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

   4.3. Programs with Dynamic Pointers ................................ 34
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. \( P \)-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,0) ............................................. 83
  Computing bind(call,(x, y)) ...................................... 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 $COND_{node}^{alias\_pair}$ 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 \textit{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. \textit{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 \( CON \ D_{node}^{alias-pair} \) ............ 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 Interprocedural 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 \([L3, \{\text{false.next}, \text{no}\}] \) in Intraprocedural May Alias

4.17. 3-SAT solution iff \([L3, \{\text{true \_i} \mid 1 \leq i \leq n\}] \) in Intraprocedural May

\[ \text{Set Alias} \] .......................................................... 61

4.18. 3-SAT solution iff \([L3, \{\text{true \_i} \mid 1 \leq i \leq n\}] \) in Interprocedural May

\[ \text{Set Alias} \] .......................................................... 64

4.19. 3-SAT solution iff \([L3, \{\text{true \_i} \mid 1 \leq i \leq n\}] \) in Interprocedural May

\[ \text{Set Alias} \] .......................................................... 65

4.20. Reduction of \(L\) to an alias problem ........................................ 68

4.21. Justification of code for state \(s_j\) 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 \(\text{bind}(\text{call, } \emptyset)\) .................................. 85

5.4. Support functions for computing \(\text{bind}(\text{call, } \langle x, y \rangle)\) .......... 86

5.5. Computing \(\text{bind}(\text{call, } \langle x, y \rangle)\) .................................. 87

5.6. Computing \(\text{may}-\text{hold}\) .............................................. 88

5.7. \(\text{Aliases. introduced. by. assignment(node)}\) ............................. 89

5.8. \(\text{Aliases. introduced. by. call(node)}\) .................................... 89

5.9. \(\text{holds}\) relation at return nodes ..................................... 91

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

5.11. \(\text{may}-\text{hold}\) representation for interesting stores of Figure 5.10 ......... 96

5.12. Implication of a known \(\text{may}-\text{hold}\) at an exit node ..................... 99

5.13. Action for \(\text{may}-\text{hold}([(\text{node, assumed. alias}),(y, z)])\) for successor, \(\text{succ}\)

\[ \equiv \text{"p = q", of node and is. prefix. with. deref(q, y)} \] .............. 103

5.14. Action for \(\text{may}-\text{hold}([(\text{node, assumed. alias}),(p, v)])\) for successor, \(\text{succ}\)

\[ \equiv \text{"p = q", of node} \] .............................................. 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 $NP$-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:
\(s_0: \; i = 0;\)
\(\star p = 0;\)
while \((i < 2)\)
{
\(s_1: \; x = 5;\)
\(s_2: \; \star p = \star p + x;\)
\(i = i + 1;\)
}
\(s_3:\)

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

\(s_0: \; i = 0;\)
\(\star p = 0;\)
\(s_1: \; x = 5;\)
while \((i < 2)\)
{
\(s_2: \; \star p = \star p + x;\)
\(i = i + 1;\)
}
\(s_3:\)

If \(\star p\) is not aliased to \(x\) at \(s_0\), both versions of the code will have \(x = 5\) and \(\star p = 10\) at \(s_3\). However, if \(\star p\) is aliased to \(x\) at \(s_0\), the original version would have \(x = \star p = 10\) at \(s_3\), but the “optimized” version would have \(x = \star p = 20\) 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\textsuperscript{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

\textsuperscript{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 (\(P\) vs \(NP\)-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 \(NP\)-hard or co-\(NP\)-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\textsuperscript{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\textsuperscript{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\textsuperscript{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

\textsuperscript{2}Precision is a measure of the erroneous information contained in the solution.

\textsuperscript{3}However, it is possible for the intraprocedural alias to be in the precise solution and the interprocedural alias not to be.

\textsuperscript{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 \((\mathcal{N}, \mathcal{E}, \rho)\) where: \(\rho\) is the entry node for \textit{main}; \(\mathcal{N}\) contains one node for each statement in the program, an \textit{entry} and \textit{exit} node for each procedure, a \textit{call} and \textit{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 \textit{call} and a \textit{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 *l*ist, *l*ist(\*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 been 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
<table>
<thead>
<tr>
<th>Rule</th>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
</table>
| 1    | OBJECT₁ → *(OBJECT₂) | if can_deref(OBJECT₂.type)  
then OBJECT₁.type :=  
deref_type(OBJECT₂.type)  
else OBJECT₁.type := error |
| 2    | OBJECT₁ → (OBJECT₂).field¹ | if is_field_of(field, OBJECT₂.type)  
then OBJECT₁.type :=  
field.type(OBJECT₂.type, field)  
else OBJECT₁.type := error |
| 3    | OBJECT → var² | OBJECT.type := type of var |

¹ field is any declared field of any structure in the program.
² 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 \([O(number\_vars \times (max\_number\_fields\_in\_one\_struct)^k)]\). 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 \( k = 1 \), \( p->f₁->f₂ \) would be represented by \( p->f₁ \) (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_name₁, object_name₂) returns true iff object_name₁ can be transformed into object_name₂ 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, 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.

---

6This is the same as the value of the type attribute for object_name obtained from Figure 2.2.
<table>
<thead>
<tr>
<th>object_name_1</th>
<th>object_name_2</th>
<th>initial \ object_name_3</th>
<th>final \ object_name_3</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td>(<em>p</em>)</td>
<td>&amp;q</td>
<td>q</td>
<td>true</td>
</tr>
<tr>
<td>( p-&gt;next )</td>
<td>( p-&gt;next-&gt;next )</td>
<td>&amp;q</td>
<td>q_next</td>
<td>true</td>
</tr>
<tr>
<td>(*p-&gt;next)</td>
<td>( p-&gt;next-&gt;data )</td>
<td>q</td>
<td>q_data</td>
<td>false</td>
</tr>
<tr>
<td>(*p-&gt;next)</td>
<td>( p-&gt;next-&gt;data )</td>
<td>( *r )</td>
<td>( r-&gt;data )</td>
<td>false</td>
</tr>
<tr>
<td>( p-&gt;next )</td>
<td>( p-&gt;next-&gt;data )</td>
<td>( q_next )</td>
<td>( q_next-&gt;next )</td>
<td>true</td>
</tr>
<tr>
<td>( p-&gt;next )</td>
<td>( p-&gt;next-&gt;next )</td>
<td>( r-&gt;next )</td>
<td>( r-&gt;next-&gt;next )</td>
<td>true</td>
</tr>
</tbody>
</table>

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., \( \text{is\_prefix}(b_k, b'_k) = true \)). 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.

\texttt{alias\_type}\( (\text{alias}) \) is the type of the object which is referred to by both object names in \text{alias}. This function correctly accounts for k-limiting. For example, \texttt{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 \(\langle a, b \rangle \text{ holds} \) on the realizable path \(\rho n_1 n_2 \ldots n_i \) 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 \(\langle a, b \rangle \text{ holds} \) 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 \(\{[n, \langle a, b \rangle] \mid \exists\text{ a realizable path, } \rho n_1 n_2 \ldots n_i \ldots n_{i-1} n, \text{ in the ICFG on which } \langle a, b \rangle \text{ holds}\}\).

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

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

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

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

\(^7\)ICFG 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, \( \rho n_1 n_2 \ldots n_{i-1} n \), in the CFG on which \( \langle a, b \rangle \) does not hold\}.

visible: At a call site, an object name (for example, \( *z \)) 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 \( z \) is a local variable of procedure \( P \), then the \( z \) 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 formal parameters (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 \( S \) is in Myers' solution at a program point \( p \), then on some path to \( p \), every (and only) the aliases represented in \( S \) 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 \( O(n) \) aliases, where \( n \) is the number of variables in the program, Weihl reports \( O(n^2) \) 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 Rudnik [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 *a1,*a2,...,*a_{n-1};

main ()
{
    temp = a1;
    a1 = temp;
    temp = a2;
    a2 = temp;
    ...
    temp = a_{n-1};
    a_{n-1} = temp;
}

Program has $2(n - 1)$ non-reflexive alias pairs $\bigcup_{1 \leq i \leq n-1} \{(\text{temp, } a_i), (a_i, \text{temp})\}$ 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\textsuperscript{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 (\textit{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 \textit{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 \textit{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

\textsuperscript{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 \( k\text{-limiting} \), 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 \( k\text{-limiting} \). \( k\text{-limiting} \) 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 \( k\text{-limited} \) 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.\textsuperscript{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\textsuperscript{11}.

[HN89, HN90] is an interprocedural algorithm and in the absence of recursion\textsuperscript{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

\textsuperscript{10} Of course these matrices can be viewed as adjacency matrices, and thus viewed as graphs with labeled edges.

\textsuperscript{11} They need to be restricted so that equality of path expressions can be computed in time polynomial in the size of the expressions.

\textsuperscript{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\(^\text{13}\), 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\(^\text{14}\), [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 \( O(n) \) (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, \( \text{pred} \), 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 \( \text{pred} \), \( \langle *q, *z \rangle \) 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 \( \text{pred} \) 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 \( \langle p, \ast v \rangle \) holds then \( \langle \ast (u \rightarrow n), \ast (v \rightarrow n \rightarrow n) \rangle \) does not necessarily hold:

![Diagram showing node and successor](image)

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

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 Rudnik [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 \textit{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 \( CFG=(\mathcal{N}, \mathcal{E}, \rho) \) of the procedure to be analyzed, a \textbf{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 \textit{true} iff there is a path from \( \rho \) to \( CFG\text{-node} \) in the CFG on which \textit{alias-pair} holds.\(^{18}\) Edges in the SPPAG represent a dependence between the \textit{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 \( \rho \). Thus, if \( holds([n, \langle a, b \rangle]) = holds([m_1, \langle a, b \rangle]) \lor holds([m_2, \langle a, b \rangle]) \) then the edges \langle [m_1, \langle a, b \rangle], [n, \langle a, b \rangle] \rangle \) and \langle [m_2, \langle a, b \rangle], [n, \langle a, b \rangle] \rangle \) would be in the SPPAG. If \( holds([n, \langle a, b \rangle]) = true \) regardless of any other \textit{holds} value, then the edge \langle \rho, [n, \langle a, b \rangle] \rangle \) would be in the SPPAG.

Like Chow and Rudnik [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 \([n, \langle a, b \rangle]\) 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 \textit{object name} in the alias and \( b \) is just the other name.
- if \( a = b = *p \) then \( \text{holds}([n, (\ast p, \ast p)]) \) is true iff there is a path to an immediate predecessor of \( n \) on which \( (\ast q, \ast q) \) holds.

- if \( a \) is \( *p \) \([a \neq b]\) then \( \text{holds}([n, (\ast p, b)]) \) is true iff there is a path to an immediate predecessor of \( n \) on which \( (\ast q, b) \) holds.

- otherwise \( \text{holds}([n, (a, b)]) \) is true iff there is a path to an immediate predecessor of \( n \) on which \( (a, b) \) holds.

- if \( n \) is \( "p = \&v" \)

  - if \( a = b = *p \) then \( \text{holds}([n, (\ast p, \ast p)]) \) is true

  - if \( a \) is \( *p \) \([a \neq b]\) then \( \text{holds}([n, (\ast p, b)]) \) is true iff there is a path to an immediate predecessor of \( n \) on which \( (v, b) \) holds.

  - otherwise \( \text{holds}([n, (a, b)]) \) is true iff there is a path to an immediate predecessor of \( n \) on which \( (a, b) \) holds.

- if \( n \) is \( "p = malloc()" \) or \( "p = NULL" \).

  - if \( a = b = *p \) then \( \text{holds}([n, (\ast p, \ast p)]) \) is \( \begin{cases} 
  \text{true} & \text{for } "p = malloc()" \\
  \text{false} & \text{for } "p = NULL"
\end{cases} \)

  - if \( a \) is \( *p \) \([a \neq b]\) then \( \text{holds}([n, (\ast p, b)]) \) must be \( \text{false} \) (i.e. \( (\ast p, b) \) does not hold on any path).

  - otherwise \( \text{holds}([n, (a, b)]) \) is true iff there is a path to an immediate predecessor of \( n \) on which \( (a, b) \) 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(\(n, \langle a, b \rangle \)) to false for all \(n, \langle a, b \rangle \) 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 \(\mathcal{O}(|\mathcal{N}| \times v^2)\) 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-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.

\(\Box\)

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 been 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 (\(\land\)), instead of disjoin (\(\lor\)), the predecessors of a node. Let \(\text{holds}^\land\) 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, \(\text{holds}^\land\) 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 \( \text{holds}^\wedge([n, \langle a, b \rangle]) \) to true for all \([n, \langle a, b \rangle]\) in the SPPAG.
Calculate the fixed point of the \( \text{holds}^\wedge \) 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.

\[ \square \]

### 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 \([\text{Entry}_{\text{Main}}] [r = \text{NULL}] [\text{Call}_{A(q)}] [\text{Entry}_A] [q = f] [\text{Exit}_A] [\text{Return}_{A(q)}] [r = q] \). \[\text{Call}_{A(i)} \]

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 \( A \) 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].

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 $|\mathcal{A}| \leq 1$. If more than one level of indirection is possible, it is no longer precise to consider $|\mathcal{A}| \leq 1$, but it is *safe*. 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 $P = \ldots n_i$ (where $n_1$ is the entry node for the procedure containing $n_i$ and the number of calls on the path $n_1n_2\ldots n_{i-1}$ equals the number of returns.),

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

If

$$\text{all alias pairs in the set } \mathcal{A} = \{A_1, A_2, \ldots, A_m\} \text{ holding at } n_1 \text{ and the execution of path } P \text{ implies that } \langle a, b \rangle \text{ holds at } n_i$$

then

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

24I.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

\[ \exists k (1 \leq k \leq m) \text{ such that when assuming only the alias pair } A_k \text{ at } n_1, \]
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.

\[ \square \]

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; \([ICFG\text{-}node, \text{assumed-alias}], \text{alias-pair}\) with \text{assumed-alias} being either \( \emptyset \) or a single alias pair. The fixed point of

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

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

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

\[
\text{holds}([n, \langle a, b \rangle]) = (\text{holds}([m_1, \langle c, d \rangle]) \lor ... \lor \text{holds}([m_k, \langle c, d \rangle]))
\]

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

\(^{25}\) Again our explanation of the algorithm is illustrative, but not intended to be an efficient implementable algorithm.
holds([ \langle a, \text{assumed-alias} \rangle, \langle a, b \rangle ]) = \\
(\text{holds}([\langle \text{entry}, \text{assumed-alias} \rangle, \langle a, b \rangle ]) \lor \ldots \lor \text{holds}([\langle m_k, \text{assumed-alias} \rangle, \langle c, d \rangle ]))

Thus an understanding of the SPPAG together with the realization that for an entry node, \text{holds}([\langle \text{entry}, \langle a, b \rangle \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, \text{holds}([\langle \text{entry}, \mathcal{AA} \rangle, \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:

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

\textbf{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 \text{holds}([\langle \text{return}, \text{assumed-alias} \rangle, \langle a, b \rangle ]) is true. Clearly it is true if \langle a, b \rangle holds at the exit node conditional on \text{assumed-alias}' holding at its entry and \text{assumed-alias}', conditional on \text{assumed-alias}, holds at the call node (see Figure 4.3). Let \mathcal{ASSUME} be the set of all possible assumed aliases. \text{Holds} for a return node is defined as:

\text{holds}([\langle \text{return}, \text{assumed-alias} \rangle, \langle a, b \rangle ]) = \text{holds}([\langle \text{exit}, \emptyset \rangle, \langle a, b \rangle ]) \lor \\
\bigvee_{\mathcal{AA} \in \mathcal{ASSUME}} \left( \text{holds}([\langle \text{exit}, \mathcal{AA} \rangle, \langle a, b \rangle ]) \land \text{holds}([\langle \text{call}, \text{assumed-alias} \rangle, \mathcal{AA} \rangle]) \right)

\textbf{Modeling the effects of parameter bindings}

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

---

^{26}\text{We will in places use } \mathcal{AA} \text{ to represent assumed alias sets (A from before) that are of size 0 or 1.
\[ \text{holds}([\text{Entry}_A, \text{assumed-alias}'), \text{assumed-alias}']) = \text{true} \]

\[ \text{holds}([\text{Exit}_A, \text{assumed-alias}'), (a, b)]) = \text{true} \]

\[ \text{holds}([\text{Call}_A, \text{assumed-alias}), \text{assumed-alias}']) = \text{true} \]

\[ \text{holds}([\text{Return}_A, \text{assumed-alias}), (a, b)]) = \text{true} \]

Figure 4.3: \textit{holds} at a return node (no local variables)

\(p...[\text{call}_P][\text{entry}_P]\). \textit{back-bind} is formally specified in Appendix C.1, but we present an informal definition below.

Consider \(\text{back-bind}_{\text{call}_P}(\text{assumed-alias})\) for arbitrary \(\text{call}_P\) and \(\text{assumed-alias}\):

- \(\text{assumed-alias} = \emptyset\)

  This condition always holds.

- \(\text{assumed-alias} = (c, d)\) 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. \(\text{back-bind}_{\text{call}_P}((c, d)) = (\ast a_j, \ast a_k)^{27}\)

  where \(a_j\) is the actual corresponding to \(c\) and \(a_k\) is the actual corresponding to \(d\).

- \(\text{assumed-alias} = (c, d)\) 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. \(\text{back-bind}_{\text{call}_P}((c, d)) = (\ast a_j, d)\) where \(a_j\) is the actual corresponding to \(c\).

- \(\text{assumed-alias} = (c, d)\) and neither \(c\) nor \(d\) are locals

  \((c, d)\) holds at entry iff it held at the call node. \(\text{back-bind}_{\text{call}_P}((c, d)) = (c, d)\).

- otherwise

---

\(^{27}\) We consider \(\ast & x \equiv x\)
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 return\( P() \) 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 \( \langle w,\text{non_visible} \rangle \), 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-bind'\(_{\text{call},p},(\langle a,\text{non_visible} \rangle , b)\) specifies the alias holding on any path \( p...[\text{call}_p] \) that guarantees \( a \) will be aliased to
the non-visible object name $b$ on $\rho [\text{call}_P][\text{entry}_P]$. \textit{back-bind}' is also formally specified in Appendix C.1, but an informal definition follows.

Consider $\text{back-bind}'_{\text{call}_P}(\langle a, \text{non-visible} \rangle, b)$ for arbitrary $\text{call}_P$, $a$, and $b$:

- $a$ is not a local of the called procedure.
  
  $\text{back-bind}'_{\text{call}_P}(\langle a, \text{non-visible} \rangle, b) = \langle a, b \rangle$

- $a$ is a dereferenced formal.
  
  $\text{back-bind}'_{\text{call}_P}(\langle a, \text{non-visible} \rangle, b) = \langle \ast a_j, b \rangle$ where $\ast a_j$ is the actual corresponding to $a$.

- otherwise
  
  $\langle a, \text{non-visible} \rangle$ 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 $\text{return}_Q$. We will use the following conventions:

\[
\begin{align*}
\text{P}_{\text{return}_Q} &= \rho m_1 \ldots m_i[\text{entry}_R][m_1 \ldots m_j][\text{call}_Q][\text{entry}_Q]o_1 \ldots o_k[\text{exit}_Q][\text{return}_Q] \\
\text{P}_{\text{exit}_Q} &= \rho m_1 \ldots m_i[\text{entry}_R][m_1 \ldots m_j][\text{call}_Q][\text{entry}_Q]o_1 \ldots o_k[\text{exit}_Q] \\
\text{P}_{\text{entry}_Q} &= \rho m_1 \ldots m_i[\text{entry}_R][m_1 \ldots m_j][\text{call}_Q][\text{entry}_Q] \\
\text{P}_{\text{call}_Q} &= \rho m_1 \ldots m_i[\text{entry}_R][m_1 \ldots m_j][\text{call}_Q] \\
\text{P}_{\text{entry}_R} &= \rho m_1 \ldots m_i[\text{entry}_R]
\end{align*}
\]

where $\text{return}_Q$ is in procedure $R$, both $m_1 \ldots m_j$ and $o_1 \ldots o_k$ are realizable paths with the same number of calls as returns, and $\text{entry}_Q$, $\text{exit}_Q$, $\text{call}_Q$, and $\text{return}_Q$ are the entry, exit, call, and return nodes, respectively, associated with the call. Consider the following cases:

- $a$ and $b$ are both \textit{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 \( P_{\text{return}_Q} \) iff it holds on \( P_{\text{call}_Q} \). Thus

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

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

If \( \langle a, b \rangle \) holds on \( P_{\text{return}_Q} \), it must also hold on \( P_{\text{exit}_Q} \). By Lemma 4.2.1, either no aliases need hold on \( P_{\text{entry}_Q} \) for \( \langle a, b \rangle \) to hold on \( P_{\text{exit}_Q} \) or there exists a \( \langle c, d \rangle \) such that only \( \langle c, d \rangle \) must hold on \( P_{\text{entry}_Q} \) for \( \langle a, b \rangle \) to hold on \( P_{\text{exit}_Q} \). In the first case, \( \text{holds}(\langle \text{exit, } \emptyset, \langle a, b \rangle \rangle) \) will be true (by definition of \( \text{holds} \)). In the second case, \( \text{holds}(\langle \text{exit, } \langle c, d \rangle, \langle a, b \rangle \rangle) \) must be true and \( \text{back-bind}_{\text{call}_Q}(\langle c, d \rangle) \) must hold on \( P_{\text{call}_Q} \). Again by Lemma 4.2.1 there is an \text{assumed-alias} (either \( \emptyset \) or a single alias pair) which must hold on \( P_{\text{entry}_R} \) and by definition of \( \text{holds} \),

\[
\text{holds}(\langle \text{return, assumed-alias}, \langle a, b \rangle \rangle) = \\
\text{holds}(\langle \text{exit, } \emptyset, \langle a, b \rangle \rangle) \lor \\
\bigvee_{\text{AA} \in \text{ASSUMED}} \left( \text{holds}(\langle \text{exit, } \text{AA}, \langle a, b \rangle \rangle) \land \\
\text{holds}(\langle \text{call, assumed-alias}, \text{back-bind}_{\text{call}_Q}(\text{AA}) \rangle) \right)
\]

- Assume that \( a \) is visible but \( b \) is not.

