is no such path $P_{C_c^l}^{\langle *\mathtt{v}_i,*\mathtt{b}\rangle}$.
  - D_a: $P_{D_a}^{\langle *\mathtt{v}_i,*\mathtt{b}\rangle} = P_{C_a^n}^{\langle *\mathtt{v}_i,*\mathtt{b}\rangle} \circ D_a$

- $\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle$ $(i \neq j)$ holds on path $P_{node}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle}$ to:
  - C_b^j: $P_{C_b^j}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} = P_{C_a^j}^{\langle *\mathtt{v}_i, *\mathtt{b} \rangle} \circ C_b^j$
  - C_c^j: $P_{C_c^j}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} = P_{C_b^j}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} \circ C_c^j$
  - D_a: $P_{D_a}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} = P_{C_c^j}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} \circ C_a^{j+1} C_a^{j+2} ... C_a^n D_a$
  - D_b: $P_{D_b}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} = P_{D_a}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} \circ D_b$
  - D_c: $P_{D_c}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} = P_{D_b}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} \circ D_c$
  - W_x: $P_{W_x}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} = P_{D_a}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} \circ W_x$
  - W_e: $P_{W_e}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} = P_{W_x}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} \circ W_e$
  - C_a^l $(1 \leq l \leq n)$: $P_{C_a^l}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} = P_{W_e}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} \circ C_a^1 C_a^2 ... C_a^l$
  - C_b^l $(1 \leq l \leq n$ and $j \neq l)$: $P_{C_b^l}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} = P_{C_a^l}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} \circ C_b^l$
  - C_c^l $(1 \leq l \leq n$ and $j \neq l)$: $P_{C_c^l}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} = P_{C_b^l}^{\langle *\mathtt{v}_i, *\mathtt{v}_j \rangle} \circ C_c^l$

Thus the code from `Part II` (padded to size m with statements that cannot have any affects on aliasing) has $\Theta(n)$ aliases if no aliases hold before the code is executed and has $\Theta(m * n^2)$ if the alias $\langle *\mathtt{b}, *\mathtt{d} \rangle$ holds before the code is executed. Since any approximate algorithm A, by definition must be able to produce an erroneous alias, $\langle *\mathtt{b'}, *\mathtt{d'} \rangle$, there must be a program P that produces $\langle *\mathtt{b'}, *\mathtt{d'} \rangle$ at the exit of the program. If `b'` isn't a variable, simply assign `b'` to `b`, otherwise we can use `b'` for `b` in `Part II`. Since we can do the same for `d'` we can we can assume that `b` and `d` in `Part II` are variables.

Consider the following program: P followed by a statement `x = NULL` (for every pointer variable `x` in P except `b` and `d`), this kills all the real aliases, but cannot destroy the alias $\langle *\mathtt{b}, *\mathtt{d} \rangle$ (if the algorithm is safe), and follow this by `Part II`. The approximate algorithm must report $\Omega(m * n^2)$ aliases even though there are only $\mathcal{O}(n)$ aliases[57].

□

[57] because the number of aliases in the original P is a constant independent of m and n

6.2 Prototype

The algorithm presented in Chapter 5 has been implemented in C. Our algorithm finds aliases for a language that is a reduced version of C. The main attributes of C not handled by our algorithm are: union types, nested structures, casting[58], pointers to functions, and exception handling. The first three of these omissions are not theoretically difficult to handle, but complicate the implementation. The other two require more theoretical examination. We do allow arrays and pointer arithmetic; however, we deal with these on a very simple and naive level.

ICFG For building ICFGs, we were fortunate to have access to *ptt*, which is a program developed by Siemens Research Corporation. We thank Siemens as well as Hemant Pande, Michael Platoff, and Michael Wagner of Siemens for their assistance with *ptt*. *Ptt* generates three address code from C source and builds the ICFG of the three address code. This superficially changes the notion (and number) of program points and the number of variables since the three address code version has its own temporaries. However, the general aliasing patterns remain the same as for the C source.

Our implementation differs slightly from the algorithm as presented in Chapter 5. The difference has to do with *non_visible*. In Chapter 5, a non-visible object name at a call node would get mapped to *non_visible* at the entry node of the called procedure. In the implementation, we simply replace the variable name in the non-visible object name with *non_visible*. For example, $*x$, where x is not visible in the called procedure, would become $*non_visible$ at the entry node of the called procedure and $x\text{->}next$ would map to $non_visible\text{->}next$. The reason for this is simply so that every alias between an object name, w, and a non-visible object is represented by only one alias[59].

---

[58] Casting of *malloc()* is not a problem for our implementation.

[59] As described in Chapter 5 an alias between w and the non-visible object $*x$ could have been represented by both $\langle w, non_visible\rangle$ with *non_visible* being $*x$ and $\langle w, *non_visible\rangle$ with *non_visible* being x. In the implementation, only the latter representation is allowed.

Analyzing C Programs Later in this chapter we present empirical data documenting algorithm's behavior on C programs. Because of restrictions on the C constructs we handle and limitations of the parser, we couldn't analyze the C source directly; therefor, we modified the C sources in the following ways:

- The parser could only handle programs in a single source file. Multiple file programs were combined into a single file.
- The parser has difficulties with "#if" statements and these were resolved by hand.
- The parser isn't equipped with resolving undefined routines with the lint libraries (*/usr/lib/lint/llib-lc* on Unix systems). This resolution had to be performed by hand.
- Not handling function variables turned out to be rather restrictive, so we solved for function variables by hand and transformed the code, replacing calls with function variables to calls to the possible values of the function variables. Fortunately, all the programs we examined didn't have complicated usage of function variables.
- We assumed that for any programs with exception handling, the exceptions could not effect the alias solution. This is not necessarily a safe assumption, but it shouldn't invalidate our observations in this chapter.

6.2.1 Optimizations

Our implementation has demonstrated that the majority of the time used by our implementation comes from copying the same alias to all the nodes in the ICFG that it reaches, once it has been created. There seems to be substantial room for execution time improvement by doing a certain amount of preprocessing on the ICFG and thus eliminating much of this redundant copying. In later sections, we give timing results from our prototype implementation; they are only meant as a indication of the current efficiency of our algorithm.

We have performed one optimization to the algorithm as presented in Chapter 5. Consider a statement in the program that isn't an assignment to a pointer (for example,

`printf("Hi\n");`). This statement has no affects on aliasing, and its alias solution is simply the union of its predecessors' solutions. For nodes such as this, we simply remember that its solution is the union of its predecessors' solutions and remove it from the ICFG[60]. Thus we avoid some of the redundant copying.

We are currently implementing yet another optimization, which was not used for obtaining the timings reported in this Chapter. Pointer assignment is the only kind of statement that can introduce aliases at a node which were not present at some immediate predecessor node and prevent propagation of an alias that was present at some immediate predecessor.[61] For an assignment "$p = q$" and an alias $\langle a, b \rangle$ if is_prefix_with_deref(q,a) is $false$ and is_prefix(p,a) is $false$ (with similar restrictions on b), $\langle a, b \rangle$ holds on a path to "$p = q$" only if it holds on the same path to the immediate predecessor of "$p = q$". Say we had a group of statements, S[62], with a head node, h, such that for any node $s \in$ S, if $s \neq h$ then all the immediate predecessors of s are also in S. If we know that a and b "never occur" in the group S,[63] then we know that whenever a is aliased to b at h it is also aliased at every node in S. We can optimize our code as follows (assume a and b never occur in S):

- If we want to know if *may-hold*($[(s, \mathcal{AA}), \langle a, b \rangle]$), $s \in$ S, then
  we check if *may-hold*($[(h, \mathcal{AA}), \langle a, b \rangle]$) is $true$.

- If we ever find out that *may-hold*($[(h, \mathcal{AA}), \langle a, b \rangle]$) then
  instead of passing the information about $\langle a, b \rangle$ to the successors of h, we pass the information to the successors of S.

---

[60] By removing it from and adding its successors to its predecessor's successor lists.

[61] Call-by-value parameter bindings can be thought of as assignment statements.

[62] For simplicity assume that S is an intraprocedural portion of the ICFG.

[63] By this we mean that for every object name l, the LHS of a pointer assignment in S, is_prefix(l,a) is $false$ and for every object name r, the RHS of a pointer assignment in S, is_prefix_with_deref(r,a) is $false$.

6.3 Empirical Comparison to Weihl's Algorithm

6.3.1 Number of Aliases

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

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

In Table 6.1 we compare the size of the *program-aliases* solution given the *may-alias* computed by our algorithm versus the number of aliases reported by Weihl's algorithm. We do this on the same ten C programs that we analyzed using Weihl's algorithm to justify our development of a new approximation algorithm [LR90][64]; a short description of each program can be found in Figure 6.2 on p. 122. As expected Weihl's algorithm reports more program aliases than our algorithm. In 8 out of 10 cases, Weihl reported more than ten times the number of aliases that we reported and for the programs in the table, on average Weihl reported 47.2 times as many aliases[65]. For both our and Weihl's algorithm, all object names were k-limited with $k = 1$. Given our implementation of Weihl's algorithm we cannot do a comparison with any other value for k.

6.3.2 Time

We would like to compare the time of our algorithm versus Weihl's algorithm, but unfortunately, our implementation Weihl's algorithm does not lend itself to timing. Weihl's algorithm is basically a two stage algorithm. First, the set AFFECT is initialized from the source. Second, aliases are computed by a transitive closure (ALIAS = AFFECT* $\circ$ (AFFECT*)^T). We are only able to time the second stage of the calculation, which on large programs dominates execution time. In Table 6.2, we compare the time of our algorithm versus the time for Weihl's algorithm (second stage only). Again, this is a

[64] In [LR90], the number of aliases found by Weihl are reported as twice what they are reported here because aliases were counted twice (i.e., once for $\langle a, b\rangle$ and once for $\langle b, a\rangle$).

[65] This number is not statistically significant. It is not meant as a claim over all programs, but is simply an observation about the ten programs in Table 6.1.

Program	Lines	Weihl	*program-aliases*	Weihl / *program-aliases*
ul	523	4,851	340	14.2
pokerd	1,354	62,225	300	207.4
compress	1,488	6,316	305	20.7
loader	1,522	39,059	472	82.7
learn	1,642	61,845	813	76.0
ed	1,772	1,796	1,664	1.0
diff	1,793	44,366	1,323	33.5
tbl	2,545	4,401	1,169	3.7
make	2,952	207,906	9,274	22.4
lex	3,315	9,490	925	10.2

Table 6.1: Number of Aliases: Comparison to Weihl

comparison of two prototype implementations and is meant to give a rough feel for the timing differences; both implementations could be optimized further. As expected, our algorithm takes longer than Weihl's algorithm. In one case, our algorithm was faster than Weihl's algorithm and in five of ten cases we do not take more than twice the time of Weihl's algorithm. For the ten programs in Table 6.2, on average our algorithm uses 9.8 times as much time. In Table 6.3, we present both the time and number of aliases ratios. *Ul*, *ed*, and *tbl* have timing ratios much higher than average. This is because we compute point-specific aliases and the time our algorithm takes is approximately proportional to the number of point-specific aliases found. For *ul*, *ed*, and *tbl* the ratio of number of point-specific aliases found by our algorithm to the number of aliases reported by Weihl is the largest.[66]

6.4 May Alias Solution

6.4.1 Empirically Measured Solution Size

In this and the remaining sections of this chapter, we present various empirical results of experiments on 19 C programs. This suite is a superset of the programs that we

[66]The ratios were: *ed*, 84.8; *tbl*, 84.4; *ul*, 16.2; *lex*, 12.6; *make*, 10.3; *compress*, 4.0; *diff*, 1.8; *learn*, 1.1; *loader*, 0.1; *pokerd*, 0.1

Program	Lines	Weihl	*program-aliases*	*program-aliases* / Weihl
ul	523	3s	58s	19.3
pokerd	1,354	1m 24s	10s	0.1
compress	1,488	4s	6s	1.5
loader	1,522	36s	1m 1s	1.6
learn	1,642	46s	1m 34s	2.0
ed	1,772	6s	2m 33s	25.5
diff	1,793	58s	1m 49s	1.8
tbl	2,545	10s	4m 59s	29.9
make	2,952	17m 21s	3h 3m 55s	10.6
lex	3,315	18s	2m 5s	6.9

Table 6.2: Time: Comparison to Weihl

Program	Lines	Number of Aliases	Time
		Weihl / *program-aliases*	*program-aliases* / Weihl
ul	523	14.2	19.3
pokerd	1,354	207.4	0.1
compress	1,488	20.7	1.5
loader	1,522	82.7	1.6
learn	1,642	76.0	2.0
ed	1,772	1.0	25.5
diff	1,793	33.5	1.8
tbl	2,545	3.7	29.9
make	2,952	22.4	10.6
lex	3,315	10.2	6.9

Table 6.3: Number of Aliases and Time: Comparison to Weihl

compared to Weihl's algorithm in Chapter 6.3. By historical accident, our implementation of Weihl's algorithm has more restrictions on C then the implementation of our algorithm; thus we could not compare all 19 programs to Weihl's algorithm. These programs came from a pool of available C programs that was collected for an ongoing empirical study of the structure of C programs [RP88]. Programs were selected from that pool if they met the following restrictions:

- Program size not much larger than 5,000 lines.
- Use of function variables was simple enough to resolve by hand.
- Little or no pointer casting.
- Few nested structures.

The sample is by no means large enough to draw general conclusions about algorithm behavior, but it is large enough to indicate that our algorithm performs well over a limited domain of C programs. Figure 6.2 contains a short description of the programs we used (many of these descriptions can be found in [RP88]).

Since our implementation is defined in terms of k-limiting, where k is a run-time constant, our studies include a comparison of our algorithm's behavior on the same program for different values of k. We tried $k = 1, 2, 3$, and 4. Because of an unfortunate choice for the implementation of object names, we did more limited experiments on some programs:

- $k \leq 3$ for *assembler*, *loader*, *poker*, and *simulator*
- $k = 1$ for *make*

Table 6.4 gives the size of the May Alias solution found by our algorithm for $k = 2$. Most interesting is the *Aliases/node* and the *Max Aliases at one Node* columns of this table. *Aliases/node* gives the average number of aliases found at each program point in the program. It ranges from 0.7 to 238.7 with an average of 41.4.[67] *Max Aliases*

[67] For make with $k = 1$, it is 674.8. The average is 74.7 if make is included.

Program	Description
allroots	Find all the roots of a polynomial
assembler	SIC assembler [Bec85]
compress	UNIX: compress data
diff	UNIX: show file differences
diffh	fast diff; UNIX diff -h
ed	UNIX line editor
football	a football statistics program
fixoutput	a simple translator
learn	introduction to UNIX
lex	generator of a lexical analyzer
lex315	a scanner for a subset of C
loader	SIC loader [Bec85]
make	UNIX: helps maintain programs
poker	a game of cards
pokerd	a variant of poker
simulator	SIC machine simulator/debugger [Bec85]
tbl	UNIX: format tables for nroff and troff
tp	tape formatting
ul	UNIX: underline

Figure 6.2: Description of programs in our empirical study

Program	ICFG Nodes	May Aliases	Aliases/ Node	Max Aliases at one Node
allroots	359	*240*	*0.7*	*3*
fixoutput	584	*1,856*	*3.2*	*12*
diffh	587	*7,184*	*12.2*	*45*
poker	865	1,962	2.3	108
ul	1,256	*79,555*	*63.3*	*101*
lex315	1,308	*5,712*	*4.4*	*12*
loader	1,355	55,588	41.0	133
compress	1,513	5,844	3.9	21
tp	1,531	*60,700*	*39.7*	*190*
pokerd	1,726	16,997	9.9	241
learn	2,573	103,464	40.2	110
ed	3,003	*150,369*	*50.1*	*98*
assembler	3,137	748,849	238.7	442
make ($k = 1$)	3,171	2,155,188	674.8	2,062
diff	3,455	83,964	24.3	88
simulator	4,857	454,802	93.6	292
football	5,511	234,752	42.6	92
tbl	5,601	337,833	60.3	120
lex	6,121	85,017	13.9	53

Highlighted numbers are valid for the precise solution.

Table 6.4: Size of our May Alias Solution ($k = 2$)

at one Node gives the maximum number of aliases found at any program point in the program. It ranges from 3 to 442 with an average of 120.[68] Both of these numbers indicate that the number of aliases at individual program points is small which implies that the information provided might be precise enough to be useful. For example, since there are few aliases at any program point, the set of variables modified at a statement which stores indirectly through a pointer is likely to be small also.

Table 6.5 gives the information in Table 6.4 ranging over various values of k; this gives an idea of how increasing k affects the alias solution. One unexpected observation from this table is that for many of the programs (11) the $k = 2$ and $k = 3$ solutions had the same size. When this occurred we omitted the $k = 4$ solution because it is the same as well. For two other programs (*football* and *tbl*), the $k = 3$ and $k = 4$ solution

[68] For make ($k = 1$) it is 2,062 which would raise the average to 222.

have the same size.

In general, we expect the size of the reported alias solution to vary directly with the value of k. For a program where aliases $\langle *a, *b \rangle$ and $\langle **a, **b \rangle$ hold on paths to program point n; when $k = 1$, only the alias $\langle *a, *b \rangle$ needs to be reported at n; when $k = 2$ both must be reported. However, an examination of Table 6.5 shows that for *compress, ed, tbl,* and *lex*, the number of aliases reported decreased when k increased. *compress* and *tbl* show a smaller number of aliases because each contained a statement that is not an assignment to a pointer but gets 1-limited to a pointer assignment. This case is illustrated by the following example: "$**a = **b$" where both a and b are of type "int **" gets 1-limited to "$*a = *b$" which is an assignment to a pointer. Also, if there is an assignment "$*a = ...$", $\langle **a, b \rangle$ holds on some path before "$*a = ...$" is executed, then $\langle **a, b \rangle$ is killed by this assignment. However, if $k = 1$, then the alias is represented by $\langle *a, b \rangle$ and is not killed by the assignment. We suspect that this is what causes the decrease in aliases for *ed* and *lex*, but the aliasing is too complex to verify by this claim hand.

We expected most programs to have some sort of recursive data structure; however, on our data set this was not the case. This allows us to make some very strong claims. In Chapter 6.4.2, we are able to make claims about the precision of our solution for the programs in Table 6.5. We can conclude for 9 programs (7 with $k = 3$ and 2 with $k = 4$) that the solution found by our algorithm is indeed the precise solution (as defined in Chapter 2.5). Thus the *highlighted* entries in all tables in this chapter represent precise information about the aliasing solution.

6.4.2 Measurement of Empirical Precision

In Chapter 4, we mentioned four sources of approximation. They are:

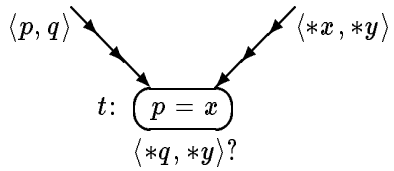
1. In any program that contains recursive data structures, there are a potentially infinite number of objects which can have aliases. We represent all possible objects by a finite (polynomial) number of objects.

2.

Program (Nodes)	May Aliases			Aliases/ Node			Max Aliases at one Node		
	k=1	k=2	k=3	k=1	k=2	k=3	k=1	k=2	k=3
allroots (359)	240	240	*240*	0.7	0.7	*0.7*	3	3	*3*
fixoutput (584)	1,840	1,856	*1,856*	3.2	3.2	*3.2*	10	12	*12*
diffh (587)	5,986	7,184	*7,184*	10.2	12.2	*12.2*	34	45	*45*
poker (865)	770	1,962	4,362	0.9	2.3	5.0	34	108	258
ul (1,256)	79,003	79,555	*79,555*	62.9	63.3	*63.3*	95	101	*101*
lex315 (1,308)	5,696	5,712	*5,712*	4.4	4.4	*4.4*	10	12	*12*
loader (1,355)	25,418	55,588	102,121	18.8	41.0	75.4	55	133	241
compress (1,513)	6,404	5,844	5,844	4.2	3.9	3.9	21	21	21
tp (1,531)	51,256	60,700	*60,700*	33.5	39.7	*39.7*	163	190	*190*
pokerd (1,726)	6,517	16,997	36,419	3.8	9.9	21.1	82	241	523
		for k=4:	58,257		for k=4:	33.8		for k=4:	838
learn (2,573)	73,699	103,464	103,464	28.7	40.2	40.2	89	110	110
ed (3,003)	150,392	150,369	*150,369*	50.1	50.1	*50.1*	98	98	*98*
assembler (3,137)	283,770	748,849	1,436,120	90.5	238.7	457.8	155	442	914
make (3,171)	2,155,188	-	-	674.8	-	-	2,062	-	-
diff (3,455)	81,543	83,964	83,964	23.6	24.3	24.3	86	88	88
simulator (4,857)	225,550	454,802	767,813	46.4	93.6	158.1	121	292	539
football (5,511)	180,414	234,752	234,762	32.7	42.6	42.6	72	92	92
		for k=4:	*234,762*		for k=4:	*42.6*		for k=4:	*92*
tbl (5,601)	371,818	337,833	338,743	66.4	60.3	60.5	130	120	120
		for k=4:	*338,743*		for k=4:	*60.5*		for k=4:	*120*
lex (6,121)	120,205	85,017	85,017	19.6	13.9	13.9	52	53	53

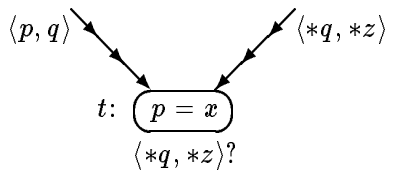
Highlighted numbers are valid for the precise solution.

Table 6.5: Size of our May Alias Solution



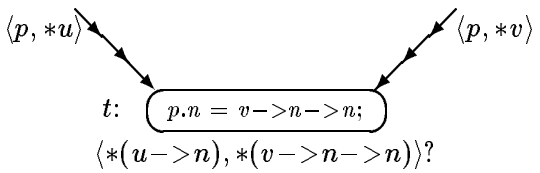
If both $\langle p, q\rangle$ and $\langle *x, *y\rangle$ occur on the **same** path, then $\langle *q, *y\rangle$ holds on that path extended by t; therefore, we conclude this, even though it may not be true.

3.



If on at least one path to an immediate predecessor, $\langle *q, *z\rangle$ holds and neither $\langle p, q\rangle$ nor $\langle p, z\rangle$ does, then $\langle *q, *z\rangle$ holds on that path extended by t. However, if on all paths to immediate predecessors on which $\langle *q, *z\rangle$ holds, $\langle p, q\rangle$ also holds, then $\langle *q, *z\rangle$ does not necessarily hold on any path to t. In either case, for safety, we assume $\langle *q, *z\rangle$ holds on some path to t.

4.



Normally, $\langle *(u{->}n), *(v{->}n{->}n)\rangle$ should hold on a path to t because assigning $v{->}n{->}n$ to $p.n$ is also assigning $v{->}n{->}n$ to $u{->}n$[69]. However, $\langle *(u{->}n), *(v{->}n{-}\ n)\rangle$ need not hold. If, for example, $\langle p, *u\rangle$ and $\langle p, *v\rangle$ hold on the same path then $\langle *(u{->}n), *(v{->}n{->}n)\rangle$ does not necessarily hold.

Lemma E.1.1 (p. 245) states that given Assumption 5.1.1 (NULL, p. 81) these four cases are the only sources of imprecision in our algorithm.

[69] on the path on which $\langle p, *u\rangle$ holds

We've modified our algorithm to count these cases. Instead of having a $true/false$ lattice, we used a YES/MAYBE/NO lattice where YES $\sqsubseteq$ MAYBE $\sqsubseteq$ NO. With this implementation, $may\text{-}hold([(node, \mathcal{AA}), \mathcal{PA}]) =$

- NO iff $may\text{-}hold([(node, \mathcal{AA}), \mathcal{PA}])$ is $false$
- MAYBE, if $may\text{-}hold([(node, \mathcal{AA}), \mathcal{PA}])$ is $true$ because of case 2, 3, or 4 above
- MAYBE, if $may\text{-}hold([(node, \mathcal{AA}), \mathcal{PA}])$ is implied by some $may\text{-}hold([(node', \mathcal{AA}'), \mathcal{PA}']) =$ MAYBE, but not by any $may\text{-}hold([(node'', \mathcal{AA}''), \mathcal{PA}'']) =$ YES
- YES otherwise

Now define $\%YES_k(P)$ for *may-hold* computed from P with k-limit constant k:

$$\%YES_k(P) = 100 * \left(\frac{\left| \left\{ (node, \mathcal{PA}) \;\middle|\; (\exists \mathcal{AA}) may\text{-}hold([(node, \mathcal{AA}), \mathcal{PA}]) = \text{YES} \right\} \right|}{\left| \left\{ (node, \mathcal{PA}) \;\middle|\; \begin{array}{l} (\exists \mathcal{AA}) may\text{-}hold([(node, \mathcal{AA}), \mathcal{PA}]) = \text{YES} \\ \text{or } may\text{-}hold([(node, \mathcal{AA}), \mathcal{PA}]) = \text{MAYBE} \end{array} \right\} \right|} \right)$$

Given that there are only the aforementioned four types of approximations and that the fourth type does not occur; we claim that

$$\left\{ (node, \mathcal{PA}) \;\middle|\; (\exists \mathcal{AA}) may\text{-}hold([(node, \mathcal{AA}), \mathcal{PA}]) = \text{YES} \right\} \subseteq limit_k(\text{precise solution})$$

This is true because type 1 approximation does not matter since we are using $limit_k$; any alias that is the result of or implied by a type 2 or 3 approximation has the value MAYBE. Any alias marked with a YES must be in $limit_k$(precise solution) since by Lemma E.1.1 and our assumption about type 4 approximations, no other type of approximation can occur. Thus $\%YES_k(P) \leq 100 * (1/precision_k(landi, P))$.[70] and thus can use $\%YES_k(P)$ to bound the precision of our solution.

The above discussion is only true when k is greater than the maximum number of dereferences in any object name of pointer type that syntactically appears in P

[70] Remember, when k is greater than or equal to the maximum number of dereferences in any object name with an alias in the precise solution, $precision_k(landi, P) = precision(landi, P)$ (see Chapter 6.1).

(i.e., is the LHS or RHS of an assignment or an actual in a procedure call).[71] In our implementation if we had two two-dimensional arrays of integers A and B, then the non-pointer assignment "A[i,j] = B[i,j]" would be represented as "**A = **B". However, if k was 1, we would see this statement as "*A = *B" which we is considered a pointer assignment. This would certainly introduce imprecision that is not accounted for by $\%YES_k(P)$.

Table 6.6 presents empirical precision results for the 19 programs we've been examining. For all but one of the programs we can get a bound on the precision of our solution[72]. For nine programs we were able to find the precise solution. In general the precision results are very encouraging.

6.5 May Alias Solution Size vs Conditional May Alias Size

Originally, our research goal was design an algorithm for computing May Alias. Instead we decided to solve for Conditional May Alias from which we could trivially extract May Alias. If May Alias is the problem we want to solve, then the Conditional May Alias solution should be about the same size as the May Alias solution, because the former is what we compute and store. Table 6.7 indicates that this is indeed the case in terms of the number of relations computed. The maximum ratio of our Conditional May Alias solution to our May Alias solution is 5.9; for $k = 2$ the average is 1.6. As indicated in [PRL91], the additional information in Conditional May Alias can lead to more precise results in problems like Reaching Definitions. Thus, Conditional May Alias could well be more useful than May Alias and the double relation size between them is bearable.

---

[71] This is a limit of our implementation, it is not fundamental to the algorithm itself.

[72] We could not get a bound for *make* because we were not able to solve for aliases with a sufficiently large k.

Program	ICFG Nodes	May Aliases k=1	May Aliases k=2	May Aliases k=3	$\%YES_k$(Program) k=1	$\%YES_k$(Program) k=2	$\%YES_k$(Program) k=3
allroots	359	240	240	240	100	100††	100††
fixoutput	584	1,840	1,856	1,856	100	100††	100††
diffh	587	5,986	7,184	7,184	100	100	100††
poker	865	770	1,962	4,362	100	100†	100†
ul	1,256	79,003	79,555	79,555	100	100	100††
lex315	1,308	5,696	5,712	5,712	100	100††	100††
loader	1,355	25,418	55,588	102,121	99	90	77†
compress	1,513	6,404	5,844	5,844	85	72	72††
tp	1,531	51,256	60,700	60,700	100	100	100††
pokerd	1,726	6,517	16,997	36,419	78	53	40
			for k=4:	58,257		for k=4:	35†
learn	2,573	73,699	103,464	103,464	99.9	99.9	99.9††
ed	3,003	150,392	150,369	150,369	100	100	100††
assembler	3,137	283,770	748,849	1,436,120	20	16	13†
make	3,171	2,155,188	-	-	74	-	-
diff	3,455	81,543	83,964	83,964	91	91	91††
simulator	4,857	225,550	454,802	767,813	91	99†	97†
football	5,511	180,414	234,752	234,762	100	100	100
			for k=4:	234,762		for k=4:	100 ††
tbl	5,601	371,818	337,833	338,743	88	100	100
			for k=4:	338,743		for k=4:	100††
lex	6,121	120,205	85,017	85,017	88	84	84††

$^{\dagger}\%YES_k(Program) \leq 100 * (1/precision_k(landi, Program))$
$^{\dagger\dagger}\%YES_k(Program) \leq 100 * (1/precision(landi, Program))$

Table 6.6: Precision of our May Alias Solution

Program (Nodes)	May Aliases			*may-hold*			*may-hold*/ May Alias		
	k=1	k=2	k=3	k=1	k=2	k=3	k=1	k=2	k=3
allroots (359)	240	240	240	955	955	955	4.0	4.0	4.0
fixoutput (584)	1,840	1,856	1,856	1,840	1,856	1,856	1.0	1.0	1.0
diffh (587)	5,986	7,184	7,184	7,580	8,842	8,842	1.3	1.2	1.2
poker (865)	770	1,962	4,362	904	2,096	4,496	1.2	1.1	1.0
ul (1,256)	79,003	79,555	79,555	89,776	92,532	92,532	1.1	1.2	1.2
lex315 (1,308)	5,696	5,712	5,712	5,836	5,852	5,852	1.0	1.0	1.0
loader (1,355)	25,418	55,588	102,121	61,221	187,337	508,924	2.4	3.4	5.0
compress (1,513)	6,404	5,844	5,844	7,799	7,346	7,346	1.2	1.3	1.3
tp (1,531)	51,256	60,700	60,700	74,367	92,550	92,550	1.5	1.5	1.5
pokerd (1,726)	6,517	16,997	36,419	14,448	26,547	46,054	2.2	1.6	1.3
		for k=4:	58,257		for k=4:	67,997		for k=4:	1.2
learn (2,573)	73,699	103,464	103,464	103,776	142,761	142,761	1.4	1.4	1.4
ed (3,003)	150,392	150,369	150,369	201,331	201,284	201,284	1.3	1.3	1.3
assembler (3,137)	283,770	748,849	1,436,120	900,372	2,385,797	6,625,431	3.2	3.2	4.6
make (3,171)	2,155,188	-	-	12,624,949	-	-	5.9	-	-
diff (3,455)	81,543	83,964	83,964	113,836	109,345	109,345	1.4	1.3	1.3
simulator (4,857)	225,550	454,802	767,813	273,519	543,305	922,463	1.2	1.2	1.2
football (5,511)	180,414	234,752	234,762	324,042	414,222	414,232	1.8	1.8	1.8
		for k=4:	234,762		for k=4:	414,232		for k=4:	1.8
tbl (5,601)	371,818	337,833	338,743	425,969	401,167	401,938	1.1	1.2	1.2
		for k=4:	338,743		for k=4:	401,938		for k=4:	1.1
lex (6,121)	120,205	85,017	85,017	145,643	109,717	109,717	1.2	1.3	1.3

Table 6.7: Size of our May Alias Solution vs Size of *may-hold*

6.6 Algorithm Time Performance

6.6.1 Theoretical Issues

While designing our algorithm, we were trying to make the complexity $\mathcal{O}$(size of the solution). This is not possible in the worst case. We devised an algorithm that only does work for those elements of the Conditional May Alias solution whose value is *true*; however, we can not guarantee a constant amount of work for each of the following:

- bind (Figure 5.3 and Figure 5.5) uses doubly nested loops over formal/actual lists. These lists tend to grow very slowly, if at all, with program size. They can be considered constant size [CK89].

- alias_at_exit_implies (Appendix D.3) contains loops that must be executed once for each call site to the procedure that contains the exit node. This will be a problem only if the number of *true* Conditional May Aliases generated by the loop is less than the number of call sites. In general, we do not expect this to be the case, but this loop could be replaced by one that depends only on the number of *true* cases generated and not on the number of call sites.

- There are multiple ways that $\mathit{may\text{-}hold}([(\mathit{node}, \mathcal{AA}), \mathcal{PA}])$ can be generated. It is possible for work to be done once for each different way that it can be generated. If *node* is not a return node, this is not a problem because the number of different ways that $\mathit{may\text{-}hold}([(\mathit{node}, \mathcal{AA}), \mathcal{PA}])$ can become true is $\mathcal{O}\left(\left|\left\{\ m\ \middle|\ \ll m, \mathit{node} \gg \text{ is an edge in the ICFG}\ \right\}\right|\right)$. Since in programs, the number of edges is order the number of nodes, in an amortized sense there is a constant number of ways that a Conditional May Alias can become *true*.

  If *node* is a return node (for simplicity, ignoring visibility):

$$\mathit{may\text{-}hold}([(\mathit{node}, \mathcal{AA}), \mathcal{PA}]) =$$
$$\mathit{holds}([(\mathit{exit}, \emptyset), \mathcal{PA}]) \vee \bigvee_{AA' \in \mathcal{ASSUMED}} \begin{pmatrix} \mathit{holds}([(\mathit{exit}, AA'), \mathcal{PA}]) \wedge \\ \mathit{holds}([(\mathit{call}, \mathcal{AA}), \mathit{back\text{-}bind}_{\mathit{call}}(AA')]) \end{pmatrix}$$

Thus the number of different ways that *may-hold*($[(node, \mathcal{AA}), \mathcal{PA}]$) can become true is $1 + |\{AA' \,|\, \textit{may-hold}([(exit, AA'), \mathcal{PA}]) \text{ is } true\}|$ which can be at most Max-assumed which we define as

$$1 + max_{(\text{all possible aliases, } \mathcal{PA})} \left| \left\{ \mathcal{AA} \;\middle|\; \begin{array}{c} (\exists\ node)\ node \in \text{ICFG and} \\ \textit{may-hold}([(node, \mathcal{AA}), \mathcal{PA}]) \\ \text{is } true \end{array} \right\} \right|$$

Unfortunately, Table 6.8 indicates that Max-assumed is not constant. Table 6.8 is sorted by the size of *may-hold*, so that it is easier to observe how Max-assumed increases as *may-hold* increases. It does appear to be a slowly growing function (34 when there are 2,385,797 true Conditional May Aliases [73]). Thus, there is hope that the algorithm's running behavior will be linear in the size of the Conditional May Alias solution.