The \( \langle a, b \rangle \) holds on \( P_{\text{return}_Q} \) iff \( \langle a, \text{non-visible} \rangle \) holds on \( P_{\text{exit}_Q} \) and \( b \) is the non-visible object name “\text{non-visible}”. By Lemma 4.2.1 there is a single \( \langle a', \text{non-visible} \rangle \) which must hold on path \( P_{\text{entry}_Q} \) and thus \( \text{back-bind}_{\text{call}_Q}^{'}(\langle a', \text{non-visible} \rangle, b) \) must hold on path \( P_{\text{call}_Q} \). By Lemma 4.2.1 there is an \text{assumed-alias} (either \( \emptyset \) or a single alias pair) which must hold on \( P_{\text{entry}_R} \) and by definition of \( \text{holds} \), \( \text{holds}(\langle \text{call, assumed-alias}, \text{back-bind}_{\text{call}_Q}^{'}(\langle a', \text{non-visible} \rangle, b) \rangle) \) is true. Thus:

\[
\text{holds}(\langle \text{return, assumed-alias}, \langle a, b \rangle \rangle) = \\
\bigvee_{\langle a', \text{non-visible} \rangle \in \text{ASSUMED}} \\
\left( \text{holds}(\langle \text{exit, } \langle a', \text{non-visible} \rangle, \langle a, \text{non-visible} \rangle \rangle) \land \\
\text{holds}(\langle \text{call, assumed-alias}, \text{back-bind}_{\text{call}_Q}^{'}(\langle a', \text{non-visible} \rangle, b) \rangle) \right)
\]

A formal specification for the PAG and \( \text{holds} \) relation can be found in Appendix C.3. The fixed point of \( \text{holds} \) for the program in Figure 2.1 is in Figure 4.5. The assumed-alias \( \langle *r, *r \rangle, \langle *q, *r \rangle \), and \( \langle *r, \text{non-visible} \rangle \) are not included in \text{main}'s table because
each involves at least one local and thus these aliases could never hold on a path to \( E N T R Y_{\text{main}} \). 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 \( \text{true}^* \) entry means the value of \( \text{holds}([\text{ICFG-node, assumed-alias}], \text{possible-alias}] \) is \( \text{true} \) 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 \( \text{bind}_{\text{call}} \) functions to model parameter effects. \( \text{bind}_{\text{call}}(A) \) is the set of aliases which hold on the path \( \rho_{n_1...n_{i-2}}[\text{call}][\text{entry}_p] \) if \( A \) holds on \( \rho_{n_1...n_{i-2}}[\text{call}] \).

\( \text{bind}_{\text{call}} \) is formally specified in Appendix C.2, but may be thought of as:

\[
\text{bind}_{\text{call}}(A) = \begin{cases} 
\langle c,d \rangle & \text{[back-bind}_{\text{call}}(\langle c,d \rangle) = \emptyset] \text{ or } \\
\langle c,d \rangle & \text{[} \exists (a,b) \in A \text{ (back-bind}_{\text{call}}(\langle c,d \rangle) = \langle a,b \rangle) \text{]} 
\end{cases}
\]

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

- \( \text{may-alias}(\rho) = \emptyset \)

- if \( n \) is an entry node then \( \text{may-alias}(n) = \bigcup_{m \ll n, n} \in E \text{ (bind}_m(\text{may-alias}(m)) \)

- otherwise, \( \text{may-alias}(n) = \)

\[
\begin{cases} 
\langle a,b \rangle & \text{neither a nor b is non-visible and } \\
\langle a,b \rangle & \text{holds([\langle a,0 \rangle, \langle a,b \rangle]) = \text{true}}] \text{} \\
\langle a,b \rangle & \text{[(} \exists AA \in \text{may-alias(entry}(n)) \text{) holds([\langle n,AA \rangle, \langle n,b \rangle]) = \text{true}]}
\end{cases}
\]

**Theorem 4.2.3** There exists a polynomial time algorithm for determining precise Interprocedural May Alias sets in the presence of single level pointers.
<table>
<thead>
<tr>
<th>ICFG-node assumed-alias</th>
<th>( &lt;p, q&gt; )</th>
<th>( &lt;p, r&gt; )</th>
<th>( &lt;q, r&gt; )</th>
<th>( &lt;q, \text{nv}&gt; )</th>
<th>( &lt;r, \text{nv}&gt; )</th>
<th>( &lt;q, q&gt; )</th>
<th>( &lt;r, r&gt; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTRY( _{main} ) ( q )</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>( \tau = \text{NULL} ) ( q )</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>CALL( _{A(q)} ) ( q )</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>RETURN( _{A(q)} ) ( q )</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>( \tau = q ) ( q )</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>CALL( _{A(&amp;p)} ) ( q )</td>
<td>\text{true}^*</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>RETURN( _{A(&amp;p)} ) ( q )</td>
<td>\text{true}^*</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>EXIT( _{main} ) ( q )</td>
<td>\text{true}^*</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

\( q, \text{nv} \) \text{true}^* \text{false} \text{false} \text{false} \text{false} \text{false} \text{false}

<table>
<thead>
<tr>
<th>ICFG-node assumed-alias</th>
<th>( &lt;q, f&gt; )</th>
<th>( &lt;q, \text{nv}&gt; )</th>
<th>( &lt;f, \text{nv}&gt; )</th>
<th>( &lt;q, q&gt; )</th>
<th>( &lt;f, f&gt; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTRY( _A ) ( q )</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>( q = f ) ( q )</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>EXIT( _A ) ( q )</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>ENTRY( _A ) ( *f, f )</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>\text{true}^*</td>
</tr>
<tr>
<td>( q = f ) ( *f, f )</td>
<td>\text{true}^*</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>\text{true}^*</td>
</tr>
<tr>
<td>EXIT( _A ) ( *f, f )</td>
<td>\text{true}^*</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>\text{true}^*</td>
</tr>
<tr>
<td>ENTRY( _A ) ( *q, f )</td>
<td>\text{true}^*</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>( q = f ) ( *q, f )</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>EXIT( _A ) ( *q, f )</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>ENTRY( _A ) ( *q, \text{nv} )</td>
<td>false</td>
<td>\text{true}^*</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>( q = f ) ( *q, \text{nv} )</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>EXIT( _A ) ( *q, \text{nv} )</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>ENTRY( _A ) ( *f, \text{nv} )</td>
<td>false</td>
<td>false</td>
<td>\text{true}^*</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>( q = f ) ( *f, \text{nv} )</td>
<td>\text{true}^*</td>
<td>\text{true}^*</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>EXIT( _A ) ( *f, \text{nv} )</td>
<td>\text{true}^*</td>
<td>\text{true}^*</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

\( \text{true}^* \equiv \text{non-visible} \)

Figure 4.5: holds([\( \text{ICFG-node, assumed-alias} \), \( \text{possible-alias} \)]) for the program in Figure 2.1
Build the PAG. Initialize $\text{holds}([\mathbf{n}, \mathbf{A}], \mathcal{P} \mathcal{A})$ to $\text{false}$ for all $[(\mathbf{n}, \mathbf{A}), \mathcal{P} \mathcal{A}] \in \text{PAG}$. Calculate the fixed point of $\text{holds}$. Initialize $\text{may-alias}(n)$ to $\emptyset$ for all $n$ in the ICFG. Calculate the fixed point of $\text{may-alias}$.

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 $\text{holds}$ 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, $\text{may-alias}$ can change its value at most $O(v^2)$ times, where $v$ is the number of variables in the program. The precision of our algorithm stems from Lemma 4.2.1\textsuperscript{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 $\text{may-alias}$ 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.

\textsuperscript{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 $\text{MFP} \equiv \text{MOP}$ [Kil73].
<table>
<thead>
<tr>
<th>ICFG-node</th>
<th>may-alias(ICFG-node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTRY_{main}</td>
<td>∅</td>
</tr>
<tr>
<td>τ = NULL</td>
<td>∅</td>
</tr>
<tr>
<td>CALL A(q)</td>
<td>∅</td>
</tr>
<tr>
<td>RETURN_{A(q)}</td>
<td>∅</td>
</tr>
<tr>
<td>τ = q</td>
<td>∅</td>
</tr>
<tr>
<td>CALL A(&amp;p)</td>
<td>∅</td>
</tr>
<tr>
<td>RETURN_{A(&amp;p)}</td>
<td>{(*p, *q), (*q, *q)}</td>
</tr>
<tr>
<td>EXIT_{main}</td>
<td>{(*p, *q), (*q, *q)}</td>
</tr>
<tr>
<td>ENTRY_{A}</td>
<td>{(*f, *f)}</td>
</tr>
<tr>
<td>q = f</td>
<td>{(*q, *f), (*f, *f), (*q, *q)}</td>
</tr>
<tr>
<td>EXIT_{A}</td>
<td>{(*q, *f), (*f, *f), (*q, *q)}</td>
</tr>
</tbody>
</table>

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 *p, x \rangle holds on all paths to \( n_1 \) both \langle *q, x \rangle and \langle *r, x \rangle must be assumed at entryp. We will use \( CD^{alias-pair}_{node} \) to refer the precise Conditional Must Alias information.

\( CD^{alias-pair}_{node} \) always has a value. If, on some path to node, alias-pair doesn’t hold then \( CD^{alias-pair}_{node} \) 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 \( \mathcal{A}_i \) which is either ∅ (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 \( \mathcal{A}_i \) (i.e., the union of assumptions over all individual paths).

The possible values for \( CD^{alias-pair}_{node} \) are false and any subset of the set of all
int x, *p, *q, *r;

...  
  p()
  {
    if (x == 0)
      p = q;
    else p = r;
    n_1:
  }

On all paths to \( n_1 \langle *p, x \rangle \) holds iff both 
\( \langle *q, x \rangle \) and \( \langle *r, x \rangle \) hold on all paths to \( \text{entry}_p \).

Figure 4.8: Conditional Must Alias: an example

possible aliases (\textit{POSSIBLE-ALIASES}). These values when ordered by the restrictiveness of the Conditional Must Aliases form a lattice. \textit{false} is most restrictive since it means that no conditions will force \textit{alias-pair} to hold at node. Thus, we are just lifting the superset lattice and adding a new bottom (\textit{false}). That is \( a \subset b \) iff \( a \) is \textit{false} or \( a \supseteq b \). The meet (\( \cap \)) of two elements of the lattice is \textit{false}, if either element is \textit{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 \textit{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 \( COND_{node}^{alias-pair} \) values. Conditional Must Alias information will be computed by a maximal fixed point calculation of the \textit{must-holds} relation:

To simplify the formulas below,

- Let \( \text{must-holds}(n, \text{false}) = \text{false} \) for all nodes \( n \) in the ICFG.

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

\footnote{If \( b \) is \textit{false}, \( a \subset b \) iff \( a \) is \textit{false} also.}
Let \( \text{POSSIBLE-ALIASES} \) be the set of all possible aliases.
Define the lattice \( L = (S, \subseteq, \cap, \perp, \top) \) as:

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

Figure 4.9: Lattice for possible values of \( CON\overline{P}_{\text{node}}^{\text{alias-pair}} \)

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

- \( n \) is an entry node
  
  Clearly \( \{ (a, b) \} \) is the unique minimal set of aliases which must hold at entry.

- \( n \) is a return node (\textit{return}_Q)
  
  Assume that we are interested in whether \( (a, b) \) holds on all realizable paths \( P \) to \( \text{return}_Q \). Where, as on page 38:

\[
\begin{align*}
P_{\text{return}_Q} &= \rho_{n_1 \ldots n_r[\text{entry}_R]} m_1 \ldots m_j[\text{call}_Q][\text{entry}_Q] o_1 \ldots o_k[\text{exit}_Q][\text{return}_Q] \\
P_{\text{exit}_Q} &= \rho_{n_1 \ldots n_r[\text{entry}_R]} m_1 \ldots m_j[\text{call}_Q][\text{entry}_Q] o_1 \ldots o_k[\text{exit}_Q] \\
P_{\text{entry}_Q} &= \rho_{n_1 \ldots n_r[\text{entry}_R]} m_1 \ldots m_j[\text{call}_Q][\text{entry}_Q] \\
P_{\text{call}_Q} &= \rho_{n_1 \ldots n_r[\text{entry}_R]} m_1 \ldots m_j[\text{call}_Q] \\
P_{\text{entry}_R} &= \rho_{n_1 \ldots n_r[\text{entry}_R]} \\
\end{align*}
\]

where \( \text{return}_Q \) is in procedure \( R \), both \( m_1 \ldots m_j \) and \( o_1 \ldots o_k \) are realizable paths with the same number of calls as returns, and \( \text{entry}_Q, \text{exit}_Q, \text{call}_Q, \) and \( \text{return}_Q \) 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 \( P_{\text{return}_Q} \) if \( \langle a, b \rangle \) must hold on all \( P_{\text{exit}_Q} \) whenever \( Q \) is invoked by call site of \( \text{return}_Q \). \( \langle a, b \rangle \) must hold on all \( P_{\text{exit}_Q} \) if all aliases in \( \text{must-holds}(\text{exit}_Q, \langle a, b \rangle) \) hold on all \( P_{\text{entry}_Q} \). This is true if for each \( \langle c, d \rangle \) in \( \text{must-holds}(\text{exit}_Q, \langle a, b \rangle) \), \( \text{back-bind}_{\text{call}_Q}(\langle c, d \rangle) \) must hold on all \( P_{\text{call}_Q} \). This must be the case if each alias in

\[
\prod_{\langle c, d \rangle \in \text{must-holds}(\text{exit}_Q, \langle a, b \rangle)} \left( \text{must-holds}(\text{call}_Q, \text{back-bind}_{\text{call}_Q}(\langle c, d \rangle)) \right)
\]

holds on all \( P_{\text{entry}_H} \).

- Either \( a \) or \( b \) but not both is visible in the procedure which was called.

\( \langle a, b \rangle \) must hold on all \( P_{\text{return}_Q} \) if \( \langle a, \text{non-visible} \rangle \) must hold on all \( P_{\text{exit}_Q} \) and "non-visible" is the non-visible object \( b \). \( \langle a, \text{non-visible} \rangle \) must hold on all \( P_{\text{exit}_Q} \) if all aliases in \( \text{must-holds}(\text{exit}_Q, \langle a, \text{non-visible} \rangle) \) hold on all \( P_{\text{entry}_Q} \).

This is true if for each \( \langle c, \text{non-visible} \rangle \) in \( \text{must-holds}(\text{exit}_Q, \langle a, \text{non-visible} \rangle) \), \( \text{back-bind}_{\text{call}_Q}(\langle c, \text{non-visible} \rangle, b) \) must hold on all \( P_{\text{call}_Q} \). This must be the case if each alias in

\[
\prod_{\langle c, \text{non-visible} \rangle \in \text{must-holds}(\text{exit}_Q, \langle a, \text{non-visible} \rangle)} \left( \text{must-holds}(\text{call}_Q, \text{back-bind}_{\text{call}_Q}(\langle c, \text{non-visible} \rangle, b)) \right)
\]

holds on all \( P_{\text{entry}_H} \).

- Both \( a \) and \( b \) are not visible in the procedure which was called.

\( \langle a, b \rangle \) must hold on all \( P_{\text{return}_Q} \) iff \( \langle a, b \rangle \) must hold on all \( P_{\text{call}_Q} \).

- 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 \( \rho n_1 n_2 \cdots n_i \) (\( n = n_i \)) in the ICFG, \( \langle a, b \rangle \) holds at \( n_i \) iff \( \langle c, d \rangle \) holds on the path \( \rho n_1 \cdots n_{i-1} \) (see Section 4.2.3 and Section 4.2.1). Thus \( \text{must-holds}(n, \langle a, b \rangle) \) 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 \text{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 \( n \).

\text{must-holds} is formally specified in Appendix C.4.2. The fixed point of \text{must-holds} for the program in Figure 2.1 is in Figure 4.10.
<table>
<thead>
<tr>
<th>ICFG-node</th>
<th>possible-alias</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( p, q )</td>
</tr>
<tr>
<td>ENTRY_{main}</td>
<td>{ ( p, q ) }</td>
</tr>
<tr>
<td>( r = \text{NULL} )</td>
<td>{ ( p, q ) }</td>
</tr>
<tr>
<td>( \text{CALL}_{A(q)} )</td>
<td>{ ( p, q ) }</td>
</tr>
<tr>
<td>( \text{RETURN}_{A(q)} )</td>
<td>{ ( p, q ) }</td>
</tr>
<tr>
<td>( r = q )</td>
<td>{ ( p, q ) }</td>
</tr>
<tr>
<td>( \text{CALL}_{A(&amp;p)} )</td>
<td>{ ( p, q ) }</td>
</tr>
<tr>
<td>( \text{RETURN}_{A(&amp;p)} )</td>
<td>{ ( q, n, v ) }</td>
</tr>
<tr>
<td>( \text{EXIT}_{main} )</td>
<td>{ ( p, q ) }</td>
</tr>
</tbody>
</table>

\( \uparrow \) \( n, v \equiv \text{non-visible} \).
\( \uparrow \) because \( \text{must-holds}(\text{EXIT}_A, \langle q, n, v \rangle) = \langle f, n, v \rangle \),
back-bind_{\text{CALL}_{A(\&p)}}(\langle f, n, v \rangle, p) = \langle p, \rangle,
and \( \text{must-holds}(\text{CALL}_{A(\&p)}, \langle p, \rangle) = \emptyset \).
\( \uparrow \) because \( \text{must-holds}(\text{EXIT}_A, \langle q, n, v \rangle) = \langle f, n, v \rangle \),
back-bind_{\text{CALL}_{A(\&p)}}(\langle f, n, v \rangle, \langle *r \rangle) = \langle p, *r \rangle,
and \( \text{must-holds}(\text{CALL}_{A(\&p)}, \langle p, *r \rangle) = \langle p, *q \rangle \).

Figure 4.10: \( \text{must-holds}(\text{ICFG-node,possible-alias}) \) 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:

- \( \text{must-alias}(\varnothing) = \emptyset \)
- if \( n \) is an entry node then
  \[
  \text{must-alias}(n)^{30} = \bigcap_{\langle m, n \rangle \in \mathcal{E}} \left\{ \langle a, b \rangle \mid \langle a, b \rangle \in \text{bind}_m(\text{must-alias}(m)) \text{ and neither } a \text{ nor } b \text{ contain non_visible} \right\}
  \]
- otherwise,
  \[
  \text{must-alias}(n) = \{ \langle a, b \rangle \mid \text{must-holds}(n, \langle a, b \rangle) \subseteq^{31} \text{must-alias(entry(n))} \}
  \]

**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.

\[\square\]

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:

\[\text{An alias has to hold on all calls.}\]

\[\text{must-alias}(X) = \begin{cases} \text{false} & \text{if } X = \text{false} \\ X \subseteq Y & \text{otherwise} \end{cases}\]

\[\text{Both lattices have height } \mathcal{O}(v^2), \text{ where } v \text{ is the number of variables in the program.}\]
For all $n$ in the ICFG, all $(a, b) \in POSSIBLE\text{-}ALIASES$

initialize $must\text{-}holds(n,(a, b))$ to $\emptyset$.

Calculate the fixed point of $must\text{-}holds$.

For all $n$ in the ICFG, initialize $must\text{-}alias(n)$ to $POSSIBLE\text{-}ALIASES$.

Calculate the fixed point of $must\text{-}alias$.

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

<table>
<thead>
<tr>
<th>ICFG-node</th>
<th>must-alias(ICFG-node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ENTRY_{main}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$r = NULL$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$CALL_{A(q)}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$RETURN_{A(q)}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$r = q$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$CALL_{A(&amp;p)}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$RETURN_{A(&amp;p)}$</td>
<td>${p, *q, *q}$</td>
</tr>
<tr>
<td>$EXIT_{main}$</td>
<td>${p, *q, *q}$</td>
</tr>
<tr>
<td>$ENTRY_{A}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q = f$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$EXIT_{A}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Figure 4.12: $must\text{-}alias(ICFG-node)$ for the program in Figure 2.1
• For every procedure P, for every reference formal r in P
  
  – Replace the declaration “ref\textsuperscript{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 = p_{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.

\[ \square \]

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.

\textsuperscript{33} ref indicates reference formal
Follows from Lemma 4.3.1 and Theorem 4.2.3.

\[ \square \]

**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.

\[ \square \]

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.\(^3\)

The proof of Theorem 4.4.1 is by reduction from the 3-SAT problem for \( \bigwedge_{i=1}^{n} (i_{1} \lor l_{i,2} \lor l_{i,3}) \) with propositional variables \( \{v_{1}, v_{2}, \ldots, v_{m}\} \). 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

---

\(^3\)The intraprocedural version of this problem (Theorem 4.4.1) is a subproblem of the interprocedural version.
int **v_1, **v_2, ...;  // *v_1, *v_2, ...
int *true, *false;
int yes, no;

/* A path through this section of code 
corresponds to a truth assignment */

L1:
if (-) {v_1 = &true; v_bar_1 = &false}
else {v_1 = &false; v_bar_1 = &true}
if (-) {v_2 = &true; v_bar_2 = &false}
else {v_2 = &false; v_bar_2 = &true}
...  // 
if (-) {v_m = &true; v_bar_m = &false}
else {v_m = &false; v_bar_m = &true}

L2: false = &no;

/* The code below will break the \textit{false, no} alias iff the 
truth assignment from above makes the formula false */

if (-) *l_1 = &yes else if (-) *l_1 = &yes else *l_1 = &yes;
if (-) *l_2 = &yes else if (-) *l_2 = &yes else *l_2 = &yes;
...  // 
if (-) *l_n = &yes else if (-) *l_n = &yes else *l_n = &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, \textit{false, no}] 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} \lor l_{i,2} \lor l_{i,3}).

If we interpret \ast v_i aliased to \textit{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, (*false, no)] must be in Intraprocedural
May Alias. However if the formula is unsatisfiable then every truth assignment has a
clause, say (l_i,1 ∨ l_i,2 ∨ 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 (*false, yes) must hold on all paths to L3 and thus (*false, no) 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 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 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, (*false, yes)] 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
(*false, no) holds, and thus [L3, (*false, yes)] is not in Intraprocedural Must Alias. If
the 3-SAT formula is unsatisfiable then on every path to L3, (*false, yes) holds and
thus [L3, (*false, yes)] 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 $\bigwedge_{i=1}^{n}(l_{i,1} \lor l_{i,2} \lor l_{i,3})$ with variables $\{v_1, v_2, \ldots, v_m\}$. It is conceptually identical to the proof of Theorem 4.4.1. The reduction is specified by the program in Figure 4.14 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,(*false, no)] must be in Interprocedural May Alias. However if the formula is unsatisfiable then every truth assignment has a clause, say $(l_{i,1} \lor l_{i,2} \lor l_{i,3})$, 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

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

alias (*false, yes) must hold on all paths to L3 and thus (*false, no) 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, \overline{v_1})
    ref‡ int *v_1, *\overline{v_1};
{  
   if (¬) var_2(true, false) else var_2(false, true);
}
var_2(v_2, \overline{v_2})
    ref int *v_2, *\overline{v_2};
{  
   if (¬) var_3(true, false) else var_3(false, true)
}
...
var_m(v_m, \overline{v_m})
    ref int *v_m, *\overline{v_m};
{  
   partb()
}

/* The code below will break the \langle *false, no \rangle 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 */
var1(v1, \overline{v_1})
    ref int *v1, *\overline{v_1};
{
    if (-) var2(v1, \overline{v_1}, true, false) else var2(v1, \overline{v_1}, false, true);
}
var2(v1, \overline{v_1}, v2, \overline{v_2})
    ref int *v1, *\overline{v_1}, *v2, *\overline{v_2};
{
    if (-) var3(v1, \overline{v_1}, v2, \overline{v_2}, true, false)
        else var3(v1, \overline{v_1}, v2, \overline{v_2}, false, true)
}

... 

var_m(v1, \overline{v_1}, v2, \overline{v_2}, ..., v_m, \overline{v_m})
    ref int *v1, *\overline{v_1}, *v2, *\overline{v_2}, ..., *v_m, *\overline{v_m};
{
    partb(v1, \overline{v_1}, v2, \overline{v_2}, ..., v_m, \overline{v_m})
}

/* The code below will break the \{false, no\} alias iff the truth assignment from above makes the formula false */
partb(v1, \overline{v_1}, v2, \overline{v_2}, ..., v_m, \overline{v_m})
    ref int *v1, *\overline{v_1}, *v2, *\overline{v_2}, ..., *v_m, *\overline{v_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 (-) var1(true, false) else var1(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.

\(\square\)

### 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 Interprocedural 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 \( \land_{i=1}^{n} (l_{i,1} \lor l_{i,2} \lor l_{i,3}) \) 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 \(*\langle v_i, next \rangle\) 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, \overline{v_1}, v_2, \overline{v_2}, \ldots, v_m, \overline{v_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; \overline{v_1}.next = \&false\}
  else \{v_1.next = \&false; \overline{v_1}.next = \&true\}
  if (¬) \{v_2.next = \&true; \overline{v_2}.next = \&false\}
  else \{v_2.next = \&false; \overline{v_2}.next = \&true\}
  \ldots
  if (¬) \{v_m.next = \&true; \overline{v_m}.next = \&false\}
  else \{v_m.next = \&false; \overline{v_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;

\ldots

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,\*(false.next, no)] 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,*(false.next, no)]` must be in Intraprocedural May Alias. However if the formula is unsatisfiable then every truth assignment has a clause, say \((l_{i,1} \lor l_{i,2} \lor l_{i,3})\), 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

\[
\begin{align*}
\text{if } (\cdot) & \cdot (l_{i,1}.next).next = \&yes \\
\text{else if } (\cdot) & \cdot (l_{i,2}.next).next = \&yes \\
\text{else } & \cdot (l_{i,3}.next).next = \&yes;
\end{align*}
\]

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

\[\Box\]

### 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.

\[\Box\]
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 \( \rho n_1 n_2 \ldots n_i \) iff for each \( (a, b) \in \mathcal{A} \), \( a \) 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

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

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

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

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 \( \wedge_{i=1}^{n} (l_{i,1} \vee l_{i,2} \vee l_{i,3}) \) with variables \( \{v_1, v_2, \ldots, v_m\} \). 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 *vi* 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
```c
int *v_1,*\overline{v_1},*v_2,*\overline{v_2},\ldots,*v_m,*\overline{v_m};
int *c_1,*c_2,\ldots,*c_n;
int true,false;

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

L2:

/\* The code below will create a \{\langle *c_i, true \rangle \mid 1 \leq i \leq 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};
    \ldots
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 \([L3, \{ \langle c_i, true \rangle \mid 1 \leq i \leq n \}]\) must be
in Intraprocedural May Set Alias. However if the formula is unsatisfiable then every
truth assignment has a clause, say \((l_{j,1} \lor l_{j,2} \lor l_{j,3})\), with 3 false literals. This means
\( \ast l_{j,1}, \ast l_{j,2}, \) and \( \ast l_{j,3}\) are all aliased to false. Because every path from L2 to L3 must go
through the statement

\[
\text{if} \ (-) \ c_j = l_{j,1} \ \text{else} \ (-) \ c_j = l_{j,2} \ \text{else} \ c_j = l_{i,3};
\]

the alias \( \{ \langle c_i, true \rangle \mid 1 \leq i \leq n \} \) 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.

\[ \square \]

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 Intraprocedural May Set Alias is \( \mathcal{NP} \)-hard.

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

**Corollary 4.7.3** In the presence of multiple level pointers the problem of deter-
mining precise Intraprocedural 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 Intraprocedural 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 Intraprocedural May Set Alias is \( \mathcal{NP} \)-hard.
The proof of Theorem 4.7.2 is by reduction from the 3-SAT problem for \( \bigwedge_{i=1}^{n} (l_{i,1} \lor l_{i,2} \lor l_{i,3}) \) with variables \( \{v_1, v_2, \ldots, v_m\} \). 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 \( \text{true} \) 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 \( \text{clause}_{i-1} \) passes the true literal of the \( i^{th} \) clause to \( \text{clause}_i \). This call corresponds to passing \( \text{true} \) to \( c_i \) and [L3, \( \{ \langle c_i, \text{true} \rangle \mid 1 \leq i \leq n \} \)] must be in Interprocedural May Set Alias. However if the formula is unsatisfiable then every truth assignment has a clause, say \( (l_{j,1} \lor l_{j,2} \lor l_{j,3}) \), with 3 false literals. This means \( l_{j,1}, l_{j,2}, \) and \( l_{j,3} \) are all aliased to \( \text{false} \). Because every path from L2 to L3 must go through \( \text{clause}_{j-1} \) the alias \( \{ \langle c_i, \text{true} \rangle \mid 1 \leq i \leq n \} \) 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.

\[ \Box \]

### 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

\[ \text{Figure 4.19 is an alternate reduction that works for C scope rules.} \]
int true, false;

/* A path through this section of code corresponds to a truth assignment */
var1(v1, \overline{v1})
    ref int v1, \overline{v1};
{
    if (-) var2(true, false) else var2(false, true);
}
var2(v2, \overline{v2})
    ref int v2, \overline{v2};
{
    if (-) var3(true, false) else var3(false, true)
}

varm(vm, \overline{vm})
    ref int vm, \overline{vm};
{
    clause6()
}

/* The code below will create a \{ \langle ci, true \rangle \mid 1 \leq i \leq n \} alias iff the truth assignment from above makes the formula true */
clause6()
{
    if (-) clause1(l1, 1) else if (-) clause1(l1, 2) else clause1(l1, 3)
}
clause1(c1)
    ref int c1;
{
    if (-) clause2(l2, 1) else if (-) clause2(l2, 2) else clause2(l2, 3)
}
clause2(c2)
    ref int c2;
{
    if (-) clause3(l3, 1) else if (-) clause3(l3, 2) else clause3(l3, 3)
}

clausen(cn)
    ref int cn;
{
    clause6()
}
L3: }
L1: main()
{
    if (-) var1(true, false) else var1(false, true);
}

Note: In order for proof to work clause6() and all clausek(c_k)[1 \leq k \leq n] must be in the scope of v1, v2, ..., vm, \overline{v1}, ..., and \overline{vm}. 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, \overline{v_1})
ref\^1 int v_1, \overline{v_1};
{
  if (-) var_2(v_1, \overline{v_1}, true, false) else var_2(v_1, \overline{v_1}, false, true);
}
var_2(v_1, \overline{v_1}, v_2, \overline{v_2})
ref int v_1, \overline{v_1}, v_2, \overline{v_2};
{
  if (-) var_3(v_1, \overline{v_1}, v_2, \overline{v_2}, true, false) else var_3(v_1, \overline{v_1}, v_2, \overline{v_2}, false, true)
}

var_m(v_1, \overline{v_1}, v_2, \overline{v_2}, ..., v_m, \overline{v_m})
ref int v_1, \overline{v_1}, v_2, \overline{v_2}, ..., v_m, \overline{v_m};
{
  clause_6(v_1, \overline{v_1}, v_2, \overline{v_2}, ..., v_m, \overline{v_m})
}

/* The code below will create a \{ \langle c_i, true \rangle \mid 1 \leq i \leq n \} alias iff  
the truth assignment from above makes the formula true */
clause_6(v_1, ..., \overline{v_m})
ref int v_1, ..., \overline{v_m};
{
  L2: if (-) clause_1(v_1, ..., \overline{v_m}, l_1, 1) else if (-) clause_1(v_1, ..., \overline{v_m}, l_1, 2)
  else clause_1(v_1, ..., \overline{v_m}, l_1, 3)
}
clause_1(v_1, ..., \overline{v_m}, c_1)
ref int v_1, ..., \overline{v_m}, c_1;
{
  if (-) clause_2(v_1, ..., \overline{v_m}, c_1, l_2, 1) else if (-) clause_2(v_1, ..., \overline{v_m}, c_1, l_2, 2)
  else clause_2(v_1, ..., \overline{v_m}, c_1, l_2, 3)
}
clause_2(v_1, ..., \overline{v_m}, c_1, c_2)
ref int v_1, ..., \overline{v_m}, c_1, c_2;
{
  if (-) clause_3(v_1, ..., \overline{v_m}, c_1, c_2, l_3, 1) else if (-) clause_3(v_1, ..., \overline{v_m}, c_1, c_2, l_3, 2)
  else clause_3(v_1, ..., \overline{v_m}, c_1, c_2, l_3, 3)
}

clause_n(v_1, ..., \overline{v_m}, c_1, c_2, ..., c_n)
ref int v_1, ..., \overline{v_m}, c_1, c_2, ..., c_n;
{
L3: } }
main()
L1: if (-) var_1(true, false) else var_1(false, true);
}

\^1 ref indicates reference formal
\^2 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 \( P \)-space hard, if four levels of indirection are possible. We show that some are \( P \)-space complete, but for many of these problems, it is open as to whether they are complete for \( P \)-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 \( P \)-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 \( P \)-space under the common assumptions of static analysis.

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

First we show that both problems are in \( P \)-space. Consider the following polynomial space bounded NDTM \( M \): Given a Control Flow Graph (CFG) \( G \) with start node \( \rho \), \( M \) starts with the path \( \rho \) 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, \([\text{node}, \langle a, b \rangle]\), \([\text{node}, \langle a, b \rangle]\) is in Intraprocedural May Alias iff \( M \) can generate some path ending in \( \text{node} \) for which \( \langle a, b \rangle \) is in the alias solution. \([\text{node}, \langle a, b \rangle]\) is in the complement of Intraprocedural Must Alias iff \( M \) can generate some path ending in \( \text{node} \) and \( \langle a, b \rangle \) is not in the alias solution. Since there are only a polynomial number of such pairs, by asking if \([\text{node}, \langle a, b \rangle]\) 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 \( P \)-space is closed under complementation) are both in \( P \)-space.
To show that these problems are \( P \)-space hard is more complicated. Our proof uses the fact that a certain language is known to be \( P \)-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 \( L(R) \neq \emptyset \). \( \mathcal{L} \) is complete for \( P \)-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 \( O(n) \) such that \( M \) accepts \( L(R) \), \( |S| \leq 2n \), 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.

\[\square\]

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 \( \sigma \) (\( |\Sigma| \leq n \)) 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 \( \Sigma \) 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 \( \epsilon\cdot\text{closeur}() \) macro which is of size \( O(|S|*|\Delta|) \)

\( = O(n^3) \). It can be generated from \( R \) in time polynomial with respect to \( n \).
/* Given regular expression \( R \), \( M = (S, \Sigma, \Delta, s_0, F) \) is the NFA of Lemma 4.8.1. */

\[
\textbf{define macro} \ \epsilon-\text{closure}() \\
\textbf{repeat physically} |S| \text{ times} \\
\quad \textbf{if} (-) \\
\quad \quad \{ \star s_j = \star s_i; \ s_j = s_i; \} \\
\quad \text{else} \ \star s_i = \star s_j; \\
\quad \textbf{not\_in\_state} = \&\text{no}; \\
\quad \textbf{if} (-) \\
\quad \quad \{ \star s_j = \star s_i; \ \star s_k = \star s_i; \ s_j = s_i; \} \\
\quad \text{else if} (-) \\
\quad \quad \{ \star s_j = \star s_k; \ \star s_i = \star s_k; \ s_j = s_k; \} \\
\quad \text{else} \ \{ \star s_i = \star s_j; \ \star s_k = \star s_j; \} \\
\quad \textbf{not\_in\_state} = \&\text{no}; \\
\textbf{end repeat} \\
\textbf{end macro}
\]

\[
\text{int} \ \star\star\star\text{nc}; \\
\text{int} \ \star\star\sigma; \quad \text{once for each} \ \sigma \in \Sigma \\
\text{int} \ \star\star s, \star\star s'; \quad \text{once for each} \ s \in S \\
\text{int} \ \text{in\_state}, \text{not\_in\_state}; \\
\text{int} \ \text{yes, no, \star\star temp}; \\
\text{in\_state} = \&\text{yes}; \ \text{not\_in\_state} = \&\text{no}; \\
\text{\sigma} = \&\text{temp}; \quad \text{once for each} \ \sigma \in \Sigma \\
\text{\text{\text{s}}} = \&\text{not\_in\_state}; \quad \text{once for each} \ s \in S \\
\text{\text{s}_0} = \&\text{in\_state}; \\
\epsilon\text{-\text{\text{closure}()} } \\
\text{l}_1: \ \text{while} (-) \{ \\
\quad \text{if} (-) \text{nc} = \&\sigma_1; \quad \text{Let} \ \Sigma = \{\sigma_1, \ldots, \sigma_m\} \\
\quad \text{else if} (-) \text{nc} = \&\sigma_2; \quad \text{once for each} \ s \in S \\
\quad \quad \ldots \\
\quad \text{else} \text{nc} = \&\sigma_m; \\
\quad \text{s'} = \&\text{not\_in\_state}; \quad \text{once for each} \ s \in S \\
\quad \text{\sigma} = \&\text{\text{s}_j}; \\
\quad \text{\star\star\text{nc}} = \text{\text{s}_i}; \quad \text{once for each} \ (s_i, \sigma, s_j) \in \Delta \ (\sigma \in \Sigma) \\
\quad \text{\sigma} = \&\text{\text{\text{s}}}'; \\
\quad \text{\text{s}} = \text{\text{s}'}; \quad \text{once for each} \ s \in S \\
\text{l}_4: \ \epsilon\text{-\text{\text{closure}()} } \\
\} \\
\text{l}_2: \ \star s_f = \&\text{no}; \quad \text{once for each} \ s_f \in F \\
\text{l}_3: \\
\]

Figure 4.20: Reduction of \( L \) to an alias problem
The following variables are declared in Figure 4.20.

\[
\begin{align*}
\text{int } **\ast nc; \\
\text{int } **\ast \sigma; & \quad \text{once for each } \sigma \in \Sigma \\
\text{int } **s,**s'; & \quad \text{once for each } s \in S \\
\text{int *in\_state, *not\_in\_state;} \\
\text{int yes, no, **temp;}
\end{align*}
\]

In the program each letter \( \sigma \in \Sigma \) has a variable (called \( \sigma \)). Some string will be generated by the \texttt{while} loop; each pass through the loop adds another character to the string. \( \ast nc \) is always aliased to the next character of the string. For any path \( P \) to \( l_2 \) where \( l_1 \) has been executed at least \( m + 1 \) times, let \( string_m = c_1c_2...c_m \) where \( c_i \in \Sigma (1 \leq i \leq m) \) is the character referred to by \( nc \) on the \( i^{th} \) iteration of the \texttt{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 \( \ast s \) aliased to \texttt{in\_state} iff \( (s_0, string) \models_M (s, \epsilon) \), and thus simulate \( M \). This wasn’t possible, but we came up with a sufficient compromise.) We use the alias \( (\ast\text{in\_state, yes}) \) 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., \( \ast\text{in\_state} \) is aliased to \( \text{yes} \) on \( P \)) then \( \ast s \) is aliased to \texttt{in\_state} iff \( (s_0, string_m) \models_M (s, \epsilon) \), where \( l_1 \) appears \( m + 1 \) times on \( P \). All \( s' \), as well as \texttt{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,

\( \langle \ast\text{in\_state, yes} \rangle \) holds on \( P \) iff \( \langle \ast\text{in\_state, no} \rangle \) does not.

\[\]

There are only two “int” variables in the program; \texttt{yes} and \texttt{no}. Since \texttt{in\_state} is type “\texttt{int *}”, it has to refer to one or the other or nothing at all. An examination of the code makes it obvious that \texttt{in\_state} must always refer to something.

\[\]

**Lemma 4.8.3** For any path \( P \) up to (but not including) the code for \texttt{e\_closure()}\(^{37}\), let \( S_P = \left\{ s \mid (s, \text{in\_state}) \text{ holds on } P \right\} \).

\[\]

\(^{37}\)That is \texttt{e\_closure()} is the very next thing to be executed.
• If \( \langle \text{in.state, yes} \rangle \) holds on \( P \) then there is a path \( P' \) through \( \epsilon\text{-closure()} \) such that \( \langle \text{in.state, yes} \rangle \) holds on \( P \circ P' \)

• For all paths \( P' \) through \( \epsilon\text{-closure()} \), if \( \langle \text{in.state, yes} \rangle \) holds on \( P \circ P' \) then for all \( s_i \) \( \langle s_i, \text{in.state} \rangle \) iff \( s_i \in \left\{ s' \mid \text{for some } s \in S_P, (s, \epsilon) \vdash_M (s', \epsilon) \right\} \).

Consider \( (s_i, \epsilon, s_j) \in \Delta \) where \( \text{in-degree}(s_i) = 1 \). The following code represents adding \( s_i \) to the \( \epsilon\text{-closure()} \) if \( s_j \) is in it:

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

Besides adding \( s_j \) to the \( \epsilon\text{-closure()} \) if \( s_i \) is in it, the above code needs to insure that if \( s_i \) was in the \( \epsilon\text{-closure()} \) before this code, it is also in the \( \epsilon\text{-closure()} \) after the code. There are four possible relevant alias conditions before this code is executed (remember \( \langle \text{not.in.state, no} \rangle \) holds on all paths\(^3\)).

<table>
<thead>
<tr>
<th>aliases before executed</th>
<th>then path</th>
<th>else path</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle s_j, \text{not.in.state} \rangle )</td>
<td>( \langle s_j, \text{not.in.state} \rangle )</td>
<td>( \langle s_j, \text{not.in.state} \rangle )</td>
</tr>
<tr>
<td>( \langle s_i, \text{not.in.state} \rangle )</td>
<td>( \langle s_j, \text{not.in.state} \rangle )</td>
<td>( \langle s_j, \text{not.in.state} \rangle )</td>
</tr>
<tr>
<td>( \langle s_j, \text{in.state} \rangle )</td>
<td>( \langle s_j, \text{in.state} \rangle )</td>
<td>( \langle s_j, \text{not.in.state} \rangle )</td>
</tr>
<tr>
<td>( \langle s_i, \text{not.in.state} \rangle )</td>
<td>( \langle s_j, \text{in.state} \rangle )</td>
<td>( \langle s_i, \text{not.in.state} \rangle )</td>
</tr>
<tr>
<td>( \langle s_i, \text{in.state} \rangle )</td>
<td>( \langle s_j, \text{in.state} \rangle )</td>
<td>( \langle s_i, \text{in.state} \rangle )</td>
</tr>
</tbody>
</table>

To achieve the desired effect, \( *s_j \) should be aliased to \( \text{not.in.state} \) after execution of this code iff \( *s_j \) and \( *s_i \) are aliased to \( \text{not.in.state} \) 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 \( \langle \text{in.state, yes} \rangle \) by creating \( \langle \text{in.state, no} \rangle \). 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:

\(^3\)with possible local exceptions
if (-)
    \{ *s_j = *s_i; *s_k = *s_i; s_j = s_i; \} \quad \text{for some}
    (s_i, s_j) \in \Delta \quad \text{and}
    (s_k, s_j) \in \Delta \quad \text{such that}
    s_i \neq s_k
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;

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 in_state, no \rangle holds on paths
indicated by Figure 4.21. When \langle in_state, no \rangle holds on a path, \langle in_state, 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 in_state, 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; \} \quad \text{once for each}
    (s_i, s_j) \in \Delta \quad \text{such that}
    \text{in-degree}(s_j) = 1
else *s_i = *s_j;
not_in_state = \&no;

if (-)
    \{ *s_j = *s_i; *s_k = *s_i; s_j = s_i; \} \quad \text{once for each}
    (s_i, s_j) \in \Delta \quad \text{such that}
    \text{in-degree}(s_j) = 2
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;

in \epsilon\text{-closure()} (by simple induction on \( i \)):

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

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 \( \epsilon \)-transitions and we don't need
to consider paths with \( \epsilon \)-cycles. Therefore, the code for \( \epsilon\text{-closure()} \) satisfies Lemma
4.8.3.

\( \square \)
Let \((s_i, e, s_j) \in \Delta\), \((s_k, e, s_j) \in \Delta\), and \(s_i \neq s_k\)

Let \(s_k\text{-in} \equiv (*s_k, \text{in\_state})\) for \(k = 1, 2,\) or \(j\)

Let \(s_k\text{-not\_in} \equiv (*s_k, \text{not\_in\_state})\) for \(k = 1, 2,\) or \(j\)

Let error \(\equiv (*\text{in\_state}, \text{no})\) and thus breaking (*in\_state, yes)

<table>
<thead>
<tr>
<th>aliases before executed</th>
<th>desired (s_j) alias after execution</th>
<th>then path</th>
<th>else-then path</th>
<th>else-else path</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s_j\text{-not-in}) (s_i\text{-not-in}) (s_k\text{-not-in})</td>
<td>(s_j\text{-not-in}) (s_j\text{-not-in}) (s_j\text{-not-in}) (s_j\text{-not-in})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(s_j\text{-not-in}) (s_i\text{-not-in}) (s_k\text{-in})</td>
<td>(s_j\text{-in}) (s_j\text{-not-in}) error (s_j\text{-in}) (s_j\text{-not-in}) error</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(s_j\text{-not-in}) (s_i\text{-in}) (s_k\text{-not-in})</td>
<td>(s_j\text{-in}) (s_j\text{-in}) (s_j\text{-not-in}) error (s_j\text{-not-in}) error</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(s_j\text{-not-in}) (s_i\text{-in}) (s_k\text{-in})</td>
<td>(s_j\text{-in}) (s_j\text{-in}) (s_j\text{-in}) (s_j\text{-not-in}) error</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(s_j\text{-in}) (s_i\text{-not-in}) (s_k\text{-not-in})</td>
<td>(s_j\text{-in}) (s_j\text{-not-in}) error (s_j\text{-not-in}) error (s_j\text{-in})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(s_j\text{-in}) (s_i\text{-not-in}) (s_k\text{-in})</td>
<td>(s_j\text{-in}) (s_j\text{-not-in}) error (s_j\text{-in}) (s_j\text{-in})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(s_j\text{-in}) (s_i\text{-in}) (s_k\text{-not-in})</td>
<td>(s_j\text{-in}) (s_j\text{-in}) (s_j\text{-not-in}) error (s_j\text{-in})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(s_j\text{-in}) (s_i\text{-in}) (s_k\text{-in})</td>
<td>(s_j\text{-in}) (s_j\text{-in}) (s_j\text{-in}) (s_j\text{-in})</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.21: Justification of code for state \(s_j\) with 2 in-edges
Lemma 4.8.4 For all $w \in \Sigma^n$ there exists a path $P$ through the program in Figure 4.20 to $l_2$ on which $l_1$ appears $m$ ($m = |w| + 1$) times, $\text{string}_{m-1} = w$, and $\langle \text{in\_state, yes} \rangle$ 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 $l_2$, and in $\varepsilon$-closure(). Since $\text{in\_state}$ is of type "int *", the only place where the alias $\langle \text{in\_state, yes} \rangle$ can be destroyed (before $l_2$ is executed) is in $\varepsilon$-closure(). Thus by Lemma 4.8.3, Lemma 4.8.4 holds.

\(\Box\)

Lemma 4.8.5 For all paths $P$ through the program in Figure 4.20 to $l_1$ on which $l_1$ appears $m$ times, if $\langle \text{in\_state, yes} \rangle$ holds on $P$ then $\langle s, \text{in\_state} \rangle$ holds on $P$ iff $(s_0, \text{string}_{m-1}) \vdash^* M (s, \varepsilon)$.

Consider the meaning of a path $P_m = P_{m-1} \circ \ldots \circ l_1$ through the program in Figure 4.20 to $l_1$ where $l_1$ occurs $m$ times and if $m > 0$ then $l_1$ occurs $m - 1$ times on $P_{m-1}$ which also ends at $l_1$. Proof of Lemma 4.8.5 is by induction on $m$.

**Basis:**

The beginning of the initialization phase is:

```plaintext
in\_state = &yes;
not\_in\_state = &no;
\sigma = &temp; \quad \text{once for each } \sigma \in \Sigma
```

The alias $\langle \text{in\_state, yes} \rangle$ represents the fact that the path is thus far a legal simulation of $M$. The alias $\langle \text{not\_in\_state, no} \rangle$ is an invariant. There are actually a few very local places where $\langle \text{not\_in\_state, yes} \rangle$, but when this happens the invariant alias is immediately restored. The assignment "$\sigma = \&\text{temp};"$ simply points all $\sigma$ to some unimportant location.

The rest of the initialization phase is:

```plaintext
s = &not\_in\_state; \quad \text{once for each } s \in S
s_0 = &in\_state;
\varepsilon\text{-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, \( \epsilon\text{-closure()} \) either destroys the alias \( \langle \text{in.state}, \text{yes} \rangle \) 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 \( \langle \text{in.state}, \text{yes} \rangle \) 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 FACT1: \( (s_0, \text{string}_{m-2}) \vdash^*_M (s, \epsilon) \) iff \( \langle s, \text{in.state} \rangle \) holds on \( P_{m-1} \).