Table 6.9 contains the same information as Table 6.8 for different values of k. It is also sorted by size of *may-hold* (for $k = 2$). In some cases, Max-assumed increases as k increases, but in others Max-assumed decreases. Max-assumed can increase if an alias holds conditional on two different assumed aliases that get k-limited into the same alias as in this program fragment.

```
P() {
    temp = g;
n:while (temp != NULL)
      temp = temp->next;
}
```

$\langle$*temp, *h$\rangle$ holds on a path to n
conditional on $\langle$*g, *h$\rangle$ or $\langle$*(g->next), *h$\rangle$
or $\langle$*(g->next->next), *h$\rangle$ or ...
holding at entry of P

Max-assumed can decrease if for smaller k, multiple aliases are k-limited into one alias, thus unioning their assumed alias sets, but for larger k they are not unioned.

6.6.2 Conclusions

We wanted to claim that our algorithm performs linearly in the size of *may-hold*, but we do not have a theoretical justification for such a claim. Thus, we decided to

[73] and 31 for make ($k = 1$) with 12,624,949 Conditional May Aliases

Program	*may-hold*	Max-assumed	Avg-assumed
allroots	955	1	1.0
fixoutput	1,856	1	1.0
poker	2,096	1	1.0
lex315	5,852	1	1.0
compress	7,346	1	1.0
diffh	8,842	2	1.0
pokerd	26,547	1	1.0
ul	92,532	2	1.0
tp	92,550	2	1.0
diff	109,345	4	1.0
lex	109,717	3	1.0
learn	142,761	2	1.0
loader	187,337	11	1.6
ed	201,284	4	1.1
tbl	401,167	6	1.0
football	414,222	2	1.0
simulator	543,305	10	1.0
assembler	2,385,797	34	1.9
make ($k = 1$)	12,624,949	31	3.6

$$\text{Max-assumed} \equiv max_{(\text{all possible aliases, } \mathcal{PA})} \left| \left\{ \mathcal{AA} \;\middle|\; \begin{array}{c} (\exists\ node)\ node \in \text{ICFG and} \\ may\text{-}hold([(node, \mathcal{AA}), \mathcal{PA}]) \\ \text{is } true \end{array} \right\} \right|$$

$$\text{Avg-assumed} \equiv avg_{(\text{all possible aliases, } \mathcal{PA})} \left| \left\{ \mathcal{AA} \;\middle|\; \begin{array}{c} (\exists\ node)\ node \in \text{ICFG and} \\ may\text{-}hold([(node, \mathcal{AA}), \mathcal{PA}]) \\ \text{is } true \end{array} \right\} \right|$$

Table 6.8: Number of Assumptions per Alias ($k = 2$)

Program	*may-hold*			Max-assumed			Avg-assumed		
	k=1	k=2	k=3	k=1	k=2	k=3	k=1	k=2	k=3
allroots	955	955	955	1	1	1	1.0	1.0	1.0
fixoutput	1,840	1,856	1,856	1	1	1	1.0	1.0	1.0
poker	904	2,096	4,496	1	1	1	1.0	1.0	1.0
lex315	5,836	5,852	5,852	1	1	1	1.0	1.0	1.0
compress	7,799	7,346	7,346	1	1	1	1.0	1.0	1.0
diffh	7,580	8,842	8,842	2	2	2	1.0	1.0	1.0
pokerd	14,448	26,547	46,054	1	1	1	1.0	1.0	1.0
		for k=4:	67,997		for k=4:	1		for k=4:	1.0
ul	89,776	92,532	92,532	2	2	2	1.0	1.0	1.0
tp	74,367	92,550	92,550	2	2	2	1.0	1.0	1.0
diff	113,836	109,345	109,345	5	4	4	1.1	1.0	1.0
lex	145,643	109,717	109,717	4	3	3	1.1	1.0	1.0
learn	103,776	142,761	142,761	2	2	2	1.0	1.0	1.0
loader	61,221	187,337	508,924	3	11	21	1.1	1.6	2.4
ed	201,331	201,284	201,284	4	4	4	1.1	1.1	1.1
tbl	425,969	401,167	401,938	4	6	3	1.0	1.0	1.0
		for k=4:	401,938		for k=4:	3		for k=4:	1.0
football	324,042	414,222	414,232	2	2	12	1.0	1.0	1.0
		for k=4:	414,232		for k=4:	2		for k=4:	1.0
simulator	273,519	543,305	922,463	4	10	19	1.0	1.0	1.1
assembler	900,372	2,385,797	6,625,431	34	34	66	1.0	1.9	2.7
make	12,624,949	-	-	31	-	-	3.6	-	-

$$\text{Max-assumed} \equiv max_{(\text{all possible aliases, } \mathcal{PA})} \left| \left\{ \mathcal{AA} \;\middle|\; \begin{array}{c} (\exists\ node)\ node \in \text{ICFG and} \\ may\text{-}hold([(node, \mathcal{AA}), \mathcal{PA}]) \\ \text{is } true \end{array} \right\} \right|$$

$$\text{Avg-assumed} \equiv avg_{(\text{all possible aliases, } \mathcal{PA})} \left| \left\{ \mathcal{AA} \;\middle|\; \begin{array}{c} (\exists\ node)\ node \in \text{ICFG and} \\ may\text{-}hold([(node, \mathcal{AA}), \mathcal{PA}]) \\ \text{is } true \end{array} \right\} \right|$$

Table 6.9: Number of Assumptions per Alias

Program	*may-hold*	CPU Time	*may-hold*/sec
allroots	955	(.78s) 1s	1,224
fixoutput	1,856	(1.48s) 1s	1,254
poker	2,096	14s	150
lex315	5,852	(3.88s) 4s	1,508
compress	7,346	(6.20s) 6s	1,185
diffh	8,842	(6.22s) 6s	1,422
pokerd	26,547	20s	1,327
ul	92,532	60s	1,542
tp	92,550	60s	1,543
diff	109,345	101s	1,083
lex	109,717	95s	1,155
learn	142,761	123s	1,170
loader	187,337	195s	961
ed	201,284	132s	1,525
tbl	401,167	277s	1,448
football	414,222	241s	1,719
simulator	543,305	566s	977
assembler	2,385,797	3,299s	723
make ($k = 1$)	12,624,949	11,035s	1,144

Table 6.10: Timings for our Algorithm ($k = 2$)

examine the empirical behavior of our algorithm. In Table 6.10, we present the timings of our algorithm on the 19 C programs. The table is sorted by size of *may-hold* and lists "*may-hold*/sec". If our algorithm is indeed linear in the size of *may-hold*, *may-hold*/sec should be independent of the size of *may-hold*. Table 6.10 seems to indicate that is is true. Another indication that the algorithm may indeed be linear in the size of *may-hold* is that in Table 6.11, *may-hold*/sec does not vary significantly as k changes for a given program, even when the size of *may-hold* changes significantly. Finally, we list *may-hold* and *may-hold*/sec sorted by *may-hold*/sec in Table 6.12. This seems to indicate that the properties of the program are more important then the size of *may-hold* in determining *may-hold*/sec.

Program	*may-hold*			CPU Time			*may-hold*/sec		
	k=1	k=2	k=3	k=1	k=2	k=3	k=1	k=2	k=3
allroots	955	955	955	1s	1s	1s	1,224	1,224	1,224
fixoutput	1,840	1,856	1,856	2s	1s	1s	1,219	1,254	1,254
poker	904	2,096	4,496	5s	14s	38s	171	150	118
lex315	5,836	5,852	5,852	4s	4s	4s	1,512	1,508	1,477
compress	7,799	7,346	7,346	6s	6s	6s	1,272	1,185	1,193
diffh	7,580	8,842	8,842	5s	6s	6s	1,399	1,422	1,422
pokerd	14,448	26,547	46,054	10s	20s	39s	1,445	1,327	1,181
		for k=4:	67,997		for k=4:	62s		for k=4:	1,097
ul	89,776	92,532	92,532	59s	60s	60s	1,522	1,542	1,542
tp	74,367	92,550	92,550	49s	60s	61s	1,518	1,543	1,517
diff	113,836	109,345	109,345	109s	101s	102s	1,044	1,083	1,072
lex	145,643	109,717	109,717	120s	95s	96s	1,214	1,155	1,143
learn	103,776	142,761	142,761	94s	122s	126s	1,104	1,170	1,133
loader	61,221	187,337	508,924	61s	195s	625s	1,004	961	814
ed	201,331	201,284	201,284	153s	132s	130s	1,316	1,525	1,548
tbl	425,969	401,167	401,938	299s	277s	273s	1,425	1,448	1,472
		for k=4:	401,938		for k=4:	284s		for k=4:	1,415
football	324,042	414,222	414,232	185s	241s	240s	1,752	1,719	1,726
		for k=4:	414,232		for k=4:	242s		for k=4:	1,712
simulator	273,519	543,305	922,463	278s	566s	984s	984	960	937
assembler	900,372	2,385,797	6,625,431	1,159s	3,299s	7,103s	777	723	933
make	12,624,949	-	-	11,035s	-	-	1,144	-	-

Table 6.11: Timings for our Algorithm

Program	*may-hold*	*may-hold*/sec	Program	*may-hold*	*may-hold*/sec
poker ($k = 3$)	4,496	118	allroots ($k = 3$)	955	1,224
poker ($k = 2$)	2,096	150	fixoutput ($k = 2$)	1,856	1,254
poker ($k = 1$)	904	171	fixoutput ($k = 3$)	1,856	1,254
assembler ($k = 2$)	2,385,797	723	compress ($k = 1$)	7,799	1,272
assembler ($k = 1$)	900,372	777	ed ($k = 1$)	201,331	1,316
loader ($k = 3$)	508,924	814	pokerd ($k = 2$)	26,547	1,327
assembler ($k = 3$)	6,625,431	933	diffh ($k = 1$)	7,580	1,399
simulator ($k = 3$)	922,463	937	tbl ($k = 4$)	401,938	1,415
simulator ($k = 2$)	543,305	960	diffh ($k = 2$)	8,842	1,422
loader ($k = 2$)	187,337	961	diffh ($k = 3$)	8,842	1,422
simulator ($k = 1$)	273,519	984	tbl ($k = 1$)	425,969	1,425
loader ($k = 1$)	61,221	1,004	pokerd ($k = 1$)	14,448	1,445
diff ($k = 1$)	113,836	1,044	tbl ($k = 2$)	401,167	1,448
diff ($k = 3$)	109,345	1,072	tbl ($k = 3$)	401,938	1,472
diff ($k = 2$)	109,345	1,083	lex315 ($k = 3$)	5,852	1,477
pokerd ($k = 4$)	67,997	1,097	lex315 ($k = 2$)	5,852	1,508
learn ($k = 1$)	103,776	1,104	lex315 ($k = 1$)	5,836	1,512
learn ($k = 3$)	142,761	1,133	tp ($k = 3$)	92,550	1,517
lex ($k = 3$)	109,717	1,143	tp ($k = 1$)	74,367	1,518
make ($k = 1$)	12,624,949	1,144	ul ($k = 1$)	89,776	1,522
lex ($k = 2$)	109,717	1,155	ed ($k = 2$)	201,284	1,525
learn ($k = 2$)	142,761	1,170	ul ($k = 2$)	92,532	1,542
pokerd ($k = 3$)	46,054	1,181	ul ($k = 3$)	92,532	1,542
compress ($k = 2$)	7,346	1,185	tp ($k = 2$)	92,550	1,543
compress ($k = 3$)	7,346	1,193	ed ($k = 3$)	201,284	1,548
lex ($k = 1$)	145,643	1,214	football ($k = 4$)	414,232	1,712
fixoutput ($k = 1$)	1,840	1,219	football ($k = 2$)	414,222	1,719
allroots ($k = 1$)	955	1,224	football ($k = 3$)	414,232	1,726
allroots ($k = 2$)	955	1,224	football ($k = 1$)	324,042	1,752

Table 6.12: *may-hold*/sec

Chapter 7

Conclusions and Future Work

7.1 Summary of Thesis

This thesis can be divided into three major areas: theoretical investigation of the alias problem, a safe Interprocedural May Alias algorithm, and a theoretical and empirical examination of our algorithm. All three of these are interrelated and each has contributed useful results.

Our classification of the alias problem in the presence of various alias mechanisms yielded an understanding of what makes aliasing difficult (i.e., two or more levels of indirection). More importantly, it lent insight into a good method for tracking interprocedural alias information (i.e., by summarizing the runtime stack with a single assumed alias). It also gave some feel for the types of approximation that any safe algorithm must have. This was useful for understanding other approximation algorithms and made it possible to empirically track the imprecision of our own algorithm. Finally, this research yielded a precise algorithm for an interprocedural flow sensitive data flow problem.

Our development of a safe algorithm for solving for Interprocedural May Alias has been promising. It justified the use of Conditional May Alias as a method for the interprocedural aspects of the alias problem. Conditional May Alias is precise for single level pointers and was developed in the classification portion of this thesis. While it is no longer precise in the presence of general pointers, it is still safe and it does not introduce any imprecision that is not attributable to the existence of an alias that may not exist for intraprocedural reasons.

Finally, the theoretical and empirical investigation of our algorithm showed that for at least one definition of precision, in the worst case no algorithm can be more precise

than our algorithm. It also empirically yielded very encouraging precision results and showed great improvement over the extant technique [Wei80a].

7.2 Future Work

The main thrust of the thesis has been the development of a good algorithm for approximating aliases in the presence of pointers and while the initial results are promising, the algorithm presented here is by no means the last word on the subject. An incomplete list of additional work that still needs to be done follows.

- While the precision results in this thesis are encouraging, the timing results are only initial measurements. We are optimizing our algorithm and its prototype implementation. We intend to obtain additional timings.
- k-limiting the alias solution is sufficient for certain data flow problems and appears to be more than adequate for a family of C programs regardless of the problem at hand. However, we still need to investigate the possibility of using other mechanisms for limiting the representation for dynamic stores.
- We will continue and generalize the research in [PRL91] by investigating the utility of Conditional May Alias in various data flow problems, especially MOD.
- We wish to remove the restrictions we placed on C, especially to allow the presence of function variables.
- We want to extend our precise algorithm for Interprocedural Must Alias in the presence of single-level pointers to a safe algorithm in the presence of general pointers. Also we need to investigate the utility of such information.
- We wish to investigate alias analysis in object-oriented programming paradigms, for example pointers to class members [LS88].

Appendix A

Dictionary of Functions

address_type(*type***)** returns the type of objects which can point to *type*. (Chapter 2.2)

alias_consequences(*alias***)** is the set of all aliases that are implied by *alias*. An algorithm for computing consequences_alias can be found in Figure 5.1. (Chapter 5.1.3)

alias_type(*alias***)** is the type of the object which is referred to by both object names in *alias*. This function correctly accounts for k-limiting. (Chapter 2.4)

amp_object_name(*object_name***)** is &*(object_name)*, if *object_name* does not start with an &, otherwise *error*. (Chapter 2.3)

apply_trans(*object_name*$_1$**,***object_name*$_2$**,***object_name*$_3$**)** : $object_name_1$ and $object_name_3$ must have the same type and $is_prefix(object_name_1, object_name_2)$ must be *true*. The function applies to $object_name_3$ the sequence of dereferences and field accesses necessary to transform $object_name_1$ into $object_name_2$. It returns *true* iff any dereference occurs somewhere in the sequence. Some examples of *apply_trans* can be found in Figure 2.3. (Chapter 2.3)

approximate-when-both-non_visible (call-node, $\langle nv_1,\ nv_2 \rangle$, non_visible-name1-is, non_visible-name2-is, name1-AA, name2-AA) substitutes non_visible-name1-is for non_visible in nv_1, substitutes non_visible-name2-is for non_visible in nv_2, and uses the assumed aliases name1-AA and name2-AA to establish a safe assumed alias condition at procedure entry. (Appendix D.1)

back_bind_true(*call*,*assumed_alias***)** =

$$\begin{cases} \left\{ (\mathcal{AA}, nv) \;\middle|\; \begin{array}{l} (\exists alias)\ \textit{may-hold}([(call, \mathcal{AA}), alias]) \text{ and} \\ (assumed_alias, nv) \in \text{bind}_{call}(alias) \end{array} \right\} & \text{if } \mathcal{AA} \text{ has an object name with non_visible in it} \\ \left\{ \mathcal{AA} \;\middle|\; \begin{array}{l} (\exists alias)\ \textit{may-hold}([(call, \mathcal{AA}), alias]) \\ \text{and } assumed_alias \in \text{bind}_{call}(alias) \end{array} \right\} & \text{otherwise} \end{cases}$$

(Chapter 5.2.3)

BBT : see back_bind_true.

can_deref(*type***)** is *true* iff *type* can be legally dereferenced without casting. (Chapter 2.2)

deref(*object_name***)** removes the & if *object_name* starts with an & and otherwise returns $*(object_name)$. This corresponds to rule 1 of Figure 2.2. (Chapter 2.3)

deref_type(*type***)** If *type* can be dereferenced, **deref_type** returns the type of the objects to which *type* may point. (Chapter 2.2)

field_access(*object_name*,*field_name***)** is *(object_name).field_name*, if *object_name* does not start with an &, otherwise error. This corresponds to rule 2 of Figure 2.2. (Chapter 2.3)

field_type(*type*,*field***)** If *type* is a structure with *field*, **field_type** returns the type of *field*. (Chapter 2.2)

is_field_of(*field*,*type***)** is *true* iff *type* is a structure type and *field* is a legal field of *type*. (Chapter 2.2)

is_k-limited(*object_name***)** is *true* iff object_type(*object_name*) can be dereferenced, *object_name* contains at least k dereferences, and *object_name* does not start with an address operator. (Chapter 2.3)

is_prefix($object_name_1$, $object_name_2$**)** returns *true* iff $object_name_1$ can be transformed into $object_name_2$ by a (possibly empty) sequence of dereferences and field accesses (i.e. applications of rule 1 and rule 2 of Figure 2.2). (Chapter 2.3)

is_prefix_with_deref($object_name_1$, $object_name_2$**)** returns *true* iff $object_name_1$ can be transformed into $object_name_2$ by a sequence of dereferences and field accesses (i.e. applications of rule 1 and rule 2 of Figure 2.2) with AT LEAST ONE dereference. (Chapter 2.3)

is_struct($type$**)** is *true* iff $type$ is a structure. (Chapter 2.2)

$may\text{-}hold([(n_i, \mathcal{AA}), \langle a, b\rangle])$ is *true* iff $\langle a, b\rangle$ holds on some path from $entry(n_i)$, the entry of the procedure containing n_i, to n_i assuming there is a path from entry of main to $entry(n_i)$ on which the assumed alias $\mathcal{AA}$ holds **and there is a path from entry of main to the** $entry(n_i)$ **on which** $\mathcal{AA}$ **holds**. (Chapter 5.1.1)

object_type($object_name$**)** is the type of data that can be stored in objects with $object_name$,[74] if $object_name$ does not start with "&". If, for some $name$, $object_name = \&name$ then it is address_type($name$). (Chapter 2.3)

safely_make_alias($node$, on_1, $\mathcal{AA}_1$, on_2, $\mathcal{AA}_2$**)** When safely_make_alias is invoked we know that $\langle on_1, on_2\rangle$ holds on a path to $node$. However, we also know that two assumptions $\mathcal{AA}_1$ and $\mathcal{AA}_2$ are necessary. We do not allow multiple assumptions, so this routine safely approximates this situation. If both assumptions contain non_visible then this will be the special case of an alias in which both components contain non_visible (as discussed on p. 94). Otherwise both assumptions are individually necessary and either can be safely chosen. If one assumption contains non_visible, then use that one; otherwise use either (i.e., only use the assumption $\emptyset$ if both are $\emptyset$).

simple_object_name($variable$**)** is the object name which is just $variable$, i.e. rule 3 of Figure 2.2. (Chapter 2.3)

---

[74] This is the same as the value of the type attribute for $object_name$ obtained from Figure 2.2.

true_under_assumption($exit$,$\mathcal{AA}$) =

$$\{\langle x, y\rangle \mid \textit{may-hold}([(exit, \mathcal{AA}), \langle x, y\rangle]) \text{ is } true\}$$

(Chapter 5.2.3)

TUA : see true_under_assumption.

Appendix B

Intraprocedural Aliasing in the Presence of Single Level Pointers

There are many inductive proofs based on iteration of a fixed point calculation in this appendix. All of these proofs assume that the values of the function in the i^{th} iteration are computed using the values of the function in the $(i-1)^{th}$ iteration.

B.1 Building the SPPAG

Let $\mathcal{O} = \{*p \mid p \text{ is a pointer in the program}\} \bigcup$

$\{v \mid v \text{ is a non-pointer variable in the program}\}$.

$\mathcal{O}$ is the set of all *object names* in the program which may have aliases.

Let $\mathcal{POSSIBLE\text{-}ALIASES} = (\mathcal{O} \times \mathcal{O})$

Given CFG=($\mathcal{N}$,$\mathcal{E}$,ρ), construct SPPAG=($\mathcal{N}'$,$\mathcal{E}'$,ρ') and $holds : \mathcal{N}' \rightarrow \{true, false\}$.

$\mathcal{N}' = \{[n, \langle a, b\rangle] \mid n \in \mathcal{N} \text{ and } \langle a, b\rangle \in \mathcal{POSSIBLE\text{-}ALIASES}\} \cup \{\rho'\}$

Specification of $\mathcal{E}'$ and $holds$. For each $[n, \langle a, b\rangle]$ in $\mathcal{N}'$:

1. $a = b$ and a is not a dereferenced pointer

   Add $\ll \rho', [n, \langle a, b\rangle] \gg$ to $\mathcal{E}'$

   $holds([n, \langle a, b\rangle]) = holds^{\wedge}([n, \langle a, b\rangle]) = true$

2. $n = \rho$ [$a \neq b$ or a is a dereferenced pointer]

   Add $\ll \rho', [n, \langle a, b \rangle] \gg$ to $\mathcal{E}'$

   $holds([n, \langle a, b \rangle]) = holds^{\wedge}([n, \langle a, b \rangle]) = false$

3. n is not an assignment to a pointer [($a \neq b$ or a is a dereferenced pointer) and $n \neq \rho$]

   $\forall \ll m, n \gg \in \mathcal{E}$, add $\ll [m, \langle a, b \rangle], [n, \langle a, b \rangle] \gg$ to $\mathcal{E}'$

   $holds([n, \langle a, b \rangle]) = \bigvee_{\ll m,n \gg \in \mathcal{E}} \left(holds([m, \langle a, b \rangle]) \right)$

   $holds^{\wedge}([n, \langle a, b \rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle a, b \rangle]) \right)$

4. n is the pointer assignment "$p = q$" [($a \neq b$ or a is a dereferenced pointer) and $n \neq \rho$]

   (a) if $a = b = *p$

   $\forall \ll m, n \gg \in \mathcal{E}$, add $\ll [m, \langle *q, *q \rangle], [n, \langle *p, *p \rangle] \gg$ to $\mathcal{E}'$

   $holds([n, \langle *p, *p \rangle]) = \bigvee_{\ll m,n \gg \in \mathcal{E}} \left(holds([m, \langle *q, *q \rangle]) \right)$

   $holds^{\wedge}([n, \langle *p, *p \rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle *q, *q \rangle]) \right)$

   (b) if $a = *p$ [$b \neq a$]

   $\forall \ll m, n \gg \in \mathcal{E}$, add $\ll [m, \langle *q, b \rangle], [n, \langle *p, b \rangle] \gg$ to $\mathcal{E}'$

   $holds([n, \langle *p, b \rangle]) = \bigvee_{\ll m,n \gg \in \mathcal{E}} \left(holds([m, \langle *q, b \rangle]) \right)$

   $holds^{\wedge}([n, \langle *p, b \rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle *q, b \rangle]) \right)$

   (c) if $a \neq *p$ and $b \neq *p$

   $\forall \ll m, n \gg \in \mathcal{E}$, add $\ll [m, \langle a, b \rangle], [n, \langle a, b \rangle] \gg$ to $\mathcal{E}'$

   $holds([n, \langle a, b \rangle]) = \bigvee_{\ll m,n \gg \in \mathcal{E}} \left(holds([m, \langle a, b \rangle]) \right)$

   $holds^{\wedge}([n, \langle a, b \rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle a, b \rangle]) \right)$

5. n is the pointer assignment "$p = \&v$" [($a \neq b$ or a is a dereferenced pointer) and $n \neq \rho$]

   (a) if $a = b = *p$

   Add $\ll \rho', [n, \langle *p, *p \rangle] \gg$ to $\mathcal{E}'$

   $holds([n, \langle *p, *p \rangle]) = holds^{\wedge}([n, \langle *p, *p \rangle]) = true$

(b) if $a = *p$ $[b \neq a]$

$\forall \ll m, n \gg \in \mathcal{E}$, add $\ll [m, \langle v, b \rangle], [n, \langle *p, b \rangle] \gg$ to $\mathcal{E}'$

$holds([n, \langle *p, b \rangle]) = \bigvee_{\ll m,n \gg \in \mathcal{E}} \left(holds([m, \langle v, b \rangle]) \right)$

$holds^{\wedge}([n, \langle *p, b \rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle v, b \rangle]) \right)$

(c) if $a \neq *p$ and $b \neq *p$

$\forall \ll m, n \gg \in \mathcal{E}$, add $\ll [m, \langle a, b \rangle], [n, \langle a, b \rangle] \gg$ to $\mathcal{E}'$

$holds([n, \langle a, b \rangle]) = \bigvee_{\ll m,n \gg \in \mathcal{E}} \left(holds([m, \langle a, b \rangle]) \right)$

$holds^{\wedge}([n, \langle a, b \rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle a, b \rangle]) \right)$

6. n is the pointer assignment "$p = malloc()$" or "$p = NULL$" [($a \neq b$ or a is a dereferenced pointer) and $n \neq \rho$]

(a) if $a = b = *p$

Add $\ll \rho', [n, \langle *p, *p \rangle] \gg$ to $\mathcal{E}'$

$$holds([n, \langle *p, *p \rangle]) = holds^{\wedge}([n, \langle *p, *p \rangle]) = \begin{cases} true & \text{for "p=malloc()"} \\ false & \text{for "p=NULL"} \end{cases}$$

(b) if $a = *p$ $[b \neq a]$

$\forall \ll m, n \gg \in \mathcal{E}$, add $\ll \rho', [n, \langle *p, b \rangle] \gg$ to $\mathcal{E}'$

$holds([n, \langle *p, b \rangle]) = holds^{\wedge}([n, \langle *p, b \rangle]) = false$

(c) $a \neq *p$ and $b \neq *p$

$\forall \ll m, n \gg \in \mathcal{E}$, add $\ll [m, \langle a, b \rangle], [n, \langle a, b \rangle] \gg$ to $\mathcal{E}'$

$holds([n, \langle a, b \rangle]) = \bigvee_{\ll m,n \gg \in \mathcal{E}} \left(holds([m, \langle a, b \rangle]) \right)$

$holds^{\wedge}([n, \langle a, b \rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle a, b \rangle]) \right)$

B.2 SPPAG based algorithm is polynomial time and precise for May Alias

Let CFG=($\mathcal{N}$,$\mathcal{E}$,ρ).

Let SPPAG=($\mathcal{N}'$,$\mathcal{E}'$,ρ') and $holds : \mathcal{N}' \rightarrow \{true, false\}$ be as specified in Appendix B.1.

Lemma B.2.1 *The Single Procedure Pointer Alias Graph (SPPAG) is polynomial in the size of the CFG.*

Each variable in the program contributes one *object name* which may be involved in aliases; for each pointer p, $*p$ may have aliases and for each non-pointer v, v itself may be aliased. Thus the number of possible aliases pairs in a program is $\mid variables \mid^2$. Since a program can use at most $\mathcal{O}(\mid \mathcal{N} \mid)$ variables, the number of possible aliases is $\mathcal{O}(\mid \mathcal{N} \mid^2)$.

$\mid \mathcal{N}' \mid = \mid \mathcal{N} \mid * \mid$ POSS-ALIASES $\mid$

$\mid \mathcal{E}' \mid \leq \mid \mathcal{E} \mid * \mid$ POSS-ALIASES $\mid + \mid \mathcal{N}' \mid$

□

Lemma B.2.2 *The SPPAG can be built from the CFG in polynomial time.*

The specification of SPPAG in Appendix B.1 lends itself to an algorithm for building the SPPAG. Each node in $\mathcal{N}'$ requires at most $\mathcal{O}(\mathcal{E})$ work.

□

Lemma B.2.3 *Fixed point calculation of holds and holds$^\wedge$ on the SPPAG takes time polynomial in the size of the program.*

Follows from Lemma B.2.1 (p. 146) and the observation that for any node $n' \in \mathcal{N}'$, each of $holds(n')$ and $holds^\wedge(n')$ can change its value at most once.

□

Lemma B.2.4 *For every path in the CFG, $\rho n_1 n_2 ... n_{i-1} n$, if $\langle a, b \rangle$ holds on the path then $holds([n, \langle a, b \rangle]) = true$ after the fixed point calculation.*

Proof by induction on i, the path length.

basis: $i = 0$

> The path is simply ρ. Only reflexive aliases for variables, a, hold. $holds([\rho, \langle a, a \rangle]) = true$ by Rule 1 of Appendix B.1. Since $\ll \rho', [\rho, \langle a, a \rangle] \gg \in \mathcal{E}'$ (also by Rule 1) the fixed point calculation must have set $holds([\rho, \langle a, a \rangle])$ to $true$. Thus Lemma B.2.4 holds.

induction hypothesis: Lemma B.2.4 holds for $j < i$.

induction step: Let $\langle a, b \rangle$ hold on $\rho n_1 n_2 ... n_{i-1} n$.

1. $a = b$ and a is not a dereferenced pointer

   $holds([n, \langle a, a\rangle]) = true$ by definition of $holds$ and $\ll\rho', [n, \langle a, a\rangle]\gg \in \mathcal{E}'$.

3. n is not an assignment to a pointer [$a \neq b$ or a is a dereferenced pointer]

   Since $\langle a, b\rangle$ holds on $\rho n_1 n_2 ... n_{i-1} n$ it must also hold on $\rho n_1 n_2 ... n_{i-1}$. By induction hypothesis, $holds([n_{i-1}, \langle a, b\rangle]) = true$. Since $holds([n, \langle a, b\rangle]) = \bigvee_{\ll m,n\gg \in \mathcal{E}} \big(holds([m, \langle a, b\rangle])\big)$ and $\ll n_{i-1}, n\gg \in \mathcal{E}$, $holds([n, \langle a, b\rangle])$ must also be true.

4. n is a pointer assignment "$p = q$" [$a \neq b$ or a is a dereferenced pointer]

   (a) if $a = b = *p$

   $\langle *q, *q\rangle$ must hold on $\rho n_1 n_2 ... n_{i-1}$. By definition of hold this means that $*q$ is a name of a location. This is true as long as q is not $NULL$. By induction hypothesis $holds([n_{i-1}, \langle *q, *q\rangle]) = true$. $holds([n, \langle *p, *p\rangle]) = \bigvee_{\ll m,n\gg \in \mathcal{E}} \big(holds([m, \langle *q, *q\rangle])\big)$. Thus $holds([n, \langle *p, *p\rangle]) = true$.

   (b) if $a = *p$ [$a \neq b$]

   $\langle *q, b\rangle$ must hold on $\rho n_1 n_2 ... n_{i-1}$. By induction hypothesis,
   $holds([n_{i-1}, \langle *q, b\rangle]) = true$. $holds([n, \langle *p, b\rangle]) =$
   $\bigvee_{\ll m,n\gg \in \mathcal{E}} \big(holds([m, \langle *q, b\rangle])\big)$. Thus $holds([n, \langle *p, b\rangle]) = true$.

   (c) if $a \neq *p$ and $b \neq *p$

   $\langle a, b\rangle$ must hold on $\rho n_1 ... n_{i-1}$. By induction hypothesis, $holds([n_{i-1}, \langle a, b\rangle])$ $= true$. Since $holds([n, \langle a, b\rangle]) = \bigvee_{\ll m,n\gg \in \mathcal{E}} \big(holds([m, \langle a, b\rangle])\big)$, $holds([n, \langle a, b\rangle])$ must also be $true$.

5. n is a pointer assignment "$p = \&v$" [$a \neq b$ or a is a dereferenced pointer]

   (a) if $a = b = *p$

   By definition $holds([n, \langle *p, *p\rangle]) = true$ and $\ll\rho', [n, \langle *p, *p\rangle]\gg$ is in $\mathcal{E}$.

   (b) if $a = *p$ [$a \neq b$]

   $\langle v, b\rangle$ must hold on $\rho n_1 n_2 ... n_{i-1}$. By induction hypothesis,
   $holds([n_{i-1}, \langle v, b\rangle]) = true$. $holds([n, \langle *p, b\rangle]) =$
   $\bigvee_{\ll m,n\gg \in \mathcal{E}} \big(holds([m, \langle v, b\rangle])\big)$. Thus $holds([n, \langle *p, b\rangle]) = true$.

(c) if $a \neq *p$ and $b \neq *p$

$\langle a, b\rangle$ must hold on $\rho n_1 n_2 ... n_{i-1}$. By induction hypothesis,

$holds([n_{i-1}, \langle a, b\rangle]) = true$. Since $holds([n, \langle a, b\rangle]) =$

$\bigvee_{\ll m,n \gg \in \mathcal{E}} \left(holds([m, \langle a, b\rangle])\right)$, $holds([n, \langle a, b\rangle])$ must also be true.

6. n is a pointer assignment "$p = malloc()$" or "$p = NULL$" [$a \neq b$ or a is a dereferenced pointer]

(a) if $a = b = *p$

In this case, n has to be "$p = malloc()$" and by definition of holds $holds([n, \langle *p, *p\rangle]) = true$ and $\ll \rho', [n, \langle *p, *p\rangle] \gg$ is in $\mathcal{E}$.

(c) if $a \neq *p$ and $b \neq *p$

$\langle a, b\rangle$ must hold on $\rho n_1 n_2 ... n_{i-1}$. By induction hypothesis,

$holds([n_{i-1}, \langle a, b\rangle]) = true$. Since $holds([n, \langle a, b\rangle]) =$

$\bigvee_{\ll m,n \gg \in \mathcal{E}} \left(holds([m, \langle a, b\rangle])\right)$, $holds([n, \langle a, b\rangle])$ must also be true.

□

Lemma B.2.5 *If* $holds([n, \langle a, b\rangle]) = true$ *after the fixed point calculation then for some path in the CFG,* $\rho n_1 n_2 ... n_{i-1} n$, $\langle a, b\rangle$ *holds.*

Proof by number of iterations of the fixed point algorithm.

basis: After initialization.

For all $n \in \mathcal{N}'$, $holds(n)$ is initially false by definition (see Figure 4.1).

Lemma B.2.5 holds vacuously.

induction hypothesis:

Lemma B.2.5 holds for iterations $j < i$ of the fixed point algorithm.

induction step: Let $holds([n, \langle a, b\rangle])$ be made *true* in iteration i.

1. $a = b$ and a is not a dereferenced pointer

Lemma trivially holds.

2. $n = \rho$ [$a \neq b$ or a is a dereferenced pointer]

   By definition $holds([n, \langle a, b \rangle])$ can never be made true.