The code:

\[
\begin{align*}
\text{if (\( (-) \) nc = &\sigma_1; } \\
\text{ else if (\( (-) \) nc = &\sigma_2; } \\
\text{ ... eln \( = \) &\sigma_m; }
\end{align*}
\]

Let \( \Sigma = \{ \sigma_1, \ldots, \sigma_m \} \)

simply generates the next character of the string being considered. FACT2: The code

\[
\begin{align*}
\text{\( s' \) = &\text{not.in.state};} & \quad \text{once for each } s \in S \\
\sigma = &\text{s'j;} & \quad \text{once for each} \\
\text{**nc = si;} & \quad (s_i, \sigma, s_j) \in \Delta \text{ (} \sigma \in \Sigma \) \\
\sigma = &\text{temp;} & \quad \text{unique for } s_j \text{ by Lemma 4.8.1}
\end{align*}
\]

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

FACT1 and FACT2 imply \( (s_0, \text{string}_{m-2}) \vdash^*_M (s, nc_{m-1}) \vdash^*_M (s', \epsilon) \) iff \( \langle s', \text{in.state} \rangle \) holds on \( P_m \) up to the last \( l_4 \). The end of the loop is

\[
\begin{align*}
\text{\( s = s' \);} & \quad \text{once for each } s \in S \\
l_4: & \quad \epsilon\text{-closure()}
\end{align*}
\]
and thus given Lemma 4.8.3, Lemma 4.8.5 holds for $P_m$.

\[\square\]

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

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

Claim 2: $R \in \mathcal{L}$ iff $[l_3, \langle \textit{in\_state, no} \rangle]$ 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 $w \notin L(R)$. By Lemma 4.8.4 there is a path $P = \rho \ldots l_1 l_2$ on which $l_1$ is executed $m$ times, $\text{string}_{m-1} = w$, and $\langle \textit{in\_state, yes} \rangle$ holds on $P$. By Lemma 4.8.5 (because the same aliases that hold on $P = \rho \ldots l_1 l_2$ also hold on $\rho \ldots l_1$), for all states $s \langle s, \textit{in\_state} \rangle$ holds on $P$ iff $(s_0, \text{string}_{m-1}) \vdash_M^* (s, e)$. Since $w \notin L(R)$, all $s$ aliased to 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 in_state and $\langle \textit{in\_state, yes} \rangle$ holds on $P$ extended to $l_3$. Therefore, $[l_3, \langle \textit{in\_state, yes} \rangle]$ is in May Alias and $[l_3, \langle \textit{in\_state, no} \rangle]$ is in the complement of Must Alias.

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

Assume for the program in Figure 4.20 there was a path $P = P_{l_1} \circ l_2 \ldots l_3 (P_{l_1} = \rho \ldots l_1)$ on which $\langle \textit{in\_state, yes} \rangle$ holds and $l_1$ appears $m$ times. Since the code after $l_2$ can only assign &no (and yes can never be an alias of no), $\langle \textit{in\_state, yes} \rangle$ also holds on $P_{l_1}$. By Lemma 4.8.5, for all states $s \langle s, \textit{in\_state} \rangle$ holds on $P_{l_1}$ iff $(s_0, \text{string}_{m-1}) \vdash_M^* (s, e)$. Since $w \in L(R)$, for some $s_f \in F \langle s_f, \textit{in\_state} \rangle$ must hold on $P_{l_1}$. The code between $l_2$ and $l_1$ must assign $s_f$ and therefore in_state the value &no and $\langle \textit{in\_state, yes} \rangle$ can not hold on $P$. Contradiction. Thus on all paths $P \langle \textit{in\_state, yes} \rangle$ does not hold and by Lemma 4.8.2 $\langle \textit{in\_state, no} \rangle$ does. Therefore, $[l_3, \langle \textit{in\_state, yes} \rangle]$ is not in May Alias and $[l_3, \langle \textit{in\_state, no} \rangle]$ is not in the complement of Must Alias.

\[\square\]
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 𝑀̂ 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 \( \text{holds}([n_i, \mathcal{AA}, \langle a, b \rangle]) \). 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 \( \mathcal{P}A \) 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 \( \text{holds}([\text{node}, \mathcal{A}A, \mathcal{P}A]) \) for every possible triple: node in the ICFG, \( \mathcal{A}A \), and \( \mathcal{P}A \). So for a program with \( O(n) \) ICFG nodes and \( O(v) \) variables (and thus \( O(v^2) \) different possible \( \mathcal{A}A \) and \( O(v^2) \) possible \( \mathcal{P}A \)), this means a minimum of \( O(n \times v^4) \) 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 \( \text{holds} \) in a demand driven fashion so that we will only concern ourselves with \( \text{holds}([\text{node}, \mathcal{A}A, \mathcal{P}A]) \) which have the value \textit{true}. Since most of the \text{holds} relation will have the value \textit{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 \( \mathcal{A}A \) 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 \( \mathcal{A}A \) 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 \( \mathcal{A}A \) holds.

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

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

\(^{39}\)Actually, when you consider the work necessary at return nodes the complexity is \( O(n \times v^6) \)
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 \((s_p, s_q)\) 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 \((s_p, s_q)\) 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 \(\text{may-\text{hold}}([\text{node}, \mathcal{AA}], \mathcal{PA}]\) to \text{false} for all possible \text{node}, \mathcal{AA}, and \mathcal{PA}.

- Find the value of \(\text{may-\text{hold}}([\text{node}, \mathcal{AA}], \mathcal{PA}]\) for a given \text{node}, \mathcal{AA}, and \mathcal{PA}.

- Set the value of \(\text{may-\text{hold}}([\text{node}, \mathcal{AA}], \mathcal{PA}]\) for a given \text{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 \(O(n \times v^4)\) 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 \(\text{may-\text{hold}}([\text{node}, \mathcal{AA}], \mathcal{PA}]\) to \text{false} for all possible \text{node}, \mathcal{AA}, and \mathcal{PA}.

Simply initialize the hash table, which takes constant time.

---

40This “optimization” can be removed from the algorithm without great difficulty.

41This 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 \( \text{may-hold}([\text{node}, \mathcal{A}, \mathcal{P}, \mathcal{A}]) \) for a given \( \text{node}, \mathcal{A}, \mathcal{P}, \mathcal{A} \).

Look up \( \text{node}, \mathcal{A}, \mathcal{P}, \mathcal{A} \) in the hash table; if it is there, then return the value associated with it; otherwise return \text{false}.

• Set the value of \( \text{may-hold}([\text{node}, \mathcal{A}, \mathcal{P}, \mathcal{A}]) \) for a given \( \text{node}, \mathcal{A}, \mathcal{P}, \mathcal{A} \).

If \( \text{node}, \mathcal{A}, \mathcal{P}, \mathcal{A} \) is in the hash table, then simply change the value associated with it; otherwise add it.

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

5.1.3 Alias Consequences

Consider an alias between \(*p\) and \(*q\) where \(p\) and \(q\) are declared by

\[
\text{struct list \{} \text{int data; struct list } *\text{next} \text{\} } *p, *q;
\]

This alias yields the following situation:

\[
\begin{array}{c}
p \\
\downarrow \\
\hline \\
q
\end{array}
\]

Thus the alias between \(*p\) and \(*q\) implies the existence of other aliases; for example, between \(p->\text{next}\) and \(q->\text{next}\) and between \(p->\text{next}->\text{next}\) and \(q->\text{next}->\text{next}\). The existence of this last alias is based on the assumption that \(p->\text{next}\) and \(q->\text{next}\) are not \text{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,0) will be all the aliases on entry of a called procedure that must exist because of parameter bindings, while bind(call,(a, b)) 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 \( q,r, f \) are all type "int *"):
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 $(*b, x)$ and destroys the alias $(*a, x)$ at $\text{return}_P()$ 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 $\text{non-visible(type)}^3$. $\text{non-visible(type)}$ 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 $\text{non-visible}$ and leave the type implicit.

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

---

$^3$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 (*b, x) and destroys the alias pair (*a, x).

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

The occurrence of (*q, non_visible), *r) in bind(call_{P(q)}, (*q, *r)) 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, \emptyset)

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 (f, a)\textsuperscript{44} is in bind(call, \emptyset). The second occurs if two distinct formals are passed two

\textsuperscript{44}(*f, non_visible), *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 \(\text{call}\) is \(P(a, *a)\) then \(<**f_1, *f_2>\) is in \(\text{bind}(\text{call}, \emptyset)\). The algorithm for computing \(\text{bind}(\text{call}, \emptyset)\) is straightforward and is in Figure 5.3.

**Computing \(\text{bind}(\text{call}, \langle x, y \rangle)\)**

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

In this example, since \(*a_2\) is aliased to \(*a_1\) at \(P(a_1, a_2)\), \(*f_2\) 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 \(P(a_1, a_2)\), \(*f_2\) is aliased to \(*f_1\) at \(P(f_1, f_2)\). 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 \(\text{bind}(\text{call}, \langle x, y \rangle)\) is a straightforward encoding of these three cases. It can be found in Figure 5.5.
```c
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, 0)
{ 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\textsuperscript{1} form\textsubscript{i} = f_i
        apply.trans(a_i, a_j, form\textsubscript{i})
        if (can_deref(object_type(f_j)))
          bind = bind ∪ alias_consequences({deref(f_j), deref(form\textsubscript{i})})
        if (is_struct(object_type(f_j)))
          for each field (field) of object_type(f_j) which can be derefered:
            bind = bind ∪
              alias_consequences (\langle deref(field_access(f_j, field)), deref(field_access(form_i, field)) \rangle)
        }
    return bind
}
\textsuperscript{1} ‘‘let a = b’’ puts a copy of b in a.
```

Figure 5.3: Computing bind(call, 0)
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,(x,y))
/* Uses functions defined in Figure 5.4 */
bind(call, \{x, y\})
{ bind = \emptyset
  if both \(x\) and \(y\) are visible in the called procedure
      bind = bind \cup \{\{x, y\}\} \quad 1^{st} \text{ way in Chapter 5.1.4}
  if \(x\) or \(y\) (assume \(x\)) is visible in the called procedure but the other is not
      bind = bind \cup \{(x, \text{non_visible}), y\} \quad 1^{st} \text{ way in Chapter 5.1.4}
  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 \(\text{form}_i = f_i\)
            if (apply.trans(a_i, x, \text{form}_i))\(^1\)
              bind = bind \cup \text{same.object}(call, \text{form}_i, y)
            else bind = bind \cup \text{same.value}(call, \text{form}_i, y) \quad 2^{nd} \text{ way in Chapter 5.1.4}
        }
      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 \(\text{form}_i = f_i\); let \(\text{form}_j = f_j\)
              if (apply.trans(a_i, x, \text{form}_i)) and
                  apply.trans(a_j, y, \text{form}_j))\(^1\)\(^1\)
                bind = bind \cup \text{same.object}(call, \text{form}_i, \text{form}_j)
              else bind = bind \cup \text{same.value}(call, \text{form}_i, \text{form}_j) \quad 3^{rd} \text{ way in Chapter 5.1.4}
            }
    }
}

\(^1\) “let \(a = b\)” puts a copy of \(b\) in \(a\).
\(^2\) When at least one dereference occurs \(\text{form}_i\) and \(y\) must be aliases, otherwise they just contain the same value.
\(^1\) Both \(\text{form}_i\) and \(\text{form}_j\) must be dereferenced at least once for them to be aliases.
          If either \(\text{form}_i\) or \(\text{form}_j\) was not dereferenced, they contain the same value.

Figure 5.5: Computing bind(call, \{x, y\})
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
                aliases.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(\{"p = q", ∅\}, ⟨*p, *q⟩) is true.\(^{45}\) Once we have this initial set, we compute the set of all true may-holds 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(\{"p = q", ∅\}, ⟨*p, *q⟩) 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, θ>, <u, v>]) to true
      add <node, θ, <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,θ)
  { 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, θ>, deref(p), deref(q)]) and all may-hold([<node, θ>, <u, v>]) such that <u, v>
in alias.consequences(deref(p), deref(q)) are true unless p is a prefix of q. For example,
the assignment “p = p->next” does not create an alias <*p, *(p->next)>; 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 \texttt{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). \texttt{may\_hold([\texttt{call, assumed\_alias}, possible\_alias])} has effects on both its corresponding entry and return nodes.

\textbf{Effects on corresponding entry node (entry)} While \texttt{holds} as defined in Chapter 4.2.3 had the nice property that the \texttt{holds} relations which were true at a call node did not affect the \texttt{hold} relations at the entry of the called procedure, this is not true for \texttt{may\_hold} since it requires the existence of a path to the entry node with certain characteristics whereas \texttt{holds} simply assumed the existence of such a path. The effects are very simple, for each alias \(\langle a, b \rangle\) in \texttt{bind(call, possible\_alias)}, if \texttt{may\_hold([entry, \langle a, b \rangle), \langle a, b \rangle])} is \texttt{false}\footnote{In other words, \texttt{(entry, \langle a, b \rangle, \langle a, b \rangle)} has not yet been added to the worklist.}, we set \texttt{may\_hold([entry, \langle a, b \rangle], \langle a, b \rangle)} to \texttt{true} and add \texttt{(entry, \langle a, b \rangle, \langle a, b \rangle)} to the worklist.

\textbf{Effects on corresponding return node (return)} Before we can go into the details of how to handle the effects on \texttt{return}, we have to introduce the functions \texttt{back\_bind} and \texttt{back\_bind'} which have the following definitions:

\texttt{back\_bind\_call(\langle a, b \rangle)} specifies the alias on any path to \texttt{call} that guarantees \(a\) is aliased to \(b\) after control flows to corresponding entry node.

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

These definitions imply that \texttt{back\_bind\_call(\langle a, b \rangle)} = \texttt{\langle c, d \rangle} iff \texttt{\langle a, b \rangle} \in \texttt{bind\_call(\langle c, d \rangle)} and \texttt{back\_bind\'_call(\langle a, non\_visible \rangle, o)} = \texttt{\langle c, d \rangle} iff \texttt{\langle a, non\_visible \rangle} \in \texttt{bind\_call(\langle c, d \rangle)} where the \texttt{non\_visible} is the non-visible object name \(o\).
1. Rule 1 If \( x \) and \( y \) are both not visible in the called procedure:

\[
holds([\text{return, assumed} \text{ alias}], \langle x, y \rangle]) = holds([\text{call, assumed} \text{ alias}], \langle x, y \rangle])
\]

2. Rule 2 If \( x \) and \( y \) are both visible in the called procedure:

\[
holds([\text{return, assumed} \text{ alias}], \langle x, y \rangle]) = \\
holds([\text{exit, } \emptyset], \langle x, y \rangle)) \lor \\
\bigvee_{AA \in \text{ASSUMED}} \left( holds([\text{exit, } AA], \langle x, y \rangle)) \land \\
holds([\text{call, assumed} \text{ alias}, \text{back-bind}_{\text{call}}(AA)]) \right)
\]

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

\[
holds([\text{return, assumed} \text{ alias}], \langle x, y \rangle]) = \\
\bigvee_{(o, \text{non-visible}) \in \text{ASSUMED}} \left( holds([\text{exit, } \langle o, \text{non-visible} \rangle], \langle x, \text{non-visible} \rangle)) \land \\
holds([\text{call, assumed} \text{ alias}, \text{back-bind}_{\text{call}}(\langle o, \text{non-visible} \rangle, y)]) \right)
\]

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 ). \textit{may-hold} is a simple encoding of these rules with a few minor modifications. The \( holds \) rules are still valid for \textit{may-hold} because \textit{may-hold} is basically \( holds \) with the added restriction that the \textit{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 \textit{exit} with assumption not \( \emptyset \) is paired as follows:

\[
holds([\text{exit, } AA], \langle a, b \rangle)) \land holds([\text{call, assumed} \text{ alias}, \text{back-bind}_{\text{call}}(AA)])
\]

The only way that \( holds([\text{exit, } AA], \langle a, b \rangle)) \) can contribute to the value at a return node is if \( holds([\text{call, assumed} \text{ alias}, \text{back-bind}_{\text{call}}(AA)]) \) 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 \textit{exit} on which the alias \( AA \) holds. Thus this is also a valid definition for \textit{may-hold}. 

We will consider the implications of \( \text{may-hold}([\text{call, assumed\_alias}, \text{possible\_alias}]) \) on \( \text{may-hold} \) at \text{return}. Let \( \text{possible\_alias} = \langle a, b \rangle \), 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 \( \text{may-hold}([\text{return, assumed\_alias}, \langle a, b \rangle]) \) is not \text{true} then set it to \text{true} and add \( (\text{return, assumed\_alias}, \langle a, b \rangle) \) 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\(^{47}\):

   \[
   \text{may-hold}([\text{return, assumed\_alias}, \langle x, y \rangle]) = \text{may-hold}([\text{exit, } \emptyset, \langle x, y \rangle]) \lor \\
   \bigvee_{AA} (\text{may-hold}([\text{exit, } AA, \langle x, y \rangle]) \land \text{may-hold}([\text{call, assumed\_alias}, \text{back-bind\_call}(AA)]))
   \]

   We know \( \text{may-hold}([\text{call, assumed\_alias}, \langle a, b \rangle]) \).

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

   For each \( AA \) in \( \text{bind\_call}(\langle a, b \rangle) \)

   for every possible \( \langle x, y \rangle \):

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

      if \( \text{may-hold}([\text{return, assumed\_alias}, \langle x, y \rangle]) \) is \text{false} then

      \{ set \( \text{may-hold}([\text{return, assumed\_alias}, \langle x, y \rangle]) \) to \text{true}

      add \( (\text{return, assumed\_alias}, \langle x, y \rangle) \) 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 \( \text{may-hold}([\text{exit, } \emptyset, \langle x, y \rangle]) \) was \text{true}, we should make \( \text{may-hold}([\text{return, assumed\_alias}, \langle x, y \rangle]) \) \text{true} for all possible \text{assumed\_alias}. However, this is not necessary and it is sufficient to only make \( \text{may-hold}([\text{return, } \emptyset, \langle x, y \rangle]) \) \text{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 \( \text{may-hold}([[\text{exit}, AA], \langle x, y \rangle]) \).
We will do this by keeping a set \text{true_under_assumption} (TUA) for each (exit node, assumed alias) pair:

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

Given TUA, we can easily only consider the interesting \( \langle x, y \rangle \). By using the
\text{trick} we used for \text{may-hold} (Chapter 5.1.2) we can initialize all TUA to \( \emptyset \) in
constant time. Whenever we set \( \text{may-hold}([[\text{exit}', AA'], \langle x', y' \rangle]) \) to true we also add
\( \langle x', y' \rangle \) to TUA(\text{exit}', AA'). This will work if \( \text{may-hold}([[\text{exit}, AA], \langle x, y \rangle]) \) becomes
\text{true} before we need it when processing \( \text{may-hold}([[\text{call}, \text{assumed_alias}], \langle a, b \rangle]) \).\(^{48}\)
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 \text{non_visible} at \text{exit} with \( a \). Thus we would get the following action for
\text{may-hold}([[\text{call}, \text{assumed_alias}], \langle a, b \rangle]):

For each \( (AA, nv) \) in \text{bind_call}(\( (a, b) \)), for every \( \langle x, y \rangle \) on TUA(\text{exit}, AA)
(assume \( z \) contains \text{non_visible}):

\[
\{ \text{let } x' = nv; \text{apply_trans(non_visible}, x, x') \\
\quad \text{if } \text{may-hold}([[\text{return}, \text{assumed_alias}], \langle x', y \rangle]) \text{ is false then} \\
\quad \quad \{ \text{set } \text{may-hold}([[\text{return}, \text{assumed_alias}], \langle x', y \rangle]) \text{ to true} \\
\quad \quad \quad \text{add } (\text{return}, \text{assumed_alias}, \langle x', y \rangle) \text{ to the worklist.} \}
\}
\]

\(^{48}\)This may be the case when, for another call site of the same procedure, the assumption \( AA \) was
produced at the entry of the called procedure at an earlier iteration of the alias algorithm.
More Complex Effects on Return Nodes\textsuperscript{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 \textit{may\_hold} with two assumed aliases:

\[
\text{may\_hold}([[\text{exit}, \langle o_1, \text{non\_visible} \rangle, \langle o_2, \text{non\_visible} \rangle], \langle n_v_1, n_v_2 \rangle])
\]

This represents the fact that if \textit{o}_1 is aliased to non-visible object name \textit{l}_1 and \textit{o}_2 is aliased to non-visible object name \textit{l}_2 on a path to the entry of the procedure then, on some path to \textit{exit}, \textit{n}_v_1 (where the non-visible portion represents \textit{l}_1) is aliased to \textit{n}_v_2 (where the non-visible portion represents \textit{l}_2). In Figure 5.11, we show how \textit{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 \textit{may\_holds} are derived (they are indeed obtainable from our algorithm), but the figure is simply to show how \textit{may\_hold} can represent stores with aliases between non-visible object names. With this encoding scheme we can derive \textit{may\_hold}([[\textit{n}_6, \langle \ast\textit{g}_1, \text{non\_visible} \rangle], \langle \ast\textit{g}_2, \text{non\_visible} \rangle]) in Figure 5.11 because the following are true:

- \textit{may\_hold}([[\textit{n}_7, \langle \ast\textit{g}_1, \text{non\_visible} \rangle, \langle \text{non\_visible}, \text{non\_visible} \rangle]])
- \textit{may\_hold}([[\textit{n}_4, \langle \ast\textit{g}_1, \text{non\_visible} \rangle]) and \langle \ast\textit{g}_1, \textit{l}_1 \rangle = \text{back\_bind}_{\textit{m}_4} (\langle \ast\textit{g}_1, \text{non\_visible} \rangle, \textit{l}_1)
- \textit{may\_hold}([[\textit{n}_4, \langle \ast\textit{g}_2, \text{non\_visible} \rangle]) and \langle \ast\textit{g}_2, \textit{l}_2 \rangle = \text{back\_bind}_{\textit{m}_4} (\langle \ast\textit{g}_2, \text{non\_visible} \rangle, \textit{l}_2)

Fortunately, we only need this type of \textit{may\_hold} at exit nodes\textsuperscript{51}. In order to handle this, we change TUA slightly. For \textit{may\_hold}([[\text{exit}, \langle o_1, \text{non\_visible} \rangle, \langle o_2, \text{non\_visible} \rangle], \langle n_v_1, n_v_2 \rangle])

\textsuperscript{50}For example, \textit{x} = \ast\text{non\_visible}.

\textsuperscript{51}We are still considering the case where \textit{a} is not visible in the called procedure but \textit{b} is.

\textsuperscript{51}However, in Figure 5.11, we included an alias of this type at \textit{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
Figure 5.11: \textit{may-hold} representation for interesting stores of Figure 5.10
we would add \((nv_1, nv_2)\), \((o_2, \text{non-visible})\) being the other condition necessary for \((nv_1, nv_2)\), to TUA\((\text{exit}, o_1, \text{non-visible})\) and \((nv_1, nv_2)\), \((o_1, \text{non-visible})\) to TUA\((\text{exit}, o_2, \text{non-visible})\). To handle \(\text{may-hold}([\text{call}, \text{assumed-alias}], \langle a, b \rangle)\) correctly we need the following function:

\[
\text{approximate-when-both-non-visible} \quad \text{(call-node, } \langle nv_1, nv_2 \rangle, \non\text{visible-name1-is, non-visible-name2-is, name1-AA, name2-AA)}
\]

substitutes \(\text{non-visible-name1-is}\) for \(\text{non-visible}\) in \(nv_1\), substitutes \(\text{non-visible-name2-is}\) for \(\text{non-visible}\) in \(nv_2\), and uses the assumed aliases \(\text{name1-AA}\) and \(\text{name2-AA}\) 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 \(\text{may-hold}([\text{call}, \text{assumed-alias}], \langle a, b \rangle)\) is:

For each \((\text{AA}, nv)\) in \(\text{bind-call}([\text{\langle a, b \rangle}])\)

for every \(\langle x, y \rangle\) or \(\langle x, y \rangle, \text{other-condition}\) on TUA\((\text{exit}, \text{AA})\)

if both \(x\) and \(y\) contain \(\text{non-visible}\)

for each \((\text{AA}, nv_2)\) such that \(\text{may-hold}([\text{\langle \text{AA} \rangle, alias}])\) and

\((\text{\langle non-visible, other-condition \rangle}, nv_2)\) in \(\text{bind-call}(\text{alias})\)

\(\text{approximate-when-both-non-visible}(\text{call}, \langle x, y \rangle, nv, nv_2, \text{assumed-alias, AA})\)

else (assume \(x\) contains \(\text{non-visible}\)):

let \(x' = nv\); apply\_trans(\text{non-visible}, x, x')

if \(\text{may-hold}([\text{\langle \text{return-alias} \rangle, \langle x', y \rangle}])\) is \text{false} then

set \(\text{may-hold}([\text{\langle \text{return-alias} \rangle, \langle x', y \rangle}])\) to \text{true}

add \(\text{\langle return-alias, \langle x', y \rangle} \rangle\) to the worklist.

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

\((\text{AA}, nv_2)\) such that for some \(\text{alias}\), \(\text{may-hold}([\text{\langle \text{call-AA} \rangle, \text{alias} \rangle})\)

and \((\text{\langle non-visible, other-condition \rangle}, nv_2)\) in \(\text{bind-call}(\text{alias})\)
in time linear to the number of such \((\mathcal{A}, n, v_2)\). We will do this with a data structure \texttt{back_bind_true} (BBT) which is defined as follows:

\[
\texttt{back bind true}(\texttt{call, assumed alias}) =
\begin{cases}
(\mathcal{A}, n, v) & (\exists \texttt{alias}) \texttt{ may hold}([\texttt{call}, \mathcal{A}, \texttt{alias}]) \text{ and } (\texttt{assumed alias}, n, v) \in \text{bind}_{\text{call}}(\texttt{alias}) \\
\mathcal{A} & (\exists \texttt{alias}) \texttt{ may hold}([\texttt{call}, \mathcal{A}, \texttt{alias}]) \text{ and } \texttt{assumed alias} \in \text{bind}_{\text{call}}(\texttt{alias})
\end{cases}
\]

If \(\mathcal{A}\) has an object name with non_visible in it otherwise

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 \texttt{may hold} to true. We will continue to omit this check in our code. Appendix D.2 contains the final algorithm for computing \texttt{Alias at call implies(node, assumed alias, possible alias)}.

5.2.4 \texttt{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 \texttt{Alias at call implies}. Given that we know \texttt{may hold}([\texttt{exit}, \mathcal{A}, \{x, y\}]) is true, specifies the rule in Figure 5.9 leaving only one free variable, \texttt{assumed alias} (see Figure 5.12). Since \texttt{Alias at exit implies} presents no problems that weren’t already addressed by \texttt{Alias at call implies}, we omit an informal description of this routine. Pseudocode for it can be found in Appendix D.3.
\[
\text{may-\textit{hold}}([\text{return, assumedalias}], \langle x, y \rangle) = \text{may-\textit{hold}}([\text{exit, } \emptyset], \langle x, y \rangle) \lor \\
\bigvee_{AA} (\text{may-\textit{hold}}([\text{exit, } AA], \langle x, y \rangle) \land \text{may-\textit{hold}}([\text{call, assumedalias}], \text{back-bindcall}(AA)))
\]

We know \text{may-\textit{hold}}([\text{exit, } AA], \langle x, y \rangle).

Figure 5.12: Implication of a known \text{may-\textit{hold}} at an exit node

5.2.5 \textbf{Any\_other\_alias\_implies(node, assumedalias, possiblealias)}

The implications of \text{may-\textit{hold}}([\text{node, assumedalias}, possiblealias]) depends strongly on its successors. These implications must be considered separately for each successor. Since we have already examined the cases where \textit{node} is a call or exit node, the successors of \textit{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).

\textbf{Successor is a call node, exit node, or a program statement which is not an assignment to a pointer:} These nodes simply collect \text{may-\textit{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 \text{may-\textit{hold}}([\text{node, assumedalias}, possiblealias]) is simple in the case where its successor, \textit{succ}, is of one of these types.

\[
\text{if } \text{may-\textit{hold}}([\text{succ, assumedalias}, possiblealias]) \text{ is false}
\]

\[
\text{set } \text{may-\textit{hold}}([\text{succ, assumedalias}, possiblealias]) \text{ to true}
\]

\[
\text{add (succ, assumedalias, possiblealias) to the worklist}
\]

\textbf{Successor (succ) is an assignment to a pointer:} This case encompasses the major intraprocedural affects of pointers on aliasing. The effects of \text{may-\textit{hold}}([\text{node, assumedalias}, possiblealias]) depends on the relationship of the object names in \textit{possiblealias} and the object names involved in the pointer assignment. In the following discussion we will consider \textit{succ} to the be statement \textquotedblleft p = q\textquotedblright, where \textit{p} and \textit{q} are arbitrary object names of pointer type (not necessarily simple variable names). What follows is a case analysis. The effects of \textit{possiblealias} 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 \( q \).

   ![Diagram](attachment:node-succ.png)

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

2. \texttt{possible_alias} = \( \langle y, z \rangle \) where \texttt{is.p}refix. \_with. \_deref(q, y).

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

i. \texttt{not is.p}refix(p, z)

ii. \texttt{is.p}refix(p, z)

iii. Interaction of \( \langle \star q, z \rangle \) 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 \texttt{is.p}refix(p, z). The two types of effects are characterized by the following examples:

In case 2.i, \texttt{may.hold([\texttt{node}, \_assumed.\_alias], \langle \star q, z \rangle \rangle)} implies \texttt{may.hold([\texttt{succ}, \_assumed.\_alias], \langle \star q, z \rangle \rangle)} and \texttt{may.hold([\texttt{succ}, \_assumed.\_alias], \langle \star p, z \rangle \rangle)}.

While in
case 2.ii, \textit{may-hold}([[\textit{node}, \textit{assumed\_alias}], \langle *p, *q \rangle]) gives no information about the aliasing that occurs at \textit{succ}. Thus it would seem that the action when \textit{may-hold}([[\textit{node}, \textit{assumed\_alias}], \langle y, z \rangle]) is \textit{true} for \textit{succ} = "p = q" where is\_prefix\_with\_deref\(q, y)\) would be:

\[
\begin{align*}
\text{if } & \text{!is\_prefix}(p, z) \\
& /* \text{is\_prefix}(p, z) \Rightarrow \text{case 2.ii}; \text{!is\_prefix}(p, z) \Rightarrow \text{case 2.i*} /
\\
& \{ \text{let } p' = p \\
& \quad \text{apply\_trans}(q, y, p') \\
& \quad \text{if } \text{may\_hold}([[\textit{succ}, \textit{assumed\_alias}], \langle p', z \rangle]) \text{ is false} \\
& \quad \{ \text{set } \text{may\_hold}([[\textit{succ}, \textit{assumed\_alias}], \langle p', z \rangle]) \text{ to true} \\
& \quad \quad \text{add } (\textit{succ}, \textit{assumed\_alias}, \langle p', z \rangle) \text{ to the worklist} \\
& \quad \}
\}
\end{align*}
\]

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

\[
\begin{align*}
\text{node: } & (\quad ) \\
\text{succ: } & \langle p = q; \rangle \\
\end{align*}
\]

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

\footnote{The only alias that holds at \textit{succ} in case 2.ii. is \langle *p, *q \rangle which holds regardless of the alias situation at \textit{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, on1, AA1, on2, AA2) When safely.make_alias is invoked we know that ⟨on1, on2⟩ holds on a path to node. However, we also know that two assumptions AA1 and AA2 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

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

can be implemented efficiently, similarly to BBT and TUA. As with those two, this action will only work correctly if \textit{may\textendash hold([\langle node.\mathcal{AA}, p, v \rangle]) becomes true before}

\textit{may\textendash hold([\langle node.\mathcal{AA}, \langle y, z \rangle \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:

![Diagram of node and succ with p = q]
Case 3.i: \( \textit{may-hold}([\text{node, assumed_alias}], \langle p, \ast u \rangle) \) implies
\( \textit{may-hold}([\text{succ, assumed_alias}], \langle p, \ast u \rangle) \) and, unless \( u \) or \( p \) is a prefix of \( q \),
\( \textit{may-hold}([\text{succ, assumed_alias}], \langle \ast q, \ast \ast u \rangle) \).

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

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

\[
\begin{aligned}
\text{node:} & \quad p & q \\
\text{succ:} & \quad p = q \\
\end{aligned}
\]

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

\[
\text{for each } ((y, z), AA) \text{ such that } \textit{may-hold}([\text{node, AA}], \langle y, z \rangle) \text{ and } \\
is\text{-prefix}_\text{with}\text{-deref}(q, y) \text{ and } !\text{is\text{-prefix}}(p, z)
\]

efficiently.

5.3 May Alias

Computing May Alias given \( \textit{may-hold} \) is extremely simple:

\[
\textit{may-alias}(\text{node}) = \{ P.A | (\exists AA) \textit{may-hold}(\text{node, AA, PA}) = \text{true} \}
\]

\(^{53}\)when \( u \) or \( p \) is a prefix of \( q \), we do not want to create \( \langle \ast u, \ast q \rangle \) for the same reason we do not want to create \( \langle \ast p, \ast(p\rightarrow\text{next}) \rangle \) for “\( p = p\rightarrow\text{next} \)”.
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\([(\text{succ, assumed_alias}), (a, b)]\) is false
  set may-hold\([(\text{succ, assumed_alias}), (a, b)]\) to true
  add \((\text{succ, assumed_alias}, (a, b))\) to the worklist
}
}
for each \((\text{AA}, (y, z))\) such that may-hold\([(\text{node, AA}), (y, z)]\) 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, \(v = p->\text{next}(\equiv z)\). In this case, we do not need to do anything about the relationship between \(v->\text{next}\) and \(p->\text{next}\) since \(v\) and \(p\) are both 'shifted' by \->\text{next}. */

/* Rules out effects of cases equivalent to, for example, \(v = v->\text{next}(\equiv z)\) */

if \(v\) contains non_visible or \!is_prefix\(v, z\)
{ let \(v' = v\)
  apply_trans\(q, y, v'\)
  safely_make_alias\(\text{succ, v', assumed_alias, z, AA}\)
}

Figure 5.14: Action for may-hold\([(\text{node, assumed_alias}), (p, v)]\) for successor, \(\text{succ} = \text{"p = q\" of node}\)
This can clearly be computed in time linear in the size of the \textit{may-}\textit{hold} solution. In fact, given a clever data structure for \textit{may-}\textit{hold}, no additional computation (nor space) is needed to get the solution for \textit{may-}\textit{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:

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

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.

\[
\text{limit}_k(\text{alias solution}) = \begin{cases} 
  (\text{node}, \langle a', b' \rangle) & \text{if } (\text{node}, \langle a, b \rangle) \in \text{alias solution}, \\
  \text{a' is the k-limited representation} & \text{for } a, \text{ and } b' \text{ is the k-limited representation for } b 
\end{cases}
\]

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

\[
\text{precision}_k(A, P) = \frac{|\text{limit}_k(\{A's solution for } P\})|}{|\text{limit}_k(\{\text{precise solution for } P\})|}
\]

For programs where all object names in the precise solution have k (the k-limiting constant) or fewer dereferences, then \(\text{precision}_k(A, P) = \text{precision}(A, P)\).

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 \(\Theta(n^2)\) 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 \(|S| = \Theta(n^2)\) aliases to our solution and none to the precise solution. Such a padded program (of size \(m = \Omega(n)\), including additional statements) would have \(\Theta(n^3) / \ast\) from Figure 6.1 \(\ast / + (m - \Theta(n)) \ast \Theta(n^2) / \ast \Theta(n^2)\) for each added statement \(\ast / = \Theta(m \ast n^2)\) aliases reported with only \(\Theta(n)\) aliases in the precise solution, thus \(\text{precision}(\text{landi, Figure 6.1}) = \Theta(m \ast n)\). 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.
<table>
<thead>
<tr>
<th>node</th>
<th>Precise May Alias at node</th>
<th>may-alias(node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int **a, *b, *c, *d;</td>
<td>(omitting reflexive)</td>
<td></td>
</tr>
<tr>
<td>int *v[i]; (1 ≤ i ≤ n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/* - Part I - */</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a = malloc();</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b = malloc();</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c = malloc();</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d = malloc();</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>if (-)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a = &amp;b;</td>
<td>{(*a, b)}</td>
<td>{(*a, b), (**a, *b)}</td>
</tr>
<tr>
<td>else c = d;</td>
<td>{(*c, *d)}</td>
<td>{(*c, *d)}</td>
</tr>
<tr>
<td>*a = c;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{(*a, b), (**a, *b),</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(*c, *d), (**a, *c)},</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(**a, *d), (*c, *b)}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a = NULL;</td>
<td>{(*c, *d), (*c, *b)},</td>
<td>{(*c, *d)}</td>
</tr>
<tr>
<td>c = NULL;</td>
<td>0</td>
<td>{(*b, *d)}</td>
</tr>
<tr>
<td>/* - Part II - */</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W_c: while (-) {</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>for all k, 1 ≤ k ≤ n:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_a^k: if (-)</td>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>C_b^k:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v_k = b;</td>
<td>{(*v_k, *b)}</td>
<td>S</td>
</tr>
<tr>
<td>C_c^k: b = NULL;</td>
<td>0</td>
<td>{(*v_i, *d)</td>
</tr>
<tr>
<td>/* end for all */</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D_a: if (-)</td>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>D_b:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b = d;</td>
<td>{(*b, *d)}</td>
<td>S</td>
</tr>
<tr>
<td>D_c: d = NULL;</td>
<td>0</td>
<td>{(*v_i, *b)</td>
</tr>
<tr>
<td>W_x:</td>
<td>0</td>
<td>S</td>
</tr>
</tbody>
</table>

Number of program point aliases in the precise solution of May Alias: n + 11
Number of aliases in \textit{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), $\text{precision}_k(landi, P)$ is $O(m \times n)$. I.e., our algorithm reports $O(m \times n^2)$ 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 $O(m \times n_1^2)$ point specific aliases.
   In the worst possible case everything is reported aliased to everything else $(n_1 \times (n_1 - 1)$ 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 $\text{alias}\_\text{consequences}(\langle *p, *q \rangle)$ which by our assumption must be aliases in the precise solution.
   - For statements $p = &v$ we create all the aliases in $\text{alias}\_\text{consequences}(\langle *p, v \rangle)$ which must be in the precise solution by our assumption
   - When pointer formal $f$ is passed actual $a$ we create all the aliases in $\text{alias}\_\text{consequences}(\langle *f, *a \rangle)$ 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\(^5\), in which case we could easily prove that \( n_1 = n \).

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

\[ \square \]

**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 \( \Omega(m \cdot n) \). I.e., \( A \) reports \( \Omega(m \cdot n^2) \) 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 anywhere in the loop. However, when the alias \( \langle *b, *d \rangle \) holds on entry to the loop the first time then

\[ S = \{ \langle *b, *d \rangle \} \cup \{ \langle *v_i, *b \rangle | 1 \leq i \leq n \} \cup \{ \langle *v_i, *d \rangle | 1 \leq i \leq n \} \cup \begin{cases} \{ \langle *v_i, *v_j \rangle | 1 \leq i, j \leq n \} \\ \text{and } i \neq j \end{cases} \]

is the precise alias solution at every program point in Part II except at “\( b = \text{NULL} \)” where \( S \) holds except \( b \) is not aliased to anything and at “\( d = \text{NULL} \)” where \( S \) holds except \( d \) is not aliased to anything. The proof of this follows:

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

\(^5\)but such a version of the algorithm would be much less efficient.
- $D_a$: $P_{D_a}^{(b,d)\ast} = P_{C_a}^{(b,d)\ast} \circ D_a$
- $D_b$: $P_{D_b}^{(b,d)\ast} = P_{D_a}^{(b,d)\ast} \circ D_b$
- $D_c$: There is no such $P_{D_c}^{(b,d)\ast}$.
- $W_x$: $P_{W_x}^{(b,d)\ast} = P_{D_a}^{(b,d)\ast} \circ W_x$

$\langle \ast v_i, \ast d \rangle$ holds on path $P_{node}^{(\ast v_i, \ast d)}$ to:

- $C_b^i$: $P_{C_b^i}^{(\ast v_i, \ast d)} = P_{C_a^i}^{(b,d)\ast} \circ C_b^i$
- $C_c^i$: $P_{C_c^i}^{(\ast v_i, \ast d)} = P_{C_b^i}^{(\ast v_i, \ast d)} \circ C_c^i$
- $D_a$: $P_{D_a}^{(\ast v_i, \ast d)} = P_{C_a^i}^{(\ast v_i, \ast d)} \circ C_a^{l+1}C_a^{l+2}...C_a^n\circ D_a$
- $D_b$: $P_{D_b}^{(\ast v_i, \ast d)} = P_{D_a}^{(\ast v_i, \ast d)} \circ D_b$
- $D_c$: There is no such path $P_{D_c}^{(\ast v_i, \ast d)}$.
- $W_x$: $P_{W_x}^{(\ast v_i, \ast d)} = P_{D_a}^{(\ast v_i, \ast d)} \circ W_x$
- $W_c$: $P_{W_c}^{(\ast v_i, \ast d)} = P_{W_x}^{(\ast v_i, \ast d)} \circ W_c$
- $C_a^l (1 \leq l \leq n)$: $P_{C_a^l}^{(\ast v_i, \ast d)} = P_{W_c}^{(\ast v_i, \ast d)} \circ C_a^1C_a^2...C_a^l$
- $C_b^l (1 \leq l \leq n$ and $i \neq l)$: $P_{C_b^l}^{(\ast v_i, \ast d)} = P_{C_a^l}^{(\ast v_i, \ast d)} \circ C_b^l$
- $C_c^l (1 \leq l \leq n$ and $i \neq l)$: $P_{C_c^l}^{(\ast v_i, \ast d)} = P_{C_b^l}^{(\ast v_i, \ast d)} \circ C_c^l$

$\langle \ast v_i, \ast b \rangle$ holds on path $P_{node}^{(\ast v_i, \ast b)}$ to:

- $D_b$: $P_{D_b}^{(\ast v_i, \ast b)} = P_{D_a}^{(\ast v_i, \ast b)} \circ D_b$
- $D_c$: $P_{D_c}^{(\ast v_i, \ast b)} = P_{D_b}^{(\ast v_i, \ast b)} \circ D_c$
- $W_x$: $P_{W_x}^{(\ast v_i, \ast b)} = P_{D_c}^{(\ast v_i, \ast b)} \circ W_x$
- $W_c$: $P_{W_c}^{(\ast v_i, \ast b)} = P_{W_x}^{(\ast v_i, \ast b)} \circ W_c$
- $C_a^l (1 \leq l \leq n)$: $P_{C_a^l}^{(\ast v_i, \ast b)} = P_{W_c}^{(\ast v_i, \ast b)} \circ C_a^1C_a^2...C_a^l$
- $C_b^l (1 \leq l \leq n)$: $P_{C_b^l}^{(\ast v_i, \ast b)} = P_{C_a^l}^{(\ast v_i, \ast b)} \circ C_b^l$
- $C_c^l (1 \leq l \leq n)$: There is no such path $P_{C_c^l}^{(\ast v_i, \ast b)}$.
- $D_a$: $P_{D_a}^{(\ast v_i, \ast b)} = P_{C_a^l}^{(\ast v_i, \ast b)} \circ D_a$
\( \langle *v_i, *v_j \rangle \ (i \neq j) \) holds on path \( P_{node}^{\langle *v_i, *v_j \rangle} \) to:

- \( C_b^j: P_{C_b}^{\langle *v_i, *v_j \rangle} = P_{C_b}^{\langle *v_i, *b \rangle} \circ C_b^j \)
- \( C_b^j: P_{C_b}^{\langle *v_i, *v_j \rangle} = P_{C_b}^{\langle *v_b, *v_j \rangle} \circ C_b^j \)
- \( D_a: P_{D_a}^{\langle *v_i, *v_j \rangle} = P_{C_b}^{\langle *v_i, *v_j \rangle} \circ C_a^{j+1} C_a^{j+2} \ldots C_a^n D_a \)
- \( D_b: P_{D_b}^{\langle *v_i, *v_j \rangle} = P_{D_a}^{\langle *v_i, *v_j \rangle} \circ D_b \)
- \( D_c: P_{D_c}^{\langle *v_i, *v_j \rangle} = P_{D_b}^{\langle *v_i, *v_j \rangle} \circ D_c \)
- \( W_x: P_{W_x}^{\langle *v_i, *v_j \rangle} = P_{D_a}^{\langle *v_i, *v_j \rangle} \circ W_x \)
- \( W_c: P_{W_c}^{\langle *v_i, *v_j \rangle} = P_{W_x}^{\langle *v_i, *v_j \rangle} \circ W_c \)
- \( C_a^l \ (1 \leq l \leq n): P_{C_a^l}^{\langle *v_i, *v_j \rangle} = P_{W_a}^{\langle *v_i, *v_j \rangle} \circ C_a^{l} C_a^{2} \ldots C_a^{l} \)
- \( C_b^l \ (1 \leq l \leq n \text{ and } j \neq l): P_{C_b^l}^{\langle *v_i, *v_j \rangle} = P_{C_a^l}^{\langle *v_i, *v_j \rangle} \circ C_b^{l} \)
- \( C_c^l \ (1 \leq l \leq n \text{ and } j \neq l): P_{C_c^l}^{\langle *v_i, *v_j \rangle} = P_{C_a^l}^{\langle *v_i, *v_j \rangle} \circ C_c^{l} \)

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 \( \Theta(m \times n^2) \) if the alias \( \langle *b, *d \rangle \) holds before the code is executed. Since any approximate algorithm \( A \), by definition must be able to produce an erroneous alias, \( \langle *b', *d' \rangle \), there must be a program \( P \) that produces \( \langle *b', *d' \rangle \) 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 \( x = \text{NULL} \) (for every pointer variable \( x \) in \( P \) except \( b \) and \( d \)), this kills all the real aliases, but cannot destroy the alias \( \langle *b, *d \rangle \) (if the algorithm is safe), and follow this by Part II. The approximate algorithm must report \( \Omega(m \times n^2) \) aliases even though there are only \( O(n) \) aliases\(^{57}\).