3. n is not an assignment to a pointer [($a \neq b$ or a is not a dereferenced pointer) and $n \neq \rho$]

   $holds([n, \langle a, b \rangle]) = \bigvee_{\ll m,n \gg \in \mathcal{E}} \left(holds([m, \langle a, b \rangle]) \right)$ Thus for some $\ll m, n \gg \in \mathcal{E}$, $holds([m, \langle a, b \rangle])$ must be true at iteration i-1. By the induction hypothesis, there is a path $\rho n_1 n_2 ... m$ on which $\langle a, b \rangle$ holds. Thus $\langle a, b \rangle$ must also hold on the path $\rho n_1 n_2 ... mn$.

4. n is the pointer assignment "$p = q$" [($a \neq b$ or a is not a dereferenced pointer) and $n \neq \rho$]

   (a) if $a = b = *p$

   $holds([n, \langle *p, *p \rangle]) = \bigvee_{\ll m,n \gg \in \mathcal{E}} \left(holds([m, \langle *q, *q \rangle]) \right)$ Thus for some $\ll m, n \gg \in \mathcal{E}$, $holds([m, \langle *q, *q \rangle])$ must be true at iteration i-1. By the induction hypothesis, there is a path $\rho n_1 n_2 ... m$ on which $\langle *q, *q \rangle$ holds. Thus $\langle *p, *p \rangle$ must hold on the path $\rho n_1 n_2 ... mn$.

   (b) if $a = *p$ [$a \neq b$]

   $holds([n, \langle *p, b \rangle]) = \bigvee_{\ll m,n \gg \in \mathcal{E}} \left(holds([m, \langle *q, b \rangle]) \right)$ Thus for some $\ll m, n \gg \in \mathcal{E}$, $holds([m, \langle *q, b \rangle])$ must be true at iteration i-1. By the induction hypothesis, there is a path $\rho n_1 n_2 ... m$ on which $\langle *q, b \rangle$ holds. Thus $\langle *p, b \rangle$ must hold on the path $\rho n_1 n_2 ... mn$.

   (c) if $a \neq *p$ and $b \neq *p$

   $holds([n, \langle a, b \rangle]) = \bigvee_{\ll m,n \gg \in \mathcal{E}} \left(holds([m, \langle a, b \rangle]) \right)$ Thus for some $\ll m, n \gg \in \mathcal{E}$, $holds([m, \langle a, b \rangle])$ must be true at iteration i-1. By the induction hypothesis, there is a path $\rho n_1 n_2 ... m$ on which $\langle a, b \rangle$ holds. Thus $\langle a, b \rangle$ must also hold on the path $\rho n_1 n_2 ... mn$.

5. n is the pointer assignment "$p = \&v$" [($a \neq b$ or a is not a dereferenced pointer) and $n \neq \rho$]

(a) if $a = b = *p$

$\langle *p, *p \rangle$ must hold on all paths to n.

(b) if $a = *p$ $[a \neq b]$

$holds([n, \langle *p, b \rangle]) = \bigvee_{\ll m,n \gg \in \mathcal{E}} \left(holds([m, \langle v, b \rangle]) \right)$ Thus for some $\ll m, n \gg \in \mathcal{E}$, $holds([m, \langle v, b \rangle])$ must be true at iteration i-1. By the induction hypothesis, there is a path $\rho n_1 n_2 ... m$ on which $\langle v, b \rangle$ holds. Thus $\langle *p, b \rangle$ must hold on the path $\rho n_1 n_2 ... mn$.

(c) if $a \neq *p$ and $b \neq *p$

$holds([n, \langle a, b \rangle]) = \bigvee_{\ll m,n \gg \in \mathcal{E}} \left(holds([m, \langle a, b \rangle]) \right)$ Thus for some $\ll m, n \gg \in \mathcal{E}$, $holds([m, \langle a, b \rangle])$ must be true at iteration i-1. By the induction hypothesis, there is a path $\rho n_1 n_2 ... m$ on which $\langle a, b \rangle$ holds. Thus $\langle a, b \rangle$ must also hold on the path $\rho n_1 n_2 ... mn$.

6. n is the pointer assignment "$p = malloc()$" or "$p = NULL$" [($a \neq b$ or a is not a dereferenced pointer) and $n \neq \rho$]

(a) if $a = b = *p$

- n is "$p = malloc()$"

  $\langle *p, *p \rangle$ must hold on all paths to n.

- n is "$p = NULL$"

  By definition it is impossible for $holds([n, \langle *p, *p \rangle])$ to be $true$.

(b) if $a = *p$ $[a \neq b]$

By definition it is impossible for $holds([n, \langle *p, b \rangle])$ to be $true$.

(c) if $a \neq *p$ and $b \neq *p$

$holds([n, \langle a, b \rangle]) = \bigvee_{\ll m,n \gg \in \mathcal{E}} \left(holds([m, \langle a, b \rangle]) \right)$ Thus for some $\ll m, n \gg \in \mathcal{E}$, $holds([m, \langle a, b \rangle])$ must be true at iteration i-1. By the induction hypothesis, there is a path $\rho n_1 n_2 ... m$ on which $\langle a, b \rangle$ holds. Thus $\langle a, b \rangle$ must also hold on the path $\rho n_1 n_2 ... mn$.

□

Theorem 4.2.1 *There exists a polynomial time algorithm for determining precise* **Intraprocedural May Alias** *sets in the presence of single level pointers.*

Lemmas B.2.2 (p. 147) and B.2.3 (p. 147) together imply that the algorithm in Figure 4.1 takes time polynomial in the size of the CFG. Lemmas B.2.4 (p. 147) and B.2.5 (p. 149) imply that it finds the precise Intraprocedural May Alias sets.

□

B.3 SPPAG based algorithm is polynomial time and precise for Must Alias

Lemma B.3.1 *For every path in the CFG, $\rho n_1 n_2 ... n_{i-1} n$, if $\langle a, b \rangle$ does not hold then $holds^{\wedge}([n, \langle a, b \rangle]) = false$ after the fixed point calculation.*

Proof by induction on i, the path length.

basis: $i = 0$

> The path is simply ρ. Only reflexive aliases for non-dereferenced variables hold. $holds^{\wedge}([\rho, \langle a, b \rangle]) = false$ ($a \neq b$ or a is a dereferenced variable) by Rule 2 of $holds^{\wedge}$ (Appendix B.1). Since $\ll \rho', [\rho, \langle a, b \rangle] \gg \in \mathcal{E}'$ the fixed point calculation must have set $holds^{\wedge}([\rho, \langle a, b \rangle])$ to $false$. Thus Lemma B.3.1 holds.

induction hypothesis: Lemma B.3.1 holds for $j < i$.

induction step: Let $\langle a, b \rangle$ be an alias that does not hold on $\rho n_1 n_2 ... n_{i-1} n$.

1. $a = b$ and a is not a dereferenced variable

   This is not possible since reflexive aliases between non-dereferenced variables must always hold.

2. $n = \rho$ [$a \neq b$ or a is a dereferenced variable]

   Handled by the basis.

3. n is not an assignment to a pointer [($a \neq b$ or a is a dereferenced variable) and $n \neq \rho$]

   Since $\langle a, b \rangle$ does not hold on $\rho n_1 n_2 ... n_{i-1} n$, it cannot hold on $\rho n_1 n_2 ... n_{i-1}$. By induction hypothesis, $holds^{\wedge}([n_{i-1}, \langle a, b \rangle]) = false$. Since $holds^{\wedge}([n, \langle a, b \rangle]) =$

$\bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle a, b \rangle]) \right)$ and $\ll n_{i-1}, n \gg \in \mathcal{E}$, $holds^{\wedge}([n, \langle a, b \rangle])$ must also be $false$.

4. n is a pointer assignment "$p = q$" [($a \neq b$ or a is a dereferenced variable) and $n \neq \rho$]

   (a) if $a = b = *p$

   $\langle *q, *q \rangle$ cannot hold on $\rho n_1 n_2 ... n_{i-1}$. By induction hypothesis, $holds^{\wedge}([n_{i-1}, \langle *q, *q \rangle]) = false$. $holds^{\wedge}([n, \langle *p, *p \rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle *q, *q \rangle]) \right)$. Thus $holds^{\wedge}([n, \langle *p, *p \rangle]) = false$.

   (b) if $a = *p$ [$a \neq b$]

   $\langle *q, b \rangle$ cannot hold on $\rho n_1 n_2 ... n_{i-1}$. By induction hypothesis, $holds^{\wedge}([n_{i-1}, \langle *q, b \rangle]) = false$. $holds^{\wedge}([n, \langle *p, b \rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle *q, b \rangle]) \right)$. Thus $holds^{\wedge}([n, \langle *p, b \rangle]) = false$.

   (c) if $a \neq *p$ and $b \neq *p$

   $\langle a, b \rangle$ cannot hold on $\rho n_1 n_2 ... n_{i-1}$. By induction hypothesis, $holds^{\wedge}([n_{i-1}, \langle a, b \rangle]) = false$. $holds^{\wedge}([n, \langle a, b \rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle a, b \rangle]) \right)$. Thus $holds^{\wedge}([n, \langle a, b \rangle])$ must also be $false$.

5. n is a pointer assignment "$p = \&v$" [($a \neq b$ or a is a dereferenced variable) and $n \neq \rho$]

   (a) if $a = b = *p$

   $\langle *p, *p \rangle$ must be aliased on all paths to n thus the theorem vacuously holds.

   (b) if $a = *p$ [$a \neq b$]

   $\langle v, b \rangle$ cannot hold on $\rho n_1 n_2 ... n_{i-1}$. By induction hypothesis, $holds^{\wedge}([n_{i-1}, \langle v, b \rangle]) = false$. $holds^{\wedge}([n, \langle *p, b \rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle v, b \rangle]) \right)$. Thus $holds^{\wedge}([n, \langle *p, b \rangle]) = false$.

   (c) if $a \neq *p$ and $b \neq *p$

   $\langle a, b \rangle$ cannot hold on $\rho n_1 n_2 ... n_{i-1}$. By induction hypothesis, $holds^{\wedge}([n_{i-1}, \langle a, b \rangle]) = false$. $holds^{\wedge}([n, \langle a, b \rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle a, b \rangle]) \right)$. Thus $holds^{\wedge}([n, \langle a, b \rangle])$ must also be $false$.

6. n is a pointer assignment "$p = malloc()$" or "$p = NULL$" [($a \neq b$ or a is a dereferenced variable) and $n \neq \rho$]

   (a) if $a = b = *p$

   n must be "$p = NULL$" in which case $holds^{\wedge}([n, \langle *p, *p \rangle]) = false$ by definition.

   (b) if $a = *p$ [$a \neq b$]

   By definition of $holds^{\wedge}$, $holds^{\wedge}([n, \langle *p, b \rangle]) = false$.

   (c) if $a \neq *p$ and $b \neq *p$

   $\langle a, b \rangle$ cannot hold on $\rho n_1 n_2 ... n_{i-1}$. By induction hypothesis, $holds^{\wedge}([n_{i-1}, \langle a, b \rangle]) = false$. $holds^{\wedge}([n, \langle a, b \rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle a, b \rangle]) \right)$. Thus $holds^{\wedge}([n, \langle a, b \rangle])$ must also be $false$.

□

Lemma B.3.2 *If $holds^{\wedge}([n, \langle a, b \rangle]) = false$ after the fixed point calculation then for some path in the CFG, $\rho n_1 n_2 ... n_{i-1} n$, $\langle a, b \rangle$ does not hold.*

Proof by number of iteration of the fixed point algorithm.

basis: After initialization.

For all $n \in \mathcal{N}'$, $holds^{\wedge}(n)$ is initially true. Lemma B.3.2 holds vacuously.

induction hypothesis:

Lemma B.3.2 holds for iterations $j < i$ of the fixed point algorithm.

induction step: Let $holds^{\wedge}([n, \langle a, b \rangle])$ be made $false$ in iteration i.

1. $a = b$ and a is not a dereferenced variable

   By definition of $holds^{\wedge}$, $holds^{\wedge}([n, \langle a, a \rangle]) = true$. Thus $holds^{\wedge}([n, \langle a, a \rangle])$ cannot be $false$.

2. $n = \rho$ [$a \neq b$ or a is a dereferenced variable]

   Only reflexive aliases between variable names hold at the entry of the procedure. Thus the lemma follows.

3. n is not an assignment to a pointer [($a \neq b$ or a is a dereferenced variable) and $n \neq \rho$]

   $holds^{\wedge}([n, \langle a, b\rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle a, b\rangle])\right)$ Thus for some $\ll m, n \gg \in \mathcal{E}$, $holds^{\wedge}([m, \langle a, b\rangle])$ must be $false$ at iteration i-1. By the induction hypothesis, there is a path $\rho n_1 n_2 ... m$ on which $\langle a, b\rangle$ does not hold. Thus $\langle a, b\rangle$ cannot hold on the path $\rho n_1 n_2 ... mn$.

4. n is the pointer assignment "$p = q$" [($a \neq b$ or a is a dereferenced variable) and $n \neq \rho$]

   (a) if $a = b = *p$

   $holds^{\wedge}([n, \langle *p, *p\rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle *q, *q\rangle])\right)$ Thus for some $\ll m, n \gg \in \mathcal{E}$, $holds^{\wedge}([m, \langle *q, *q\rangle])$ must be $false$ at iteration i-1. By the induction hypothesis, there is a path $\rho n_1 n_2 ... m$ on which $\langle *q, *q\rangle$ does not hold. Thus $\langle *p, *p\rangle$ cannot hold on the path $\rho n_1 n_2 ... mn$.

   (b) if $a = *p$ [$a \neq b$]

   $holds^{\wedge}([n, \langle *p, b\rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle *q, b\rangle])\right)$ Thus for some $\ll m, n \gg \in \mathcal{E}$, $holds^{\wedge}([m, \langle *q, b\rangle])$ must be $false$ at iteration i-1. By the induction hypothesis, there is a path $\rho n_1 n_2 ... m$ on which $\langle *q, b\rangle$ does not hold. Thus $\langle *p, b\rangle$ cannot hold on the path $\rho n_1 n_2 ... mn$.

   (c) if $a \neq *p$ and $b \neq *p$

   $holds^{\wedge}([n, \langle a, b\rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle a, b\rangle])\right)$ Thus for some $\ll m, n \gg \in \mathcal{E}$, $holds^{\wedge}([m, \langle a, b\rangle])$ must be $false$ at iteration i-1. By the induction hypothesis, there is a path $\rho n_1 n_2 ... m$ on which $\langle a, b\rangle$ does not hold. Thus $\langle a, b\rangle$ cannot hold on the path $\rho n_1 n_2 ... mn$.

5. n is the pointer assignment "$p = \&v$" [($a \neq b$ or a is a dereferenced variable) and $n \neq \rho$]

   (a) if $a = b = *p$

   $holds^{\wedge}([n, \langle *p, *p\rangle]) = true$ by definition and cannot possibly be $false$.

(b) if $a = *p$ $[a \neq b]$

$holds^{\wedge}([n, \langle *p, b\rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle v, b\rangle])\right)$ Thus for some $\ll m, n \gg \in \mathcal{E}$, $holds^{\wedge}([m, \langle v, b\rangle])$ must be $false$ at iteration i-1. By the induction hypothesis, there is a path $\rho n_1 n_2 ... m$ on which $\langle v, b\rangle$ does not hold. Thus $\langle *p, b\rangle$ cannot hold on the path $\rho n_1 n_2 ... mn$.

(c) if $a \neq *p$ and $b \neq *p$

$holds^{\wedge}([n, \langle a, b\rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle a, b\rangle])\right)$ Thus for some $\ll m, n \gg \in \mathcal{E}$, $holds^{\wedge}([m, \langle a, b\rangle])$ must be $false$ at iteration i-1. By the induction hypothesis, there is a path $\rho n_1 n_2 ... m$ on which $\langle a, b\rangle$ does not hold. Thus $\langle a, b\rangle$ cannot hold on the path $\rho n_1 n_2 ... mn$.

6. n is the pointer assignment "$p = malloc()$" or "$p = NULL$" [($a \neq b$ or a is a dereferenced variable) and $n \neq \rho$]

(a) if $a = b = *p$

- n is "$p = malloc()$"

  $holds^{\wedge}([n, \langle *p, *p\rangle]) = true$ by definition and can never be $false$.

- n is "$p = NULL$"

  $\langle *p, *p\rangle$ cannot possibly hold at n.

(b) if $a = *p$ $[a \neq b]$

No aliases for $*p$ can possibly hold at n.

(c) if $a \neq *p$ and $b \neq *p$

$holds^{\wedge}([n, \langle a, b\rangle]) = \bigwedge_{\ll m,n \gg \in \mathcal{E}} \left(holds^{\wedge}([m, \langle a, b\rangle])\right)$ Thus for some $\ll m, n \gg \in \mathcal{E}$, $holds^{\wedge}([m, \langle a, b\rangle])$ must be $false$ at iteration i-1. By the induction hypothesis, there is a path $\rho n_1 n_2 ... m$ on which $\langle a, b\rangle$ does not hold. Thus $\langle a, b\rangle$ cannot hold on the path $\rho n_1 n_2 ... mn$.

□

Theorem 4.2.2 *There exists a polynomial time algorithm for determining precise* **Intraprocedural Must Alias** *sets in the presence of single level pointers.*

Lemmas B.2.2 (p. 147) and B.2.3 (p. 147) together imply that the algorithm in Figure

4.2 takes time polynomial in the size of the CFG. Lemmas B.3.1 (p. 152) and B.3.2 (p. 154) imply that it finds the precise Intraprocedural Must Alias sets.

□

Appendix C

Interprocedural Aliasing in the Presence of Single Level Pointers

There are many inductive proofs based on iteration of a fixed point calculation in this appendix. All of these proofs assume that the values of the function in the i^{th} iteration are computed using the values of the function in the $(i-1)^{th}$ iteration.

C.1 Back-Bind

C.1.1 Definition

$back\text{-}bind_{call_P}(alias\text{-}pair)$ specifies which alias holding on $\rho n_1 ... n_{i-1}[call_P]$ forces $alias\text{-}pair$ to hold on $\rho n_1 ... n_{i-1}[call_P][entry_P]$.

Let $back\text{-}bind_n$, n a call node in the ICFG, be defined as:

$back\text{-}bind_n(\emptyset) = \emptyset$

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

$back\text{-}bind'_{call_P}(\langle a, non_visible\rangle, o)$ specifies the alias holding on any path $\rho...[call_P]$ that guarantees a will be aliased to the non-visible object o on $\rho...[call_P][entry_P]$.

$$back\text{-}bind'_n(\langle a, non_visible\rangle, o) = \begin{cases} \langle a,o\rangle & \text{if } a \text{ is } \textbf{visible} \text{ in the called procedure} \\ \langle *a_j,o\rangle & \text{if } a \text{ is the dereferenced formal } *f_j \text{ with corresponding actual } a_j \\ false & \text{otherwise } (a \text{ is a local [non-parameter] object name)} \end{cases}$$

C.1.2 Proof of correctness

Let $called(n)$, n a call node, be the entry node of the procedure called by n.

Lemma C.1.1 *For all call nodes, n:*

- $back\text{-}bind_n(\emptyset) = \emptyset$

  On all paths $\rho n_1 n_2 ... n[called(n)]$, $\emptyset$ *is considered to 'hold'.*

- *For all possible alias pairs,* $\langle a, b \rangle$ ($a \neq non_visible$ *and* $b \neq non_visible$)*:*

  - *If* $back\text{-}bind_n(\langle a, b \rangle) = false$ *then*

    On all paths $\rho n_1 n_2 ... n[called(n)]$, $\langle a, b \rangle$ *does not hold.*

  - *If* $back\text{-}bind_n(\langle a, b \rangle) = \langle c, d \rangle$ *then*

    On all paths $\rho n_1 n_2 ... n[called(n)]$ $\langle a, b \rangle$ *holds iff* $\langle c, d \rangle$ *holds on* $\rho n_1 n_2 ... n$.

- *For all possible alias pairs* $\langle a, non_visible \rangle$ ($a \neq non_visible$)*, and all possible* o *(*o *is an object name not visible in the called procedure):*

  - *If* $back\text{-}bind'_n(\langle a, non_visible \rangle, o) = false$ *then*

    On all paths $\rho n_1 n_2 ... n[called(n)]$, a *can never be aliased to the non-visible object name* o.

  - *If* $back\text{-}bind'_n(\langle a, non_visible \rangle, o) = \langle c, o \rangle$ *then*

    On all paths $\rho n_1 n_2 ... n[called(n)]$, a *is aliased to the non-visible object name* o *iff* $\langle c, o \rangle$ *holds on* $\rho n_1 n_2 ... n$.

Proof:

For all call nodes, n:

1. $back\text{-}bind_n(\emptyset) = \emptyset$

   In conditional may alias $\emptyset$ is really the absence of any assumed aliases.

   Thus it is safe to assume that $\emptyset$ holds on every path.

2. For all possible alias pairs, $\langle a, b \rangle$ ($a \neq non_visible$ and $b \neq non_visible$):

   (a) $back\text{-}bind_n(\langle a, b \rangle) = false$

   This only occurs when either a or b are local (non-parameter) object names. In this case, on all paths $\rho n_1 n_2 ... n[called(n)]$, $\langle a, b \rangle$ does not hold.

   (b) $back\text{-}bind_n(\langle a, b \rangle) = \langle c, d \rangle$

i. a and b are both visible in the called procedure

By definition $\langle c, d\rangle = \langle a, b\rangle$.

Clearly on all paths $\rho n_1 n_2 ... n[called(n)]$, $\langle a, b\rangle$ holds iff $\langle a, b\rangle$ holds on $\rho n_1 n_2 ... n$.

ii. a is visible in the called procedure, and b is the dereferenced formal $*f_j$ with actual a_j

By definition $\langle c, d\rangle = \langle a, *a_j\rangle$.

Clearly on all paths $\rho n_1 n_2 ... n[called(n)]$, $\langle a, *f_j\rangle$ holds iff $\langle a, *a_j\rangle$ holds on $\rho n_1 n_2 ... n$.

iii. a is the dereferenced formal $*f_j$ with actual a_j, and b is the dereferenced formal $*f_k$ with actual a_k

By definition $\langle c, d\rangle = \langle *a_j, *a_k\rangle$.

Clearly on all paths $\rho n_1 n_2 ... n[called(n)]$, $\langle *f_j, *f_k\rangle$ holds iff $\langle *a_j, *a_k\rangle$ holds on $\rho n_1 n_2 ... n$.

3. For all possible alias pairs $\langle a, non_visible\rangle$ ($a \neq non_visible$), and all possible o (o is an object name not visible in the called procedure):

(a) $back\text{-}bind'_n(\langle a, non_visible\rangle, o) = false$

This only occurs when a is a local (non-parameter) object name. Thus on all paths $\rho n_1 n_2 ... n[called(n)]$, a can never be aliased to anything.

(b) $back\text{-}bind'_n(\langle a, non_visible\rangle, o) = \langle c, non_visible\rangle$

i. if a is visible in the called procedure

By definition $\langle c, non_visible\rangle = \langle a, non_visible\rangle$.

On all paths $\rho n_1 n_2 ... n[called(n)]$, a is aliased to the non-visible object name o iff $\langle a, o\rangle$ holds on $\rho n_1 n_2 ... n$.

ii. if a is the dereferenced formal $*f_j$ with actual a_j

By definition $\langle c, non_visible\rangle = \langle *a_j, non_visible\rangle$.

On all paths $\rho n_1 n_2 ... n[called(n)]$, $*f_j$ is aliased to the non-visible object name o iff $\langle *a_j, o\rangle$ holds on $\rho n_1 n_2 ... n$.

□

C.1.3 Computable in constant time

Lemma C.1.2 *Given a call node n and an alias* $\langle a, b\rangle$, $back\text{-}bind_n(\langle a, b\rangle)$ *can be computed in constant time.*
Given a call node n, an alias $\langle c, non_visible\rangle$, *and object name o which is not visible in the called procedure,* $back\text{-}bind'_n(\langle c, non_visible\rangle, o)$ *can be computed in constant time.*

The naive encoding of the definitions of $back\text{-}bind_n(\langle a, b\rangle)$ and $back\text{-}bind'_n(\langle c, non_visible\rangle, o)$ yields constant time algorithms.

□

C.2 Bind

C.2.1 Definition

$bind_{call_P}(alias\text{-}set)$ specifies for all paths $\rho n_1 ... n_i[call_P][entry_P]$ which aliases hold assuming all aliases in *alias-set* hold on $\rho n_1 ... n_i[call_P]$.

Let $\mathcal{O}' = \{*p \mid p \text{ is a pointer in the program}\} \bigcup$
$\{v \mid v \text{ is a non-pointer variable in the program}\}$.
$\mathcal{O}'$ is the set of all object names in the program which may have aliases.

Let $\mathcal{POSSIBLE\text{-}ALIASES}' = (\mathcal{O}' \times \mathcal{O}')$.
$\mathcal{POSSIBLE\text{-}ALIASES}'$ is the set of all possible aliases.

Let $bind_n(A)$, n a call node in the ICFG and $A \subseteq \mathcal{POSSIBLE\text{-}ALIASES}'$, be defined as: $bind_n(A) = bind'_n(\emptyset) \cup \bigcup_{\langle a,b\rangle \in A}(bind'_n(\langle a, b\rangle))$

$$bind'_n(\emptyset) = \left(\left\{ \langle *f_i, *f_j \rangle \;\middle|\; \begin{array}{l} f_i \text{ and } f_j \text{ are pointer formals with actuals} \\ a_i \text{ and } a_j \text{ respectively, and } a_i = a_j \text{ and} \\ \text{neither } a_i \text{ nor } a_j \text{ is a pointer variable}^{75} \end{array} \right\} \cup \left\{ \langle *f_i, *a_i \rangle \;\middle|\; \begin{array}{l} f_i \text{ is a pointer formal with actual } a_i, \\ \text{and } a_i \text{ is visible in the called procedure} \\ \text{and } a_i \text{ is not a pointer variable} \end{array} \right\} \right)$$

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

C.2.2 Proof of correctness

Let *called*(*call_node*) be the entry node of the procedure called by *call_node*.

Lemma C.2.1 *For a call node n and a set of aliases alias-set, $bind_n$(alias-set) is the set of aliases that hold on $\rho n_1 ... n_{i-1} n[called(n)]$ given that alias-set holds on $\rho n_1 ... n_{i-1} n$.*

> Lemma 4.2.1 states that we can consider the effect of each alias in *alias-set* separately and considering the true-on-all-paths case ($\emptyset$). This is precisely what the function $bind_n$ does.

[75] a_i can be $\&v$ for some non-pointer v, but a_i can not be p for a pointer variable p.

The only aliases that hold on paths to $[called(n)]$ even if no aliases hold at n are aliases between dereferenced pointer formals that are passed the same (non-pointer) actual[76] and aliases between dereferenced pointer formals and their dereferenced (non-pointer) actuals[76] (assuming the actual is visible in the called procedure). This is obviously captured by the definition of $bind'_n(\emptyset)$.

The only aliases that hold on paths to $[called(n)]$ given that $\langle a, b\rangle$ hold at n are:

- $\langle a, b\rangle$ if both a and b are visible in the called procedure.
- $\langle a, *f_i\rangle$ if a is visible in the called procedure and $*f_i$ is a pointer formal with actual a_i and $*a_i = b$.
- $\langle *f_i, *f_j\rangle$ if f_i and f_j are pointer formals with corresponding actuals a_i and a_j, and $*a_i = a$ and $*a_j = b$.

These cases are all captured by the definition of $bind'_n(\langle a, b\rangle)$.

□

C.2.3 Computable in constant time

Lemma C.2.2 *Given a call node n and an alias set alias-set, $bind_n$(alias-set) can be computed in time polynomial in $\mathcal{O}(|$alias-set$| * |($number of formals in the called procedure$)^2|)$.*

The naive encoding of the definition of $bind_n(alias\text{-}set)$ satisfies the above time constraints.

□

[76] The actual must be $\&v$ for some variable v. Its dereferenced value is v.

C.3 A Precise Polynomial Time Algorithm for Computing Interprocedural May Alias Sets in the Presence of Single Level Pointers

C.3.1 Terminology

Let $\mathcal{O}$ = $\{*p \mid p$ is a pointer in the program$\}$ $\bigcup$
$\{v \mid v$ is a non-pointer variable in the program$\}$ $\bigcup$
$\{non_visible\}$.

$\mathcal{O}$ is the set of all object names in the program which may have aliases.

"*non_visible*" represents object names which are not visible.

Let $\mathcal{POSSIBLE\text{-}ALIASES} = (\mathcal{O} \times \mathcal{O}) - \{\langle non_visible, non_visible \rangle\}$

and $\mathcal{ASSUMED} = \mathcal{POSSIBLE\text{-}ALIASES} \cup \{\emptyset\}$

Given an ICFG for a program:

Let $exit(n)$, n a return node in the ICFG, be the exit node corresponding to n.
Let $call(n)$, n a return node in the ICFG, be the call node corresponding to n.
Let $entry(n)$, n an ICFG node, be the entry node of the procedure containing n.

Let $back\text{-}bind_{call}$, *call* a call node in the ICFG, be defined as in Appendix C.1.
$back\text{-}bind_{call}(alias\text{-}pair)$ specifies which alias holding at the *call* site forces *alias-pair* to hold at the entry of the called procedure.

$n_1 n_2 ... n_i \circ m_1 m_2 ... m_j = n_1 n_2 ... n_i m_1 m_2 ... m_j$
$n_1 n_2 ... n_i \bullet m_1 m_2 ... m_j = n_1 n_2 ... n_i m_2 ... m_j$ [n_i must equal m_1]

C.3.2 Building the Pointer Alias Graph (PAG)

Given ICFG=($\mathcal{N}$,$\mathcal{E}$,ρ), construct PAG=($\mathcal{N}'$,$\mathcal{E}'$,ρ') and $holds : \mathcal{N}' \rightarrow \{true, false\}$.
$holds([(node, \mathcal{AA}), \mathcal{PA}]) = true$ iff

$\exists$ a realizable path $P = [entry(node)]n_1 n_2 ... n_i[node]$ in the ICFG, where $n_1 n_2 ... n_i$ contains the same number of calls as returns, such that for every realizable path $P' = \rho m_1 ... m_i[entry(node)]$, $\mathcal{AA}$ holding[77] on P' means $\mathcal{PA}$ must hold on $P' \bullet P$ (*i.e.*, $\rho m_1 ... m_i[entry(node)]n_1 n_2 ... n_i[node]$).

In order to simply if the algorithm description and proof we expand the definition of *holds*. For all $node \in$ ICFG, all $\mathcal{AA} \in \mathcal{ASSUMED}$, $\underline{holds([(node, \mathcal{AA}), false]) = false.}$

$$\mathcal{N}' = \left\{ [(node, \mathcal{AA}), \mathcal{PA}] \;\middle|\; \begin{array}{l} node \in \mathcal{N}, \mathcal{AA} \in \mathcal{ASSUMED}, \\ \text{and } \mathcal{PA} \in \mathcal{POSSIBLE\text{-}ALIASES} \end{array} \right\} \cup \{\rho'\}$$

Specification of $\mathcal{E}'$ and *holds*.
For all $node \in \mathcal{N}$, for all $\langle a, b \rangle \in \mathcal{POSSIBLE\text{-}ALIASES}$, for all $\mathcal{AA} \in \mathcal{ASSUMED}$:

1. $a = b$ and a is not a dereferenced pointer

   (Reflexive aliases for variables always hold)

   Add $\ll \rho', [(node, \mathcal{AA}), \langle a, b \rangle] \gg$ to $\mathcal{E}'$

   $holds([(node, \mathcal{AA}), \langle a, b \rangle]) = true$

2. $a \neq b$ or a is a dereferenced pointer

   (a) *node* is an entry node

   $$\left(\begin{array}{l} \textit{An alias conditionally holds at an entry node iff we assume it does} \\ \textit{or it is a reflexive alias for a variable.} \end{array} \right)$$

   Add $\ll \rho', [(node, \mathcal{AA}), \langle a, b \rangle] \gg$ to $\mathcal{E}'$

   $$holds([(node, \mathcal{AA}), \langle a, b \rangle]) = \begin{cases} true & \text{if } \mathcal{AA} = \langle a, b \rangle \\ false & \text{otherwise} \end{cases}$$

   (b) *node* is an exit node

   (On any path, $\langle a, b \rangle$ holds at an exit node iff it held before the exit node)

   For every $\ll m, node \gg \in \mathcal{E}$, add $\ll [(m, \mathcal{AA}), \langle a, b \rangle], [(node, \mathcal{AA}), \langle a, b \rangle] \gg$ to $\mathcal{E}'$

   $$holds([(node, \mathcal{AA}), \langle a, b \rangle]) = \bigvee_{\ll m, node \gg \in \mathcal{E}} \left(holds([(m, \mathcal{AA}), \langle a, b \rangle]) \right)$$

[77] We consider the assumed alias $\emptyset$ to hold on all paths

(c) *node* is a call node

(On any path, $\langle a,b\rangle$ holds at a call node iff it held before the call node)

For every $\ll m, node \gg \in \mathcal{E}$, add $\ll [(m,\mathcal{AA}),\langle a,b\rangle],[(node,\mathcal{AA}),\langle a,b\rangle] \gg$ to $\mathcal{E}'$

$\boldsymbol{holds}([(node,\mathcal{AA}),\langle a,b\rangle]) = \bigvee_{\ll m,node\gg \in \mathcal{E}} \left(\boldsymbol{holds}([(m,\mathcal{AA}),\langle a,b\rangle])\right)$

(d) *node* is a return node

i. a and b are both not visible in the called procedure

$$\left(\begin{array}{l} \textit{The procedure call cannot affect this alias. The alias} \\ \textit{holds after the call iff it held before the call.} \end{array}\right)$$

Add $\ll [(call(node),\mathcal{AA}),\langle a,b\rangle],[(node,\mathcal{AA}),\langle a,b\rangle] \gg$ to $\mathcal{E}'$

$\boldsymbol{holds}([(node,\mathcal{AA}),\langle a,b\rangle]) = \boldsymbol{holds}([(call(node),\mathcal{AA}),\langle a,b\rangle])$

ii. a is visible in the called procedure, but b is not

$$\left(\begin{array}{l} \langle a,b\rangle \textit{ holds on a path to node iff } \langle a, non_visible\rangle \\ \textit{holds at } exit(node) \textit{ and ``non_visible'' is } b \end{array}\right)$$

For every $\langle c, non_visible\rangle \in \mathcal{ASSUMED}$:

if $back\text{-}bind'_{call(node)}(\langle c, non_visible\rangle, b) \neq false$ then add

$\ll [(exit(node),\langle c, non_visible\rangle),\langle a, non_visible\rangle],[(node,\mathcal{AA}),\langle a,b\rangle] \gg$ and

$\ll [(call(node),\mathcal{AA}), back\text{-}bind'_{call(node)}(\langle c, non_visible\rangle, b)],[(node,\mathcal{AA}),\langle a,b\rangle] \gg$

to $\mathcal{E}'$

$$\boldsymbol{holds}([(node,\mathcal{AA}),\langle a,b\rangle]) = \bigvee_{\langle c,non_visible\rangle \in \mathcal{ASSUMED}} \left(\begin{array}{l} holds([(exit(node),\langle c, non_visible\rangle),\langle a, non_visible\rangle]) \bigwedge \\ holds([(call(node),\mathcal{AA}), back\text{-}bind'_{call(node)}(\langle c, non_visible\rangle, b)]) \end{array}\right)$$

iii. a and b are visible in the called procedure

($\langle a,b\rangle$ holds on a path to node iff $\langle a,b\rangle$ holds at $exit(node)$)

Add $\ll [(exit(node),\emptyset),\langle a,b\rangle],[(node,\mathcal{AA}),\langle a,b\rangle] \gg$ to $\mathcal{E}'$

For every $\langle c,d\rangle \in \mathcal{ASSUMED}$ (c and $d \neq$ "*non_visible*"):

if $back\text{-}bind_{call(node)}(\langle c,d\rangle) \neq false$ then add

$\ll [(exit(node),\langle c,d\rangle),\langle a,b\rangle],[(node,\mathcal{AA}),\langle a,b\rangle] \gg$ and

$\ll [(call(node),\mathcal{AA}), back\text{-}bind_{call(node)}(\langle c,d\rangle)],[(node,\mathcal{AA}),\langle a,b\rangle] \gg$ to $\mathcal{E}'$

$\boldsymbol{holds}([(node,\mathcal{AA}),\langle a,b\rangle]) = \boldsymbol{holds}([(exit(node),\emptyset),\langle a,b\rangle]) \vee$

$$\bigvee_{\langle c,d\rangle \in \mathcal{ASSUMED}} \left(\begin{array}{l} holds([(exit(node), \langle c,d\rangle), \langle a,b\rangle]) \bigwedge \\ holds([(call(node), \mathcal{AA}), back\text{-}bind_{call(node)}(\langle c,d\rangle)]) \end{array} \right)$$

(e) otherwise (*node* is a statement node)

i. *node* is not an assignment to a pointer

(On any path, $\langle a,b\rangle$ holds at node iff it held immediately before node)

For all $\ll m, node \gg \in \mathcal{E}$, add $\ll [(m,\mathcal{AA}), \langle a,b\rangle], [(node,\mathcal{AA}), \langle a,b\rangle] \gg$ to $\mathcal{E}'$

$holds([(node,\mathcal{AA}), \langle a,b\rangle]) = \bigvee_{\ll m,node \gg \in \mathcal{E}} \left(holds([(m,\mathcal{AA}), \langle a,b\rangle]) \right)$

ii. *node* is the pointer assignment "$p = q$"

A. if $a = b = *p$

$$\left(\begin{array}{l} \langle *p, *p\rangle \text{ holds on a path to node iff } \langle *q, *q\rangle \\ \text{held immediately before node} \end{array} \right)$$

For all $\ll m, node \gg \in \mathcal{E}$ add $\ll [(m,\mathcal{AA}), \langle *q, *q\rangle], [(node,\mathcal{AA}), \langle *p, *p\rangle] \gg$ to $\mathcal{E}'$

$holds([(node,\mathcal{AA}), \langle *p, *p\rangle]) = \bigvee_{\ll m,node \gg \in \mathcal{E}} \left(holds([(m,\mathcal{AA}), \langle *q, *q\rangle]) \right)$

B. if $a = *p$ $[a \neq b]$

$$\left(\begin{array}{l} \langle *p, b\rangle \text{ holds on a path to node iff } \langle *q, b\rangle \\ \text{held immediately before node} \end{array} \right)$$

For all $\ll m, node \gg \in \mathcal{E}$ add $\ll [(m,\mathcal{AA}), \langle *q, b\rangle], [(node,\mathcal{AA}), \langle *p, b\rangle] \gg$ to $\mathcal{E}'$

$holds([(node,\mathcal{AA}), \langle *p, b\rangle]) = \bigvee_{\ll m,node \gg \in \mathcal{E}} \left(holds([(m,\mathcal{AA}), \langle *q, b\rangle]) \right)$

C. if $(a \neq *p)$ and $(b \neq *p)$

($\langle a,b\rangle$ holds on a path to node iff $\langle a,b\rangle$ held immediately before node)

For all $\ll m, node \gg \in \mathcal{E}$ add $\ll [(m,\mathcal{AA}), \langle a,b\rangle], [(node,\mathcal{AA}), \langle a,b\rangle] \gg$ to $\mathcal{E}'$

$holds([(node,\mathcal{AA}), \langle a,b\rangle]) = \bigvee_{\ll m,node \gg \in \mathcal{E}} \left(holds([(m,\mathcal{AA}), \langle a,b\rangle]) \right)$

iii. *node* is the pointer assignment "$p = \&v$"

A. if $a = b = *p$

($\langle *p, *p\rangle$ holds on all paths to node)

Add $\ll \rho', [(node,\mathcal{AA}), \langle *p, *p\rangle] \gg$ to $\mathcal{E}'$

$holds([(node,\mathcal{AA}), \langle *p, *p\rangle]) = true$

B. if $a = *p$ $[a \neq b]$

$$\left(\begin{array}{l} \langle *p, b\rangle \textit{ holds on a path to node iff } \langle v, b\rangle \\ \\ \textit{held immediately before node} \end{array} \right)$$

For all $\ll m, node \gg \in \mathcal{E}$ add $\ll [(m, \mathcal{AA}), \langle v, b\rangle], [(node, \mathcal{AA}), \langle *p, b\rangle] \gg$ to $\mathcal{E}'$

$$\textit{holds}([(node, \mathcal{AA}), \langle *p, b\rangle]) = \bigvee_{\ll m,node \gg \in \mathcal{E}} \left(\textit{holds}([(m, \mathcal{AA}), \langle v, b\rangle]) \right)$$

C. if $(a \neq *p)$ and $(b \neq *p)$

($\langle a, b\rangle$ holds on a path to node iff $\langle a, b\rangle$ held immediately before node)

For all $\ll m, node \gg \in \mathcal{E}$ add $\ll [(m, \mathcal{AA}), \langle a, b\rangle], [(node, \mathcal{AA}), \langle a, b\rangle] \gg$ to $\mathcal{E}'$

$$\textit{holds}([(node, \mathcal{AA}), \langle a, b\rangle]) = \bigvee_{\ll m,node \gg \in \mathcal{E}} \left(\textit{holds}([(m, \mathcal{AA}), \langle a, b\rangle]) \right)$$

iv. *node* is the pointer assignment "$p = malloc()$" or "$p = NULL$"

A. if $a = b = *p$

Add $\ll \rho', [(node, \mathcal{AA}), \langle *p, *p\rangle] \gg$ to $\mathcal{E}'$

$$\textit{holds}([(node, \mathcal{AA}), \langle *p, *p\rangle]) = \left\{ \begin{array}{ll} true & node \text{ is "} p = malloc()\text{"} \\ false & node \text{ is "} p = NULL\text{"} \end{array} \right.$$

B. if $a = *p$ $[a \neq b]$

($*p$ can never be aliased to anything on any path to node)

Add $\ll \rho', [(node, \mathcal{AA}), \langle *p, b\rangle] \gg$ to $\mathcal{E}'$

$\textit{holds}([(node, \mathcal{AA}), \langle *p, b\rangle]) = false$

C. if $(a \neq *p)$ and $(b \neq *p)$

($\langle a, b\rangle$ holds on a path to node iff $\langle a, b\rangle$ held immediately before node)

For all $\ll m, node \gg \in \mathcal{E}$ add $\ll [(m, \mathcal{AA}), \langle a, b\rangle], [(node, \mathcal{AA}), \langle a, b\rangle] \gg$ to $\mathcal{E}'$

$$\textit{holds}([(node, \mathcal{AA}), \langle a, b\rangle]) = \bigvee_{\ll m,node \gg \in \mathcal{E}} \left(\textit{holds}([(m, \mathcal{AA}), \langle a, b\rangle]) \right)$$

C.3.3 Algorithm for Computing Precise May Alias Sets in the Presence of Single Level Pointers

Let $\mathcal{O}' = \{*p \mid p \text{ is a pointer in the program}\} \bigcup$

$\{v \mid v \text{ is a non-pointer variable in the program}\}$.

$\mathcal{O}'$ is the set of all object names in the program which may have aliases.

Let $\mathcal{POSSIBLE\text{-}ALIASES}' = (\mathcal{O}' \times \mathcal{O}')$.

$\mathcal{POSSIBLE}$-$\mathcal{ALIASES}'$ is the set of all possible aliases.

Define the lattice [Hec77] $\mathcal{L}' =$

$$(powerset(\mathcal{POSSIBLE}\text{-}\mathcal{ALIASES}'), \supseteq (\sqsubseteq), \cup(\sqcap), \mathcal{POSSIBLE}\text{-}\mathcal{ALIASES}' \ (\bot), \emptyset \ (\top))$$

Let $bind_n(\mathcal{AA})$, n a call node in the ICFG and $\mathcal{AA} \in \mathcal{POSSIBLE}$-$\mathcal{ALIASES}'$, be defined as in Appendix C.2.

Let $entry(n)$, n a node in the ICFG, be the entry node of the procedure containing n.

For every node n in the ICFG define $may\text{-}alias(n)$ as:

1. $may\text{-}alias(\rho) = \emptyset$
2. if n is an entry node then $may\text{-}alias(n) = \bigcup_{\ll m,n \gg \in \mathcal{E}} \left(bind_m(may\text{-}alias(m)) \right)$
3. otherwise, $may\text{-}alias(n) =$
$$\left\{ \langle a, b \rangle \;\middle|\; \begin{array}{l} \text{neither } a \text{ nor } b \text{ is } non_visible \text{ and} \\ [holds([(n, \emptyset), \langle a, b \rangle]) = true] \bigvee \\ [\left(\exists \mathcal{AA} \in may\text{-}alias(entry(n))\right) holds([(n, \mathcal{AA}), \langle a, b \rangle]) = true] \end{array} \right\}$$

Algorithm:

1. For all n in the ICFG, all $\langle a, b \rangle \in \mathcal{POSSIBLE}$-$\mathcal{ALIASES}$, all $\mathcal{AA} \in \mathcal{ASSUMED}$, initialize $holds([(n, \mathcal{AA}), \langle a, b \rangle])$ to $false$.
2. Calculate the fixed point of $holds$.
3. For all n in the ICFG, initialize $may\text{-}alias(n)$ to $\emptyset$.
4. Calculate the fixed point of $may\text{-}alias$.

C.3.4 Proof of Lemma 4.2.1

Lemma 4.2.1 *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 for the procedure containing n_i and the number of calls on the path $n_1 n_2 ... n_{i-1}$ equals the number of returns.),*
- *and for all possible alias pairs $\langle a, b \rangle$;*

  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 $\langle a, b \rangle$ holds at n_i*

  then

  > *either assuming no aliases at n_1 and executing path P forces $\langle a, b \rangle$ to hold at n_i*
  >
  > *or*
  >
  > *$\exists k (1 \leq k \leq m)$ such that when assuming* **only** *the alias pair A_k at n_1, executing the path P forces $\langle a, b \rangle$ to hold at n_i.*

Proof by induction on i, the path length.

basis: $i = 1$

> n_1 must be an entry node. Only aliases that are assumed at n_1 and reflexive aliases between variables (but not dereferenced pointers) hold on the path n_1 given an assumption $\mathcal{A}$. In the first case, the alias that holds must be in $\mathcal{A}$ and satisfies the Lemma. In the second case no alias assumption is necessary.

induction hypothesis: Lemma 4.2.1 holds for $j < i$.

induction step:

1. n_i is an entry node

   i must be 1 and the basis applies.

2. n_i is a call or exit node

   For all $\langle a, b\rangle$, $\langle a, b\rangle$ holds on P assuming $\mathcal{A}$ at n_1 only if $\langle a, b\rangle$ holds on $n_1 ... n_{i-1}$ assuming $\mathcal{A}$ at n_1. By induction hypothesis either no assumption or only $A_k \in \mathcal{A}$ forces $\langle a, b\rangle$ to hold on $n_1 ... n_{i-1}$ and thus the same is true for P.

3. n_i is a return node

   (a) Both a and b are not visible in the called procedure.

   Let n_j $(1 < j < i)$ be the call that corresponds to the return n_i. For all $\langle a, b\rangle$, $\langle a, b\rangle$ holds on P assuming $\mathcal{A}$ at n_1 only if $\langle a, b\rangle$ holds on $n_1 ... n_j$ assuming $\mathcal{A}$ at n_1. By induction hypothesis either no assumption or only $A_k \in \mathcal{A}$ forces $\langle a, b\rangle$ to hold on $n_1 ... n_j$ and thus the same is true for P.

   (b) a is visible in the called procedure, but b is not.

   Let n_j $(1 < j < i)$ be the call that corresponds to the return n_i. Let $\mathcal{A}'$ be the set of alias pairs that hold on $n_1 ... n_{j+1}$ (an entry node) assuming $\mathcal{A}$ at n_1. For $\langle a, b\rangle$ to hold on P assuming $\mathcal{A}$ at n_1, $\langle a, \mathit{non_visible}\rangle$ (where $\mathit{non_visible}$ is b) must hold on $n_{j+1} ... n_{i-1}$ assuming $\mathcal{A}'$ at n_{j+1}. $n_{j+1} ... n_{i-1}$ is shorter than P and meets the requirements of the Lemma thus by induction hypothesis, there is a $A'_{k'} \in \mathcal{A}'$[78] such that assuming only $A'_{k'}$ at n_{j+1} is sufficient. By Lemma C.1.1 (p. 160), $\mathit{back\text{-}bind}'(A'_{k'}, b)$ must hold on $n_1 ... n_j$ while assuming $\mathcal{A}$ at n_1 in order for $\langle a, b\rangle$ to hold on P under the same assumptions. The Lemma follows by induction because $n_1 ... n_j$ is shorter than P and meets the requirement of the Lemma.

   (c) Both a and b are visible in the called procedure.

   Let n_j $(1 < j < i)$ be the call that corresponds to the return n_i. Let $\mathcal{A}'$ be the set of alias pairs that hold on $n_1 ... n_{j+1}$ (an entry node) assuming $\mathcal{A}$ at n_1. For $\langle a, b\rangle$ to hold on P assuming $\mathcal{A}$ at n_1, $\langle a, b\rangle$ must hold on

[78] The no assumption case is not possible because $\mathit{non_visible}$ is involved.

$n_{j+1}...n_{i-1}$ assuming $\mathcal{A}'$ at n_{j+1}. $n_{j+1}...n_{i-1}$ is shorter than P and meets the requirements of the Lemma; thus, by induction hypothesis, either:

i. No assumption is necessary at n_{j+1}.

In this case no assumption is necessary at n_1 for $\langle a, b\rangle$ to hold on P.

ii. There is a $A'_{k'} \in \mathcal{A}'$ such that assuming only $A'_{k'}$ at n_{j+1} is sufficient.

By Lemma C.1.1 (p. 160) *back-bind*$(A'_{k'})$ must hold on $n_1...n_j$ assuming $\mathcal{A}$ at n_1 for $\langle a, b\rangle$ to hold on P under the same assumptions. The Lemma follows by induction because $n_1...n_j$ is shorter than P and meets the requirement of the Lemma.

4. otherwise (n_i is a statement node)

(a) n_i is not an assignment to a pointer

For all $\langle a, b\rangle$, $\langle a, b\rangle$ holds on P assuming $\mathcal{A}$ at n_1 only if $\langle a, b\rangle$ holds on $n_1...n_{i-1}$ assuming $\mathcal{A}$ at n_1. By induction hypothesis either no assumption or only $A_k \in \mathcal{A}$ forces $\langle a, b\rangle$ to hold on $n_1...n_{i-1}$ and thus the same is true for P.

(b) n_i is the pointer assignment "$p = q$"

For all $\langle a, b\rangle$,

i. if $a = b = *p$

$\langle *p, *p\rangle$ holds on P assuming $\mathcal{A}$ at n_1 only if $\langle *q, *q\rangle$ holds on $n_1...n_{i-1}$ assuming $\mathcal{A}$ at n_1. By induction hypothesis either no assumption or only $A_k \in \mathcal{A}$ forces $\langle *q, *q\rangle$ to hold on $n_1...n_{i-1}$ and thus the same is true for P.

ii. $a = *p$ $[a \neq b]$

$\langle *p, b\rangle$ holds on P assuming $\mathcal{A}$ at n_1 only if $\langle *q, b\rangle$ holds on $n_1...n_{i-1}$ assuming $\mathcal{A}$ at n_1. By induction hypothesis either no assumption or only $A_k \in \mathcal{A}$ forces $\langle *q, b\rangle$ to hold on $n_1...n_{i-1}$ and thus the same is true for P.

iii. otherwise

$\langle a, b\rangle$ holds on P assuming $\mathcal{A}$ at n_1 only if $\langle a, b\rangle$ holds on $n_1 ... n_{i-1}$ assuming $\mathcal{A}$ at n_1. By induction hypothesis either no assumption or only $A_k \in \mathcal{A}$ forces $\langle a, b\rangle$ to hold on $n_1 ... n_{i-1}$ and thus the same is true for P.

(c) n_i is the pointer assignment "$p = \&v$"

For all $\langle a, b\rangle$,

i. if $a = b = *p$

$\langle *p, *p\rangle$ holds on all paths regardless of alias assumptions. Thus $\langle *p, *p\rangle$ holds on P with no alias assumptions at n_1.

ii. $a = *p$ $[a \neq b]$

$\langle *p, b\rangle$ holds on P assuming $\mathcal{A}$ at n_1 only if $\langle v, b\rangle$ holds on $n_1 ... n_{i-1}$ assuming $\mathcal{A}$ at n_1. By induction hypothesis either no assumption or only $A_k \in \mathcal{A}$ forces $\langle v, b\rangle$ to hold on $n_1 ... n_{i-1}$ and thus the same is true for P.

iii. otherwise

$\langle a, b\rangle$ holds on P assuming $\mathcal{A}$ at n_1 only if $\langle a, b\rangle$ holds on $n_1 ... n_{i-1}$ assuming $\mathcal{A}$ at n_1. By induction hypothesis either no assumption or only $A_k \in \mathcal{A}$ forces $\langle a, b\rangle$ to hold on $n_1 ... n_{i-1}$ and thus the same is true for P.

(d) n_i is the pointer assignment "$p = malloc()$" or "$p = NULL$"

i. if $a = b = *p$

n_i must be "$p = malloc()$" and $\langle *p, *p\rangle$ holds on all paths regardless of alias assumptions. Thus $\langle *p, *p\rangle$ holds on P with no alias assumptions at n_1.

ii. $a = *p$ $[a \neq b]$

Not possible.

iii. otherwise

$\langle a, b\rangle$ holds on P assuming $\mathcal{A}$ at n_1 only if $\langle a, b\rangle$ holds on $n_1 ... n_{i-1}$ assuming $\mathcal{A}$ at n_1. By induction hypothesis either no assumption or only

$A_k \in \mathcal{A}$ forces $\langle a, b \rangle$ to hold on $n_1 ... n_{i-1}$ and thus the same is true for P.

□

C.3.5 Proof of Theorem 4.2.3

Let CFG=($\mathcal{N}$,$\mathcal{E}$,ρ).

Let PAG=($\mathcal{N}'$,$\mathcal{E}'$,ρ') and $holds : \mathcal{N}' \rightarrow \{true, false\}$ be as specified in Appendix C.3.2.

Lemma C.3.1 *The Pointer Alias Graph (PAG) is polynomial in the size of the CFG.*

Each variable in the program contributes one *object name* which may be involved in aliases; for each pointer p, $*p$ may have aliases and for each non-pointer v, v itself may be aliased. Thus the number of possible aliases pairs in a program is $\mid variables \mid^2$. Since a program can use at most $\mathcal{O}(\mid \mathcal{N} \mid)$ variables, the number of possible aliases is $\mathcal{O}(\mid \mathcal{N} \mid^2)$.

$\mid \mathcal{N}' \mid = \mid \mathcal{N} \mid * \mid \mathcal{ASSUMED} \mid * \mid \mathcal{POSSIBLE\text{-}ALIASES} \mid + 1$

$\mid \mathcal{E}' \mid \leq \mid \mathcal{E} \mid * \mid \mathcal{ASSUMED} \mid^2 * \mid \mathcal{POSSIBLE\text{-}ALIASES} \mid + \mid \mathcal{N}' \mid$

□

Lemma C.3.2 *The PAG can be built from the CFG in polynomial time.*

The specification of PAG in Appendix C.3.2 lends itself to an algorithm for building the PAG. Each node in $\mathcal{N}'$ requires at most $\mathcal{O}(max(\mathcal{E}, \mid \mathcal{ASSUMED} \mid))$ work.

□

Lemma C.3.3 *Fixed point calculation on the PAG is polynomial time.*

Follows from Lemma C.3.1 (p. 175) and the observation that for any node $n' \in \mathcal{N}'$, $holds(n')$ can change its value at most once.

□

Lemma C.3.4 $holds([(node, \mathcal{AA}), \mathcal{PA}]) = true$ *iff*

> $\exists$ *a realizable path* $P = [entry(node)]n_1 n_2 ... n_i[node]$ *in the ICFG, where* $n_1 n_2 ... n_i$ *contains the same number of calls as returns, such that for every realizable path* $P' = \rho m_1 ... m_i[entry(node)]$, $\mathcal{AA}$ *holding*[79] *on* P' *means* $\mathcal{PA}$ *must hold on* $P' \bullet P$ (*i.e.*, $\rho m_1 ... m_i[entry(node)]n_1 n_2 ... n_i[node]$).

Proof of Lemma C.3.4 (**only if**) By induction on number of iterations of the fixed point algorithm:

Assume $holds([(node, \mathcal{AA}), \langle a, b \rangle])$ became true in the ith iteration.

basis ($i = 0$): after initialization.

> For all $n \in \mathcal{N}'$, $holds(n)$ is initially false. "only if" direction of Lemma holds for iteration 0.

induction hypothesis:

> Assume the "only if" direction of Lemma C.3.4 holds for iterations $j < i$.

induction step:

1. $a = b$ and a is not a dereferenced pointer

   Reflexive aliases for variables hold on all paths.

2. $a \neq b$ or a is a dereferenced pointer

   (a) *node* is an entry node

   By definition $holds([(node, \mathcal{AA}), \langle a, b \rangle])$ is true iff $\mathcal{AA} = \langle a, b \rangle$. In this case the "only if" direction of the Lemma is clearly satisfied.

[79] We consider the assumed alias $\emptyset$ to hold on all paths

(b) *node* is an exit node

$\boldsymbol{holds}([(node, \mathcal{AA}), \langle a, b\rangle]) = \bigvee_{\ll m,node\gg \in \mathcal{E}} \left(\boldsymbol{holds}([(m, \mathcal{AA}), \langle a, b\rangle])\right)$

Thus for some m such that $\ll m, node\gg \in \mathcal{E}$, $\boldsymbol{holds}([(m, \mathcal{AA}), \langle a, b\rangle])$ was set true at an earlier iteration. By induction hypothesis, there is a path P $(entry(m)...m)$ that satisfies the Lemma for $\boldsymbol{holds}([(m, \mathcal{AA}), \langle a, b\rangle])$. Since $entry(m) = entry(node)$, $P \circ [node]$ must satisfy the Lemma for $\boldsymbol{holds}([(node, \mathcal{AA}), \langle a, b\rangle])$.

(c) *node* is an call node

Same argument as for exit nodes.

(d) *node* is a return node

i. a and b are both not visible in the called procedure

$\boldsymbol{holds}([(node, \mathcal{AA}), \langle a, b\rangle]) = \boldsymbol{holds}([(call(node), \mathcal{AA}), \langle a, b\rangle])$

$\boldsymbol{holds}([(call(node), \mathcal{AA}), \langle a, b\rangle])$ was set true at an earlier iteration. By induction hypothesis, there is a path P $(entry(call(node))...[call(node)])$ that satisfies the Lemma for $\boldsymbol{holds}([(call(node), \mathcal{AA}), \langle a, b\rangle])$. Let P^* be any realizable path from $entry(exit(node))$ to $exit(node)$. Since $entry(call(node)) = entry(node)$, $P \circ P^* \circ [node]$ must satisfy the Lemma for $\boldsymbol{holds}([(node, \mathcal{AA}), \langle a, b\rangle])$.

ii. a is visible in the called procedure, but b is not

$$\boldsymbol{holds}([(node, \mathcal{AA}), \langle a, b\rangle]) = \bigvee_{\langle c, non_visible\rangle \in \mathcal{ASSUMED}} \left(\begin{array}{l} holds([(exit(node), \langle c, non_visible\rangle), \langle a, non_visible\rangle]) \bigwedge \\ holds([(call(node), \mathcal{AA}), back\text{-}bind'_{call(node)}(\langle c, non_visible\rangle, b)]) \end{array} \right)$$

Since $\boldsymbol{holds}([(node, \mathcal{AA}), \langle a, b\rangle])$ is true, for some $\langle c, non_visible\rangle$ (1) $\boldsymbol{holds}([(exit(node), \langle c, non_visible\rangle), \langle a, non_visible\rangle])$ and (2) $\boldsymbol{holds}([(call(node), \mathcal{AA}), back\text{-}bind'_{call(node)}(\langle c, non_visible\rangle, b)]$ must have been made true at a earlier iterations. By induction, (1) implies that there is a path P_1 $([entry(exit(node))]$ ... $[exit(node)])$ such that when $\langle c, non_visible\rangle$ holds on a path P'_1 to $entry(exit(node))$, $\langle a, non_visible\rangle$ holds on $P'_1 \bullet P_1$. Also by induction, (2) implies there is a path P_2

$([entry(node)]$ ... $[call(node)])$[80] such that when $\mathcal{AA}$ holds on a path P_2' to $entry(node)$, $back\text{-}bind'_{call(node)}(\langle c, non_visible\rangle, b)$ holds on $P_2' \bullet P_2$. This along with Lemma C.1.1 (p. 160) imply that $P_2 \circ P_1 \circ [node]$ is a path that satisfies the Lemma for $holds([(node, \mathcal{AA}), \langle a, b\rangle])$.

iii. a and b are both visible in the called procedure

$holds([(node, \mathcal{AA}), \langle a, b\rangle]) = holds([(exit(node), \emptyset), \langle a, b\rangle]) \vee$
$\bigvee_{\langle c,d\rangle \in \mathcal{ASSUMED}} \left(\begin{array}{l} holds([(exit(node), \langle c, d\rangle), \langle a, b\rangle]) \bigwedge \\ holds([(call(node), \mathcal{AA}), back\text{-}bind_{call(node)}(\langle c, d\rangle)]) \end{array} \right)$

Since $holds([(node, \mathcal{AA}), \langle a, b\rangle])$ is true either $holds([(exit(node), \emptyset), \langle a, b\rangle])$ was set true in a previous iteration or for some $\langle c, d\rangle$ (1) $holds([(exit(node), \langle c, d\rangle), \langle a, b\rangle])$ and (2) $holds([(call(node), \mathcal{AA}), back\text{-}bind_{call(node)}(\langle c, d\rangle)]$ must have been made true at some earlier iterations. In the former case, by induction there is a path $P_\emptyset$ $([entry(exit(node))]$ ... $[exit(node)])$ such that for any that path $P'_\emptyset$ to $[entry(exit(node))]$, $\langle a, b\rangle$ holds on $P'_\emptyset \bullet P_\emptyset$. Thus for any path P^* $([entry(node)]$ $n_1 ... n_i$ $[call(node)])$ that is realizable and has an equal number of calls and returns on $n_1 ... n_i$, $P^* \circ P_\emptyset \circ [node]$ satisfies the Lemma for $holds([(node, \mathcal{AA}), \langle a, b\rangle])$. In the latter case by induction, (1) implies that there is a path P_1 $([entry(exit(node))]$ ... $[exit(node)])$ such that when $\langle c, d\rangle$ holds on a path P'_1 to $entry(exit(node))$, $\langle a, b\rangle$ holds on $P'_1 \bullet P_1$. Also by induction, (2) implies there is a path P_2 $([entry(node)]$ ... $[call(node)])$ such that when $\mathcal{AA}$ holds on a path P'_2 to $entry(node)$, $back\text{-}bind_{call(node)}(\langle c, d\rangle)$ holds on $P'_2 \bullet P_2$. This along with Lemma C.1.1 (p. 160) imply that $P_2 \circ P_1 \circ [node]$ is a path that satisfies the Lemma for $holds([(node, \mathcal{AA}), \langle a, b\rangle])$.

(e) otherwise ($node$ is a statement node)

i. $node$ is not an assignment to a pointer

$holds([(node, \mathcal{AA}), \langle a, b\rangle]) = \bigvee_{\ll m, node \gg \in \mathcal{E}} \left(holds([(m, \mathcal{AA}), \langle a, b\rangle]) \right)$

Thus for some m such that $\ll m, node \gg \in \mathcal{E}$, $holds([(m, \mathcal{AA}), \langle a, b\rangle])$ was

[80] $entry(node) = entry(call(node))$.

set true at an earlier iteration. By induction hypothesis, there is a path P $(entry(m)...m)$ that satisfies the Lemma for $holds([(m, \mathcal{AA}), \langle a, b\rangle])$. Since $entry(m) = entry(node)$, $P \circ [node]$ must satisfy the Lemma for $holds([(node, \mathcal{AA}), \langle a, b\rangle])$.

ii. *node* is the pointer assignment "$p = q$"

A. if $a = b = *p$

$holds([(node, \mathcal{AA}), \langle *p, *p\rangle]) = \bigvee_{\ll m,node\gg \in \mathcal{E}} \left(holds([(m, \mathcal{AA}), \langle *q, *q\rangle])\right)$

Thus for some m such that $\ll m, node\gg \in \mathcal{E}$, $holds([(m, \mathcal{AA}), \langle *q, *q\rangle])$ was set true at an earlier iteration. By induction hypothesis, there is a path P $(entry(m)...m)$ that satisfies the Lemma for $holds([(m, \mathcal{AA}), \langle *q, *q\rangle])$. Since $entry(m) = entry(node)$, $P \circ [node]$ must satisfy the Lemma for $holds([(node, \mathcal{AA}), \langle *p, *p\rangle])$.

B. if $a = *p$ $[a \neq b]$

$holds([(node, \mathcal{AA}), \langle *p, b\rangle]) = \bigvee_{\ll m,node\gg \in \mathcal{E}} \left(holds([(m, \mathcal{AA}), \langle *q, b\rangle])\right)$

Thus for some m such that $\ll m, node\gg \in \mathcal{E}$, $holds([(m, \mathcal{AA}), \langle *q, b\rangle])$ was set true at an earlier iteration. By induction hypothesis, there is a path P $(entry(m)...m)$ that satisfies the Lemma for $holds([(m, \mathcal{AA}), \langle *q, b\rangle])$. Since $entry(m) = entry(node)$, $P \circ [node]$ must satisfy the Lemma for $holds([(node, \mathcal{AA}), \langle *p, b\rangle])$.

C. if $(a \neq *p)$ and $(b \neq *p)$

$holds([(node, \mathcal{AA}), \langle a, b\rangle]) = \bigvee_{\ll m,node\gg \in \mathcal{E}} \left(holds([(m, \mathcal{AA}), \langle a, b\rangle])\right)$

Thus for some m such that $\ll m, node\gg \in \mathcal{E}$, $holds([(m, \mathcal{AA}), \langle a, b\rangle])$ was set true at an earlier iteration. By induction hypothesis, there is a path P $(entry(m)...m)$ that satisfies the Lemma for $holds([(m, \mathcal{AA}), \langle a, b\rangle])$. Since $entry(m) = entry(node)$, $P \circ [node]$ must satisfy the Lemma for $holds([(node, \mathcal{AA}), \langle a, b\rangle])$.

iii. *node* is the pointer assignment "$p = \&v$"

A. if $a = b = *p$

$\langle *p, *p\rangle$ holds on all paths to *node*. Thus any path P^* $([entry(node)]$ $n_1...n_i$ $[node])$ that is realizable and has an equal number of calls and

returns on $n_1 ... n_i$, satisfies the Lemma for $holds([(node, \mathcal{AA}), \langle *p, *p \rangle])$.

B. if $a = *p$ $[a \neq b]$

$holds([(node, \mathcal{AA}), \langle *p, b \rangle]) = \bigvee_{\ll m, node \gg \in \mathcal{E}} \left(holds([(m, \mathcal{AA}), \langle v, b \rangle]) \right)$

Thus for some m such that $\ll m, node \gg \in \mathcal{E}$, $holds([(m, \mathcal{AA}), \langle v, b \rangle])$ was set true at an earlier iteration. By induction hypothesis, there is a path P $(entry(m)...m)$ that satisfies the Lemma for $holds([(m, \mathcal{AA}), \langle v, b \rangle])$. Since $entry(m) = entry(node)$, $P \circ [node]$ must satisfy the Lemma for $holds([(node, \mathcal{AA}), \langle *p, b \rangle])$.

C. if $(a \neq *p)$ and $(b \neq *p)$

$holds([(node, \mathcal{AA}), \langle a, b \rangle]) = \bigvee_{\ll m, node \gg \in \mathcal{E}} \left(holds([(m, \mathcal{AA}), \langle a, b \rangle]) \right)$

Thus for some m such that $\ll m, node \gg \in \mathcal{E}$, $holds([(m, \mathcal{AA}), \langle a, b \rangle])$ was set true at an earlier iteration. By induction hypothesis, there is a path P $(entry(m)...m)$ that satisfies the Lemma for $holds([(m, \mathcal{AA}), \langle a, b \rangle])$. Since $entry(m) = entry(node)$, $P \circ [node]$ must satisfy the Lemma for $holds([(node, \mathcal{AA}), \langle a, b \rangle])$.

iv. *node* is the pointer assignment "$p = malloc()$" or "$p = NULL$"

A. if $a = b = *p$

By definition, *node* must be "$p = malloc()$". In which case, $\langle *p, *p \rangle$ holds on all paths to *node*. Thus any path P^* $([entry(node)]\ n_1 ... n_i\ [node])$ that is realizable and has an equal number of calls and returns on $n_1 ... n_i$, satisfies the Lemma for $holds([(node, \mathcal{AA}), \langle *p, *p \rangle])$.

B. if $a = *p$ $[a \neq b]$

By definition, $holds([(node, \mathcal{AA}), \langle *p, b \rangle])$ can never be true.

C. if $(a \neq *p)$ and $(b \neq *p)$

$holds([(node, \mathcal{AA}), \langle a, b \rangle]) = \bigvee_{\ll m, node \gg \in \mathcal{E}} \left(holds([(m, \mathcal{AA}), \langle a, b \rangle]) \right)$

Thus for some m such that $\ll m, node \gg \in \mathcal{E}$, $holds([(m, \mathcal{AA}), \langle a, b \rangle])$ was set true at an earlier iteration. By induction hypothesis, there is a path P $(entry(m)...m)$ that satisfies the Lemma for $holds([(m, \mathcal{AA}), \langle a, b \rangle])$. Since $entry(m) = entry(node)$, $P \circ [node]$ must satisfy the Lemma for $holds([(node, \mathcal{AA}), \langle a, b \rangle])$.