\( \square \)

\(^{57}\) 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\textsuperscript{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 \textit{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 \textit{ptt}. \textit{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 \textit{non_visible}. In Chapter 5, a non-visible object name at a call node would get mapped to \textit{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 \textit{non_visible}. For example, *\textit{x}, where \textit{z} is not visible in the called procedure, would become *\textit{non_visible} at the entry node of the called procedure and \textit{z->next} would map to \textit{non_visible->next}. The reason for this is simply so that every alias between an object name, \textit{w}, and a non-visible object is represented by only one alias\textsuperscript{59}.

\textsuperscript{58} Casting of \textit{malloc()} is not a problem for our implementation.

\textsuperscript{59} As described in Chapter 5 an alias between \textit{w} and the non-visible object *\textit{z} could have been represented by both \textit{(w, non_visible)} with \textit{non_visible} being *\textit{z} and \textit{(w, *non_visible)} with \textit{non_visible} being \textit{z}. 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; therefore, 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/lnint/lib-ic 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. 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. For an assignment “p = q” and an alias \((a, b)\) if is_prefix_with_deref\((q, a)\) is false and is_prefix\((p, a)\) is false (with similar restrictions on \(b\)), \((a, b)\) 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, \(S\), 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 \(S\), 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 \(\text{may-hold}([s, AA], [a, b])\), \(s \in S\), then we check if \(\text{may-hold}([h, AA], [a, b])\) is true.

- If we ever find out that \(\text{may-hold}([h, AA], [a, b])\) then instead of passing the information about \((a, b)\) 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 \(i\), the LHS of a pointer assignment in \(S\), is_prefix\((i, a)\) 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:

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

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 \((\text{ALIAS} = \text{AFFECT}^* \circ (\text{AFFECT}^*)^T)\). 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.
<table>
<thead>
<tr>
<th>Program</th>
<th>Lines</th>
<th>Weihl</th>
<th>program-aliases</th>
<th>Weihl / program-aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td>ul</td>
<td>523</td>
<td>4.851</td>
<td>340</td>
<td>14.2</td>
</tr>
<tr>
<td>pokerd</td>
<td>1,354</td>
<td>62.225</td>
<td>300</td>
<td>207.4</td>
</tr>
<tr>
<td>compress</td>
<td>1,488</td>
<td>6.316</td>
<td>305</td>
<td>20.7</td>
</tr>
<tr>
<td>loader</td>
<td>1,522</td>
<td>39.059</td>
<td>472</td>
<td>82.7</td>
</tr>
<tr>
<td>learn</td>
<td>1,642</td>
<td>61.845</td>
<td>813</td>
<td>76.0</td>
</tr>
<tr>
<td>ed</td>
<td>1,772</td>
<td>1.796</td>
<td>1,664</td>
<td>1.0</td>
</tr>
<tr>
<td>diff</td>
<td>1,793</td>
<td>44.366</td>
<td>1,323</td>
<td>33.5</td>
</tr>
<tr>
<td>tbl</td>
<td>2,545</td>
<td>4,401</td>
<td>1,169</td>
<td>3.7</td>
</tr>
<tr>
<td>make</td>
<td>2,952</td>
<td>207.906</td>
<td>9,274</td>
<td>22.4</td>
</tr>
<tr>
<td>lex</td>
<td>3,315</td>
<td>9,490</td>
<td>925</td>
<td>10.2</td>
</tr>
</tbody>
</table>

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.\(^{65}\)

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

\(^{65}\) 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
<table>
<thead>
<tr>
<th>Program</th>
<th>Lines</th>
<th>Weihl</th>
<th>\textit{program-aliases}</th>
<th>\textit{program-aliases} / Weihl</th>
</tr>
</thead>
<tbody>
<tr>
<td>ul</td>
<td>523</td>
<td>3s</td>
<td>58s</td>
<td>19.3</td>
</tr>
<tr>
<td>pokerd</td>
<td>1,354</td>
<td>1m 24s</td>
<td>10s</td>
<td>0.1</td>
</tr>
<tr>
<td>compress</td>
<td>1,488</td>
<td>4s</td>
<td>6s</td>
<td>1.5</td>
</tr>
<tr>
<td>loader</td>
<td>1,522</td>
<td>36s</td>
<td>1m 1s</td>
<td>1.6</td>
</tr>
<tr>
<td>learn</td>
<td>1,642</td>
<td>46s</td>
<td>1m 34s</td>
<td>2.0</td>
</tr>
<tr>
<td>ed</td>
<td>1,772</td>
<td>6s</td>
<td>2m 33s</td>
<td>25.5</td>
</tr>
<tr>
<td>diff</td>
<td>1,793</td>
<td>58s</td>
<td>1m 49s</td>
<td>1.8</td>
</tr>
<tr>
<td>tbl</td>
<td>2,545</td>
<td>10s</td>
<td>4m 59s</td>
<td>29.9</td>
</tr>
<tr>
<td>make</td>
<td>2,952</td>
<td>17m 21s</td>
<td>3h 3m 55s</td>
<td>10.6</td>
</tr>
<tr>
<td>lex</td>
<td>3,315</td>
<td>18s</td>
<td>2m 5s</td>
<td>6.9</td>
</tr>
</tbody>
</table>

Table 6.2: Time: Comparison to Weihl

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

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 ≤ 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.\textsuperscript{67} Max Aliases

\textsuperscript{67}For make with k = 1, it is 674.8. The average is 74.7 if make is included.
<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allroots</td>
<td>Find all the roots of a polynomial</td>
</tr>
<tr>
<td>assembler</td>
<td>SIC assembler [Bec85]</td>
</tr>
<tr>
<td>compress</td>
<td>UNIX: compress data</td>
</tr>
<tr>
<td>diff</td>
<td>UNIX: show file differences</td>
</tr>
<tr>
<td>diffh</td>
<td>fast diff; UNIX diff -h</td>
</tr>
<tr>
<td>ed</td>
<td>UNIX line editor</td>
</tr>
<tr>
<td>football</td>
<td>a football statistics program</td>
</tr>
<tr>
<td>fixoutput</td>
<td>a simple translator</td>
</tr>
<tr>
<td>learn</td>
<td>introduction to UNIX</td>
</tr>
<tr>
<td>lex</td>
<td>generator of a lexical analyzer</td>
</tr>
<tr>
<td>lex315</td>
<td>a scanner for a subset of C</td>
</tr>
<tr>
<td>loader</td>
<td>SIC loader [Bec85]</td>
</tr>
<tr>
<td>make</td>
<td>UNIX: helps maintain programs</td>
</tr>
<tr>
<td>poker</td>
<td>a game of cards</td>
</tr>
<tr>
<td>pokerd</td>
<td>a variant of poker</td>
</tr>
<tr>
<td>simulator</td>
<td>SIC machine simulator/debugger [Bec85]</td>
</tr>
<tr>
<td>tbl</td>
<td>UNIX: format tables for nroff and troff</td>
</tr>
<tr>
<td>tp</td>
<td>tape formatting</td>
</tr>
<tr>
<td>ul</td>
<td>UNIX: underline</td>
</tr>
</tbody>
</table>

Figure 6.2: Description of programs in our empirical study
<table>
<thead>
<tr>
<th>Program</th>
<th>ICFG Nodes</th>
<th>May Aliases</th>
<th>Aliases/Node</th>
<th>Max Aliases at one Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>allroots</td>
<td>359</td>
<td>240</td>
<td>0.7</td>
<td>3</td>
</tr>
<tr>
<td>fixoutput</td>
<td>584</td>
<td>1,856</td>
<td>3.2</td>
<td>12</td>
</tr>
<tr>
<td>diff</td>
<td>587</td>
<td>7,184</td>
<td>12.2</td>
<td>45</td>
</tr>
<tr>
<td>poker</td>
<td>865</td>
<td>1,962</td>
<td>2.3</td>
<td>108</td>
</tr>
<tr>
<td>ul</td>
<td>1,256</td>
<td>79,555</td>
<td>63.3</td>
<td>101</td>
</tr>
<tr>
<td>lex315</td>
<td>1,308</td>
<td>5,712</td>
<td>4.4</td>
<td>12</td>
</tr>
<tr>
<td>loader</td>
<td>1,355</td>
<td>55,588</td>
<td>41.0</td>
<td>133</td>
</tr>
<tr>
<td>compress</td>
<td>1,513</td>
<td>5,844</td>
<td>3.9</td>
<td>21</td>
</tr>
<tr>
<td>tp</td>
<td>1,531</td>
<td>60,700</td>
<td>39.7</td>
<td>190</td>
</tr>
<tr>
<td>pokerd</td>
<td>1,726</td>
<td>16,997</td>
<td>9.9</td>
<td>241</td>
</tr>
<tr>
<td>learn</td>
<td>2,573</td>
<td>103,464</td>
<td>40.2</td>
<td>110</td>
</tr>
<tr>
<td>ed</td>
<td>3,003</td>
<td>150,369</td>
<td>50.1</td>
<td>98</td>
</tr>
<tr>
<td>assembler</td>
<td>3,137</td>
<td>748,849</td>
<td>238.7</td>
<td>442</td>
</tr>
<tr>
<td>make (k = 1)</td>
<td>3,171</td>
<td>2,155,188</td>
<td>674.8</td>
<td>2,062</td>
</tr>
<tr>
<td>diff</td>
<td>3,455</td>
<td>83,964</td>
<td>24.3</td>
<td>88</td>
</tr>
<tr>
<td>simulator</td>
<td>4,857</td>
<td>454,802</td>
<td>93.6</td>
<td>292</td>
</tr>
<tr>
<td>football</td>
<td>5,511</td>
<td>234,752</td>
<td>42.6</td>
<td>92</td>
</tr>
<tr>
<td>tbl</td>
<td>5,601</td>
<td>337,833</td>
<td>60.3</td>
<td>120</td>
</tr>
<tr>
<td>lex</td>
<td>6,121</td>
<td>85,017</td>
<td>13.9</td>
<td>53</td>
</tr>
</tbody>
</table>

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

Highlighted numbers are valid for the precise solution.

Table 6.5: Size of our May Alias Solution
If both \(<p, q>\) and \(<*x, y>\) occur on the same path, then \(<*q, y>\) holds on that path extended by \(t\); therefore, we conclude this, even though it may not be true.

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

Normally, \(<*(u \rightarrow n), *(v \rightarrow n \rightarrow n)\>) should hold on a path to \(t\) because assigning \(v \rightarrow n \rightarrow n\) to \(p \cdot n\) is also assigning \(v \rightarrow n \rightarrow n\) to \(u \rightarrow n\). However, \(<*(u \rightarrow n), *(v \rightarrow n \rightarrow n)\>) need not hold. If, for example, \(<p, *u>\) and \(<p, *v>\) hold on the same path then \(<*(u \rightarrow n), *(v \rightarrow n \rightarrow n)\>) 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.

\[^{69}\text{on the path on which }<p, *u>\text{ holds}\]
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 ⊆ MAYBE ⊆ NO. With this implementation, $may\cdot hold([(node, AA), PA]) =$

- NO iff $may\cdot hold([(node, AA), PA])$ is *false*
- MAYBE, if $may\cdot hold([(node, AA), PA])$ is *true* because of case 2, 3, or 4 above
- MAYBE, if $may\cdot hold([(node, AA), PA])$ is implied by some
  $may\cdot hold([(node', AA'), PA']) = MAYBE$, but not by any
  $may\cdot hold([(node'', AA''), PA'']) = YES$
- YES otherwise

Now define $YES_k(P)$ for $may\cdot hold$ computed from $P$ with k-limit constant $k$:

\[
%YES_k(P) = 100 \times \left\{ \left\{ \begin{array}{c}
(node, PA) \quad (\exists AA) may\cdot hold([(node, AA), PA]) = YES \\
(node, PA) \quad (\exists AA) may\cdot hold([(node, AA), PA]) = YES \\
\quad or may\cdot hold([(node, AA), PA]) = MAYBE
\end{array} \right\} \right\}
\]

Given that there are only the aforementioned four types of approximations and that the fourth type does not occur, we claim that

\[
\left\{ (node, PA) \quad (\exists AA) may\cdot hold([(node, AA), PA]) = YES \right\} \subseteq limit_k(\text{precise solution})
\]

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 $lim_k(\text{precise solution})$ since by Lemma E.1.1 and our assumption about type 4 approximations, no other type of approximation can occur. Thus $%YES_k(P) \leq 100 \times (1/precision_k(landi, P))$.\(^{70}\) 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).\textsuperscript{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 \(\ast\ast A = \ast\ast B\). However, if \(k\) was 1, we would see this statement as \(\ast A = \ast B\) which we is considered a pointer assignment. This would certainly introduce imprecision that is not accounted for by \(\%Y E S_k(P)\).

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\textsuperscript{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.

\textsuperscript{71}This is a limit of our implementation, it is not fundamental to the algorithm itself.

\textsuperscript{72}We could not get a bound for \textit{make} because we were not able to solve for aliases with a sufficiently large \(k\).
<table>
<thead>
<tr>
<th>Program</th>
<th>ICFG Nodes</th>
<th>May Aliases</th>
<th>%YES_k(Program)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>k=1</td>
<td>k=2</td>
<td>k=3</td>
</tr>
<tr>
<td>allroots</td>
<td>359</td>
<td>240</td>
<td>240</td>
</tr>
<tr>
<td>fixoutput</td>
<td>584</td>
<td>1,840</td>
<td>1,856</td>
</tr>
<tr>
<td>diffh</td>
<td>587</td>
<td>5,986</td>
<td>7,184</td>
</tr>
<tr>
<td>poker</td>
<td>865</td>
<td>770</td>
<td>1,962</td>
</tr>
<tr>
<td>ul</td>
<td>1,256</td>
<td>79,003</td>
<td>79,555</td>
</tr>
<tr>
<td>lex315</td>
<td>1,308</td>
<td>5,696</td>
<td>5,712</td>
</tr>
<tr>
<td>loader</td>
<td>1,355</td>
<td>25,418</td>
<td>55,588</td>
</tr>
<tr>
<td>compress</td>
<td>1,513</td>
<td>6,404</td>
<td>5,844</td>
</tr>
<tr>
<td>tp</td>
<td>1,531</td>
<td>51,256</td>
<td>60,700</td>
</tr>
<tr>
<td>pokerd</td>
<td>1,726</td>
<td>6,517</td>
<td>16,997</td>
</tr>
<tr>
<td>learn</td>
<td>2,573</td>
<td>73,699</td>
<td>103,464</td>
</tr>
<tr>
<td>ed</td>
<td>3,003</td>
<td>150,392</td>
<td>150,369</td>
</tr>
<tr>
<td>assembler</td>
<td>3,137</td>
<td>283,770</td>
<td>748,849</td>
</tr>
<tr>
<td>make</td>
<td>3,171</td>
<td>2,155,188</td>
<td>-</td>
</tr>
<tr>
<td>diff</td>
<td>3,455</td>
<td>81,543</td>
<td>83,964</td>
</tr>
<tr>
<td>simulator</td>
<td>4,857</td>
<td>225,550</td>
<td>454,802</td>
</tr>
<tr>
<td>football</td>
<td>5,511</td>
<td>180,414</td>
<td>234,752</td>
</tr>
<tr>
<td>for k=4:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tbl</td>
<td>5,601</td>
<td>371,818</td>
<td>337,833</td>
</tr>
<tr>
<td>for k=4:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lex</td>
<td>6,121</td>
<td>120,205</td>
<td>85,017</td>
</tr>
<tr>
<td>for k=4:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

†%YES_k(Program) ≤ 100 * (1/precision_k(landi, Program))
††%YES_k(Program) ≤ 100 * (1/precision(landi, Program))

Table 6.6: Precision of our May Alias Solution
<table>
<thead>
<tr>
<th>Program (Nodes)</th>
<th>May Aliases</th>
<th>( k=1 )</th>
<th>( k=2 )</th>
<th>( k=3 )</th>
<th>( may)-hold</th>
<th>( may)-hold/ May Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td>allroots (359)</td>
<td>240</td>
<td>240</td>
<td>240</td>
<td>955</td>
<td>955</td>
<td>955</td>
</tr>
<tr>
<td>fixoutput (584)</td>
<td>1.840</td>
<td>1.856</td>
<td>1.856</td>
<td>1.840</td>
<td>1.856</td>
<td>1.856</td>
</tr>
<tr>
<td>diffh (587)</td>
<td>5.986</td>
<td>7.184</td>
<td>7.184</td>
<td>7.580</td>
<td>8.842</td>
<td>8.842</td>
</tr>
<tr>
<td>poker (865)</td>
<td>770</td>
<td>1.962</td>
<td>4.362</td>
<td>904</td>
<td>2.096</td>
<td>4.496</td>
</tr>
<tr>
<td>ul (1.256)</td>
<td>79.003</td>
<td>79.555</td>
<td>79.555</td>
<td>89.776</td>
<td>92.532</td>
<td>92.532</td>
</tr>
<tr>
<td>lex315 (1.308)</td>
<td>5.696</td>
<td>5.712</td>
<td>5.712</td>
<td>5.836</td>
<td>5.852</td>
<td>5.852</td>
</tr>
<tr>
<td>loader (1.355)</td>
<td>25.418</td>
<td>55.588</td>
<td>102.121</td>
<td>61.221</td>
<td>187.337</td>
<td>508.924</td>
</tr>
<tr>
<td>compress (1.513)</td>
<td>6.404</td>
<td>5.844</td>
<td>5.844</td>
<td>7.799</td>
<td>7.346</td>
<td>7.346</td>
</tr>
<tr>
<td>tp (1.531)</td>
<td>51.256</td>
<td>60.700</td>
<td>60.700</td>
<td>74.367</td>
<td>92.550</td>
<td>92.550</td>
</tr>
<tr>
<td>poked (1.726)</td>
<td>6.517</td>
<td>16.997</td>
<td>36.419</td>
<td>14.448</td>
<td>26.547</td>
<td>46.054</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>for ( k=4 ):</td>
<td>58.257</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>for ( k=4 ):</td>
<td>67.997</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>for ( k=4 ):</td>
<td>1.2</td>
</tr>
<tr>
<td>learn (2.573)</td>
<td>73.699</td>
<td>103.464</td>
<td>103.464</td>
<td>103.776</td>
<td>142.761</td>
<td>142.761</td>
</tr>
<tr>
<td>ed (3.003)</td>
<td>150.392</td>
<td>150.369</td>
<td>150.369</td>
<td>201.331</td>
<td>201.284</td>
<td>201.284</td>
</tr>
<tr>
<td>assembler (3.137)</td>
<td>283.770</td>
<td>748.849</td>
<td>1,436.120</td>
<td>900.372</td>
<td>2,385.797</td>
<td>6,625.431</td>
</tr>
<tr>
<td>make (3.171)</td>
<td>2,155.188</td>
<td>-</td>
<td>-</td>
<td>12,624.949</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>diff (3.455)</td>
<td>81.543</td>
<td>83.964</td>
<td>83.964</td>
<td>113.836</td>
<td>109.345</td>
<td>109.345</td>
</tr>
<tr>
<td>simulator (4.857)</td>
<td>225.550</td>
<td>454.802</td>
<td>767.813</td>
<td>273.519</td>
<td>543.305</td>
<td>922.463</td>
</tr>
<tr>
<td>football (5.511)</td>
<td>180.414</td>
<td>234.752</td>
<td>234.762</td>
<td>324.042</td>
<td>414.222</td>
<td>414.232</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for ( k=4 ):</td>
<td>234.762</td>
<td>414.232</td>
<td>414.232</td>
<td>for ( k=4 ): 1.8</td>
</tr>
<tr>
<td>tbl (5.601)</td>
<td>371.818</td>
<td>337.833</td>
<td>338.743</td>
<td>425.969</td>
<td>401.167</td>
<td>401.938</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for ( k=4 ):</td>
<td>338.743</td>
<td>401.938</td>
<td>401.938</td>
<td>for ( k=4 ): 1.1</td>
</tr>
<tr>
<td>lex (6.121)</td>
<td>120.205</td>
<td>85.017</td>
<td>85.017</td>
<td>145.643</td>
<td>109.717</td>
<td>109.717</td>
</tr>
</tbody>
</table>

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 $O$ (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 $may\cdot hold([\text{node}, \text{AA}], \mathcal{P}, \mathcal{A})$ can be generated. It is possible for work to be done once for each different way that it can be generated. If $\text{node}$ is not a return node, this is not a problem because the number of different ways that $may\cdot hold([\text{node}, \text{AA}], \mathcal{P}, \mathcal{A})$ can become true is $O \left( \left\{ m \mid \ll m, \text{node} \gg \text{is an edge in the ICFG} \right\} \right)$. 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 $\text{node}$ is a return node (for simplicity, ignoring visibility):

\[
may\cdot hold([\text{node}, \text{AA}], \mathcal{P}, \mathcal{A}) = \\
holds([\text{exit}, \emptyset], \mathcal{P}, \mathcal{A}) \lor \bigvee_{\text{AA'} \in \text{ASSUMED}} \left( holds([\text{exit, AA'}], \mathcal{P}, \mathcal{A}) \land \\
holds([\text{call, AA}], \text{back-bind-call}(\text{AA'})) \right)
\]
Thus the number of different ways that $may\cdot hold([\text{node, } AA], \mathcal{P}, A)$ can become true is $1 + |\{AA' | may\cdot hold([\text{exit, } AA'}, \mathcal{P}, A) is true \}|$ which can be at most Max-assumed which we define as

$$1 + \max_{\text{all possible aliases, } \mathcal{P}, A} \left( \left\{ \begin{array}{c} (\exists \text{ node}) \text{ node } \in \text{ICFG and} \\
may\cdot hold([\text{node, } AA], \mathcal{P}, A) \\
is \text{true} \end{array} \right\} \right)$$