Proof of Lemma C.3.4 (**if**) by induction on length of path P:

basis: $\mid P \mid = 1$

$P = entry_node$. There are only two possibilities for $\mathcal{PA}$ holding on $P' \bullet P$ given that $\mathcal{AA}$ holds on P'. The first is that $\mathcal{AA} = \mathcal{PA}$. By case 2.a of Appendix C.3.2 $holds([(entry_node, \mathcal{AA}), \mathcal{PA}]) = true$ if $\mathcal{AA} = \mathcal{PA}$. The only other possibility is if $\mathcal{PA}$ is a reflexive alias of a variable (but not a dereferenced pointer). In this event, case 1 of Appendix C.3.2 defines $holds([(entry_node, \mathcal{AA}), \mathcal{PA}])$ to be true.

induction hypothesis:

Assume the "if" direction of Lemma C.3.4 holds for paths of length $< \mid P \mid$.

induction step:

Let $P = [entry(node)]n_1...n_i[node]$ where $n_1...n_i$ has the same number of calls as returns. Let $\langle a, b \rangle$ be any alias that holds on $P' \bullet P$ for all $P' = \rho m_1...m_i[entry(node)]$ such that $\mathcal{AA}$ holds on P' and $m_1...m_i$ has the same number of calls as returns.

1. $a = b$ and a is not a dereferenced pointer

   By definition $holds([(node, \mathcal{AA}), \langle a, b \rangle])$ is $true$.

2. $a \neq b$ or a is a dereferenced pointer

   (a) *node* is an entry node

   $\mid P \mid$ must be 1 and the basis applies.

   (b) *node* is an exit node

   Since $\langle a, b \rangle$ holds on all paths $P' \bullet [entry(node)]n_1...n_i[node]$ it must also hold on $P' \bullet [entry(node)]n_1...n_i$. Thus by the induction hypothesis, $holds([(n_i, \mathcal{AA}), \langle a, b \rangle])$ is $true$. By definition, $holds([(node, \mathcal{AA}), \langle a, b \rangle]) = \bigvee_{\ll m, node \gg \in \mathcal{E}} \left(holds([(m, \mathcal{AA}), \langle a, b \rangle]) \right)$. Thus $holds([(node, \mathcal{AA}), \langle a, b \rangle])$ is true.

   (c) *node* is a call node

   Same argument as for an exit node.

(d) *node* is a return node

i. a and b are both not visible in the called procedure

a and b are aliased on a path after return from the called procedure iff they were aliases before the procedure was invoked. Let n_j be the call node that corresponds to the return, *node*. Since $\langle a, b\rangle$ holds on all paths $P' \bullet [entry(node)]n_1...n_i[node]$ it also holds on $P' \bullet [entry(node)]n_1...n_j$. Thus by the induction hypothesis, $holds([(n_j, \mathcal{AA}), \langle a,b\rangle])$ is *true*. By definition, $holds([(node,\mathcal{AA}), \langle a,b\rangle]) = holds([(call(node),\mathcal{AA}), \langle a,b\rangle])$ where $call(node) = n_j$. Thus $holds([(node,\mathcal{AA}), \langle a,b\rangle])$ is true.

ii. a is visible in the called procedure, but b is not

For a to be aliased to b on a path to *node*, a must have been aliased to *non_visible* on the same path up to (and including) the exit of the called procedure and *non_visible* is b. Let n_j be the call node that corresponds to the return, *node*. Given that $\mathcal{AA}$ holds on P' there is a set, $\mathcal{A}$ of aliases that hold on $P' \bullet [entry(node)]n_1...n_jn_{j+1}$. Now n_{j+1} is the entry node of the called procedure and we know that $\langle a, non_visible\rangle$ holds on P' $\bullet$ $[entry(node)]n_1...n_jn_{j+1}...n_i$ thus by Lemma 4.2.1 (p. 34) there is a unique $A_k \in \mathcal{A}$[81] just assuming A_k at n_{j+1} would have been sufficient. Since $n_{j+1}...n_i$ is a shorter path than P that meets the necessary restrictions, by the induction hypothesis $holds([(n_i, A_k), \langle a, non_visible\rangle])$ is true. Now since A_k holds on $P' \bullet [entry(node)]n_1...n_jn_{j+1}$, by Lemma C.1.1 (p. 160) $back\text{-}bind'_{n_j}(A_k, b)$ holds on $P' \bullet [entry(node)]n_1 \ ... \ n_j$. By induction $holds([(n_j, \mathcal{AA}), back\text{-}bind'_{n_j}(A_k, b)])$ is true. By definition,

$$holds([(node,\mathcal{AA}), \langle a,b\rangle]) = \bigvee_{\langle c,non_visible\rangle \in \mathcal{ASSUMED}} \left(\begin{array}{l} holds([(exit(node), \langle c, non_visible\rangle), \langle a, non_visible\rangle]) \bigwedge \\ holds([(call(node), \mathcal{AA}), back\text{-}bind'_{call(node)}(\langle c, non_visible\rangle, b)]) \end{array} \right)$$

$n_j = call(node)$ and $n_i = exit(node)$. Thus $holds([(node, \mathcal{AA}), \langle a,b\rangle])$ is true.

iii. a and b are visible in the called procedure

[81] $\emptyset$ is not possible since *non_visible* is involved and A_k must be $\langle c, non_visible\rangle$ for some c.

By definition,

$$\mathit{holds}([(\mathit{node},\mathcal{AA}),\langle a,b\rangle]) = \mathit{holds}([(\mathit{exit}(\mathit{node}),\emptyset),\langle a,b\rangle]) \vee$$
$$\bigvee_{\langle c,d\rangle \in \mathcal{ASSUMED}} \left(\begin{array}{l} \mathit{holds}([(\mathit{exit}(\mathit{node}),\langle c,d\rangle),\langle a,b\rangle]) \bigwedge \\ \mathit{holds}([(\mathit{call}(\mathit{node}),\mathcal{AA}),\mathit{back\text{-}bind}_{\mathit{call}(\mathit{node})}(\langle c,d\rangle)]) \end{array} \right)$$

For a to be aliased to b on a path to *node*, a must have been aliased to b on the same path up to (and including) the exit of the called procedure. Let n_j be the call node that corresponds to the return, *node*. Given that $\mathcal{AA}$ holds on P' there is a set, $\mathcal{A}$ of aliases that hold on $P' \bullet [\mathit{entry}(\mathit{node})]n_1 ... n_j n_{j+1}$. Now n_{j+1} is the entry node of the called procedure and we know that $\langle a,b\rangle$ holds on $P' \bullet [\mathit{entry}(\mathit{node})]n_1 ... n_j n_{j+1} ... n_i$ thus by Lemma 4.2.1 (p. 34) either no assumptions are necessary at n_{j+1} or there is a unique $A_k \in \mathcal{A}$ such that just assuming A_k at n_{j+1} is sufficient. In the first case since $n_{j+1} ... n_i$ is a shorter path than P that meets the necessary restrictions, by the induction hypothesis $\mathit{holds}([(n_i,\emptyset),\langle a,b\rangle])$ is true and thus $\mathit{holds}([(\mathit{node},\mathcal{AA}),\langle a,b\rangle])$ must also be true. In the latter case, by the induction hypothesis (again on $n_{j+1} ... n_i$) $\mathit{holds}([(n_i,A_k),\langle a,b\rangle])$ is true. Now since A_k holds on $P' \bullet [\mathit{entry}(\mathit{node})]n_1 ... n_j n_{j+1}$, by Lemma C.1.1 (p. 160) $\mathit{back\text{-}bind}_{n_j}(A_k)$ holds on $P' \bullet [\mathit{entry}(\mathit{node})]n_1 \ ... \ n_j$. By induction $\mathit{holds}([(n_j,\mathcal{AA}),\mathit{back\text{-}bind}_{n_j}(A_k)])$ is true. Since $n_j = \mathit{call}(\mathit{node})$ and $n_i = \mathit{exit}(\mathit{node})$, $\mathit{holds}([(\mathit{node},\mathcal{AA}),\langle a,b\rangle])$ is true by definition.

(e) otherwise (*node* is a statement node)

i. *node* is not an assignment to a pointer

Since $\langle a,b\rangle$ holds on all paths $P' \bullet [\mathit{entry}(\mathit{node})]n_1 ... n_i[\mathit{node}]$ it must also hold on $P' \bullet [\mathit{entry}(\mathit{node})]n_1 ... n_i$. Thus by the induction hypothesis, $\mathit{holds}([(n_i,\mathcal{AA}),\langle a,b\rangle])$ is *true*. By definition,
$\mathit{holds}([(\mathit{node},\mathcal{AA}),\langle a,b\rangle]) = \bigvee_{\ll m,\mathit{node}\gg \in \mathcal{E}} \left(\mathit{holds}([(m,\mathcal{AA}),\langle a,b\rangle])\right)$.
Thus $\mathit{holds}([(\mathit{node},\mathcal{AA}),\langle a,b\rangle])$ is true.

ii. *node* is the pointer assignment "$p = q$"

A. if $a = b = *p$

Since $\langle *p, *p\rangle$ holds on all paths $P' \bullet [entry(node)]n_1 ... n_i[node]$, $\langle *q, *q\rangle$ must hold on $P' \bullet [entry(node)]n_1 ... n_i$. Thus by the induction hypothesis, $holds([(n_i, \mathcal{AA}), \langle *q, *q\rangle])$ is $true$. By definition, $holds([(node, \mathcal{AA}), \langle *p, *p\rangle]) = \bigvee_{\ll m,node \gg \in \mathcal{E}} \left(holds([(m, \mathcal{AA}), \langle *q, *q\rangle])\right)$. Thus $holds([(node, \mathcal{AA}), \langle *p, *p\rangle])$ is true.

B. if $a = *p$ $[a \neq b]$

Since $\langle *p, b\rangle$ holds on all paths $P' \bullet [entry(node)]n_1 ... n_i[node]$, $\langle *q, b\rangle$ must hold on $P' \bullet [entry(node)]n_1 ... n_i$. Thus by the induction hypothesis, $holds([(n_i, \mathcal{AA}), \langle *q, b\rangle])$ is $true$. By definition, $holds([(node, \mathcal{AA}), \langle *p, b\rangle]) = \bigvee_{\ll m,node \gg \in \mathcal{E}} \left(holds([(m, \mathcal{AA}), \langle *q, b\rangle])\right)$. Thus $holds([(node, \mathcal{AA}), \langle *p, b\rangle])$ is true.

C. if $(a \neq *p)$ and $(b \neq *p)$

Since $\langle a, b\rangle$ holds on all paths $P' \bullet [entry(node)]n_1 ... n_i[node]$ it must also hold on $P' \bullet [entry(node)]n_1 ... n_i$. Thus by the induction hypothesis, $holds([(n_i, \mathcal{AA}), \langle a, b\rangle])$ is $true$. By definition, $holds([(node, \mathcal{AA}), \langle a, b\rangle]) = \bigvee_{\ll m,node \gg \in \mathcal{E}} \left(holds([(m, \mathcal{AA}), \langle a, b\rangle])\right)$. Thus $holds([(node, \mathcal{AA}), \langle a, b\rangle])$ is true.

iii. *node* is the pointer assignment "$p = \&v$"

A. if $a = b = *p$

By definition $holds([(node, \mathcal{AA}), \langle *p, *p\rangle]) = true$.

B. if $a = *p$ $[a \neq b]$

Since $\langle *p, b\rangle$ holds on all paths $P' \bullet [entry(node)]n_1 ... n_i[node]$, $\langle v, b\rangle$ must hold on $P' \bullet [entry(node)]n_1 ... n_i$. Thus by the induction hypothesis, $holds([(n_i, \mathcal{AA}), \langle v, b\rangle])$ is $true$. By definition, $holds([(node, \mathcal{AA}), \langle *p, b\rangle]) = \bigvee_{\ll m,node \gg \in \mathcal{E}} \left(holds([(m, \mathcal{AA}), \langle v, b\rangle])\right)$. Thus $holds([(node, \mathcal{AA}), \langle *p, b\rangle])$ is true.

C. if $(a \neq *p)$ and $(b \neq *p)$

Since $\langle a, b\rangle$ holds on all paths $P' \bullet [entry(node)]n_1 ... n_i[node]$ it must also hold on $P' \bullet [entry(node)]n_1 ... n_i$. Thus by the induction hypothesis, $holds([(n_i, \mathcal{AA}), \langle a, b\rangle])$ is $true$. By definition,

$holds([(node, \mathcal{AA}), \langle a, b\rangle]) = \bigvee_{\ll m,node \gg \in \mathcal{E}} \left(holds([(m, \mathcal{AA}), \langle a, b\rangle])\right)$.

Thus $holds([(node, \mathcal{AA}), \langle a, b\rangle])$ is true.

iv. *node* is the pointer assignment "$p = malloc()$" or "$p = NULL$"

A. if $a = b = *p$

node must be "$p = malloc()$" in which case,

$holds([(node, \mathcal{AA}), \langle *p, *p\rangle]) = true$.

B. if $a = *p$ $[a \neq b]$

No such alias can hold on a path to *node*.

C. if $(a \neq *p)$ and $(b \neq *p)$

Since $\langle a, b\rangle$ holds on all paths $P' \bullet [entry(node)]n_1 ... n_i[node]$ it must also hold on $P' \bullet [entry(node)]n_1 ... n_i$. Thus by the induction hypothesis, $holds([(n_i, \mathcal{AA}), \langle a, b\rangle])$ is $true$. By definition,

$holds([(node, \mathcal{AA}), \langle a, b\rangle]) = \bigvee_{\ll m,node \gg \in \mathcal{E}} \left(holds([(m, \mathcal{AA}), \langle a, b\rangle])\right)$.

Thus $holds([(node, \mathcal{AA}), \langle a, b\rangle])$ is true.

□

Lemma C.3.5 *The fixed point of may-alias can be calculated in time polynomial in the size of the ICFG = ($\mathcal{N}$,$\mathcal{E}$,ρ).*

Each variable in the program contributes one *object name* which may be involved in aliases; for each pointer p, $*p$ may have aliases and for each non-pointer v, v itself may be aliased. Thus the number of possible alias pairs in a program is $\mid variables \mid^2$. Since a program can use at most $\mathcal{O}(\mid \mathcal{N} \mid)$ variables, the number of possible aliases is $\mathcal{O}(\mid \mathcal{N} \mid^2)$.

During the fixed point calculation for any *node*, *may-alias*(*node*) can never decrease in size. Since the maximum size of all *may-alias*(*node*) is $\mathcal{O}(\mid \mathcal{N} \mid^2)$, any *may-alias*(*node*) can change its value at most that many times. Thus we have $\mid \mathcal{N} \mid$ nodes that can change their value at most $\mathcal{O}(\mid \mathcal{N} \mid^2)$ times. Therefore the fixed point

calculation for *may-alias* takes a polynomial amount of time ($\mathcal{O}(\mid \mathcal{N} \mid^3)$).

□

Lemma C.3.6 *For all realizable paths $P = n_1 n_2 ... n_i$ (where $n_1 = entry(n_i)$ and the number of calls on $n_2 ... n_{i-1}$ equals the number of returns),*

If

> *all alias pairs in the set $\mathcal{A} = \{A_1, ..., A_m\}$ holding at n_1 and that the execution of P forces all aliases in the set $\mathcal{A}'$ to hold at n_i*

then

> *for any alias $B \notin \mathcal{A}$, assuming $\mathcal{A} \cup \{B\}$ at n_1 and that the execution of P forces all aliases in the set $\mathcal{A}''$ to hold at n_i, $\mathcal{A}'$ must be a subset of $\mathcal{A}''$.*

Additional alias assumptions on a path can only create additional aliases, they can not destroy existing ones.

□

Lemma C.3.7 *For all $n \in \mathcal{N}$ and all $\langle a, b \rangle \in \mathcal{O}'$, $\langle a, b \rangle \in may\text{-}alias(n)$ iff $[n, \langle a, b \rangle]$ is in the precise solution for Interprocedural May Alias in the presence of single level pointers.*

Proof of Lemma C.3.7 (only if) by induction on iteration of fixed point algorithm.

basis:

> For all n, *may-alias*(n) is initialized to $\emptyset$ thus the Lemma holds vacuously.

induction hypothesis:

> Assume that Lemma C.3.7 holds in the "only if" direction for iterations $j < i$ of the fixed point calculation.

induction step:

Let $\langle a, b\rangle$ be added to *may-alias*(n) in iteration i.

1. n is an entry node

   By definition, *may-alias*$(n) = \bigcup_{\ll m,n \gg \in \mathcal{E}} \left(bind_m(\textit{may-alias}(m))\right)$.

   For every $\ll m, n \gg \in \mathcal{E}$, let $\mathcal{A}^{i-1}_{\ll m,n \gg}$ be the set of $[m, \langle c, d\rangle]$ such that $\langle c, d\rangle \in$ *may-alias*(m) at iteration $i-1$. By the induction hypothesis, $\mathcal{A}^{i-1}_{\ll m,n \gg} \subseteq$ precise solution for Interprocedural May Alias. Since $\langle a, b\rangle$ was added to *may-alias*(n) in the i^{th} iteration, $\langle a, b\rangle \in \bigcup_{\ll m,n \gg \in \mathcal{E}} \left(bind_m(\left\{\langle c, d\rangle \,|\, [m, \langle c, d\rangle] \in \mathcal{A}^{i-1}_{\ll m,n \gg}\right\})\right)$ which, by Lemmas C.2.1 (p. 163) and C.3.6 (p. 186), is a subset of $\bigcup_{\ll m,n \gg \in \mathcal{E}}$ $\left(bind_m(\textit{may-alias}(m))\right)$ which equals, by definition, *may-alias*(n). Therefore, $[n, \langle a, b\rangle]$ must be in the precise solution for Interprocedural May Alias.

2. *otherwise*

   By definition, *may-alias*$(n) =$

$$\left\{ \langle a, b\rangle \;\middle|\; \begin{array}{l} \text{neither } a \text{ nor } b \text{ is } \textit{non_visible} \text{ and} \\ [\textit{holds}([(n, \emptyset), \langle a, b\rangle]) = \textit{true}] \bigvee \\ [\left(\exists \mathcal{AA} \in \textit{may-alias}(\textit{entry}(n))\right) \textit{holds}([(n, \mathcal{AA}), \langle a, b\rangle]) = \textit{true}] \end{array} \right\}$$

   Thus for $\langle a, b\rangle \in$ *may-alias*(n) either *holds*$([(n, \emptyset), \langle a, b\rangle])$ is true in which case the Lemma follows from Lemma C.3.4 (p. 176) or there is some $\mathcal{AA} \in$ *may-alias*$(\textit{entry}(n))$ at iteration $i-1$ for which *holds*$([(n, \mathcal{AA}), \langle a, b\rangle])$ is true. Since $\mathcal{AA} \in$ *may-alias*$(\textit{entry}(n))$ at iteration $i-1$, by the induction hypothesis $[\textit{entry}(n), \mathcal{AA}]$ must be in the precise solution for Interprocedural May Alias. This along with Lemma C.3.4 (p. 176) implies that $[n, \langle a, b\rangle]$ has to be in the precise solution.

Proof of Lemma C.3.7 (if) by induction on path length.

basis: path is simply ρ

No aliases hold before execution of the program, thus the Lemma holds vacuously.

induction hypothesis:

Assume that Lemma C.3.7 holds in the "if" direction for paths of length $j < i$.

induction step: Let path be $P = \rho n_1 ... n_{k'}[entry(m_k)]m_1 ... m_k$ where $m_1 ... m_{k-1}$ has the same number of calls and returns. Let $\langle a, b\rangle$ be any alias that holds on P.

1. m_k is an entry node (thus $k = 0$ and $entry(m_0) = m_0$)
   By definition, $may\text{-}alias(n) = \bigcup_{\ll m,n \gg \in \mathcal{E}} \left(bind_m(may\text{-}alias(m))\right)$.
   Let $\mathcal{A}'$ be the set off all aliases that hold on $\rho n_1 ... n_{k'}$. By induction, $\mathcal{A}' \subseteq may\text{-}alias(n_{k'})$. By Lemma C.2.1 (p. 163) $\langle a, b\rangle$ is in $bind_{n_{k'}}(\mathcal{A}')$ and thus by Lemma C.3.6 (p. 186) $\langle a, b\rangle$ is in $bind_{n_{k'}}(may\text{-}alias(n_{k'}))$ and thus by definition in $may\text{-}alias(m_0)$.

2. otherwise
   By definition, $may\text{-}alias(n) =$
   $$\left\{ \langle a, b\rangle \;\middle|\; \begin{array}{l} [holds([(n, \emptyset), \langle a, b\rangle]) = true] \bigvee \\ \left[\left(\exists \mathcal{AA} \in may\text{-}alias(entry(n))\right) holds([(n, \mathcal{AA}), \langle a, b\rangle]) = true\right] \end{array} \right\}$$
   Let $\mathcal{A}'$ be the set off all aliases that hold on $\rho n_1 ... n_{k'}[entry(m_k)]m_1 ... m_k$. By Lemma 4.2.1 (p. 34), either no assumptions are necessary or there is an $\mathcal{AA}$ in $\mathcal{A}'$ such that assuming only $\mathcal{AA}$ at $entry(m_k)$ forces a and b to be aliased after execution of $[entry(m_k)]m_1 ... m_k$. In the first case, $holds([(m_k, \emptyset), \langle a, b\rangle])$ is true by Lemma C.3.4 (p. 176) and thus $\langle a, b\rangle \in may\text{-}alias(m_k)$ by definition of *may-alias*. In the second case, $holds([(m_k, \mathcal{AA}), \langle a, b\rangle])$ is true by Lemma C.3.4 (p. 176), and since $\mathcal{AA}$ holds on $\rho n_1 ... n_{k'}[entry(m_k)]$, by the induction hypothesis $\mathcal{AA} \in may\text{-}alias(entry(m_k))$. Thus $\langle a, b\rangle$ has to be in $may\text{-}alias(m_k)$ by definition of *may-alias*.

□

Theorem 4.2.3 *There exists a polynomial time algorithm for determining precise* **Interprocedural May Alias** *sets in the presence of single level pointers.*

The algorithm in Appendix C.3.3 is such an algorithm. Lemma C.3.7 (p. 186) proves that it is precise. The algorithm consists of 4 steps. Lemma C.3.1 (p. 175) and Lemma C.3.2 (p. 175) imply that the first step can be done in time polynomial in the size of the ICFG. Lemma C.3.3 (p. 175) implies that the second step can be done in time polynomial in the size of the ICFG. The third step can obviously be done in time polynomial in the size of the ICFG. Lemma C.3.5 (p. 185) implies that the final step can also be done in time polynomial in the size of the ICFG.

□

C.4 A Precise Polynomial Time Algorithm for Computing Interprocedural Must Alias Sets in the Presence of Single Level Pointers

C.4.1 Lattice of Conditional Must Alias Sets and Definitions

Let $\mathcal{O} = \{*p \mid p \text{ is a pointer in the program}\} \bigcup$
$\{v \mid v \text{ is a non-pointer variable in the program}\} \bigcup$
$\{non_visible\}$.

$\mathcal{O}$ is the set of all object names in the program which may have aliases.

"*non_visible*" represents object names which are not visible.

Let $\mathcal{POSSIBLE\text{-}ALIASES} = (\mathcal{O} \times \mathcal{O}) - \{\langle non_visible, non_visible\rangle\}$.

$\mathcal{POSSIBLE\text{-}ALIASES}$ is the set of all possible aliases.

Define the lattice[Hec77] $\mathcal{L} = (\mathcal{S}, \sqsubseteq, \sqcap, \bot, \top)$ as

- $\mathcal{S} = powerset(\mathcal{POSSIBLE\text{-}ALIASES}) \cup false$
- $a \sqsubseteq b$ iff $(a = false)$ or $(a \supseteq b)$[82]
- $a \sqcap b = \begin{cases} false & \text{if } (a = false) \text{ or } (b = false) \\ a \cup b & \text{otherwise} \end{cases}$
- $\top = \emptyset$

[82]NOTE: If b is $false$, $a \sqsubseteq b$ iff a is $false$ also.

- $\perp = false$

Given an ICFG for a program:

Let $exit(n)$, n a return node in the ICFG, be the exit node corresponding to n.
Let $call(n)$, n a return node in the ICFG, be the call node corresponding to n.
Let $entry(n)$, n a node in the ICFG, be the entry node of the procedure containing n.

Let $back\text{-}bind_{call}$, *call* a call node in the ICFG, be defined as in Appendix C.1.
$back\text{-}bind_{call}(alias\text{-}pair)$ specifies which alias holding at the *call* site forces *alias-pair* to hold at the entry of the called procedure.

C.4.2 Algorithm for Computing Precise Conditional Must Alias Sets

Let $must\text{-}holds(n, false) = false$ for all nodes n in the ICFG.
Let $must\text{-}holds(n, \langle o, o \rangle) = \emptyset$ for all nodes n in the ICFG, all possible non-dereferenced pointer object names o.
Define the relation $must\text{-}holds(n, \langle a, b \rangle)$, n a node of the ICFG=($\mathcal{N}$,$\mathcal{E}$,ρ) and $\langle a, b \rangle \in \mathcal{POSSIBLE\text{-}ALIASES}$ ($a \neq b$ or a is a dereferenced pointer), as follows:

1. n is an entry node
   $must\text{-}holds(n, \langle a, b \rangle) = \{\langle a, b \rangle\}$

2. n is an exit node
   $must\text{-}holds(n, \langle a, b \rangle) = \sqcap_{\ll m,n \gg \in \mathcal{E}} \big(must\text{-}holds(m, \langle a, b \rangle) \big)$

3. n is a call node
   $must\text{-}holds(n, \langle a, b \rangle) = \sqcap_{\ll m,n \gg \in \mathcal{E}} \big(must\text{-}holds(m, \langle a, b \rangle) \big)$

4. n is a return node

   (a) a and b are both visible in the called procedure

$$must\text{-}holds(n, \langle a, b\rangle) = \begin{cases} false & \text{if } must\text{-}holds(exit(n), \langle a, b\rangle) = false \\ \emptyset & \text{if } must\text{-}holds(exit(n), \langle a, b\rangle) = \emptyset \\ \mathcal{R}_1 & \text{otherwise} \end{cases}$$

$$\mathcal{R}_1 : \quad \sqcap_{A \in must\text{-}holds(exit(n), \langle a,b\rangle)} \Big(must\text{-}holds(call(n), back\text{-}bind_{call(n)}(A)) \Big)$$

(b) a is visible in the called procedure, but b is not

$$must\text{-}holds(n, \langle a, b\rangle) =$$

$$\begin{cases} false & \text{if } must\text{-}holds(exit(n), \langle a, non_visible\rangle) = false \\ (not\ possible) & \text{if } must\text{-}holds(exit(n), \langle a, non_visible\rangle) = \emptyset \\ \mathcal{R}_2 & \text{otherwise} \end{cases}$$

$$\mathcal{R}_2 : \quad \prod_{\substack{\langle x, non_visible\rangle \in \\ must\text{-}holds(exit(n), \langle a, non_visible\rangle)}} \Big(must\text{-}holds(call(n), back\text{-}bind'_{call(n)}(\langle x, non_visible\rangle, b)) \Big)$$

(c) a and b are not visible in the called procedure

$$must\text{-}holds(n, \langle a, b\rangle) = must\text{-}holds(call(n), \langle a, b\rangle)$$

5. n is statement node

(a) n is not an assignment to a pointer

$$must\text{-}holds(n, \langle a, b\rangle) = \sqcap_{\ll m,n \gg \in \mathcal{E}} \Big(must\text{-}holds(m, \langle a, b\rangle) \Big)$$

(b) n is the assignment "$p = q$" (p,q pointers)

i. if $a = b = *p$

$$must\text{-}holds(n, \langle *p, *p\rangle) = \sqcap_{\ll m,n \gg \in \mathcal{E}} \Big(must\text{-}holds(m, \langle *q, *q\rangle) \Big)$$

ii. if $a = *p$ $[a \neq b]$

$$must\text{-}holds(n, \langle *p, b\rangle) = \sqcap_{\ll m,n \gg \in \mathcal{E}} \Big(must\text{-}holds(m, \langle *q, b\rangle) \Big)$$

iii. if $(a \neq *p)$ and $(b \neq *p)$

$$must\text{-}holds(n, \langle a, b\rangle) = \sqcap_{\ll m,n \gg \in \mathcal{E}} \Big(must\text{-}holds(m, \langle a, b\rangle) \Big)$$

(c) n is the assignment "$p = \&v$" (p a pointer)

i. if $a = b = *p$

$$must\text{-}holds(n, \langle *p, *p\rangle) = \emptyset$$

ii. if $a = *p$ $[a \neq b]$

$$must\text{-}holds(n, \langle *p, b\rangle) = \sqcap_{\ll m,n \gg \in \mathcal{E}}\big(must\text{-}holds(m, \langle v, b\rangle)\big)$$

iii. if $(a \neq *p)$ and $(b \neq *p)$

$$must\text{-}holds(n, \langle a, b\rangle) = \sqcap_{\ll m,n \gg \in \mathcal{E}}\big(must\text{-}holds(m, \langle a, b\rangle)\big)$$

(d) n is the assignment "$p = malloc()$" or "$p = NULL$" (p a pointer)

i. if $a = b = *p$

$$must\text{-}holds(n, \langle *p, *p\rangle) = \begin{cases} \emptyset & \text{if } n \text{ is "}p = malloc()\text{"} \\ false & \text{if } n \text{ is "}p = NULL\text{"} \end{cases}$$

ii. if $a = *p$ $[a \neq b]$

$$must\text{-}holds(n, \langle *p, b\rangle) = false$$

iii. if $(a \neq *p)$ and $(b \neq *p)$

$$must\text{-}holds(n, \langle a, b\rangle) = \sqcap_{\ll m,n \gg \in \mathcal{E}}\big(must\text{-}holds(m, \langle a, b\rangle)\big)$$

C.4.3 Algorithm for Computing Precise Must Alias Sets in the Presence of Single Level Pointers

Let $\mathcal{O}' = \{*p \mid p \text{ is a pointer in the program}\} \bigcup$

$\{v \mid v \text{ is a non-pointer variable in the program}\}$.

$\mathcal{O}'$ is the set of all object names in the program which may have aliases.

Let $\mathcal{POSSIBLE\text{-}ALIASES}' = (\mathcal{O}' \times \mathcal{O}')$.

$\mathcal{POSSIBLE\text{-}ALIASES}'$ is the set of all possible aliases.

Define the lattice $\mathcal{L}' =$

$$(powerset(\mathcal{POSSIBLE\text{-}ALIASES}'), \subseteq (\sqsubseteq), \cap(\sqcap), \emptyset\ (\bot), \mathcal{POSSIBLE\text{-}ALIASES}'\ (\top))$$

Let $bind_n(A)$, n a call node in the ICFG and $A \in \mathcal{POSSIBLE\text{-}ALIASES}'$, be defined as in Appendix C.2.

Let $entry(n)$, n a node in the ICFG, be the entry node of the procedure containing n.

Define $X \subseteq' Y$, $X \in \mathcal{L}$ and $Y \in \mathcal{L}'$ as:

- $false$ if $X = false$
- $X \subseteq Y$ otherwise

For every node n in the ICFG define $must\text{-}alias(n)$ as:

1. $must\text{-}alias(\rho) = \emptyset$
2. n is an entry node

   $$must\text{-}alias(n) = \bigcap_{\ll m,n \gg \in \mathcal{E}} \left\{ \langle a, b \rangle \;\middle|\; \begin{array}{l} \langle a, b \rangle \in bind_m(must\text{-}alias(m)) \text{ and} \\ \text{neither } a \text{ nor } b \text{ contain } non_visible \end{array} \right\}$$

3. otherwise

   $$must\text{-}alias(n) = \{\langle a, b \rangle \mid must\text{-}hold(n, \langle a, b \rangle) \subseteq' must\text{-}alias(entry(n))\}$$

Algorithm

1. For all n in the ICFG, all $\langle a, b \rangle \in \mathcal{POSSIBLE\text{-}ALIASES}$ initialize $must\text{-}hold(n, \langle a, b \rangle)$ to $\emptyset$.
2. Calculate the fixed point of *must-hold*.
3. For all n in the ICFG, initialize $must\text{-}alias(n)$ to $\mathcal{POSSIBLE\text{-}ALIASES}'$.
4. Calculate the fixed point of *must-alias*.

C.4.4 Conditional Must Alias Sets

Conditional Must Alias for a *node* of the ICFG and an *alias-pair* is the unique minimal set of assumed must aliases at entry node of the procedure containing *node* which insures that *alias-pair* must hold at *node* on all paths. If no such set exists, conditional must alias is $false$. We will use $COND_{node}^{alias\text{-}pair}$ to refer to conditional must alias information.

Let the lattice of values for Conditional Must Alias values be defined as in Appendix C.4.1.

Relationships between $COND_{node}^{alias\text{-}pair}$ values

---

Lemma C.4.1 *For all n_i in the ICFG, for all $\langle a, b\rangle$,*

if

for all paths $\rho n_1 n_2 ... n_i$

$[\langle a, b\rangle$ holds on $\rho n_1 n_2 ... n_i$ iff $\langle c, d\rangle$ holds on $\rho n_1 n_2 ... n_{i-1}]$

and *n_i is not an entry node*

then

$$COND_{n_i}^{\langle a,b\rangle} = \sqcap_{\ll m,n_i \gg \in \mathcal{E}} \left(COND_m^{\langle c,d\rangle} \right)$$

Proof:

Consider any n and $\langle a, b\rangle$ such that for all paths $\rho n_1 n_2 ... n_i$ ($n_i = n$), [$\langle a, b\rangle$ holds on $\rho n_1 n_2 ... n_i$ iff $\langle c, d\rangle$ holds on $\rho n_1 n_2 ... n_{i-1}$] **and** n_i is not an entry node (note: since n_i is not an entry node, $entry(n_i) = entry(n_{i-1})$):

- if for any immediate predecessor m, $COND_m^{\langle c,d\rangle}$ is $false$ then
  there is a path $[entry(m)]n_1 n_2 ... m$ such that $\langle c, d\rangle$ does not hold regardless of which aliases hold at $[entry(m)]$. Thus $\langle a, b\rangle$ does not hold on $[entry(m)]n_1 ... mn$ regardless of which aliases hold at $[entry(m)] = [entry(n)]$. Thus $COND_n^{\langle a,b\rangle}$ is $false$.

- Otherwise ($(\forall \ll m, n \gg \in \mathcal{E})\ COND_m^{\langle c,d\rangle} \neq false$)
  $COND_n^{\langle a,b\rangle} \subseteq [\bigcup_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle c,d\rangle} \right)]$ because, by definition of $COND$, assuming $[\bigcup_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle c,d\rangle} \right)]$ holds at $entry(n)$ means that on all paths $[entry(n)]n_1 n_2 ... m$, where $\ll m, n \gg \in \mathcal{E}$, $\langle c, d\rangle$ must hold and thus $\langle a, b\rangle$ must hold on $[entry(n)]n_1 n_2 ... mn$. Also $COND_n^{\langle a,b\rangle} \supseteq [\bigcup_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle c,d\rangle} \right)]$. Otherwise there is a m such that $\ll m, n \gg \in \mathcal{E}$ and $COND_n^{\langle a,b\rangle} \not\supseteq COND_m^{\langle c,d\rangle}$.

Thus there exists a path $[entry(m)]n_1n_2...m$ which when assuming $COND_n^{\langle a,b\rangle}$ holds at $entry(m)$, $\langle c,d\rangle$ does not hold. Under those assumptions $\langle a,b\rangle$ does not hold on $[entry(m)]n_1n_2...m$ ($entry(m) = entry(n)$). Contradiction.

□

Lemma C.4.2 *$COND$ obeys the following relationships:*

$COND_n^{false} = \mathbf{false}$ *for all nodes* n *in the ICFG.*

$COND_n^{\langle o,o\rangle} = \emptyset$ *for all nodes* n *in the ICFG, all non-dereferenced pointer object names* $o \in \mathcal{O}$.

Consider $COND_n^{\langle a,b\rangle}$, n *a node of the ICFG=*$(\mathcal{N},\mathcal{E},\rho)$ *and* $\langle a,b\rangle \in \mathcal{A}$:

1. n *is an entry node*

   $COND_n^{\langle a,b\rangle} = \{\langle a,b\rangle\}$

2. n *is an exit node*

   $$COND_n^{\langle a,b\rangle} = \prod_{\ll m,n\gg \in \mathcal{E}} \left(COND_m^{\langle a,b\rangle}\right)$$

3. n *is a call node*

   $$COND_n^{\langle a,b\rangle} = \prod_{\ll m,n\gg \in \mathcal{E}} \left(COND_m^{\langle a,b\rangle}\right)$$

4. n *is a return node*

   (a) a *and* b *are both visible in the called procedure*

   $$COND_n^{\langle a,b\rangle} = \begin{cases} \mathbf{false} & \text{if } COND_{exit(n)}^{\langle a,b\rangle} = \mathbf{false} \\ \emptyset & \text{if } COND_{exit(n)}^{\langle a,b\rangle} = \emptyset \\ \mathcal{R}_1 & \text{otherwise} \end{cases}$$

   $$\mathcal{R}_1 : \quad \prod_{A \in COND_{exit(n)}^{\langle a,b\rangle}} \left(COND_{call(n)}^{back\text{-}bind_{call(n)}(A)}\right)$$

   (b) a *is visible in the called procedure, but* b *is not*

$$COND_n^{\langle a,b\rangle} = \begin{cases} false & if\ COND_{exit(n)}^{\langle a,non_visible\rangle} = false \\ (not\ possible) & if\ COND_{exit(n)}^{\langle a,non_visible\rangle} = \emptyset \\ \mathcal{R}_2 & otherwise \end{cases}$$

$\mathcal{R}_2$:

$$\bigsqcap_{\langle x,non_visible\rangle \in COND_{exit(n)}^{\langle a,non_visible\rangle}} \left(COND_{call(n)}^{back\text{-}bind'_{call(n)}(\langle x,non_visible\rangle,b)} \right)$$

(c) a and b are not visible in the called procedure

$$COND_n^{\langle a,b\rangle} = COND_{call(n)}^{\langle a,b\rangle}$$

5. n is statement node

(a) n is not an assignment to a pointer

$$COND_n^{\langle a,b\rangle} = \bigsqcap_{\ll m,n\gg \in \mathcal{E}} \left(COND_m^{\langle a,b\rangle} \right)$$

(b) n is the assignment "p = q" (p,q pointers)

*i. if a = b = *p*

$$COND_n^{\langle *p,*p\rangle} = \bigsqcap_{\ll m,n\gg \in \mathcal{E}} \left(COND_m^{\langle *q,*q\rangle} \right)$$

*ii. if a = *p [a ≠ b]*

$$COND_n^{\langle *p,b\rangle} = \bigsqcap_{\ll m,n\gg \in \mathcal{E}} \left(COND_m^{\langle *q,b\rangle} \right)$$

*iii. if a ≠ *p and b ≠ *p*

$$COND_n^{\langle a,b\rangle} = \bigsqcap_{\ll m,n\gg \in \mathcal{E}} \left(COND_m^{\langle a,b\rangle} \right)$$

(c) n is the assignment "p = &v" (p a pointer)

*i. if a = b = *p*

$$COND_n^{\langle *p,*p\rangle} = \emptyset$$

*ii. if a = *p [a ≠ b]*

$$COND_n^{\langle *p,b\rangle} = \bigsqcap_{\ll m,n\gg \in \mathcal{E}} \left(COND_m^{\langle v,b\rangle} \right)$$

*iii. if a ≠ *p and b ≠ *p*

$$COND_n^{\langle a,b\rangle} = \bigsqcap_{\ll m,n\gg \in \mathcal{E}} \left(COND_m^{\langle a,b\rangle} \right)$$

(d) n is the assignment "p = malloc()" or "p = NULL" (p a pointer)

i. *if* $a = b = *p$

$$COND_n^{\langle *p, *p \rangle} = \begin{cases} \emptyset & \text{if } n \text{ is } \text{“}p = malloc()\text{”} \\ false & \text{if } n \text{ is } \text{“}p = NULL\text{”} \end{cases}$$

ii. *if* $a = *p$ $[a \neq b]$

$$COND_n^{\langle *p, b \rangle} = false$$

iii. *if* $a \neq *p$ *and* $b \neq *p$

$$COND_n^{\langle a,b \rangle} = \sqcap_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle a,b \rangle} \right)$$

Proof:

$COND_n^{false} = false$ for all nodes n in the ICFG.

This is just a notational convenience.

$COND_n^{\langle o,o \rangle} = \emptyset$ for all nodes n in the ICFG, all non-dereferenced pointer object names $o \in \mathcal{O}$.

Reflexive aliases always hold between variables.

Consider $COND_n^{\langle a,b \rangle}$, n a node of the ICFG=($\mathcal{N}$,$\mathcal{E}$,ρ) and $\langle a, b \rangle \in \mathcal{A}$ ($a \neq b$ or a is a dereferenced pointer):

1. n is an entry node

   $COND_n^{\langle a,b \rangle} = \{\langle a, b \rangle\}$. Clearly this is the minimal set of assumed must aliases for $\langle a, b \rangle$ to hold at n.

2. n is an exit node

   For all paths $\rho n_1 n_2 ... n_i$, $\langle a, b \rangle$ holds on $\rho n_1 n_2 ... n_i$ iff $\langle a, b \rangle$ holds on $\rho n_1 ... n_{i-1}$.
   By Lemma C.4.1 (p. 194), $COND_n^{\langle a,b \rangle} = \sqcap_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle a,b \rangle} \right)$.

3. n is a call node

   For all paths $\rho n_1 n_2 ... n_i$, $\langle a, b \rangle$ holds on $\rho n_1 n_2 ... n_i$ iff $\langle a, b \rangle$ holds on $\rho n_1 ... n_{i-1}$.
   By Lemma C.4.1 (p. 194), $COND_n^{\langle a,b \rangle} = \sqcap_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle a,b \rangle} \right)$.

4. n is a return node

(a) a and b are both visible in the called procedure

- $COND_n^{\langle a,b\rangle} = \mathit{false}$ because $COND_{exit(n)}^{\langle a,b\rangle} = \mathit{false}$

  If $\langle a,b\rangle$ can under no circumstances hold on exit of a procedure then it can never hold on return from that procedure.

- $COND_n^{\langle a,b\rangle} = \emptyset$ because $COND_{exit(n)}^{\langle a,b\rangle} = \emptyset$

  $COND_{exit(n)}^{\langle a,b\rangle} = \emptyset$ means that whenever the procedure containing $exit(n)$ is called, a is aliased at b on exit and thus a and b must also be aliased whenever n is executed.

- $COND_n^{\langle a,b\rangle} = \bigsqcap_{A \in COND_{exit(n)}^{\langle a,b\rangle}} \left(COND_{call(n)}^{\textit{back-bind}_{call(n)}(A)} \right)$

  $\langle a,b\rangle$ holds on a path $\rho...[exit(n)]n$ iff it holds on $\rho...[exit(n)]$. By definition of $COND$, $\langle a,b\rangle$ must hold at $exit(n)$ iff $COND_{exit(n)}^{\langle a,b\rangle}$ holds at $entry(exit(n))$. By Lemma C.1.1 (p. 160), $COND_{exit(n)}^{\langle a,b\rangle}$ holds at $entry(exit(n))$ iff all aliases $\{\textit{back-bind}_{call(n)}(A) \mid A \in COND_{exit(n)}^{\langle a,b\rangle}\}$ must hold at $call(n)$. This is true iff all aliases in $\bigsqcap_{A \in COND_{exit(n)}^{\langle a,b\rangle}} \left(COND_{call(n)}^{\textit{back-bind}_{call(n)}(A)} \right)$ hold at $entry(n)$.

(b) a is visible in the called procedure, but b is not

- $COND_n^{\langle a,b\rangle} = \mathit{false}$ because $COND_{exit(n)}^{\langle a,non_visible\rangle} = \mathit{false}$

  If $\langle a, non_visible\rangle$ can under no circumstances hold on exit of a procedure then $\langle a, nv\rangle$, nv a non-visible object name, can never hold on return from that procedure.

- $COND_n^{\langle a,b\rangle} = \bigsqcap_{\langle x, non_visible\rangle \in COND_{exit(n)}^{\langle a,non_visible\rangle}} \left(COND_{call(n)}^{\textit{back-bind}'_{call(n)}(\langle x,non_visible\rangle,b)} \right)$

  $\langle a,b\rangle$ holds on a path $\rho...[exit(n)]n$ iff $\langle a, non_visible\rangle$ holds on $\rho...[exit(n)]$ where "$non_visible$" is the non-visible object name "b". By definition of $COND$, $\langle a, non_visible\rangle$ must hold at $exit(n)$ iff $COND_{exit(n)}^{\langle a,non_visible\rangle}$ holds at $entry(exit(n))$. By Lemma C.1.1 (p. 160), $COND_{exit(n)}^{\langle a,non_visible\rangle}$ holds at $entry(exit(n))$ iff all aliases $\left\{ \textit{back-bind}'_{call(n)}(A,b) \,\middle|\, A \in COND_{exit(n)}^{\langle a,non_visible\rangle} \right\}$ must hold at $call(n)$. This is true iff all aliases in

$$\bigsqcap_{A \in COND_{exit(n)}^{\langle a, non_visible \rangle}} \left(COND_{call(n)}^{back\text{-}bind'_{call(n)}(A,b)} \right) \text{ hold at } entry(n).$$

(c) a and b are not visible in the called procedure

For all paths $\rho n_1 ... n_i$ ($n_i = n$), $\langle a, b \rangle$ holds at n_i iff $\langle a, b \rangle$ holds at $call(n_i)$. By an argument similar to the proof of Lemma C.4.1 (p. 194)(replace n_{i-1} with $call(n_i)$), $COND_n^{\langle a,b \rangle} = COND_{call(n)}^{\langle a,b \rangle}$.

5. n is statement node

(a) n is not an assignment to a pointer

For all paths $\rho n_1 n_2 ... n_i$, $\langle a, b \rangle$ holds on $\rho n_1 n_2 ... n_i$ iff $\langle a, b \rangle$ holds on $\rho n_1 ... n_{i-1}$. By Lemma C.4.1 (p. 194), $COND_n^{\langle a,b \rangle} = \bigsqcap_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle a,b \rangle} \right)$.

(b) n is the assignment "$p = q$" (p,q pointers)

i. if $a = b = *p$

For all paths $\rho n_1 n_2 ... n_i$, $\langle *p, *p \rangle$ holds on $\rho n_1 n_2 ... n_i$ iff $\langle *q, *q \rangle$ holds on $\rho n_1 ... n_{i-1}$.

By Lemma C.4.1 (p. 194), $COND_n^{\langle *p,*p \rangle} = \bigsqcap_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle *q,*q \rangle} \right)$.

ii. if $a = *p$ $[a \neq b]$

For all paths $\rho n_1 n_2 ... n_i$, $\langle *p, b \rangle$ holds on $\rho n_1 n_2 ... n_i$ iff $\langle *q, b \rangle$ holds on $\rho n_1 ... n_{i-1}$.

By Lemma C.4.1 (p. 194), $COND_n^{\langle *p,b \rangle} = \bigsqcap_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle *q,b \rangle} \right)$.

iii. if $a \neq *p$ and $b \neq *p$

For all paths $\rho n_1 n_2 ... n_i$, $\langle a, b \rangle$ holds on $\rho n_1 n_2 ... n_i$ iff $\langle a, b \rangle$ holds on $\rho n_1 ... n_{i-1}$.

By Lemma C.4.1 (p. 194), $COND_n^{\langle a,b \rangle} = \bigsqcap_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle a,b \rangle} \right)$.

(c) n is the assignment "$p = \&v$" (p a pointer)

i. if $a = b = *p$

On all paths $\rho n_1 ... n$ $\langle *p, *p \rangle$ has to hold. Thus $COND_n^{\langle *p,*p \rangle} = \emptyset$.

ii. if $a = *p$ $[a \neq b]$

For all paths $\rho n_1 n_2 ... n_i$, $\langle *p, b \rangle$ holds on $\rho n_1 n_2 ... n_i$ iff $\langle v, b \rangle$ holds on $\rho n_1 ... n_{i-1}$.

By Lemma C.4.1 (p. 194), $COND_n^{\langle *p,b\rangle} = \sqcap_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle v,b\rangle} \right)$.

iii. if $a \neq *p$ and $b \neq *p$

For all paths $\rho n_1 n_2 ... n_i$, $\langle a,b\rangle$ holds on $\rho n_1 n_2 ... n_i$ iff $\langle a,b\rangle$ holds on $\rho n_1 ... n_{i-1}$.

By Lemma C.4.1 (p. 194), $COND_n^{\langle a,b\rangle} = \sqcap_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle a,b\rangle} \right)$.

(d) n is the assignment "$p = malloc()$" or "$p = NULL$" (p a pointer)

i. if $a = b = *p$

- n is "$p = malloc()$"

  On all paths $\rho n_1 ... n$ $\langle *p, *p\rangle$ has to hold. Thus $COND_n^{\langle *p,*p\rangle} = \emptyset$.

- n is "$p = NULL$"

  On all paths $\rho n_1 ... n$ $\langle *p, *p\rangle$ does not hold. Thus $COND_n^{\langle *p,*p\rangle} = \mathit{false}$.

ii. if $a = *p$ $[a \neq b]$

On all paths $\rho n_1 ... n$ $\langle *p, b\rangle$ does not hold. Thus $COND_n^{\langle *p,b\rangle} = \mathit{false}$.

iii. if $a \neq *p$ and $b \neq *p$

For all paths $\rho n_1 n_2 ... n_i$, $\langle a,b\rangle$ holds on $\rho n_1 n_2 ... n_i$ iff $\langle a,b\rangle$ holds on $\rho n_1 ... n_{i-1}$.

By Lemma C.4.1 (p. 194), $COND_n^{\langle a,b\rangle} = \sqcap_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle a,b\rangle} \right)$.

□

C.4.5 Proof of Correctness of Precise Conditional Must Alias Algorithm

---

Lemma C.4.3 *For all nodes n in the ICFG and for all alias-pair, the maximal fixed point of must-holds*$(n, \textit{alias-pair}) = COND_n^{\textit{alias-pair}}$.

We show *must-holds*$(n, \textit{alias-pair}) \sqsubseteq COND_n^{\textit{alias-pair}}$ by Lemma C.4.4 (p. 201) and Lemma C.4.5 (p. 201) and *must-holds*$(n, \textit{alias-pair}) \sqsupseteq COND_n^{\textit{alias-pair}}$ by Lemma

C.4.6 (p. 206).

□

Lemma C.4.4 *For every n and $\langle a,b\rangle \in \mathcal{POSSIBLE\text{-}ALIASES}$, let $A_n^{\langle a,b\rangle}$ in $\mathcal{L}$ be such that $A_n^{\langle a,b\rangle}$ is a set B, where assuming B at the entry node of the procedure containing n insures that $\langle a,b\rangle$ holds on all paths to n, and is false, if no such set B exists. Formally, define $A_n^{\langle a,b\rangle}$ so that on all paths $P = \mathit{entry}(n_i)n_2n_3...n_i(n_i = n)$, there exists an $A' \in \mathcal{L}$ such that $A_n^{\langle a,b\rangle} \sqsubseteq A'$. If $A' = \mathit{false}$ then $\langle a,b\rangle$ can never hold after the execution of P, otherwise, when all aliases in the set A' hold at $\mathit{entry}(n_i)$, the execution of P forces $\langle a,b\rangle$ to hold at n_i. Claim: $COND_n^{\langle a,b\rangle} \sqsupseteq A_n^{\langle a,b\rangle}$.*

Assume that for some n and $\langle a,b\rangle$ the Lemma was false. Thus $COND_n^{\langle a,b\rangle} \not\sqsupseteq A_n^{\langle a,b\rangle}$, but by definition $COND_n^{\langle a,b\rangle}$ is a unique minimal set B such that assuming B at the entry node of the procedure containing n insures that $\langle a,b\rangle$ holds on all paths to n and is false if no such set exists. Contradiction[83].

□

Lemma C.4.5 *Below consider must-holds to be the maximum fixed point of the must-holds relation defined in Appendix C.4.2.*

> *Let $\mathit{must\text{-}holds}(n,\langle a,b\rangle) = A$. For all paths $P = \mathit{entry}(n_i)n_2n_3...n_i(n_i = n)$, there exists an $A' \in \mathcal{L}$ such that if $A' = \mathit{false}$ then $\langle a,b\rangle$ can never hold after the execution of P, otherwise all aliases in the set A' hold at $\mathit{entry}(n_i)$ and the execution of P forces $\langle a,b\rangle$ to hold at n_i and $A \sqsubseteq A'$.*

Proof by induction on i, the path length.

basis: $i = 1$

[83] Remember, $a \sqsubseteq b$ iff $(a = \mathit{false})$ or $(a \supseteq b)$

The path is simply n_1, where n_1 is an entry node. By definition

- if $a = b$ and a is not a dereferenced pointer

  $\mathit{must\text{-}holds}(n, \langle a, a\rangle) = \emptyset$. Lemma must hold because reflexive aliases always hold ($A' = \emptyset$).

- Otherwise

  $\mathit{must\text{-}holds}(n, \langle a, b\rangle) = \{\langle a, b\rangle\}$. Lemma must hold ($A' = \{\langle a, b\rangle\}$).

induction hypothesis: Lemma C.4.5 holds for $j < i$.

induction step:

Let $\mathcal{P} = \mathit{entry}(n_i)n_2...n_i$.

Let $\mathcal{P}_{i-1} = \mathit{entry}(n_i)n_2...n_{i-1}$.

Consider $\mathit{must\text{-}holds}(n_i, \langle a, b\rangle)$

if $a = b$ and a is not a dereferenced pointer then

$\mathit{must\text{-}holds}(n, \langle a, a\rangle) = \emptyset$. Lemma must hold because reflexive aliases always hold ($A' = \emptyset$).

otherwise

1. n_i is an entry node

   Only possible if $i = 1$. See basis.

2. n_i is an exit node

   $\mathit{must\text{-}holds}(n_i, \langle a, b\rangle) = \bigsqcap_{\ll m,n_i \gg \in \mathcal{E}} (\mathit{must\text{-}holds}(m, \langle a, b\rangle))$.

   Argument C.4.1 *By the induction hypothesis, for $\mathcal{P}_{i-1}$ there is an $A'_{i-1} \sqsupseteq \mathit{must\text{-}holds}(n_{i-1}, \langle a, b\rangle)$. $\mathit{must\text{-}holds}(n_{i-1}, \langle a, b\rangle) \sqsupseteq \mathit{must\text{-}holds}(n_i, \langle a, b\rangle)$ by definition of must-holds. A'_{i-1} must also be valid for A' on $\mathcal{P}_i$. Thus $A' = A'_{i-1} \sqsupseteq \mathit{must\text{-}holds}(n_{i-1}, \langle a, b\rangle) \sqsupseteq \mathit{must\text{-}holds}(n_i, \langle a, b\rangle)$.*

3. n_i is a call node

   $\mathit{must\text{-}holds}(n_i, \langle a, b\rangle) = \bigsqcap_{\ll m,n_i \gg \in \mathcal{E}} (\mathit{must\text{-}holds}(m, \langle a, b\rangle))$.

   Lemma holds by *Argument C.4.1 (p. 202)*.

4. n_i is a return node

   (a) a and b are both visible in the called procedure

   - $\mathit{must\text{-}holds}(n_i, \langle a, b\rangle) = \mathit{false}$ because $\mathit{must\text{-}holds}(n_{i-1}, \langle a, b\rangle) = \mathit{false}$.

     Lemma must hold because false is less than everything.

   - $\mathit{must\text{-}holds}(n_i, \langle a, b\rangle) = \emptyset$ because $\mathit{must\text{-}holds}(n_{i-1}, \langle a, b\rangle) = \emptyset$.

     By induction, whenever the procedure containing $\mathit{exit}(n)$ is called, a is aliased at b on exit and thus a and b must also be aliased whenever n is executed. Thus $A' = \emptyset$ satisfies the Lemma.

   - (1) $\mathit{must\text{-}holds}(n_i, \langle a, b\rangle) =$

     $\bigsqcap_{A \in \mathit{must\text{-}holds}(\mathit{exit}(n_i), \langle a,b\rangle)} \left(\mathit{must\text{-}holds}(\mathit{call}(n_i), \mathit{back\text{-}bind}_{\mathit{call}(n_i)}(A))\right)$

     (2) By the induction hypothesis there is an

     $A'_{i-1} \sqsupseteq \mathit{must\text{-}holds}(n_{i-1}, \langle a, b\rangle)$.

     Let n_j be the call node for the return node n_i $(j < i)$.

     (3) By the induction hypothesis for all $\langle c, d\rangle$ there is an

     $A'^{\langle c,d\rangle}_j \sqsupseteq \mathit{must\text{-}holds}(n_j, \langle c, d\rangle)$.

     Consider $A' = \bigsqcap_{X \in A'_{i-1}} \left(A'^{\mathit{back\text{-}bind}_{\mathit{call}(n_i)}(X)}_j\right)$.

     Certainly, $A' \sqsupseteq \mathit{must\text{-}holds}(n_i, \langle a, b\rangle)$ because of (1), (2), and (3) and Lemma C.1.1 (p. 160).

     Also A' holding at $\mathit{entry}(n_i)$ and executing $\mathcal{P}_i$ must force $\langle a, b\rangle$ to hold at n_i because of (2) and (3).

   (b) a is visible in the called procedure, but b is not

   - $\mathit{must\text{-}holds}(n_i, \langle a, b\rangle) = \mathit{false}$ because

     $\mathit{must\text{-}holds}(n_{i-1}, \langle a, \mathit{non_visible}\rangle) = \mathit{false}$.

     Lemma must hold because false is less than everything.

   - (1) $\mathit{must\text{-}holds}(n_i, \langle a, b\rangle) =$

     $\bigsqcap_{B \in \mathit{must\text{-}holds}(\mathit{exit}(n_i), \langle a, \mathit{non_visible}\rangle)} \left(\mathit{must\text{-}holds}(\mathit{call}(n_i), \mathit{back\text{-}bind}'_{\mathit{call}(n_i)}(B, b))\right)$

     (2) By the induction hypothesis there is an

     $A'_{i-1} \sqsupseteq \mathit{must\text{-}holds}(n_{i-1}, \langle a, \mathit{non_visible}\rangle)$.

     Let n_j be the call node for the return node n_i $(j < i)$.

(3) By the induction hypothesis for all $\langle c, d\rangle$ there is an $A'^{\langle c,d\rangle}_j \sqsupseteq \mathit{must\text{-}holds}(n_j, \langle c, d\rangle)$.
Consider $A' = \bigsqcap_{B \in A'_{i-1}} A'^{\mathit{back\text{-}bind}'_{call(n_i)}(B,b)}_j$.
Certainly, $A' \sqsupseteq \mathit{must\text{-}holds}(n_i, \langle a, b\rangle)$ because of (1), (2), and (3) and Lemma C.1.1 (p. 160).
Also A' holding at $\mathit{entry}(n_i)$ and executing $\mathcal{P}_i$ must force $\langle a, b\rangle$ to hold at n_i because of (2) and (3).

(c) a and b are both not visible in the called procedure
Lemma holds by a trivial modification of *Argument C.4.1 (p. 202)*.

5. n_i is a statement node

(a) n_i is not an assignment to a pointer
$\mathit{must\text{-}holds}(n_i, \langle a, b\rangle) = \bigsqcap_{\ll m,n_i \gg \in \mathcal{E}} (\mathit{must\text{-}holds}(m, \langle a, b\rangle))$.
Lemma holds by *Argument C.4.1 (p. 202)*.

(b) n_i is the assignment "$p = q$" (p,q pointers)

i. if $a = b = *p$
$\mathit{must\text{-}holds}(n_i, \langle *p, *p\rangle) = \bigsqcap_{\ll m,n_i \gg \in \mathcal{E}} (\mathit{must\text{-}holds}(m, \langle *q, *q\rangle))$
By the induction hypothesis, for $\mathcal{P}_{i-1}$ there is an $A'_{i-1} \sqsupseteq \mathit{must\text{-}holds}(n_{i-1}, \langle *q, *q\rangle)$. $\mathit{must\text{-}holds}(n_{i-1}, \langle *q, *q\rangle) \sqsupseteq \mathit{must\text{-}holds}(n_i, \langle *p, *p\rangle)$ by definition of *must-holds*. A'_{i-1} must also be valid for A' on $\mathcal{P}_i$ for $\mathit{must\text{-}holds}(n_i, \langle *p, *p\rangle)$ because $A' = A'_{i-1} \sqsupseteq \mathit{must\text{-}holds}(n_{i-1}, \langle *q, *q\rangle) \sqsupseteq \mathit{must\text{-}holds}(n_i, \langle *p, *p\rangle)$.

ii. if $a = *p$ $[a \neq b]$
$\mathit{must\text{-}holds}(n_i, \langle *p, b\rangle) = \bigsqcap_{\ll m,n_i \gg \in \mathcal{E}} (\mathit{must\text{-}holds}(m, \langle *q, b\rangle))$
By the induction hypothesis, for $\mathcal{P}_{i-1}$ there is an $A'_{i-1} \sqsupseteq \mathit{must\text{-}holds}(n_{i-1}, \langle *q, b\rangle)$. $\mathit{must\text{-}holds}(n_{i-1}, \langle *q, b\rangle) \sqsupseteq \mathit{must\text{-}holds}(n_i, \langle *p, b\rangle)$ by definition of *must-holds*. A'_{i-1} must also be valid for A' on $\mathcal{P}_i$ for $\mathit{must\text{-}holds}(n_i, \langle *p, b\rangle)$ because $A' = A'_{i-1} \sqsupseteq \mathit{must\text{-}holds}(n_{i-1}, \langle *q, b\rangle) \sqsupseteq \mathit{must\text{-}holds}(n_i, \langle *p, b\rangle)$.

iii. if $a \neq *p$ and $b \neq *p$

$must\text{-}holds(n_i, \langle a, b \rangle) = \bigsqcap_{\ll m,n_i \gg \in \mathcal{E}} (must\text{-}holds(m, \langle a, b \rangle))$

Lemma holds by *Argument C.4.1 (p. 202)*.

(c) n_i is the assignment "$p = \&v$" (p a pointer)

i. if $a = b = *p$

$must\text{-}holds(n_i, \langle *p, *p \rangle) = \emptyset$, but $\langle *p, *p \rangle$ holds on all paths to n_i (even with no assumptions at $entry(n_i)$), thus the Lemma holds ($A' = \emptyset$).

ii. if $a = *p$ $[a \neq b]$

$must\text{-}holds(n_i, \langle *p, b \rangle) = \bigsqcap_{\ll m,n_i \gg \in \mathcal{E}} (must\text{-}holds(m, \langle v, b \rangle))$

By the induction hypothesis, for $\mathcal{P}_{i-1}$ there is an $A'_{i-1} \sqsupseteq must\text{-}holds($ $n_{i-1}, \langle v, b \rangle)$. $must\text{-}holds(n_{i-1}, \langle v, b \rangle) \sqsupseteq must\text{-}holds(n_i, \langle *p, b \rangle)$ by definition of *must-holds*. A'_{i-1} must also be valid for A' on $\mathcal{P}_i$ for $must\text{-}holds(n_i, \langle *p, b \rangle)$ because $A' = A'_{i-1} \sqsupseteq must\text{-}holds(n_{i-1}, \langle v, b \rangle) \sqsupseteq must\text{-}holds(n_i, \langle *p, b \rangle)$.

iii. if $a \neq *p$ and $b \neq *p$

$must\text{-}holds(n_i, \langle a, b \rangle) = \bigsqcap_{\ll m,n_i \gg \in \mathcal{E}} (must\text{-}holds(m, \langle a, b \rangle))$

Lemma holds by *Argument C.4.1 (p. 202)*.

(d) n_i is the assignment "$p = malloc()$" or "$p = NULL$" (p a pointer)

i. if $a = b = *p$

- n_i is "$p = malloc()$"

  $must\text{-}holds(n_i, \langle *p, *p \rangle) = \emptyset$, but $\langle *p, *p \rangle$ holds on all paths to n_i (even with no assumptions at $entry(n_i)$), thus the Lemma holds ($A' = \emptyset$).

- n_i is "$p = NULL$"

  $must\text{-}holds(n, \langle *p, b \rangle) = false$.

  Lemma must hold because $false$ is less than everything.

ii. if $a = *p$ $[a \neq b]$

$must\text{-}holds(n, \langle *p, b \rangle) = false$.

Lemma must hold because $false$ is less than everything.

iii. if $a \neq *p$ and $b \neq *p$

$must\text{-}holds(n_i, \langle a, b\rangle) = \sqcap_{\ll m,n_i \gg \in \mathcal{E}} \ (must\text{-}holds(m, \langle a, b\rangle))$

Lemma holds by *Argument C.4.1 (p. 202)*.

□

Lemma C.4.6 *Below consider must-holds to be the maximum fixed point of the must-holds relation defined in Appendix C.4.2.*

For all $n \in ICFG$ *and* $\langle a, b\rangle \in \mathcal{POSSIBLE\text{-}ALIASES}$,
$must\text{-}holds(n, \langle a, b\rangle) \sqsupseteq COND_n^{\langle a,b\rangle}$.

Proof by induction on j, number of iterations of the maximum fixed point algorithm.

Let $must\text{-}holds^j(n, \langle a, b\rangle)$ be the value of $must\text{-}holds(n, \langle a, b\rangle)$ on the j^{th} iteration of the fixed point algorithm.

basis: $j = 0$

$must\text{-}holds^0(n, \langle a, b\rangle)$ for all n in ICFG and all $\langle a, b\rangle \in \mathcal{A}$ is initialized to $\emptyset = \top$. Lemma holds.

induction hypothesis: Lemma C.4.6 holds for $k < j$.
induction step: Say the value of $must\text{-}holds^j(n, \langle a, b\rangle)$ changed at iteration j.
if $a = b$ and a is not a dereferenced pointer then

$must\text{-}holds^j(n, \langle a, a\rangle) = \emptyset$. Lemma must hold because $\emptyset = \top$.

otherwise

1. n is an entry node
   $must\text{-}holds^j(n, \langle a, b\rangle) = \{\langle a, b\rangle\}$. By Lemma C.4.2 (p. 197) $COND_n^{\langle a,b\rangle} = \{\langle a, b\rangle\}$ and thus the Lemma holds.

2. n is an exit node

   $must\text{-}holds^j(n, \langle a, b\rangle) = \bigsqcap_{\ll m,n \gg \in \mathcal{E}} must\text{-}holds^{j-1}(m, \langle a, b\rangle)$.

   Argument C.4.2 *By the induction hypothesis for all* $\ll m, n \gg \in \mathcal{E}$,
   $must\text{-}holds^{j-1}(m, \langle a, b\rangle) \sqsupseteq COND_m^{\langle a,b\rangle}$. *By Lemma C.4.2 (p. 197),*
   $COND_n^{\langle a,b\rangle} = \bigsqcap_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle a,b\rangle}\right)$ *and by definition,*
   $must\text{-}holds^j(n, \langle a, b\rangle) = \bigsqcap_{\ll m,n \gg \in \mathcal{E}} \left(must\text{-}holds^{j-1}(m, \langle a, b\rangle)\right)$
   $\sqsupseteq \bigsqcap_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle a,b\rangle}\right) = COND_n^{\langle a,b\rangle}$. *Thus Lemma C.4.6 must hold.*

3. n is a call node

   $must\text{-}holds^j(n, \langle a, b\rangle) = \bigsqcap_{\ll m,n \gg \in \mathcal{E}} must\text{-}holds^{j-1}(m, \langle a, b\rangle)$.

   Lemma holds by *Argument C.4.2 (p. 207).*

4. n is a return node

   (a) a and b are both visible in the called procedure

   - $must\text{-}holds^j(n_i, \langle a, b\rangle) = false$

     because $must\text{-}holds^{j-1}(exit(n), \langle a, b\rangle) = false$.

     By the induction hypothesis,

     $false = must\text{-}holds^{j-1}(exit(n), \langle a, b\rangle) \sqsupseteq COND_{exit(n)}^{\langle a,b\rangle}$.

     Thus $COND_{exit(n)}^{\langle a,b\rangle} = false$, which means by Lemma C.4.2 (p. 197), $COND_n^{\langle a,b\rangle}$ must be $false$.

   - $must\text{-}holds^j(n_i, \langle a, b\rangle) = \emptyset$

     because $must\text{-}holds^{j-1}(exit(n), \langle a, b\rangle) = \emptyset$.

     Lemma must hold because $\emptyset = \top$.

   - $must\text{-}holds^j(n, \langle a, b\rangle)$ = [by definition]

     $\bigsqcap_{A \in must\text{-}holds^{j-1}(exit(n), \langle a,b\rangle)} \left(must\text{-}holds^{j-1}(call(n), back\text{-}bind_{call(n)}(A))\right)$
     $\sqsupseteq$ [by induction] $\bigsqcap_{A \in COND_{exit(n)}^{\langle a,b\rangle}} \left(must\text{-}holds^{j-1}(call(n), back\text{-}bind_{call(n)}(A))\right)$
     $\sqsupseteq$ [by induction] $\bigsqcap_{A \in COND_{exit(n)}^{\langle a,b\rangle}} \left(COND_{call(n)}^{back\text{-}bind_{call(n)}(A)}\right)$
     = [by Lemma C.4.2 (p. 197)] $COND_n^{\langle a,b\rangle}$

     Thus lemma must hold.

   (b) a is visible in the called procedure, but b is not

- $must\text{-}holds^{j}(n_i, \langle a, b\rangle) = false$

  because $must\text{-}holds^{j-1}(exit(n), \langle a, non_visible\rangle) = false$.

  By the induction hypothesis, $false =$

  $must\text{-}holds^{j-1}(exit(n), \langle a, non_visible\rangle) \sqsupseteq COND^{\langle a,non_visible\rangle}_{exit(n)}$.

  Thus $COND^{\langle a,non_visible\rangle}_{exit(n)} = false$ which means (by Lemma C.4.2 (p. 197)) $COND^{\langle a,b\rangle}_{n}$ must be $false$.