Unfortunately, Table 6.8 indicates that Max-assumed is not constant. Table 6.8 is sorted by the size of $may\cdot hold$, so that it is easier to observe how Max-assumed increases as $may\cdot hold$ increases. It does appear to be a slowly growing function (34 when there are 2,385,797 true Conditional May Aliases\textsuperscript{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 $may\cdot hold$ (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.

```c
P() {
    temp = g;
    \langle *\text{temp, } *\text{h} \rangle \text{ holds on a path to } n
    n: while (temp != NULL) \text{ conditional on } \langle *g, *\text{h} \rangle \text{ or } \langle *(g->next), *\text{h} \rangle \text{ or } ...
        temp = temp->next;
        \text{ 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 $may\cdot hold$, but we do not have a theoretical justification for such a claim. Thus, we decided to

\textsuperscript{73} and 31 for make ($k = 1$) with 12,624,949 Conditional May Aliases
<table>
<thead>
<tr>
<th>Program</th>
<th>may-hold</th>
<th>Max-assumed</th>
<th>Avg-assumed</th>
</tr>
</thead>
<tbody>
<tr>
<td>allroots</td>
<td>955</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>fixoutput</td>
<td>1,856</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>poker</td>
<td>2,096</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>lex315</td>
<td>5,852</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>compress</td>
<td>7,346</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>diffh</td>
<td>8,842</td>
<td>2</td>
<td>1.0</td>
</tr>
<tr>
<td>pokered</td>
<td>26,547</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>ul</td>
<td>92,532</td>
<td>2</td>
<td>1.0</td>
</tr>
<tr>
<td>tp</td>
<td>92,550</td>
<td>2</td>
<td>1.0</td>
</tr>
<tr>
<td>diff</td>
<td>109,345</td>
<td>4</td>
<td>1.0</td>
</tr>
<tr>
<td>lex</td>
<td>109,717</td>
<td>3</td>
<td>1.0</td>
</tr>
<tr>
<td>learn</td>
<td>142,761</td>
<td>2</td>
<td>1.0</td>
</tr>
<tr>
<td>loader</td>
<td>187,337</td>
<td>11</td>
<td>1.6</td>
</tr>
<tr>
<td>ed</td>
<td>201,284</td>
<td>4</td>
<td>1.1</td>
</tr>
<tr>
<td>tbl</td>
<td>401,167</td>
<td>6</td>
<td>1.0</td>
</tr>
<tr>
<td>football</td>
<td>414,222</td>
<td>2</td>
<td>1.0</td>
</tr>
<tr>
<td>simulator</td>
<td>543,305</td>
<td>10</td>
<td>1.0</td>
</tr>
<tr>
<td>assembler</td>
<td>2,385,797</td>
<td>34</td>
<td>1.9</td>
</tr>
<tr>
<td>make ( (k = 1) )</td>
<td>12,624,949</td>
<td>31</td>
<td>3.6</td>
</tr>
</tbody>
</table>

\[
\text{Max-assumed} \equiv \max_{\text{all possible aliases, } \mathcal{P}_A} \left\{ \mathcal{A}_A \mid (\exists \text{ node } \in \text{ICFG and } \text{may-hold}(\text{[(node, } \mathcal{A}_A), \mathcal{P}_A]\text{ is } \text{true})} \right\}
\]

\[
\text{Avg-assumed} \equiv \frac{\text{sum over all possible aliases, } \mathcal{P}_A}{\text{count of all possible aliases, } \mathcal{P}_A} \left\{ \mathcal{A}_A \mid (\exists \text{ node } \in \text{ICFG and } \text{may-hold}(\text{[(node, } \mathcal{A}_A), \mathcal{P}_A]\text{ is } \text{true})} \right\}
\]

Table 6.8: Number of Assumptions per Alias \( (k = 2) \)
\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|c|c|c|c|c|c|}
\hline
Program & \multicolumn{3}{|c|}{\textit{may-hold}} & \multicolumn{3}{|c|}{Max-assumed} & \multicolumn{3}{|c|}{Avg-assumed} \\ 
 & \(k=1\) & \(k=2\) & \(k=3\) & \(k=1\) & \(k=2\) & \(k=3\) & \(k=1\) & \(k=2\) & \(k=3\) \\ \hline
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 \\ diff & 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 \\ & \text{for } k=4: & 67.997 & \text{for } k=4: & 1 & \text{for } k=4: & 1 \\ 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 \\ c\text{d} & 201.331 & 201.284 & 201.284 & 4 & 4 & 4 & 1.1 & 1.1 & 1.1 \\ tbl & 425.999 & 401.167 & 401.938 & 4 & 6 & 3 & 1.0 & 1.0 & 1.0 \\ & \text{for } k=4: & 401.938 & \text{for } k=4: & 3 & \text{for } k=4: & 1 \\ football & 324.042 & 414.222 & 414.232 & 2 & 2 & 12 & 1.0 & 1.0 & 1.0 \\ & \text{for } k=4: & 414.232 & \text{for } k=4: & 2 & \text{for } k=4: & 1 \\ simulator & 273.519 & 543.305 & 922.463 & 4 & 10 & 19 & 1.0 & 1.0 & 1.1 \\ assembler & 900.372 & 2385.797 & 6625.431 & 34 & 34 & 66 & 1.0 & 1.9 & 2.7 \\ make & 12624.949 & - & - & 31 & - & - & 3.6 & - & - \\ \hline
\end{tabular}
\caption{Table 6.9: Number of Assumptions per Alias}
\end{table}
<table>
<thead>
<tr>
<th>Program</th>
<th>may-hold</th>
<th>CPU Time</th>
<th>may-hold/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>allroots</td>
<td>955</td>
<td>(0.78s) 1s</td>
<td>1,224</td>
</tr>
<tr>
<td>fixoutput</td>
<td>1,856</td>
<td>(1.48s) 1s</td>
<td>1,254</td>
</tr>
<tr>
<td>poker</td>
<td>2,096</td>
<td>14s</td>
<td>150</td>
</tr>
<tr>
<td>lex315</td>
<td>5,852</td>
<td>(3.88s) 4s</td>
<td>1,508</td>
</tr>
<tr>
<td>compress</td>
<td>7,346</td>
<td>(6.20s) 6s</td>
<td>1,185</td>
</tr>
<tr>
<td>diffh</td>
<td>8,842</td>
<td>(6.22s) 6s</td>
<td>1,422</td>
</tr>
<tr>
<td>poked</td>
<td>26,547</td>
<td>20s</td>
<td>1,327</td>
</tr>
<tr>
<td>ul</td>
<td>92,532</td>
<td>60s</td>
<td>1,542</td>
</tr>
<tr>
<td>tp</td>
<td>92,550</td>
<td>60s</td>
<td>1,543</td>
</tr>
<tr>
<td>diff</td>
<td>109,345</td>
<td>101s</td>
<td>1,083</td>
</tr>
<tr>
<td>lex</td>
<td>109,717</td>
<td>95s</td>
<td>1,155</td>
</tr>
<tr>
<td>learn</td>
<td>142,761</td>
<td>123s</td>
<td>1,170</td>
</tr>
<tr>
<td>loader</td>
<td>187,337</td>
<td>195s</td>
<td>961</td>
</tr>
<tr>
<td>ed</td>
<td>201,284</td>
<td>132s</td>
<td>1,525</td>
</tr>
<tr>
<td>tbl</td>
<td>401,167</td>
<td>277s</td>
<td>1,448</td>
</tr>
<tr>
<td>football</td>
<td>414,222</td>
<td>241s</td>
<td>1,719</td>
</tr>
<tr>
<td>simulator</td>
<td>543,305</td>
<td>566s</td>
<td>977</td>
</tr>
<tr>
<td>assembler</td>
<td>2,385,797</td>
<td>3,299s</td>
<td>723</td>
</tr>
<tr>
<td>make (k = 1)</td>
<td>12,624,949</td>
<td>11,035s</td>
<td>1,144</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Program</th>
<th>( \text{may-hold} )</th>
<th>( \text{CPU Time} )</th>
<th>( \text{may-hold/sec} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( k=1 )</td>
<td>( k=2 )</td>
<td>( k=3 )</td>
</tr>
<tr>
<td>allroots</td>
<td>955</td>
<td>955</td>
<td>955</td>
</tr>
<tr>
<td>fixoutput</td>
<td>1,840</td>
<td>1,856</td>
<td>1,856</td>
</tr>
<tr>
<td>poker</td>
<td>904</td>
<td>2,096</td>
<td>4,496</td>
</tr>
<tr>
<td>lex315</td>
<td>5,836</td>
<td>5,852</td>
<td>5,852</td>
</tr>
<tr>
<td>compress</td>
<td>7,799</td>
<td>7,346</td>
<td>7,346</td>
</tr>
<tr>
<td>diffh</td>
<td>7,580</td>
<td>8,842</td>
<td>8,842</td>
</tr>
<tr>
<td>poked</td>
<td>14,448</td>
<td>26,547</td>
<td>46,054</td>
</tr>
</tbody>
</table>

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 |
| football | 324,042 | 414,222 | 414,232 | 185s | 241s | 240s | 1,752 | 1,719 | 1,726 |

for \( k=4 \): 401,938 for \( k=4 \): 284s for \( k=4 \): 1,415

| simulator | 273,519 | 543,305 | 922,463 | 278s | 566s | 984s | 984 | 960 | 937 |
| assembler | 900,372 | 2,385,797 | 6,629,431 | 1,159s | 3,299s | 7,103s | 777 | 723 | 933 |
| make | 12,624,949 | - | - | 11,053s | - | - | 1,144 | - | - |

Table 6.11: Timings for our Algorithm
<table>
<thead>
<tr>
<th>Program</th>
<th>may-hold</th>
<th>may-hold/sec</th>
<th>Program</th>
<th>may-hold</th>
<th>may-hold/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>poker ((k = 3))</td>
<td>4,496</td>
<td>118</td>
<td>allroots ((k = 3))</td>
<td>955</td>
<td>1,224</td>
</tr>
<tr>
<td>poker ((k = 2))</td>
<td>2,096</td>
<td>150</td>
<td>fixoutput ((k = 2))</td>
<td>1,856</td>
<td>1,254</td>
</tr>
<tr>
<td>poker ((k = 1))</td>
<td>904</td>
<td>171</td>
<td>fixoutput ((k = 3))</td>
<td>1,856</td>
<td>1,254</td>
</tr>
<tr>
<td>assembler ((k = 2))</td>
<td>2,385,797</td>
<td>723</td>
<td>compress ((k = 1))</td>
<td>7,799</td>
<td>1,272</td>
</tr>
<tr>
<td>assembler ((k = 1))</td>
<td>900,372</td>
<td>777</td>
<td>ed ((k = 1))</td>
<td>201,331</td>
<td>1,316</td>
</tr>
<tr>
<td>loader ((k = 3))</td>
<td>508,924</td>
<td>814</td>
<td>pokerd ((k = 2))</td>
<td>26,547</td>
<td>1,327</td>
</tr>
<tr>
<td>assembler ((k = 3))</td>
<td>6,625,431</td>
<td>933</td>
<td>diffh ((k = 1))</td>
<td>7,580</td>
<td>1,399</td>
</tr>
<tr>
<td>simulator ((k = 3))</td>
<td>922,463</td>
<td>937</td>
<td>tbl ((k = 4))</td>
<td>401,938</td>
<td>1,415</td>
</tr>
<tr>
<td>simulator ((k = 2))</td>
<td>543,305</td>
<td>960</td>
<td>diffh ((k = 2))</td>
<td>8,842</td>
<td>1,422</td>
</tr>
<tr>
<td>loader ((k = 2))</td>
<td>187,337</td>
<td>961</td>
<td>diffh ((k = 3))</td>
<td>8,842</td>
<td>1,422</td>
</tr>
<tr>
<td>simulator ((k = 1))</td>
<td>273,519</td>
<td>984</td>
<td>tbl ((k = 1))</td>
<td>425,969</td>
<td>1,425</td>
</tr>
<tr>
<td>loader ((k = 1))</td>
<td>61,221</td>
<td>1,004</td>
<td>pokerd ((k = 1))</td>
<td>14,448</td>
<td>1,445</td>
</tr>
<tr>
<td>diff ((k = 1))</td>
<td>113,836</td>
<td>1,044</td>
<td>tbl ((k = 2))</td>
<td>401,167</td>
<td>1,448</td>
</tr>
<tr>
<td>diff ((k = 3))</td>
<td>109,345</td>
<td>1,072</td>
<td>tbl ((k = 3))</td>
<td>401,938</td>
<td>1,472</td>
</tr>
<tr>
<td>diff ((k = 2))</td>
<td>109,345</td>
<td>1,083</td>
<td>lex15 ((k = 3))</td>
<td>5,852</td>
<td>1,477</td>
</tr>
<tr>
<td>pokerd ((k = 4))</td>
<td>67,997</td>
<td>1,097</td>
<td>lex15 ((k = 2))</td>
<td>5,852</td>
<td>1,508</td>
</tr>
<tr>
<td>learn ((k = 1))</td>
<td>103,776</td>
<td>1,104</td>
<td>lex15 ((k = 1))</td>
<td>5,836</td>
<td>1,512</td>
</tr>
<tr>
<td>learn ((k = 3))</td>
<td>142,761</td>
<td>1,133</td>
<td>tp ((k = 3))</td>
<td>92,550</td>
<td>1,517</td>
</tr>
<tr>
<td>lex ((k = 3))</td>
<td>109,717</td>
<td>1,143</td>
<td>tp ((k = 1))</td>
<td>74,367</td>
<td>1,518</td>
</tr>
<tr>
<td>make ((k = 1))</td>
<td>12,624,949</td>
<td>1,144</td>
<td>ul ((k = 1))</td>
<td>89,776</td>
<td>1,522</td>
</tr>
<tr>
<td>lex ((k = 2))</td>
<td>109,717</td>
<td>1,155</td>
<td>ed ((k = 2))</td>
<td>201,284</td>
<td>1,525</td>
</tr>
<tr>
<td>learn ((k = 2))</td>
<td>142,761</td>
<td>1,170</td>
<td>ul ((k = 2))</td>
<td>92,532</td>
<td>1,542</td>
</tr>
<tr>
<td>pokerd ((k = 3))</td>
<td>46,054</td>
<td>1,181</td>
<td>ul ((k = 3))</td>
<td>92,532</td>
<td>1,542</td>
</tr>
<tr>
<td>compress ((k = 2))</td>
<td>7,346</td>
<td>1,185</td>
<td>tp ((k = 2))</td>
<td>92,550</td>
<td>1,543</td>
</tr>
<tr>
<td>compress ((k = 3))</td>
<td>7,346</td>
<td>1,193</td>
<td>ed ((k = 3))</td>
<td>201,284</td>
<td>1,548</td>
</tr>
<tr>
<td>lex ((k = 1))</td>
<td>145,643</td>
<td>1,214</td>
<td>football ((k = 4))</td>
<td>414,232</td>
<td>1,712</td>
</tr>
<tr>
<td>fixoutput ((k = 1))</td>
<td>1,840</td>
<td>1,219</td>
<td>football ((k = 2))</td>
<td>414,222</td>
<td>1,719</td>
</tr>
<tr>
<td>allroots ((k = 1))</td>
<td>955</td>
<td>1,224</td>
<td>football ((k = 3))</td>
<td>414,232</td>
<td>1,726</td>
</tr>
<tr>
<td>allroots ((k = 2))</td>
<td>955</td>
<td>1,224</td>
<td>football ((k = 1))</td>
<td>324,042</td>
<td>1,752</td>
</tr>
</tbody>
</table>

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 inaprocedural 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_name1, object_name2, object_name3) : object_name1 and object_name3 must have the same type and is.prefix(object_name1, object_name2) must be true. The function applies to `object_name3` the sequence of dereferences and field accesses necessary to transform `object_name1` into `object_name2`. 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, <nv1, nv2>),
on_non Visible-name1-is, non_visible-name2-is, name1-AA, name2-AA) substitutes non_visible-name1-is for non_visible in `nv1`, substitutes non_visible-name2-is for non_visible in `nv2`, and uses the assumed aliases name1-AA and name2-AA to establish a safe assumed alias condition at procedure entry. (Appendix D.1)
\textbf{back.bind.true}(\textit{call, assumed_alias}) =

\begin{align*}
&\begin{cases}
  (\mathcal{A}, n_v) \\
  (\exists \text{alias}) \ \text{may-hold}([\text{call}, \mathcal{A}, \text{alias}]) \ \text{and} \\
  (\text{assumed_alias}, n_v) \in \text{bind\_call}(\text{alias})
\end{cases} & \text{if } \mathcal{A} \text{ has an object name with non_visible in it} \\
&\begin{cases}
  \mathcal{A} \\
  (\exists \text{alias}) \ \text{may-hold}([\text{call}, \mathcal{A}, \text{alias}]) \\
  \text{and } \text{assumed_alias} \in \text{bind\_call}(\text{alias})
\end{cases} & \text{otherwise}
\end{align*}

(Chapter 5.2.3)

\textbf{BBT} : see \textbf{back.bind.true}.

can.deref(\textit{type}) is \textbf{true} iff \textit{type} can be legally dereferenced without casting. (Chapter 2.2)

deref(\textit{object.name}) removes the & if \textit{object.name} starts with an & and otherwise returns *\textit{(object.name)}. This corresponds to rule 1 of Figure 2.2. (Chapter 2.3)

deref.type(\textit{type}) If \textit{type} can be dereferenced, \textbf{deref.type} returns the type of the objects to which \textit{type} may point. (Chapter 2.2)

\textbf{field.access}(\textit{object.name}.\textit{field.name}) is \textit{(object.name).field.name}, if \textit{object.name} does not start with an &, otherwise error. This corresponds to rule 2 of Figure 2.2. (Chapter 2.3)

\textbf{field.type}(\textit{type, field}) If \textit{type} is a structure with \textit{field}, \textbf{field.type} returns the type of \textit{field}. (Chapter 2.2)

\textbf{is_field_of}(\textit{field, type}) is \textbf{true} iff \textit{type} is a structure type and \textit{field} is a legal field of \textit{type}. (Chapter 2.2)

\textbf{is_k-limited}(\textit{object.name}) is \textbf{true} iff \textit{object.type(object.name)} can be dereferenced, \textit{object.name} contains at least \textit{k} dereferences, and \textit{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\(-\)hold\( ([n_i . \mathcal{A}\mathcal{A}], \langle a , b \rangle ) \) 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 \( \mathcal{A}\mathcal{A} \) holds and there is a path from entry of main to the entry\( (n_i ) \) on which \( \mathcal{A}\mathcal{A} \) 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\( (node , on_1 , \mathcal{A}\mathcal{A}_1 , on_2 , \mathcal{A}\mathcal{A}_2 ) \) 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 \( \mathcal{A}\mathcal{A}_1 \) and \( \mathcal{A}\mathcal{A}_2 \) 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{A}) =

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

(Chapter 5.2.3)

\textbf{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 \( O = \{ *p \mid p \text{ is a pointer in the program} \} \cup \{ v \mid v \text{ is a non-pointer variable in the program} \} \).

\( O \) is the set of all object names in the program which may have aliases.

Let \( \text{POSSIBLE-ALIASES} = (O \times O) \)

Given \( \text{CFG} = (N, E, \rho) \), construct \( \text{SPPAG} = (N', E', \rho') \) and \( \text{holds} : N' \to \{ \text{true}, \text{false} \} \).

\( N' = \{ [n, \langle a, b \rangle] \mid n \in N \text{ and } \langle a, b \rangle \in \text{POSSIBLE-ALIASES} \} \cup \{ p' \} \)

Specification of \( E' \) and \( \text{holds} \). For each \([n, \langle a, b \rangle]\) in \( N'\):

1. \( a = b \) and \( a \) is not a dereferenced pointer

   Add \( \ll \rho', [n, \langle a, b \rangle] \gg \) to \( E' \)

   \( \text{holds}([n, \langle a, b \rangle]) = \text{holds}'([n, \langle a, b \rangle]) = \text{true} \)
2. \( n = \rho \ [a \neq b \text{ or } a \text{ is a dereferenced pointer}] \)
   Add \( \llcorner \rho', [n, \langle a, b \rangle] \lrcorner \) to \( \mathcal{E}' \)
   \( \text{holds}([n, \langle a, b \rangle]) = \text{holds}^\wedge([n, \langle a, b \rangle]) = \text{false} \)

3. \( n \) is not an assignment to a pointer \( [(a \neq b \text{ or } a \text{ is a dereferenced pointer}) \) and \( n \neq \rho \]
   \( \forall \llcorner m, n \lrcorner \in \mathcal{E}, \text{add } \llcorner m, \langle a, b \rangle, [n, \langle a, b \rangle] \lrcorner \to \mathcal{E}' \)
   \( \text{holds}([n, \langle a, b \rangle]) = \lor_{\llcorner m, n \lrcorner \in \mathcal{E}} \left( \text{holds}([m, \langle a, b \rangle]) \right) \)
   \( \text{holds}^\wedge([n, \langle a, b \rangle]) = \land_{\llcorner m, n \lrcorner \in \mathcal{E}} \left( \text{holds}^\wedge([m, \langle a, b \rangle]) \right) \)

4. \( n \) is the pointer assignment "\( p = q \)" \( [(a \neq b \text{ or } a \text{ is a dereferenced pointer}) \) and \( n \neq \rho \]
   (a) if \( a = b = *p \)
      \( \forall \llcorner m, n \lrcorner \in \mathcal{E}, \text{add } \llcorner m, \langle *q, *q \rangle, [n, \langle *p, *p \rangle] \lrcorner \to \mathcal{E}' \)
      \( \text{holds}([n, \langle *p, *p \rangle]) = \lor_{\llcorner m, n \lrcorner \in \mathcal{E}} \left( \text{holds}([m, \langle *q, *q \rangle]) \right) \)
      \( \text{holds}^\wedge([n, \langle *p, *p \rangle]) = \land_{\llcorner m, n \lrcorner \in \mathcal{E}} \left( \text{holds}^\wedge([m, \langle *q, *q \rangle]) \right) \)
   (b) if \( a = *p \ [b \neq a] \)
      \( \forall \llcorner m, n \lrcorner \in \mathcal{E}, \text{add } \llcorner m, \langle *q, b \rangle, [n, \langle *p, b \rangle] \lrcorner \to \mathcal{E}' \)
      \( \text{holds}([n, \langle *p, b \rangle]) = \lor_{\llcorner m, n \lrcorner \in \mathcal{E}} \left( \text{holds}([m, \langle *q, b \rangle]) \right) \)
      \( \text{holds}^\wedge([n, \langle *p, b \rangle]) = \land_{\llcorner m, n \lrcorner \in \mathcal{E}} \left( \text{holds}^\wedge([m, \langle *q, b \rangle]) \right) \)
   (c) if \( a \neq *p \text{ and } b \neq *p \)
      \( \forall \llcorner m, n \lrcorner \in \mathcal{E}, \text{add } \llcorner m, \langle a, b \rangle, [n, \langle a, b \rangle] \lrcorner \to \mathcal{E}' \)
      \( \text{holds}([n, \langle a, b \rangle]) = \lor_{\llcorner m, n \lrcorner \in \mathcal{E}} \left( \text{holds}([m, \langle a, b \rangle]) \right) \)
      \( \text{holds}^\wedge([n, \langle a, b \rangle]) = \land_{\llcorner m, n \lrcorner \in \mathcal{E}} \left( \text{holds}^\wedge([m, \langle a, b \rangle]) \right) \)

5. \( n \) is the pointer assignment "\( p = &v \)" \( [(a \neq b \text{ or } a \text{ is a dereferenced pointer}) \) and \( n \neq \rho \]
   (a) if \( a = b = *p \)
      \( \text{Add } \llcorner \rho', [n, \langle *p, *p \rangle] \lrcorner \to \mathcal{E}' \)
      \( \text{holds}([n, \langle *p, *p \rangle]) = \text{holds}^\wedge([n, \langle *p, *p \rangle]) = \text{true} \)
(b) if \( a = *p \ [b \neq a] \)
\[
\forall \ll m, n \gg \in \mathcal{E}, \text{add} \ \ll [m, \langle v, b \rangle], [n, \langle *p, b \rangle] \gg \ \text{to} \ \mathcal{E}'
\]
\[
\text{holds}([n, \langle *p, b \rangle]) = \lor \ll m, n \gg \in \mathcal{E} \ \text{holds}([m, \langle v, b \rangle])
\]
\[
\text{holds}^\wedge ([n, \langle *p, b \rangle]) = \land \ll m, n \gg \in \mathcal{E} \ \text{holds}^\wedge([m, \langle v, b \rangle])
\]