- $must\text{-}holds^{j}(n, \langle a, b\rangle)$ = [by definition]

$$\prod_{\substack{B \in \\ must\text{-}holds^{j-1}(exit(n), \langle a, non_visible\rangle)}} \left(must\text{-}holds^{j-1}(call(n), back\text{-}bind_{call(n)}(B, b))\right)$$

  $\sqsupseteq$ [by induction]
  $\prod_{B \in COND^{\langle a,non_visible\rangle}_{exit(n)}} \left(must\text{-}holds^{j-1}(call(n), back\text{-}bind_{call(n)}(B, b))\right)$

  $\sqsupseteq$ [by induction] $\prod_{B \in COND^{<a,non_visible>}_{exit(n)}} \left(COND^{back\text{-}bind_{call(n)}(B,b)}_{call(n)}\right)$

  = [by Lemma C.4.2 (p. 197)] $COND^{\langle a,b\rangle}_{n}$

  Thus Lemma must hold.

(c) a and b are not visible in the called procedure

$must\text{-}holds^{j}(n, \langle a, b\rangle)$ = [by definition]

$must\text{-}holds^{j-1}(call(n), \langle a, b\rangle) \sqsupseteq$ [by induction]
$COND^{\langle a,b\rangle}_{call(n)}$ = [by Lemma C.4.2 (p. 197)] $COND^{\langle a,b\rangle}_{n}$

Thus Lemma holds.

5. n is a statement node

(a) n is not an assignment to a pointer

$must\text{-}holds^{j}(n, \langle a, b\rangle) = \prod_{\ll m,n \gg \in \mathcal{E}} \left(must\text{-}holds^{j-1}(m, \langle a, b\rangle)\right)$.

Lemma holds by *Argument C.4.2 (p. 207)*.

(b) n is the assignment "$p = q$" (p,q pointers)

i. if $a = b = *p$

$must\text{-}holds^{j}(n, \langle *p, *p\rangle) = \prod_{\ll m,n \gg \in \mathcal{E}} \left(must\text{-}hold^{j-1}(m, \langle *q, *q\rangle)\right)$

By the induction hypothesis for all $\ll$m,n$\gg \in \mathcal{E}$,

$must\text{-}holds^{j-1}(m, \langle *q, *q \rangle) \sqsupseteq COND_m^{\langle *q, *q \rangle}$.

By Lemma C.4.2 (p. 197),

$COND_n^{\langle *p, *p \rangle} = \bigsqcap_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle *q, *q \rangle} \right)$.

Thus Lemma must hold.

ii. if $a = *p$ $[a \neq b]$

$must\text{-}holds^{j}(n, \langle *p, b \rangle) = \bigsqcap_{\ll m,n \gg \in \mathcal{E}} (must\text{-}hold^{j-1}(m, \langle *q, b \rangle))$

By the induction hypothesis for all ≪m,n≫ $\in \mathcal{E}$,

$must\text{-}holds^{j-1}(m, \langle *q, b \rangle) \sqsupseteq COND_m^{\langle *q, b \rangle}$.

By Lemma C.4.2 (p. 197),

$COND_n^{\langle *p, b \rangle} = \bigsqcap_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle *q, b \rangle} \right)$.

Thus Lemma must hold.

iii. if $a \neq *p$ and $b \neq *p$

$must\text{-}holds^{j}(n, \langle a, b \rangle) = \bigsqcap_{\ll m,n \gg \in \mathcal{E}} (must\text{-}hold^{j-1}(m, \langle a, b \rangle))$

Lemma holds by *Argument C.4.2 (p. 207)*.

(c) n is the assignment "$p = \&v$" (p a pointer)

i. if $a = b = *p$

$must\text{-}holds^{j}(n, \langle *p, *p \rangle) = \emptyset$

By Lemma C.4.2 (p. 197), $COND_n^{\langle *p, *p \rangle} = \emptyset$

Thus Lemma must hold.

ii. if $a = *p$ $[a \neq b]$

$must\text{-}holds^{j}(n, \langle *p, b \rangle) = \bigsqcap_{\ll m,n \gg \in \mathcal{E}} (must\text{-}hold^{j-1}(m, \langle v, b \rangle))$

By the induction hypothesis for all ≪m,n≫ $\in \mathcal{E}$,

$must\text{-}holds^{j-1}(m, \langle v, b \rangle) \sqsupseteq COND_m^{\langle v, b \rangle}$.

By Lemma C.4.2 (p. 197), $COND_n^{\langle *p, b \rangle} = \bigsqcap_{\ll m,n \gg \in \mathcal{E}} \left(COND_m^{\langle v, b \rangle} \right)$.

Thus Lemma must hold.

iii. if $a \neq *p$ and $b \neq *p$

$must\text{-}holds^{j}(n, \langle a, b \rangle) = \bigsqcap_{\ll m,n \gg \in \mathcal{E}} (must\text{-}hold^{j-1}(m, \langle a, b \rangle))$

Lemma holds by *Argument C.4.2 (p. 207)*.

(d) n is the assignment "$p = malloc()$" or "$p = NULL$ (p a pointer)

i. if $a = b = *p$

- n is "$p = malloc()$"

  $must\text{-}holds^j(n, \langle *p, *p \rangle) = \emptyset$

  By Lemma C.4.2 (p. 197), $COND_n^{\langle *p, *p \rangle} = \emptyset$

  Thus Lemma must hold.

- n is "$p = NULL$"

  $must\text{-}holds^j(n, \langle *p, *p \rangle) = false$

  By Lemma C.4.2 (p. 197), $COND_n^{\langle *p, *p \rangle} = false$

  Thus Lemma must hold.

ii. if $a = *p$ $[a \neq b]$

$must\text{-}holds^j(n, \langle *p, b \rangle) = false$

By Lemma C.4.2 (p. 197), $COND_n^{\langle *p, b \rangle} = false$

Thus Lemma must hold.

iii. if $a \neq *p$ and $b \neq *p$

$must\text{-}holds^j(n, \langle a, b \rangle) = \sqcap_{\ll m,n \gg \in \mathcal{E}} \ (must\text{-}hold^{j-1}(m, \langle a, b \rangle))$

Lemma holds by *Argument C.4.2 (p. 207)*.

□

Lemma C.4.7 *Fixed point calculation of must-holds is polynomial time.*

Each variable in the program contributes one *object name* which may be involved in aliases; for each pointer p, $*p$ may have aliases and for each non-pointer v, v itself may be aliased. Thus the number of possible aliases pairs in a program is $\mid variables \mid^2$. Since a program can use at most $O(\mid \mathcal{N} \mid)$ variables, the number of possible aliases is $O(\mid \mathcal{N} \mid^2)$.

For any $n \in$ ICFG and any $\langle a, b \rangle \in \mathcal{POSSIBLE\text{-}ALIASES}$ and all iterations i, $must\text{-}holds(n, \langle a, b \rangle)$ at iteration $i \sqsupseteq must\text{-}holds(n, \langle a, b \rangle)$ at iteration $i+1$. Thus $must\text{-}holds(n, \langle a, b \rangle)$ can change its value at most $O(\mid \mathcal{N} \mid^2)$ times. This means there are at most $O(\mid \mathcal{N} \mid^5)$ iterations in the fixed point algorithm and since each iteration can take

at most $O(\mid \mathcal{E} \mid + \mid \mathcal{N} \mid^2)$ time, the fixed point can be calculated in time polynomial in the size of the ICFG.

□

C.4.6 Proof of Theorem 4.2.4

Lemma C.4.8 *The fixed point of must-alias can be calculated in time polynomial in the size of the ICFG = ($\mathcal{N}$,$\mathcal{E}$,ρ).*

Each variable in the program contributes one *object name* which may be involved in aliases; for each pointer p, $*p$ may have aliases and for each non-pointer v, v itself may be aliased. Thus the number of possible aliases pairs in a program is $\mid variables \mid^2$. Since a program can use at most $O(\mid \mathcal{N} \mid)$ variables, the number of possible aliases is $O(\mid \mathcal{N} \mid^2)$.

During the fixed point calculation for any *node*, *must-alias*(*node*) can never increase in size. Since the maximum size of all *must-alias*(*node*) is $O(\mid \mathcal{N} \mid^2)$, any *must-alias*(*node*) can change its value at most that many times. Thus we have $\mid \mathcal{N} \mid$ nodes that can change their value at most $O(\mid \mathcal{N} \mid^2)$ times. Therefore the fixed point calculation for *must-alias* takes a polynomial amount of time ($O(\mid \mathcal{N} \mid^3)$).

□

Lemma C.4.9 *$\langle a, b \rangle$ not in must-alias(n) implies $\exists$ a realizable paths to n in the ICFG on which $\langle a, b \rangle$ does not hold.*

Proof by induction on j, the iteration of the fixed point algorithm.

Let *must-alias*$^j(n)$ be the value of *must-alias*(n) on the j^{th} iteration of the fixed point algorithm.

basis: $j = 0$

For all n in the ICFG, *must-alias*(n) is initialized to $\mathcal{POSSIBLE\text{-}ALIASES}'$.

induction hypothesis: Lemma C.4.9 holds for $k < j$.

induction step:

Say the value of *must-alias*$^j(n)$ changed at iteration j.

1. n is ρ

   must-alias$(\rho) = \emptyset$. No aliases hold before the program is executed.

2. n is an entry node

   must-alias$^j(n) = \bigcap_{\ll m,n \gg \in \mathcal{E}} (bind_m(\textit{must-alias}^{j-1}(m)))$.

   Let $\langle a, b\rangle$ be any arbitrary alias pair in (*must-alias*$^{j-1}(n)$ – *must-alias*$^j(n)$). For some m a predecessor of n in the ICFG, either

   - *back-bind*$_m(\langle a, b\rangle)$ is $false$

     In this case, by Lemma C.1.1 (p. 160), on all paths $\rho n_1 n_2 ... n_i m n$ $\langle a, b\rangle$ does not hold.

   - *back-bind*$_m(\langle a, b\rangle)$ not in *must-alias*$^{j-1}(m)$ (otherwise $\langle a, b\rangle$ would be in *must-alias*$^j(n)$). By induction there is a realizable path P to m on which *back-bind*$_m(\langle a, b\rangle)$ does not hold. Thus, by Lemma C.1.1 (p. 160) and Lemma C.2.1 (p. 163) $\langle a, b\rangle$ can not hold on the realizable path $P \circ n$.

3. n is not an entry node

   must-alias$(n) = \{\langle a, b\rangle \mid \textit{must-hold}(n, \langle a, b\rangle) \subseteq' \textit{must-alias}(\textit{entry}(n))\}$

   Let $\langle a, b\rangle$ be any arbitrary alias pair in (*must-alias*$^{j-1}(n)$ – *must-alias*$^j(n)$). Either

   - *must-hold*$(n, \langle a, b\rangle) = false$

     By Lemma C.4.3 (p. 200) there is a path to n on which $\langle a, b\rangle$ does not hold.

   - There is a $\langle c, d\rangle$ in *must-hold*$(n, \langle a, b\rangle)$, but not in *must-alias*$(\textit{entry}(n))$. By the induction hypothesis, there is a path, P, to

entry(n) on which $\langle c, d\rangle$ does not hold. Thus by Lemma C.4.3 (p. 200) there is a path to n on which $\langle a, b\rangle$ does not hold.

□

Lemma C.4.10 *$\exists$ a realizable path, $\mathcal{P}_i = \rho n_1 ... n_i$, in the ICFG on which $\langle a, b\rangle$ does not hold implies $\langle a, b\rangle$ not in must-alias(n_i).*

Proof by induction on i, the path length

basis: $i = 0$

$\mathcal{P}_i = \rho$. No aliases hold and *must-alias*$(\rho) = \emptyset$.

induction hypothesis: Lemma C.4.10 holds for $j < i$.

induction step:

Let $\langle a, b\rangle$ be any alias pair which does not hold on path $\mathcal{P}_i$.

1. n_i is ρ

   See the basis.

2. n_i is an entry node

   must-alias$(n_i) = \bigcap_{\ll m, n_i \gg \in \mathcal{E}} (bind_m(\textit{must-alias}(m)))$.

   - *back-bind*$_{n_{i-1}}(\langle a, b\rangle) = false$

     By Lemma C.1.1 (p. 160) and Lemma C.2.1 (p. 163), Lemma C.4.10 must hold.

   - otherwise

     back-bind$_{n_{i-1}}(\langle a, b\rangle)$ does not hold on P_{i-1}, because if it did $\langle a, b\rangle$ would hold on P_i (by Lemma C.1.1 (p. 160)). Thus by induction, *back-bind*$_{n_{i-1}}(\langle a, b\rangle)$ is not an element of *must-alias*(n_{i-1}) and, by Lemma C.2.1 (p. 163), $\langle a, b\rangle$ is not an element of *must-alias*(n_i) because $\ll n_{i-1}, n_i \gg \in \mathcal{E}$.

3. n_i is not an entry node

   must-alias$(n_i) = \{\langle a, b\rangle \mid \textit{must-hold}(n_i, \langle a, b\rangle) \subseteq' \textit{must-alias}(\textit{entry}(n_i))\}$

- $must\text{-}hold(n_i, \langle a, b \rangle) = false$

  By Lemma C.4.3 (p. 200), Lemma C.4.10 must hold.

- otherwise

  By Lemma C.4.3 (p. 200), there is a $\langle c, d \rangle \in must\text{-}hold(n_i, \langle a, b \rangle)$ that does not hold on $P_{entry(n_i)}$. By induction, $\langle c, d \rangle$ is not in $must\text{-}alias(entry(n_i))$. Thus $must\text{-}hold(n_i, \langle a, b \rangle) \not\subseteq must\text{-}alias(entry(n_i))$. Thus $\langle a, b \rangle \notin must\text{-}hold(n_i, \langle a, b \rangle)$.

□

Theorem 4.2.4 *There exists a polynomial time algorithm for determining precise* **Interprocedural Must Alias** *sets in the presence of single level pointers.*

The algorithm in Appendix C.4.3 is such an algorithm. Lemma C.4.9 (p. 211) and Lemma C.4.10 (p. 213) prove that it is precise. The algorithm consists of 4 steps. The first step can obviously be done in time polynomial in the size of the ICFG. Lemma C.4.7 (p. 210) implies that the second step can be done in time polynomial in the size of the ICFG. The third step can obviously be done in time polynomial in the size of the ICFG. Lemma C.4.8 (p. 211) implies that the final step can also be done in time polynomial in the size of the ICFG.

□

C.5 Replacing reference formals with pointers

Lemma 4.3.1 *For all realizable paths $\rho n_1 n_2 ... n_i$ in the ICFG for the original program, $\langle x, y \rangle$ holds iff*

- *case: x and y are reference formals*

  $\langle *x, *y \rangle$ holds on the same path in the transformed program.

- *case: x is a reference formal but y is not*

  $\langle *x, y \rangle$ holds on the same path in the transformed program.

- *case: neither x nor y are reference formals*

  $\langle x, y \rangle$ holds on the same path in the transformed program.

We already assume that each variable in the program has a unique name, but since we are dealing with paths we can also assume that each instance of a variable has a unique name. For example, say variable x is declared in a recursive procedure that is called 5 times on a path. We simply use x_1 to refer to the x variable from the first invocation, x_2 for the second, and so on. This can easily be done for any given path and allows us to ignore the issue of **visibility** in the following proofs.

In the following proofs, all statements of the from a is the actual for formal f refer to the original program unless explicitly stated otherwise.

Both the proofs are case analysis on n_i a node in the ICFG and an alias. The cases are the same for both directions except the "**if**" direction starts with an alias in the original program, while the "**only if**" direction starts with an alias in the transformed program. The cases are outlined below:

1. x and y are reference formals

   (a) n_i is an entry node

      i. x is f_j and y is f_k of the procedure containing n_i. Let a_j and a_k be the corresponding actuals at n_{i-1}.

         A. a_j and a_k are reference formals

         B. a_j is a reference formal, a_k is not

         C. a_j and a_k are not reference formals

      ii. x is f_j and y is not a formal of the procedure containing n_i. Let a_j be the corresponding actual at n_{i-1}.

         A. a_j is a reference formal

         B. a_j is not a reference formal

iii. x and y are not formals of the procedure containing n_i

(b) n_i is a call node, is an exit node, or is a return node

(c) otherwise (n_i is a statement node)

2. x is a reference formal but y is not

(a) n_i is an entry node

i. x is f_j, a formal of the procedure containing n_i. Let a_j be the corresponding actual at n_{i-1}.

A. a_j is a reference formal and y is a dereferenced pointer formal of the procedure containing n_i, $*f_k$ (with actual a_k)

- a_k is $\&f$, f a reference formal
- a_k is not a & reference formal

B. a_j is a reference formal and y is not a dereferenced pointer formal of the procedure containing n_i

C. a_j is not a reference formal and y is a dereferenced pointer formal of the procedure containing n_i, $*f_k$ (with actual a_k)

- a_k is $\&f$, f a reference formal
- a_k is not a & reference formal

D. a_j is not a reference formal and y is not a dereferenced pointer formal

ii. x is not a formal of the procedure containing n_i and y is a dereferenced pointer formal of the procedure containing n_i, $*f_k$ (with actual a_k)

- a_k is $\&f$, f a reference formal
- a_k is not a & reference formal

iii. x is not a formal of the procedure containing n_i and y is not a dereferenced pointer formal

(b) n_i is a call node, is an exit node, or is a return node

(c) otherwise (n_i is a statement node) It is impossible for n_i to be a pointer assignment "$p = ...$" where $x = *p$ because x is a reference formal.

i. n_i is the pointer assignment "$p = q$" in the original program

A. if $y = *p$

B. otherwise

ii. n_i is the pointer assignment "$p = \&v$" in the original program

A. if $y = *p$

B. otherwise

iii. n_i is the pointer assignment "$p = malloc()$" or "$p = NULL$" in the original program

3. x and y are not reference formals

(a) n_i is an entry node

i. $x = *f_j$ and $y = *f_k$, f_j and f_k pointer formals of the procedure containing n_i (with corresponding actuals a_j and a_k)

- a_j is $\&f_1$ and a_k is $\&f_2$, f_1 and f_2 reference formals
- a_j is $\&f_1$ (f_1 a reference formal) and a_k is not a & reference formal
- neither a_j nor a_k is a & reference formal

ii. $x = *f_j$, f_j a pointer formal (with actual a_j) and y is not a dereferenced pointer formal of the procedure containing n_i

- a_j is $\&f$, f a reference formal
- a_k is not a & reference formal

iii. neither x nor y are dereferenced pointer formals of the procedure containing n_i

(b) n_i is a call node, is an exit node, or is a return node

(c) n_i is the pointer assignment "$p = q$" in the original program

i. if $x = y = *p$

ii. if $x = *p$ $[x \neq y]$

iii. otherwise

(d) n_i is the pointer assignment "$p = \&v$" in the original program

i. if $x = y = *p$

ii. if $x = *p$ $[x \neq y]$

iii. otherwise

(e) n_i is the pointer assignment "$p = malloc()$" or "$p = NULL$" in the original program

i. if $x = y = *p$

ii. if $x = *p$ $[x \neq y]$

iii. otherwise

Proof (**only if**) by induction on i, the path length.

basis: $i = 0$

The path is simply ρ. No aliases hold before the program is executed. Lemma vacuously holds.

induction hypothesis: Lemma 4.3.1 holds in the "only if" for paths of length $j < i$.

induction step:

Let $\langle x, y \rangle$ hold on $P_i = \rho n_1 n_2 ... n_{i-1} n_i$ in the original program.

Let $P_{i-1} = \rho n_1 n_2 ... n_{i-1}$.

Let P_i^t be the path that corresponds to P_i in the transformed program.

Let P_{i-1}^t be the path that corresponds to P_{i-1} in the transformed program.

1. x and y are reference formals

(a) n_i is an entry node

i. x is f_j and y is f_k of the procedure containing n_i. Let a_j and a_k be the corresponding actuals at n_{i-1}.

A. a_j and a_k are reference formals

$\langle a_j, a_k \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle *a_j, *a_k \rangle$ holds on P_{i-1}^t in the transformed program. This requires that $\langle *f_j, *f_k \rangle \equiv \langle *x, *y \rangle$ hold on P_i^t

because in the transformed program f_j is passed a_j and f_k is passed a_k.

B. a_j is a reference formal, a_k is not

$\langle a_j, a_k \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle *a_j, a_k \rangle$ holds on P^t_{i-1} in the transformed program. This requires that $\langle *f_j, *f_k \rangle \equiv \langle *x, *y \rangle$ hold on P^t_i because in the transformed program f_j is passed a_j and f_k is passed $\&a_k$.

C. a_j and a_k are not reference formals

a_j must equal a_k (this is the only way $\langle a_j, a_k \rangle$ could hold on P_{i-1} in the original program). Thus $\langle *f_j, *f_k \rangle$ holds on P^t_i in the transformed program because both formals are passed $\&a_j$ $(= \&a_k)$.

ii. x is f_j and y is not a formal of the procedure containing n_i.

Let a_j be the corresponding actual at n_{i-1}.

A. a_j is a reference formal

$\langle a_j, y \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle *a_j, *y \rangle$ holds on P^t_{i-1} in the transformed program. Thus $\langle *f_j, *y \rangle \equiv \langle *x, *y \rangle$ holds on P^t_i since f_j is passed a_j in the transformed program.

B. a_j is not a reference formal

$\langle a_j, y \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle a_j, *y \rangle$ holds on P^t_{i-1} in the transformed program. Thus $\langle *f_j, *y \rangle \equiv \langle *x, *y \rangle$ holds on P^t_i since f_j is passed $\&a_j$ in the transformed program.

iii. x and y are not formals of the procedure containing n_i

$\langle x, y \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis, $\langle *x, *y \rangle$ holds on P^t_{i-1} in the transformed program and thus holds on P^t_i.

(b) n_i is a call node, is an exit node, or is a return node

In all cases, $\langle x, y \rangle$ holds on P_{i-1} in the original program, thus by the induction hypothesis $\langle *x, *y \rangle$ holds on P^t_{i-1} in the transformed program. This implies that it also holds on P^t_i.

(c) otherwise (n_i is a statement node)

It is impossible for n_i to be a pointer assignment "$p = ...$" where $x = *p$ or $y = *p$ because x and y are reference formals and not pointers. Thus $\langle x, y \rangle$ holds on P_{i-1} in the original program, and by the induction hypothesis, $\langle *x, *y \rangle$ holds on P^t_{i-1} in the transformed program and thus holds on P^t_i.

2. x is a reference formal but y is not

(a) n_i is an entry node

i. x is f_j, a formal of the procedure containing n_i. Let a_j be the corresponding actual at n_{i-1}.

A. a_j is a reference formal and y is a dereferenced pointer formal of the procedure containing n_i, $*f_k$ (with actual a_k)

- a_k is $\&f$, f a reference formal

  $\langle a_j, f \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle *a_j, *f \rangle$ holds on P^t_{i-1} in the transformed program. Thus $\langle *f_j, *f_k \rangle \equiv \langle *x, y \rangle$ holds on P^t_i since f_j is passed a_j and f_k is passed f in the transformed program.

- a_k is not a & reference formal

  $\langle a_j, *a_k \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle *a_j, *a_k \rangle$ holds on P^t_{i-1} in the transformed program. Thus $\langle *f_j, *f_k \rangle \equiv \langle *x, y \rangle$ holds on P^t_i since f_j is passed a_j and f_k is passed a_k in the transformed program.

B. a_j is a reference formal and y is not a dereferenced pointer formal of the procedure containing n_i

$\langle a_j, y\rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle *a_j, y\rangle$ holds on P^t_{i-1} in the transformed program. Thus $\langle *f_j, y\rangle \equiv \langle *x, y\rangle$ holds on P^t_i since f_j is passed a_j in the transformed program.

C. a_j is not a reference formal and y is a dereferenced pointer formal of the procedure containing n_i, $*f_k$ (with actual a_k)

- a_k is $\&f$, f a reference formal

  $\langle a_j, f\rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle a_j, *f\rangle$ holds on P^t_{i-1} in the transformed program. Thus $\langle *f_j, *f_k\rangle \equiv \langle *x, y\rangle$ holds on P^t_i since f_j is passed $\&a_j$ and f_k is passed f in the transformed program.

- a_k is not a & reference formal

  $\langle a_j, *a_k\rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle a_j, *a_k\rangle$ holds on P^t_{i-1} in the transformed program. Thus $\langle *f_j, *f_k\rangle \equiv \langle *x, y\rangle$ holds on P^t_i since f_j is passed $\&a_j$ and f_k is passed a_k in the transformed program.

D. a_j is not a reference formal and y is not a dereferenced pointer formal

$\langle a_j, y\rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle a_j, y\rangle$ holds on P^t_{i-1} in the transformed program. Thus $\langle *f_j, y\rangle \equiv \langle *x, y\rangle$ holds on P^t_i since f_j is passed $\&a_j$ in the transformed program.

ii. x is not a formal of the procedure containing n_i and y is a dereferenced pointer formal of the procedure containing n_i, $*f_k$ (with actual a_k)

- a_k is $\&f$, f a reference formal

  $\langle x, f \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle *x, *f \rangle$ holds on P^t_{i-1} in the transformed program. Thus $\langle *x, *f_k \rangle \equiv \langle *x, y \rangle$ holds on P^t_i since f_k is passed f in the transformed program.

- a_k is not a & reference formal

  $\langle x, *a_k \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle *x, *a_k \rangle$ holds on P^t_{i-1} in the transformed program. Thus $\langle *x, *f_k \rangle \equiv \langle *x, y \rangle$ holds on P^t_i since f_k is passed a_k in the transformed program.

iii. x is not a formal of the procedure containing n_i and y is not a dereferenced pointer formal

  $\langle x, y \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis, $\langle *x, y \rangle$ holds on P^t_{i-1} in the transformed program and thus holds on P^t_i.

(b) n_i is a call node, is an exit node, or is a return node

In all cases, $\langle x, y \rangle$ holds on P_{i-1} in the original program, thus by the induction hypothesis $\langle *x, y \rangle$ holds on P^t_{i-1} in the transformed program. This implies that it also holds on P^t_i.

(c) otherwise (n_i is a statement node)

It is impossible for n_i to be a pointer assignment "$p = ...$" where $x = *p$ because x is a reference formal.

i. n_i is the pointer assignment "$p = q$" in the original program

A. if $y = *p$

  $\langle x, *q \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis, $\langle *x, *q \rangle$ holds on P^t_{i-1} in the transformed program and thus $\langle *x, *p \rangle \equiv \langle *x, y \rangle$ holds on P^t_i.

B. otherwise

  $\langle x, y \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis, $\langle *x, y \rangle$ holds on P^t_{i-1} in the transformed

program and thus holds on P_i^t.

ii. n_i is the pointer assignment "$p = \&v$" in the original program

A. if $y = *p$

$\langle x, v \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis, $\langle *x, v \rangle$ holds on P_{i-1}^t in the transformed program and thus $\langle *x, *p \rangle \equiv \langle *x, y \rangle$ holds on P_i^t.

B. otherwise

$\langle x, y \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis, $\langle *x, y \rangle$ holds on P_{i-1}^t in the transformed program and thus holds on P_i^t.

iii. n_i is the pointer assignment "$p = malloc()$" or "$p = NULL$" in the original program

y can not be $*p$ since $*p$ can not be aliased to anything (except possibly itself). Therefore, $\langle x, y \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis, $\langle *x, y \rangle$ holds on P_{i-1}^t in the transformed program and thus holds on P_i^t.

3. x and y are not reference formals

(a) n_i is an entry node

i. $x = *f_j$ and $y = *f_k$, f_j and f_k pointer formals of the procedure containing n_i (with corresponding actuals a_j and a_k)

- a_j is $\&f_1$ and a_k is $\&f_2$, f_1 and f_2 reference formals

$\langle f_1, f_2 \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle *f_1, *f_2 \rangle$ holds on P_{i-1}^t in the transformed program. Thus $\langle *f_j, *f_k \rangle \equiv \langle x, y \rangle$ holds on P_i^t since f_j is passed f_1 and f_k is passed f_2 in the transformed program.

- a_j is $\&f_1$ (f_1 a reference formal) and a_k is not a & reference formal

$\langle f_1, *a_k \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle *f_1, *a_k \rangle$ holds on P^t_{i-1} in the transformed program. Thus $\langle *f_j, *f_k \rangle \equiv \langle x, y \rangle$ holds on P^t_i since f_j is passed f_1 and f_k is passed a_k in the transformed program.

- neither a_j nor a_k is a & reference formal

  $\langle *a_j, *a_k \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle *a_j, *a_k \rangle$ holds on P^t_{i-1} in the transformed program. Thus $\langle *f_j, *f_k \rangle \equiv \langle x, y \rangle$ holds on P^t_i since f_j is passed a_j and f_k is passed a_k in the transformed program.

ii. $x = *f_j$, f_j a pointer formal (with actual a_j) and y is not a dereferenced pointer formal of the procedure containing n_i

- a_j is $\&f$, f a reference formal

  $\langle f, y \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle *f, y \rangle$ holds on P^t_{i-1} in the transformed program. Thus $\langle *f_j, y \rangle \equiv \langle x, y \rangle$ holds on P^t_i since f_j is passed f in the transformed program.

- a_k is not a & reference formal

  $\langle *a_j, y \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis $\langle *a_j, y \rangle$ holds on P^t_{i-1} in the transformed program. Thus $\langle *f_j, y \rangle \equiv \langle x, y \rangle$ holds on P^t_i since f_j is passed a_j in the transformed program.

iii. neither x nor y are dereferenced pointer formals of the procedure containing n_i

$\langle x, y \rangle$ holds on P_{i-1} in the original program, thus by the induction hypothesis $\langle x, y \rangle$ holds on P^t_{i-1} in the transformed program. This implies that it also holds on P^t_i.

(b) n_i is a call node, is an exit node, or is a return node

In all cases, $\langle x, y \rangle$ holds on P_{i-1} in the original program, thus by

the induction hypothesis $\langle x, y\rangle$ holds on P^t_{i-1} in the transformed program. This implies that it also holds on P^t_i.

(c) n_i is the pointer assignment "$p = q$" in the original program

i. if $x = y = *p$

$\langle *q, *q\rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis, $\langle *q, *q\rangle$ holds on P^t_{i-1} in the transformed program and thus $\langle *p, *p\rangle \equiv \langle x, y\rangle$ holds on P^t_i.

ii. if $x = *p$ $[x \neq y]$

$\langle *q, y\rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis, $\langle *q, y\rangle$ holds on P^t_{i-1} in the transformed program and thus $\langle *p, y\rangle \equiv \langle x, y\rangle$ holds on P^t_i.

iii. otherwise

$\langle x, y\rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis, $\langle x, y\rangle$ holds on P^t_{i-1} in the transformed program and thus holds on P^t_i.

(d) n_i is the pointer assignment "$p = \&v$" in the original program

i. if $x = y = *p$

By definition, $\langle *p, *p\rangle \equiv \langle x, y\rangle$ holds on P^t_i in the transformed program.

ii. if $x = *p$ $[x \neq y]$

$\langle v, y\rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis, $\langle v, y\rangle$ holds on P^t_{i-1} in the transformed program and thus $\langle *p, y\rangle \equiv \langle x, y\rangle$ holds on P^t_i.

iii. otherwise

$\langle x, y\rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis, $\langle x, y\rangle$ holds on P^t_{i-1} in the transformed program and thus holds on P^t_i.

(e) n_i is the pointer assignment "$p = malloc()$" or "$p = NULL$" in the original program

i. if $x = y = *p$

By definition, $\langle *p, *p \rangle \equiv \langle x, y \rangle$ holds on P_i^t in the transformed program for "$p = malloc()$" (it is not possible for the alias to hold on P_i for "$p = NULL$").

ii. if $x = *p$ $[x \neq y]$

Not possible, since $*p$ can not be aliased to anything (except possibly itself).

iii. otherwise

$\langle x, y \rangle$ holds on P_{i-1} in the original program, thus by induction hypothesis, $\langle x, y \rangle$ holds on P_{i-1}^t in the transformed program and thus holds on P_i^t.

Proof (**if**) by induction on i, the path length.

basis: $i = 0$

The path is simply ρ. No aliases hold before the program is executed. Lemma vacuously holds.

induction hypothesis: Lemma 4.3.1 holds in the "if" for paths of length $j < i$.

induction step:

Let $\langle a, b \rangle$ hold on $P_i^t = \rho n_1 n_2 ... n_{i-1} n_i$ in the transformed program.

Let $P_{i-1}^t = \rho n_1 n_2 ... n_{i-1}$.

Let P_i be the path that corresponds to P_i in the original program.

Let P_{i-1} be the path that corresponds to P_{i-1} in the original program.

1. $\langle a, b \rangle \equiv \langle *x, *y \rangle$, x and y are reference formals in original program

   (a) n_i is an entry node

      i. x is f_j and y is f_k of the procedure containing n_i. Let a_j and a_k be the corresponding actuals in the original program at n_{i-1}.

         A. a_j and a_k are reference formals

            In the transformed program f_j is passed a_j and f_k is passed

a_k, thus $\langle *a_j, *a_k \rangle$ must hold on P^t_{i-1}, thus by induction hypothesis $\langle a_j, a_k \rangle$ holds on P_{i-1}. This requires that $\langle f_j, f_k \rangle \equiv \langle x, y \rangle$ hold on P_i.

B. a_j is a reference formal, a_k is not

In the transformed program f_j is passed a_j and f_k is passed $\&a_k$, thus $\langle *a_j, a_k \rangle$ must hold on P^t_{i-1}, thus by induction hypothesis $\langle a_j, a_k \rangle$ holds on P_{i-1}. This requires that $\langle f_j, f_k \rangle \equiv \langle x, y \rangle$ hold on P_i.

C. a_j and a_k are not reference formals

$\langle *a_j, *a_k \rangle$ must hold on P^t_{i-1}, but both a_j and a_k had to be non-pointer variables in the original program (since they were passed to non-pointer reference formals). Thus a_j must equal a_k and $\langle f_j, f_k \rangle$ holds on P_i.

ii. x is f_j and y is not a formal of the procedure containing n_i. Let a_j be the corresponding actual at n_{i-1} in the original program.

A. a_j is a reference formal

Since f_j is passed a_j in the transformed program, $\langle *a_j, *y \rangle$ holds on P^t_{i-1}. Thus by induction hypothesis $\langle a_j, y \rangle$ holds on P_{i-1}. Thus $\langle f_j, y \rangle \equiv \langle x, y \rangle$ holds on P_i.

B. a_j is not a reference formal

Since f_j is passed $\&a_j$ in the transformed program, $\langle a_j, *y \rangle$ holds on P^t_{i-1}. Thus by induction hypothesis $\langle a_j, y \rangle$ holds on P_{i-1}. Thus $\langle f_j, y \rangle \equiv \langle x, y \rangle$ holds on P_i.

iii. x and y are not formals of the procedure containing n_i

$\langle *x, *y \rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle x, y \rangle$ holds on P_{i-1} and thus holds on P_i.

(b) n_i is a call node, is an exit node, or is a return node

In all cases, $\langle *x, *y \rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle x, y \rangle$ holds on P_{i-1} and thus holds on P_i.

(c) otherwise (n_i is a statement node)

It is impossible for n_i to be a pointer assignment "$p = ...$" where $x = p$ or $y = p$ in the original program because x and y are reference formals and not pointers. Thus $\langle *x, *y \rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle x, y \rangle$ holds on P_{i-1} and thus holds on P_i.

2. $\langle a, b \rangle \equiv \langle *x, y \rangle$, x is a reference formal in original program, but y isn't

(a) n_i is an entry node

i. x is f_j, a formal of the procedure containing n_i. Let a_j be the corresponding actual at n_{i-1} in the original program.

A. a_j is a reference formal and y is a dereferenced pointer formal of the procedure containing n_i, $*f_k$ (with actual a_k in the original program)

- a_k is $\&f$ in the original program, f a reference formal

  Since f_j is passed a_j and f_k is passed f in the transformed program, $\langle *a_j, *f \rangle$ holds on P^t_{i-1}. Thus by induction hypothesis $\langle a_j, f \rangle$ holds on P_{i-1}. Thus $\langle f_j, *f_k \rangle \equiv \langle x, y \rangle$ holds on P_i.

- a_k is not a & reference formal in the original program

  Since f_j is passed a_j and f_k is passed a_k in the transformed program, $\langle *a_j, *a_k \rangle$ holds on P^t_{i-1}. Thus by induction hypothesis $\langle a_j, *a_k \rangle$ holds on P_{i-1}. Thus $\langle f_j, *f_k \rangle \equiv \langle x, y \rangle$ holds on P_i.

B. a_j is a reference formal and y is not a dereferenced pointer formal of the procedure containing n_i

Since f_j is passed a_j in the transformed program, $\langle *a_j, y \rangle$ holds on P^t_{i-1}. Thus by induction hypothesis $\langle a_j, y \rangle$ holds on P_{i-1}. Thus $\langle f_j, y \rangle \equiv \langle x, y \rangle$ holds on P_i.

C. a_j is not a reference formal and y is a dereferenced pointer formal of the procedure containing n_i, $*f_k$ (with actual a_k in the original program)

- a_k is $\&f$, f a reference formal in the original program

  Since f_j is passed $\&a_j$ and f_k is passed f in the transformed program, $\langle a_j, *f \rangle$ holds on P^t_{i-1}. Thus by induction hypothesis $\langle a_j, f \rangle$ holds on P_{i-1}. Thus $\langle f_j, *f_k \rangle \equiv \langle x, y \rangle$ holds on P_i.

- a_k is not a & reference formal in the original program

  Since f_j is passed $\&a_j$ and f_k is passed a_k in the transformed program, $\langle a_j, *a_k \rangle$ holds on P^t_{i-1}. Thus by induction hypothesis $\langle a_j, *a_k \rangle$ holds on P_{i-1}. Thus $\langle f_j, *a_k \rangle \equiv \langle x, y \rangle$ holds on P_i.

D. a_j is not a reference formal and y is not a dereferenced pointer formal of the procedure containing n_i

Since f_j is passed $\&a_j$ in the transformed program, $\langle a_j, y \rangle$ holds on P^t_{i-1}. Thus by induction hypothesis $\langle a_j, y \rangle$ holds on P_{i-1}. Thus $\langle f_j, y \rangle \equiv \langle x, y \rangle$ holds on P_i.

ii. x is not a formal of the procedure containing n_i and y is a dereferenced pointer formal of the procedure containing n_i, $*f_k$ (with actual a_k in the original program)

- In the original program a_k is $\&f$, f a reference formal

  Since f_k is passed f in the transformed program, $\langle x, *f \rangle$ holds on P^t_{i-1}. Thus by induction hypothesis $\langle x, f \rangle$ holds on P_{i-1}. Thus $\langle x, *f_k \rangle \equiv \langle x, y \rangle$ holds on P_i.

- In the original program, a_k is not a & reference formal

  Since f_k is passed a_k in the transformed program, $\langle x, *a_k \rangle$ holds on P^t_{i-1}. Thus by induction hypothesis $\langle x, *a_k \rangle$ holds on P_{i-1}. Thus $\langle x, *f_k \rangle \equiv \langle x, y \rangle$ holds on P_i.

iii. x is not a formal of the procedure containing n_i and y is not a dereferenced pointer formal of the procedure containing n_i

$\langle *x, y \rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle x, y \rangle$ holds on P_{i-1} and thus holds on P_i.

(b) n_i is a call node, is an exit node, or is a return node

In all cases, $\langle *x, y \rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle x, y \rangle$ holds on P_{i-1} and thus holds on P_i.

(c) otherwise (n_i is a statement node)

It is impossible for n_i to be a pointer assignment "$p = ...$" where $x = *p$ in the original program because x is a reference formal.

i. n_i is the pointer assignment "$p = q$" in the original program

A. if $y = *p$

$\langle *x, *q \rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle x, *q \rangle$ holds on P_{i-1} and thus $\langle x, *p \rangle \equiv \langle x, y \rangle$ holds on P_i.

B. otherwise

$\langle *x, y \rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle x, y \rangle$ holds on P_{i-1} and thus holds on P_i.

ii. n_i is the pointer assignment "$p = \&v$" in the original program

A. if $y = *p$

$\langle *x, v \rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle x, v \rangle$ holds on P_{i-1} and thus $\langle x, *p \rangle \equiv \langle x, y \rangle$ holds on P_i.

B. otherwise

$\langle *x, y \rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle x, y \rangle$ holds on P_{i-1} and thus holds on P_i.

iii. n_i is the pointer assignment "$p = malloc()$" or "$p = NULL$" in the original program

y can not be $*p$ since $*p$ can not be aliased to anything (except possibly itself). Therefore, $\langle *x, y \rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle x, y \rangle$ holds on P_{i-1} and thus holds on P_i.

3. $\langle a, b\rangle \equiv \langle x, y\rangle$, x and y are not reference formals in original program

   (a) n_i is an entry node

      i. $x = *f_j$ and $y = *f_k$, f_j and f_k pointer formals of the procedure containing n_i (with corresponding actuals a_j and a_k in the original program)

         - In the original program, a_j is $\&f_1$ and a_k is $\&f_2$, f_1 and f_2 reference formals

           Since f_j is passed f_1 and f_k is passed f_2 in the transformed program, $\langle *f_1, *f_2\rangle$ holds on P^t_{i-1}. Thus by induction hypothesis $\langle f_1, f_2\rangle$ holds on P_{i-1}. Thus, $\langle *f_j, *f_k\rangle \equiv \langle x, y\rangle$ holds on P_i.

         - In the original program, a_j is $\&f_1$ (f_1 a reference formal) and a_k is not a & reference formal

           Since f_j is passed f_1 and f_k is passed a_k in the transformed program, $\langle *f_1, *a_k\rangle$ holds on P^t_{i-1}. Thus by induction hypothesis $\langle f_1, *a_k\rangle$ holds on P_{i-1}. Thus, $\langle *f_j, *f_k\rangle \equiv \langle x, y\rangle$ holds on P_i.

         - neither a_j nor a_k is a & reference formal

           Since f_j is passed a_j and f_k is passed a_k in the transformed program, $\langle *a_j, *a_k\rangle$ holds on P^t_{i-1}. Thus by induction hypothesis $\langle *a_j, *a_k\rangle$ holds on P_{i-1}. Thus, $\langle *f_j, *f_k\rangle \equiv \langle x, y\rangle$ holds on P_i.

      ii. $x = *f_j$, f_j a pointer formal (with actual a_j) and y is not a dereferenced pointer formal

         - a_j is $\&f$, f a reference formal

           Since f_j is passed f in the transformed program, $\langle *f, y\rangle$ holds on P^t_{i-1}. Thus by induction hypothesis $\langle f, y\rangle$ holds on P_{i-1}. Thus, $\langle *f_j, y\rangle \equiv \langle x, y\rangle$ holds on P_i.

         - a_k is not a & reference formal

           Since f_j is passed a_k in the transformed program, $\langle *a_k, y\rangle$

holds on P^t_{i-1}. Thus by induction hypothesis $\langle *a_k, y\rangle$ holds on P_{i-1}. Thus, $\langle *f_j, y\rangle \equiv \langle x, y\rangle$ holds on P_i.

iii. neither x nor y are dereferenced pointer formals

$\langle x, y\rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle x, y\rangle$ holds on P_{i-1} and thus holds on P_i.

(b) n_i is a call node, is an exit node, or is a return node

In all cases, $\langle x, y\rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle x, y\rangle$ holds on P_{i-1} and thus holds on P_i.

(c) n_i is the pointer assignment "$p = q$" in the original program

i. if $x = y = *p$

$\langle *q, *q\rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle *q, *q\rangle$ holds on P_{i-1} and thus $\langle *p, *p\rangle \equiv \langle x, y\rangle$ holds on P_i.

ii. if $x = *p$ $[x \neq y]$

$\langle x, *q\rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle x, *q\rangle$ holds on P_{i-1} and thus $\langle x, *p\rangle \equiv \langle x, y\rangle$ holds on P_i.

iii. otherwise

$\langle x, y\rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle x, y\rangle$ holds on P_{i-1} and thus holds on P_i.

(d) n_i is the pointer assignment "$p = \&v$" in the original program

i. if $x = y = *p$

By definition, $\langle *p, *p\rangle \equiv \langle x, y\rangle$ holds on P_i.

ii. if $x = *p$ $[x \neq y]$

$\langle x, v\rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle x, v\rangle$ holds on P_{i-1} and thus $\langle x, *p\rangle \equiv \langle x, y\rangle$ holds on P_i.

iii. otherwise

$\langle x, y\rangle$ holds on P^t_{i-1}, thus by induction hypothesis, $\langle x, y\rangle$ holds on P_{i-1} and thus holds on P_i.

(e) n_i is the pointer assignment "$p = malloc()$" or "$p = NULL$" in the original program

i. if $x = y = *p$

By definition, $\langle *p, *p \rangle \equiv \langle x, y \rangle$ holds on P_i for "$p = malloc()$" (it is not possible for the alias to hold on P_i^t for "$p = NULL$").

ii. if $x = *p$ $[x \neq y]$

Not possible, since $*p$ can not be aliased to anything (except possibly itself).

iii. otherwise

$\langle x, y \rangle$ holds on P_{i-1}^t, thus by induction hypothesis, $\langle x, y \rangle$ holds on P_{i-1} and thus holds on P_i.

□

Appendix D

Interprocedural May Alias Approximate Algorithm Pseudo Code

```
find_aliases()
{ worklist = ∅
  /* Alias Introduction */
  for each node (node) in the ICFG
    if node is an assignment to a pointer
      aliases_introduced_by_assignment(node) /* Figure 5.7 */
    if node is a call node
      aliases_introduced_by_call(node) /* Figure 5.8 */

  /* Implied Aliases */
  while worklist is not empty
  { remove node,assumed_alias,possible_alias from worklist
    if node is a call node
      aliases_at_call_implies(node,assumed_alias,possible_alias)
    else if node is an exit node
      alias_at_exit_implies(node,assumed_alias,possible_alias)
    else any_other_alias_implies(node,assumed_alias,possible_alias)
  }
}
```

D.1 Approximate-when-both-non_visible

approximate-when-both-non_visible (call-node, $\langle nv_1,\ nv_2\rangle$,
non_visible-name1-is, non_visible-name2-is, name1-AA, name2-AA) substitutes non_visible-name1-is for non_visible in nv_1, substitutes non_visible-name2-is for non_visible in nv_2, and uses the assumed aliases name1-AA and name2-AA to establish a safe assumed alias condition at procedure entry.

```
approximate-when-both-non_visible(call-node,⟨nv1, nv2⟩,
      non_visible-name1-is,non_visible-name2-is,name1-AA,name2-AA)
{
  apply_trans(non_visible,nv1,non_visible-name1-is)
  apply_trans(non_visible,nv2,non_visible-name2-is)
  if both name1-AA and name2-AA contain a non_visible
  {
  /* Creates another alias in which both object names are non_visible */
    let exit be the exit node of the procedure containing call-node
    if may-hold([(exit, name1-AA, name2-AA), ⟨non_visible-name1-is,non_visible-name2-is⟩]) false
    {
      make may-hold([(exit, name1-AA, name2-AA), ⟨non_visible-name1-is,non_visible-name2-is⟩]) true
      add (exit, name1-AA, name2-AA, ⟨non_visible-name1-is,non_visible-name2-is⟩)
        to worklist
    }
  } else
```

```
    {
    /* Both assumptions are individually necessary and either can be
       safely chosen.  If one assumption contains non_visible, then use
       that one (because we need it to figure out which non-visible
       object name non_visible represents; otherwise use either (but only
       use the assumption ∅ if both are ∅ because requiring both ∅ and
       𝒜𝒜 is the same as requiring just 𝒜𝒜).  */
       if name1-AA is ∅ or name2-AA contains non_visible
          AA = name2-AA†
       else AA = name1-AA
       for each ⟨x, y⟩ in
       alias_consequences(⟨non_visible-name1-is,non_visible-name2-is⟩)
          let return be the return node that corresponds to call-node
          if may-hold([(return, AA), ⟨x, y⟩]) is false
          {
             set may-hold([(return, AA), ⟨x, y⟩]) to true
             add (return,AA,⟨x, y⟩) to worklist
          }
    }
}
```

---

† Both name1-AA and name2-AA are necessary aliasing conditions at the entry of the procedure for x aliased to y at *return*. We only allow one alias condition, and it is safe but approximate to arbitrarily pick either one.

D.2 Alias_at_call_implies

```
alias_at_call_implies(call, assumed_alias, ⟨a,b⟩)
{ let return be the return node that corresponds to call
  let exit be the exit node of the called procedure
  let entry be the entry node of the called procedure
  if both a and b are not visible in the called procedure /* Rule 1 */
    if may-hold([(return, assumed_alias), ⟨a,b⟩]) is false
    { set may-hold([(return, assumed_alias), ⟨a,b⟩]) to true
      add (return, assumed_alias, ⟨a,b⟩) to worklist
    } /* Below an expanded version of the code found on page 97 with
         the effects on entry added.  */
  for each 𝒜𝒜 and (𝒜𝒜, nv) in bind_call(⟨a,b⟩)
  { if may-hold([(entry, 𝒜𝒜), 𝒜𝒜]) is false /* effects on entry */
    { set may-hold([(entry, 𝒜𝒜), 𝒜𝒜]) to true
      add (entry, 𝒜𝒜, 𝒜𝒜) to worklist
    }
    for every ⟨x,y⟩ or (⟨x,y⟩,other_condition) in TUA(exit, 𝒜𝒜)
    { if neither x nor y contain non_visible /* Rule 2 */
      { if may-hold([(return, assumed_alias), ⟨x,y⟩]) is false
        { set may-hold([(return, assumed_alias), ⟨x,y⟩]) to true
          add (return, assumed_alias, ⟨x,y⟩) to the worklist.
        }
      } else
```

```
      { if both x and y contain non_visible /* Rule 3- special case */
          for each (AA, nv2) in BBT(call, ⟨non_visible,other_condition⟩)
            approximate-when-both-non_visible(call, ⟨x, y⟩, nv, nv2,
                assumed_alias, AA)
        else (assume x contains non_visible):  /* Rule 3 */
          let x' = nv; apply_trans(non_visible,x,x')
          if may-hold([(return, assumed_alias), ⟨x', y⟩]) is false
          { set may-hold([(return, assumed_alias), ⟨x', y⟩]) to true
            add (return, assumed_alias, ⟨x', y⟩) to the worklist.
          }
      }
    } /* end for each element of TUA */
  } /* end for each element of bind */
}
```

D.3 Alias_at_exit_implies

```
alias_at_exit_implies(exit, AA, ⟨x, y⟩)
{ if both x and y contain non_visible
  { let AA = ⟨a1, a2⟩
    for each call node (call) that invokes the procedure containing exit
      for each (assumed1, nv1) in BBT(call, ⟨non_visible, a1⟩)
        for each (assumed2, nv2) in BBT(call, ⟨non_visible, a2⟩)
          approximate-when-both-non_visible(call, ⟨x, y⟩, nv1, nv2, assumed1,
              assumed2)
  } else
  { if neither x nor y contains non_visible
      for each call node (call) that invokes the exit's procedure
        for each assumed in BBT(call, AA)
          let return be the return node that corresponds to call
            if ( the object names x and y at call
                 are visible in the called procedure )
            /* e.g., two aliased locals in a recursive procedure */
              if may-hold([(return, assumed), ⟨x, y⟩]) is false
              { set may-hold([(return, assumed), ⟨x, y⟩]) to true
                add (return, assumed, ⟨x, y⟩) to worklist
              }
```

```
    if x or y contains non_visible (assume x)
      for each call node (call) that invokes the procedure containing exit
        for each (assumed, nv) in BBT(call, AA)
          let return to the return node that corresponds to call
            if object name y at call is visible in the called procedure
            { apply_trans(non_visible,x,nv)
              for each PA in alias_consequences(⟨nv, y⟩)
                if may-hold([(return, assumed), PA]) is false
                { set may-hold([(return, assumed), PA]) to true
                  add (return,assumed,PA) to worklist
                }
            }
  }
}
```

D.4 Any_other_alias_implies

```
any_other_alias_implies(node, assumed_alias, ⟨y, z⟩)
{ for each successor, succ, of node
  { if succ is not an assignment to a pointer
    { if may-hold([(succ, assumed_alias), ⟨y, z⟩]) is false
      { set may-hold([(succ, assumed_alias), ⟨y, z⟩]) to true
        add (succ, assumed_alias, ⟨y, z⟩) to the worklist
      }
    } else /* let node be the assignment ``p = q'' */
    { if p is not a prefix of y and p is not a prefix of z /* Case 1 */
        if may-hold([(succ, assumed_alias), ⟨y, z⟩]) is false
        { set may-hold([(succ, assumed_alias), ⟨y, z⟩]) to true
          add (succ, assumed_alias, ⟨y, z⟩) to the worklist
        }
```

```
if is_prefix_with_deref(q, y) /* Case 2 */
{ if !is_prefix(p, z)
  {  let p' = p
    apply_trans(q, y, p')
    if may-hold([(succ, assumed_alias), ⟨p', z⟩]) is false
    {  set may-hold([(succ, assumed_alias), ⟨p', z⟩]) to true
      add (succ, assumed_alias, ⟨p', z⟩) to the worklist
      for each (AA, ⟨p, v⟩) such that
            may-hold([(node, AA), ⟨p, v⟩]) is true
        if ( v contains non_visible
             or !is_prefix(v, z) )
        {  apply_trans(q, y, v)
          safely_make_alias(succ, v, AA, z,
            assumed_alias)
        }
    }
  }
}
```

Figure 5.13

```
      if z = p /* Case 3 */
      { for each ⟨a,b⟩ in alias_consequences(⟨p,y⟩) or
            (provided !is_prefix(y,q) and !is_prefix(p,q))
            alias_consequences(⟨*y,*q⟩)
        {  if may-hold([(succ,assumed_alias),⟨a,b⟩]) is false
          {  set may-hold([(succ,assumed_alias),⟨a,b⟩]) to true
            add (succ,assumed_alias,⟨a,b⟩) to the worklist
          }
        }
        for each (AA,⟨a,b⟩) such that
              may-hold([(node,AA),⟨a,b⟩]) and
              is_prefix_with_deref(q,a) and !is_prefix(p,b)
          if y contains non_visible or !is_prefix(y,b)
          {  let y' = y
            apply_trans(q,a,y')
            safely_make_alias(succ,y',assumed_alias,b,AA)
          }
      }
    } /* node is an assignment to a pointer */
  } /* end for each successor */
}
```

Figure 5.14 with some variable renaming

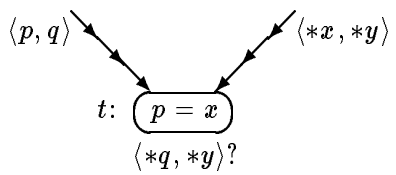
Appendix E

Safe Approximate Algorithm

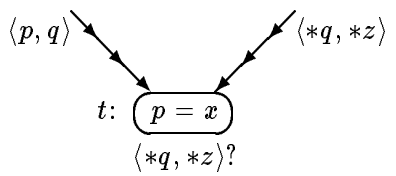
E.1 Types of Approximation

In Chapter 4, we mentioned four sources of approximation. They are:

1. In any program that contains recursive data structures, there are a potentially infinite number of objects which can have aliases. We represent all possible objects by a finite (polynomial) number of objects.

2. 

   If both $\langle p, q\rangle$ and $\langle *x, *y\rangle$ occur on the **same** path, then $\langle *q, *y\rangle$ holds on that path extended by t; therefore, we conclude this, even though it may not be true[84].

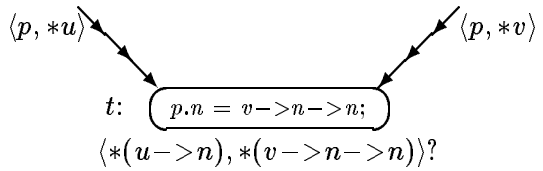
3. 

   If on at least one path to an immediate predecessor, $\langle *q, *z\rangle$ holds and neither (or both) $\langle p, q\rangle$ nor $\langle p, z\rangle$ does, then $\langle *q, *z\rangle$ holds on that path extended by t. However, if on all paths to immediate predecessors on which $\langle *q, *z\rangle$ holds, $\langle p, q\rangle$

[84] It is not enough to keep track of the arriving edge; the same problem would arise at edges arriving at predecessors.

(but not $\langle p, z\rangle$) also holds, then $\langle *q, *z\rangle$ does not necessarily hold on any path to t. In either case, for safety, we assume $\langle *q, *z\rangle$ holds on some path to t.

4.



Normally, $\langle *(u{->}n), *(v{->}n{->}n)\rangle$ should hold on a path to t because assigning $v{->}n{->}n$ to $p.n$ is also assigning $v{->}n{->}n$ to $u{->}n$[85]. However, $\langle *(u{->}n), *(v{->}n{-}\ n)\rangle$ need not hold. If, for example, $\langle p, *u\rangle$ and $\langle p, *v\rangle$ hold on the same path then $\langle *(u{->}n), *(v{->}n{->}n)\rangle$ does not necessarily hold.

Lemma E.1.1 *Let k be the k-limit constant used by our algorithm (Figure 5.6). Given that arrays are treated as aggregates and Assumption 5.1.1 (NULL, p. 81).*

If may-hold($[(node, \mathcal{AA}), \mathcal{PA}]$) is true then either $\mathcal{PA}$ is a type 2, 3, or 4 approximation, derived (directly or indirectly) from a type 2, 3 or 4 approximation, or $(node, \mathcal{PA}) \in limit_k$(precise solution).[86]

Proof of Lemma E.1.1 by induction on i, the iteration number of the algorithm.

basis: $i = 0$

- Aliases_intro_by_assignment(*node*) (Chapter 5.2.1)

  When p is not a prefix of q (let *node* be the assignment "$p = q$") the lemma follows from Assumption 5.1.1 (NULL). When p is a prefix of q no new aliases are created by our algorithm.

---

[85] on the path on which $\langle p, *u\rangle$ holds

[86] Somehow we must account for aliases involving non-visible object names. Assume that *non_visible* is used to represent non-visible objects names in the precise solution.

- Alias_intro_by_call(*node*) (Chapter 5.2.2)

  There are two ways aliases can be generated here (Chapter 5.1.4):

  - Because an actual is passed a formal
  - Because a formal f_1 is passed a_1 and another formal f_2 is passed a_2 and is_prefix(a_1,a_2)

  In both cases, the lemma follows from Assumption 5.1.1 (NULL).

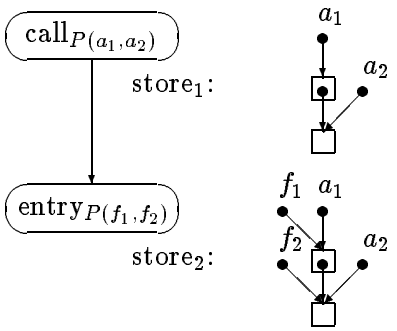
induction hypothesis: Lemma E.1.1 holds for $j < i$.

induction step:

1. Alias_at_call_implies(*call*,*assumed_alias*,*possible_alias*) (Chapter 5.2.3)

   By the induction hypothesis we know *possible_alias* is in $limit_k$(precise solution) or is (or derived by) a type 2, 3, or 4 approximation. If it is a type 2, 3, or 4 approximation than any *may-holds* that is set *true* is set *true* because of *possible_alias* and thus is derived by a type 2, 3, or 4 approximation and satisfies the lemma. Assume this is not the case (i.e., *possible_alias* is in $limit_k$(precise solution)):

   - Effects on corresponding entry node *entry*: *may-hold*($[(entry, \langle a,b \rangle), \langle a,b \rangle]$) is set *true* for each $\langle a,b \rangle$ in bind(*call*,*possible_alias*). $\langle a,b \rangle$ has to be the result of one of the following (from Chapter 5.1.4):



1^{st} way: $\langle **a_1, *a_2 \rangle$ at call $\Rightarrow$ $\langle **a_1, *a_2 \rangle$ at entry

2^{nd} way: $\langle **a_1, *a_2 \rangle$ at call $\Rightarrow$ $\langle *f_2, **a_1 \rangle$ at entry

3^{rd} way: $\langle **a_1, *a_2 \rangle$ at call $\Rightarrow$ $\langle **f_1, *f_2 \rangle$ at entry

   - 1^{st} way: Since *possible_alias* is on a path to *call* it must also hold on a path to *entry* (making sure to account for visibility). Thus lemma must hold.
   - 2^{nd} way: If an actual dereferenced has an alias at the call, clearly the formal dereferenced must be aliased to the same thing at *entry* (modulo

visibility). Thus the lemma must hold.

- 3^{rd} way: If two actuals are aliases, clearly their corresponding formals (when dereferenced) must also be dereferenced. The lemma must hold.

- Effects on corresponding return node *return*:

  This is simply a clever encoding of the *holds* relation in Figure 5.9. By the induction hypothesis *may-hold*($[(exit, \mathcal{AA}), \langle x, y \rangle]$), where $\mathcal{AA}$ is in $bind_{call}$(*possible_alias*), is either a type 2, 3, or 4 approximation or ($exit$, $\langle x, y \rangle$) is in $limit_k$(precise solution). In the first case, by definition may-hold($[(return, assumed_alias), \langle x, y \rangle]$) is derived from a type 2, 3, or 4 approximation. The second case is more complicated. We indirectly proved in Appendix C.3.5 that if an alias holds on a path to *exit* requires no assumptions or a single assumed alias at *entry* then the relation in Figure 5.9 is valid. Thus if $\langle x, y \rangle$ depends only on $\mathcal{AA}$ at *entry* for some path the encoding is valid and the lemma follows. If $\langle x, y \rangle$ depends on $\mathcal{AA}$ plus other assumptions on some path then $\langle x, y \rangle$ is an approximation at *exit*. This is because we only have information about single aliases on a path so anything that requires information about multiple aliases must be an approximation. Since $\langle x, y \rangle$ is an approximation at *exit*, by induction (*may-hold*($[(exit, \mathcal{AA}), \langle x, y \rangle]$) must have been made true at an earlier iteration) $\langle x, y \rangle$ must be a type 2, 3, or 4 approximation and thus $\langle x, y \rangle$ is derived from a type 2, 3 or 4 approximation.

2. Alias_at_exit_implies($exit, assumed_alias, possible_alias$) (Chapter 5.2.4)

   By the induction hypothesis we know *possible_alias* is in $limit_k$(precise solution) or is (or derived by) a type 2, 3, or 4 approximation. If it is a type 2, 3, or 4 approximation than any *may-holds* that is set true is set true because of *possible_alias* and thus is derived by a type 2, 3, or 4 approximation and satisfies the lemma. Assume this is not the case (i.e., *possible_alias* is in $limit_k$(precise solution)). This routine is also simply a clever encoding of the *holds* relation in Figure 5.9. By the induction hypothesis *may-hold*($[(call, \mathcal{AA}), \mathcal{PA}]$),

where *assumed_alias* in $bind_{call}(\mathcal{PA})$, is either a type 2, 3, or 4 approximation or (*call*, PA) is in $limit_k$(precise solution). In the first case, by definition may-hold($[(return, assumed_alias), \langle x, y\rangle]$) is derived from a type 2, 3, or 4 approximation. The second case is more complicated. We indirectly proved in Appendix C.3.5 that if an alias holds on a path to *exit* requires no assumptions or a single assumed alias at *entry* then the relation in Figure 5.9 is valid. Thus if $\mathcal{PA}$ depends only on $\mathring{A}$ at *call* for some path the encoding is valid and the lemma follows. If $\mathcal{PA}$ depends on $\mathcal{AA}$ plus other assumptions on some path then $\mathcal{PA}$ is an approximation at *exit*. This is because we only have information about single aliases on a path so anything that requires information about multiple aliases must be an approximation. Since $\mathcal{PA}$ is an approximation at *call*, by induction (*may-hold*($[(call, \mathcal{AA}), \mathcal{PA}]$) must have been made true at an earlier iteration) $\mathcal{PA}$ must be a type 2, 3, or 4 approximation and thus $\mathcal{PA}$ is derived from a type 2, 3, or 4 approximation.

3. Any_other_alias_implies(*node*,*assumed_alias*,*possible_alias*) (Chapter 5.2.5)
   Let *possible_alias* $\equiv \langle y, z\rangle$. By the induction hypothesis we know *possible_alias* is in $limit_k$(precise solution) or is (or derived by) a type 2, 3, or 4 approximation. If it is a type 2, 3, or 4 approximation than any *may-holds* that is set true is set true because of *possible_alias* and thus is derived by a type 2, 3, or 4 approximation and satisfies the lemma. Assume this is not the case (i.e., *possible_alias* is in $limit_k$(precise solution)).

   - Successor is a call node, exit node, or a program statement which is not an assignment to a pointer:
     These nodes simply collect *may-hold* information from their parents, thus since the lemma holds for *possible_alias* at *node*, it also holds for this case.
   - Successor (*succ*) is an assignment to a pointer: (e.g., $p = q$)
     There are three cases to consider.
     - Does the assignment preserve the alias?
       This is true when p is a prefix of neither y nor z.

To make the following discussion easier, let $y \equiv *y_1$ and $z \equiv *z_1$. The only way that the alias $\langle *y_1, *z_1 \rangle$ cannot hold on a path to *succ* given that it holds on a path to *node* is if either y_1 or z_1 is redefined by *succ* ("$p = q$") on every path to *succ* on which $\langle *y_1, *z_1 \rangle$ holds. Clearly for y_1 (or z_1) to get redefined by this statement, y_1 must be an alias of p. Thus $\langle *y_1, *z_1 \rangle$ is approximate at *succ* only if $\langle p, y_1 \rangle$ or $\langle p, z_1 \rangle$ holds on every path up to (but not including) *succ* on which $\langle *y_1, *z_1 \rangle$ does. That is the definition of the 3^{rd} type of approximation and thus the lemma holds.

– What are the effects of an alias of $*q$?

This case is applicable when is_prefix_with_deref(q,y).

Consider $y \equiv *q$. If z is something like $p{->}next$ (case 2.ii, Chapter 5.2.5) our algorithm doesn't create any new aliases. Otherwise (case 2.i, Chapter 5.2.5), $\langle *p, z \rangle$ must hold on a path to *succ* whenever $\langle *q, z \rangle$ holds to *node*. Clearly $*p$ must refer to the same object after the assignment as z referred to before the assignment. To make the discussion easier, let $z \equiv *z_1$. $*z_1$ is a different object after the assignment only if $\langle p, z_1 \rangle$ holds on the same path to *succ*. However, when $\langle p, z_1 \rangle$ holds on a path to *succ*, $\langle *p, *z_1 \rangle$ must also hold (by our Assumption 5.1.1 (NULL)) and thus $\langle *p, *z_1 \rangle$ must be in the precise solution.

This leaves only case 2.iii (Chapter 5.2.5), but that case specifically handles the 2^{nd} source of approximation and the lemma must follow.

– What are the effects of an alias of p?

This case is applicable when $y = p$.

By the induction hypothesis we know *possible_alias* is in $limit_k$(precise solution) or is (or derived by) a type 2, 3, or 4 approximation. If it is a type 2, 3, or 4 approximation than any *may-holds* that is set true is set true because of *possible_alias* and thus is derived by a type 2, 3, or 4 approximation and satisfies the lemma. Assume this is not the case (i.e., *possible_alias* is in $limit_k$(precise solution)).

For case 3.i (Chapter 5.2.5), $\langle *z, *q \rangle$ holds on a path to *succ* if $\langle y, z \rangle$ holds to *node* unless $*q$ refers to a different object after the assignment. Case 3.i does nothing in the cases where p or z is a prefix of q, so for case 3.i to generate an erroneous alias, $\langle p, v \rangle$ (where v is a prefix of q) also holds on the same path. This is the 4^{th} type of approximation, and thus the lemma follows.

Any alias generated by case 3.ii (Chapter 5.2.5), meets the requirements of the lemma at *succ* by the Assumption 5.1.1 (NULL).

This leaves only case 3.iii (Chapter 5.2.5), but that case specifically handles the 2^{nd} source of approximation and the lemma must follow.

□

References

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Ban78] J. P. Banning. *A Method for Determining the Side Effects of Procedure Calls*. PhD thesis, Department of Electrical Engineering, Stanford University, 1978.

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

[Bec85] L. L. Beck. *System Software*. Addison-Wesley, Reading, Massachusetts, 1985.

[Bur90] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[Car88] M. D. Carroll. A new pointer-removing program transformation. Unpublished manuscript, 1988.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[CK84] 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.

[CK87a] K. Cooper and K. Kennedy. Complexity of interprocedural side-effect analysis. Computer Science Department Technical Report TR87-61, Rice University, October 1987.

[CK87b] K. Cooper and K. Kennedy. Efficient computation of flow-insensitive interprocedural summary information: A correction. Computer Science Department Technical Report TR87-60, Rice University, October 1987.

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

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

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

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

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

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

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

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

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

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977.

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

[HN90] 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.

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

[HRB88] 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.

[HS90] 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.

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

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

[Kil73] G. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, January 1973.

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

[Lar89] J. R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors.* PhD thesis, University of California Berkeley, May 1989.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, July 1988. SIGPLAN NOTICES, Vol. 23, No. 7.

[Lom77] D. Lomet. Data flow analysis in the presence of procedure calls. *Journal of Research and Development*, 21(6):559–571, November 1977.

[LR90] 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.

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

[LS88] S. B. Lippman and B. Stroustrup. Pointers to class members in c++. In *Proceedings of the 1988 USENIX C++ Conference*, pages 305–323, 1988.

[MR90] 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.

[MR91] T. J. Marlowe and B. G. Ryder. Hybrid incremental alias algorithms. In *Proceedings of the Twentyfourth Hawaii International Conference on System Sciences, Volume II, Software*, pages 428–437, January 1991.

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

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

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

[PW86] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

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

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

[SP81] 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.

[Wei80a] 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.

[Wei80b] 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.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

Vita

William Alexander Landi

1982-86 B.S. in Computer Science, Cook College of Rutgers University, New Brunswick, NJ.

1986-88 M.S. in Computer Science, Rutgers University New Brunswick, NJ.

1986-87 Teaching Assistant, Department of Computer Science.

1988 Lecturer/Course Coordinator for Introduction to Computer Science.

1988-90 Research Assistant to Professor Barbara Ryder.

1990-91 Research Fellow. Siemens Corporate Research.

1990 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.

1991 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*, January 1991.

1991 H.D. Pande, B.G. Ryder, and W. Landi. Interprocedural Def-Use Associations in C Programs. In *Proceedings of the Fifth Testing, Analysis, and Verification Symposium*, October 1991.

1992 Ph.D. in Computer Science.