(c) if \( a \neq *p \) and \( b \neq *p \)
\[
\forall \ll m, n \gg \in \mathcal{E}, \text{add} \ \ll [m, \langle a, b \rangle], [n, \langle a, b \rangle] \gg \ \text{to} \ \mathcal{E}'
\]
\[
\text{holds}([n, \langle a, b \rangle]) = \lor \ll m, n \gg \in \mathcal{E} \ \text{holds}([m, \langle a, b \rangle])
\]
\[
\text{holds}^\wedge ([n, \langle a, b \rangle]) = \land \ll m, n \gg \in \mathcal{E} \ \text{holds}^\wedge([m, \langle a, b \rangle])
\]

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 \( a = b = *p \)
\[
\text{Add} \ \ll \rho', [n, \langle *p, *p \rangle] \gg \ \text{to} \ \mathcal{E}'
\]
\[
\text{holds}([n, \langle *p, *p \rangle]) = \text{holds}^\wedge([n, \langle *p, *p \rangle]) = \begin{cases} 
true & \text{for "p=malloc()"} \\
false & \text{for "p= NULL"}
\end{cases}
\]

(b) if \( a = *p \ [b \neq a] \)
\[
\forall \ll m, n \gg \in \mathcal{E}, \text{add} \ \ll \rho', [n, \langle *p, b \rangle] \gg \ \text{to} \ \mathcal{E}'
\]
\[
\text{holds}([n, \langle *p, b \rangle]) = \text{holds}^\wedge([n, \langle *p, b \rangle]) = false
\]

(c) \( a \neq *p \) and \( b \neq *p \)
\[
\forall \ll m, n \gg \in \mathcal{E}, \text{add} \ \ll [m, \langle a, b \rangle], [n, \langle a, b \rangle] \gg \ \text{to} \ \mathcal{E}'
\]
\[
\text{holds}([n, \langle a, b \rangle]) = \lor \ll m, n \gg \in \mathcal{E} \ \text{holds}([m, \langle a, b \rangle])
\]
\[
\text{holds}^\wedge ([n, \langle a, b \rangle]) = \land \ll m, n \gg \in \mathcal{E} \ \text{holds}^\wedge([m, \langle a, b \rangle])
\]

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

Let \( \text{CFG} = (\mathcal{N}, \mathcal{E}, \rho) \).

Let \( \text{SPPAG} = (\mathcal{N}', \mathcal{E}', \rho') \) and \( \text{holds} : \mathcal{N}' \to \{true, false\} \) 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 $|\text{variables}|^2$. Since a program can use at most $O(|\mathcal{N}|)$ variables, the number of possible aliases is $O(|\mathcal{N}|^2)$.

\[
|\mathcal{N}'| = |\mathcal{N}| \ast |\text{POSS-ALIASES}|
\]
\[
|\mathcal{E}'| \leq |\mathcal{E}| \ast |\text{POSS-ALIASES}| + |\mathcal{N}'|
\]

\[\square\]

**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 $O(\mathcal{E})$ work.

\[\square\]

**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 $\text{holds}(n')$ and $\text{holds}^\wedge(n')$ can change its value at most once.

\[\square\]

**Lemma B.2.4** For every path in the CFG, $\rho n_1 n_2 \ldots n_{i-1} n$, if $\langle a, b \rangle$ holds on the path then $\text{holds}([n, \langle a, b \rangle]) = \text{true}$ after the fixed point calculation.

Proof by induction on $i$, the path length.

**basis:** $i = 0$

The path is simply $\rho$. Only reflexive aliases for variables, $a$, hold. $\text{holds}(\rho, \langle a, a \rangle) = \text{true}$ 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 $\text{holds}(\rho, \langle a, a \rangle)$ to $\text{true}$. 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 $\rho n_1 n_2 \ldots n_{i-1} n$. 
1. \( a = b \) and \( a \) is not a dereferenced pointer

\[ \text{holds}([n, \langle a, a \rangle]) = \text{true} \text{ by definition of holds and } \ll \rho', [n, \langle a, a \rangle] \gg \in \mathcal{E}'. \]

3. \( n \) is not an assignment to a pointer \([a \neq b \text{ or } a \text{ is a dereferenced pointer}]

Since \( \langle a, b \rangle \) holds on \( \rho n_1 n_2 \ldots n_{i-1} n \) it must also hold on \( \rho n_1 n_2 \ldots n_{i-1} \). By induction hypothesis, \( \text{holds}([n_{i-1}, \langle a, b \rangle]) = \text{true} \). Since \( \text{holds}([n, \langle a, b \rangle]) = \bigvee_{\ll m.n \gg} \in \mathcal{E} \left( \text{holds}([m, \langle a, b \rangle]) \right) \) and \( \ll n_{i-1}, n \gg \in \mathcal{E} \), \( \text{holds}([n, \langle a, b \rangle]) \) must also be true.

4. \( n \) is a pointer assignment \( "p = q" \) \([a \neq b \text{ or } a \text{ is a dereferenced pointer}]

(a) if \( a = b = *p \)

\( \langle *q, *q \rangle \) must hold on \( \rho n_1 n_2 \ldots n_{i-1} \). By definition of holds this means that \( *q \) is a name of a location. This is true as long as \( q \) is not \( \text{NULL} \). By induction hypothesis \( \text{holds}([n_{i-1}, \langle *q, *q \rangle]) = \text{true} \). \( \text{holds}([n, \langle *p, *p \rangle]) = \bigvee_{\ll m.n \gg} \in \mathcal{E} \left( \text{holds}([m, \langle *q, *q \rangle]) \right) \). Thus \( \text{holds}([n, \langle *p, *p \rangle]) = \text{true} \).

(b) if \( a = *p \) \([a \neq b] \)

\( \langle *q, b \rangle \) must hold on \( \rho n_1 n_2 \ldots n_{i-1} \). By induction hypothesis,

\( \text{holds}([n_{i-1}, \langle *q, b \rangle]) = \text{true} \). \( \text{holds}([n, \langle *p, b \rangle]) = \bigvee_{\ll m.n \gg} \in \mathcal{E} \left( \text{holds}([m, \langle *q, b \rangle]) \right) \). Thus \( \text{holds}([n, \langle *p, b \rangle]) = \text{true} \).

(c) if \( a \neq *p \) and \( b \neq *p \)

\( \langle a, b \rangle \) must hold on \( \rho n_1 \ldots n_{i-1} \). By induction hypothesis,

\( \text{holds}([n_{i-1}, \langle a, b \rangle]) = \text{true} \). Since \( \text{holds}([n, \langle a, b \rangle]) = \bigvee_{\ll m.n \gg} \in \mathcal{E} \left( \text{holds}([m, \langle a, b \rangle]) \right) \), \( \text{holds}([n, \langle a, b \rangle]) \) must also be \( \text{true} \).

5. \( n \) is a pointer assignment \( "p = &v" \) \([a \neq b \text{ or } a \text{ is a dereferenced pointer}]

(a) if \( a = b = *p \)

By definition \( \text{holds}([n, \langle *p, *p \rangle]) = \text{true} \) and \( \ll \rho', [n, \langle *p, *p \rangle] \gg \in \mathcal{E} \).

(b) if \( a = *p \) \([a \neq b] \)

\( \langle v, b \rangle \) must hold on \( \rho n_1 n_2 \ldots n_{i-1} \). By induction hypothesis,

\( \text{holds}([n_{i-1}, \langle v, b \rangle]) = \text{true} \). \( \text{holds}([n, \langle *p, b \rangle]) = \bigvee_{\ll m.n \gg} \in \mathcal{E} \left( \text{holds}([m, \langle v, b \rangle]) \right) \). Thus \( \text{holds}([n, \langle *p, b \rangle]) = \text{true} \).
(c) if \( a \neq *p \) and \( b \neq *p \)

\[
\langle a, b \rangle \text{ must hold on } \rho n_1 n_2 \ldots n_{i-1}. \text{ By induction hypothesis,}
holds([n_{i-1}, \langle a, b \rangle]) = \text{true. Since } holds([n, \langle a, b \rangle]) =\]

\[
\forall (\ll m, n \gg) \in \mathcal{E} \left( holds([m, \langle a, b \rangle]), holds([n, \langle a, b \rangle]) \right) \text{ must also be true.}
\]

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

\[
holds([n, \langle *p, *p \rangle]) = \text{true and } \ll \rho', [n, \langle *p, *p \rangle] \gg \text{ is in } \mathcal{E}.
\]

(c) if \( a \neq *p \) and \( b \neq *p \)

\[
\langle a, b \rangle \text{ must hold on } \rho n_1 n_2 \ldots n_{i-1}. \text{ By induction hypothesis,}
holds([n_{i-1}, \langle a, b \rangle]) = \text{true. Since } holds([n, \langle a, b \rangle]) =\]

\[
\forall (\ll m, n \gg) \in \mathcal{E} \left( holds([m, \langle a, b \rangle]), holds([n, \langle a, b \rangle]) \right) \text{ must also be true.}
\]

\[\square\]

**Lemma B.2.5** If \( holds([n, \langle a, b \rangle]) = \text{true} \) after the fixed point calculation then for some path in the CFG, \( \rho n_1 n_2 \ldots n_{i-1} n, \langle a, b \rangle \text{ holds.} \)

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 \( \text{true} \) 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 \( \text{holds}(\langle n, \langle a, b \rangle \rangle) \) 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 \( n \neq \rho \)] 

\( \text{holds}(\langle n, \langle a, b \rangle \rangle) = \bigvee_{\langle m, n \rangle} \in \mathcal{E} \left( \text{holds}(\langle m, \langle a, b \rangle \rangle) \right) \) Thus for some \( \langle m, n \rangle \in \mathcal{E} \), \( \text{holds}(\langle m, \langle a, b \rangle \rangle) \) must be true at iteration \( i-1 \). By the induction hypothesis, there is a path \( \rho n_1 n_2 \ldots m \) on which \( \langle a, b \rangle \) holds. Thus \( \langle a, b \rangle \) must also hold on the path \( \rho n_1 n_2 \ldots mn \).

4. \( n \) is the pointer assignment "\( p = q \)" [(\( a \neq b \) or \( a \) is not a dereferenced pointer) and \( n \neq \rho \)]

(a) if \( a = b = *p \) 

\( \text{holds}(\langle n, \langle *p, *p \rangle \rangle) = \bigvee_{\langle m, n \rangle} \in \mathcal{E} \left( \text{holds}(\langle m, \langle *q, *q \rangle \rangle) \right) \) Thus for some \( \langle m, n \rangle \in \mathcal{E} \), \( \text{holds}(\langle m, \langle *q, *q \rangle \rangle) \) must be true at iteration \( i-1 \). By the induction hypothesis, there is a path \( \rho n_1 n_2 \ldots m \) on which \( \langle *q, *q \rangle \) holds. Thus \( \langle *p, *p \rangle \) must hold on the path \( \rho n_1 n_2 \ldots mn \).

(b) if \( a = *p \) [\( a \neq b \)] 

\( \text{holds}(\langle n, \langle *p, b \rangle \rangle) = \bigvee_{\langle m, n \rangle} \in \mathcal{E} \left( \text{holds}(\langle m, \langle *q, b \rangle \rangle) \right) \) Thus for some \( \langle m, n \rangle \in \mathcal{E} \), \( \text{holds}(\langle m, \langle *q, b \rangle \rangle) \) must be true at iteration \( i-1 \). By the induction hypothesis, there is a path \( \rho n_1 n_2 \ldots m \) on which \( \langle *q, b \rangle \) holds. Thus \( \langle *p, b \rangle \) must hold on the path \( \rho n_1 n_2 \ldots mn \).

(c) if \( a \neq *p \) and \( b \neq *p \) 

\( \text{holds}(\langle n, \langle a, b \rangle \rangle) = \bigvee_{\langle m, n \rangle} \in \mathcal{E} \left( \text{holds}(\langle m, \langle a, b \rangle \rangle) \right) \) Thus for some \( \langle m, n \rangle \in \mathcal{E} \), \( \text{holds}(\langle m, \langle a, b \rangle \rangle) \) must be true at iteration \( i-1 \). By the induction hypothesis, there is a path \( \rho n_1 n_2 \ldots m \) on which \( \langle a, b \rangle \) holds. Thus \( \langle a, b \rangle \) must also hold on the path \( \rho n_1 n_2 \ldots mn \).

5. \( n \) is the pointer assignment "\( p = &v \)" [(\( a \neq b \) or \( a \) is not a dereferenced pointer) and \( n \neq \rho \)]
(a) if \( a = b = *p \)
\[
\langle *p, *p \rangle \text{ must hold on all paths to } n.
\]

(b) if \( a = *p \ [a \neq b] \)
\[
\text{holds}(\langle m, \langle *p, b \rangle \rangle) = \bigvee_{\langle m, n \rangle \in \mathcal{E}} \left( \text{holds}(\langle m, \langle v, b \rangle \rangle) \right) \text{ Thus for some } \langle m, n \rangle \in \mathcal{E}, \text{ holds}(\langle m, \langle v, b \rangle \rangle) \text{ must be true at iteration } i-1. \text{ By the induction hypothesis, there is a path } \rho n_1 n_2 \ldots m \text{ on which } \langle v, b \rangle \text{ holds. Thus } \langle *p, b \rangle \text{ must hold on the path } \rho n_1 n_2 \ldots mn.
\]

(c) if \( a \neq *p \) and \( b \neq *p \)
\[
\text{holds}(\langle m, \langle a, b \rangle \rangle) = \bigvee_{\langle m, n \rangle \in \mathcal{E}} \left( \text{holds}(\langle m, \langle a, b \rangle \rangle) \right) \text{ Thus for some } \langle m, n \rangle \in \mathcal{E}, \text{ holds}(\langle m, \langle a, b \rangle \rangle) \text{ must be true at iteration } i-1. \text{ By the induction hypothesis, there is a path } \rho n_1 n_2 \ldots m \text{ on which } \langle a, b \rangle \text{ holds. Thus } \langle a, b \rangle \text{ must also hold on the path } \rho n_1 n_2 \ldots mn.
\]

6. \( n \) is the pointer assignment “\( p = \text{malloc()} \)” or “\( p = \text{NULL} \)” [(\( a \neq b \) or \( a \) is not a dereferenced pointer) and \( n \neq p \)]

(a) if \( a = b = *p \)
\[
\begin{itemize}
  \item \( n \) is “\( p = \text{malloc()} \)”
  \[
  \langle *p, *p \rangle \text{ must hold on all paths to } n.
  \]
  \item \( n \) is “\( p = \text{NULL} \)”
  
  By definition it is impossible for \( \text{holds}(\langle n, \langle *p, *p \rangle \rangle) \) to be true.
\end{itemize}

(b) if \( a = *p \ [a \neq b] \)

By definition it is impossible for \( \text{holds}(\langle n, \langle *p, b \rangle \rangle) \) to be true.

(c) if \( a \neq *p \) and \( b \neq *p \)
\[
\text{holds}(\langle n, \langle a, b \rangle \rangle) = \bigvee_{\langle m, n \rangle \in \mathcal{E}} \left( \text{holds}(\langle m, \langle a, b \rangle \rangle) \right) \text{ Thus for some } \langle m, n \rangle \in \mathcal{E}, \text{ holds}(\langle m, \langle a, b \rangle \rangle) \text{ must be true at iteration } i-1. \text{ By the induction hypothesis, there is a path } \rho n_1 n_2 \ldots m \text{ on which } \langle a, b \rangle \text{ holds. Thus } \langle a, b \rangle \text{ must also hold on the path } \rho n_1 n_2 \ldots mn.
\]

\(\square\)

**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.

\[\square\]

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

**Lemma B.3.1** For every path in the CFG, \( pn_1n_2...n_{i-1}n \), 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 \( \rho \). 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 \( \ll \rho', [\rho, \langle a, b \rangle] \gg \in E' \) 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 \( pn_1n_2...n_{i-1}n \).

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 \( pn_1n_2...n_{i-1}n \), it cannot hold on \( pn_1n_2...n_{i-1} \). By induction hypothesis, \( holds^\wedge([n_{i-1}, \langle a, b \rangle]) = false \). Since \( holds^\wedge([n, \langle a, b \rangle]) = \)
\[ \land \llangle m, n \rrangle \in \mathcal{E} \left( \text{holds}^\wedge([m, \langle a, b \rangle]) \right) \text{ and } \llangle n_{i-1}, n \rrangle \in \mathcal{E}, \text{ holds}^\wedge([n, \langle a, b \rangle]) \text{ must also be false.} \]

4. \( n \) is a pointer assignment "\( p = q \)" [(\( a \neq b \) or \( a \) is a dereferenced variable) and \( n \neq p \)]

(a) if \( a = b = \ast p \)

\( \langle \ast q, \ast q \rangle \) cannot hold on \( \rho n_1 n_2 \ldots n_{i-1} \). By induction hypothesis,

\[ \text{holds}^\wedge([n_{i-1}, \langle \ast q, \ast q \rangle]) = \text{false}. \text{ holds}^\wedge([n, \langle \ast p, \ast p \rangle]) = \land \llangle m, n \rrangle \in \mathcal{E} \left( \text{holds}^\wedge([m, \langle \ast q, \ast q \rangle]) \right). \text{ Thus holds}^\wedge([n, \langle \ast p, \ast p \rangle]) = \text{false}. \]

(b) if \( a = \ast p \ [a \neq b] \)

\( \langle \ast q, b \rangle \) cannot hold on \( \rho n_1 n_2 \ldots n_{i-1} \). By induction hypothesis,

\[ \text{holds}^\wedge([n_{i-1}, \langle \ast q, b \rangle]) = \text{false}. \text{ holds}^\wedge([n, \langle \ast p, b \rangle]) = \land \llangle m, n \rrangle \in \mathcal{E} \left( \text{holds}^\wedge([m, \langle \ast q, b \rangle]) \right). \text{ Thus holds}^\wedge([n, \langle \ast p, b \rangle]) = \text{false}. \]

(c) if \( a \neq \ast p \) and \( b \neq \ast p \)

\( \langle a, b \rangle \) cannot hold on \( \rho n_1 n_2 \ldots n_{i-1} \). By induction hypothesis,

\[ \text{holds}^\wedge([n_{i-1}, \langle a, b \rangle]) = \text{false}. \text{ holds}^\wedge([n, \langle a, b \rangle]) = \land \llangle m, n \rrangle \in \mathcal{E} \left( \text{holds}^\wedge([m, \langle a, b \rangle]) \right). \text{ Thus holds}^\wedge([n, \langle a, b \rangle]) \text{ must also be false.} \]

5. \( n \) is a pointer assignment "\( p = \& v \)" [(\( a \neq b \) or \( a \) is a dereferenced variable) and \( n \neq p \)]

(a) if \( a = b = \ast p \)

\( \langle \ast p, \ast p \rangle \) must be aliased on all paths to \( n \) thus the theorem vacuously holds.

(b) if \( a = \ast p \ [a \neq b] \)

\( \langle v, b \rangle \) cannot hold on \( \rho n_1 n_2 \ldots n_{i-1} \). By induction hypothesis,

\[ \text{holds}^\wedge([n_{i-1}, \langle v, b \rangle]) = \text{false}. \text{ holds}^\wedge([n, \langle \ast p, b \rangle]) = \land \llangle m, n \rrangle \in \mathcal{E} \left( \text{holds}^\wedge([m, \langle v, b \rangle]) \right). \text{ Thus holds}^\wedge([n, \langle \ast p, b \rangle]) = \text{false}. \]

(c) if \( a \neq \ast p \) and \( b \neq \ast p \)

\( \langle a, b \rangle \) cannot hold on \( \rho n_1 n_2 \ldots n_{i-1} \). By induction hypothesis,

\[ \text{holds}^\wedge([n_{i-1}, \langle a, b \rangle]) = \text{false}. \text{ holds}^\wedge([n, \langle a, b \rangle]) = \land \llangle m, n \rrangle \in \mathcal{E} \left( \text{holds}^\wedge([m, \langle a, b \rangle]) \right). \text{ Thus holds}^\wedge([n, \langle a, b \rangle]) \text{ must also be false.} \]
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 \( \text{holds}^\wedge([n,\langle *p, *p \rangle]) = false \) by definition.

(b) if \( a = *p \ [a \neq b] \)

By definition of \( \text{holds}^\wedge \), \( \text{holds}^\wedge([n,\langle *p, b \rangle]) = false \).

(c) if \( a \neq *p \) and \( b \neq *p \)

\( \langle a, b \rangle \) cannot hold on \( \rho n_1n_2...n_{i-1} \). By induction hypothesis,

\[
\text{holds}^\wedge([n_{i-1},\langle a, b \rangle]) = false. \text{ holds}^\wedge([n,\langle a, b \rangle]) = \land_{m,n \in \mathcal{E}} \text{holds}^\wedge([m,\langle a, b \rangle]).
\]

Thus \( \text{holds}^\wedge([n,\langle a, b \rangle]) \) must also be \( false \).

\[\square\]

**Lemma B.3.2** If \( \text{holds}^\wedge([n,\langle a, b \rangle]) = false \) after the fixed point calculation then for some path in the CFG, \( \rho n_1n_2...n_{i-1}n \), \( \langle a, b \rangle \) does not hold.

Proof by number of iteration of the fixed point algorithm.

**basis:** After initialization.

For all \( n \in \mathcal{N}' \), \( \text{holds}^\wedge(n) \) 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 \( \text{holds}^\wedge([n,\langle a, b \rangle]) \) be made \( false \) in iteration \( i \).

1. \( a = b \) and \( a \) is not a dereferenced variable

   By definition of \( \text{holds}^\wedge \), \( \text{holds}^\wedge([n,\langle a, a \rangle]) = true. \) Thus \( \text{holds}^\wedge([n,\langle a, a \rangle]) \) 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 \( [(a \neq b \text{ or } a \text{ is a dereferenced variable}) \text{ and} \ n \neq \rho] \) 
\[ \text{holds}^\wedge([n, \langle a, b \rangle]) = \land_{<m,n> \in E} \left( \text{holds}^\wedge([m, \langle a, b \rangle]) \right) \] 
Thus for some \( <m,n> \in E \), \( \text{holds}^\wedge([m, \langle a, b \rangle]) \) must be \textit{false} at iteration i-1. By the induction hypothesis, there is a path \( \rho n_1 n_2 \ldots m \) on which \( \langle a, b \rangle \) does not hold. Thus \( \langle a, b \rangle \) cannot hold on the path \( \rho n_1 n_2 \ldots mn \).

4. \( n \) is the pointer assignment “\( p = q \)” \( [(a \neq b \text{ or } a \text{ is a dereferenced variable}) \text{ and} \ n \neq \rho] \) 
\( (a) \) if \( a = b = \ast p \) 
\[ \text{holds}^\wedge([n, \langle \ast p, \ast p \rangle]) = \land_{<m,n> \in E} \left( \text{holds}^\wedge([m, \langle \ast q, \ast q \rangle]) \right) \] 
Thus for some \( <m,n> \in E \), \( \text{holds}^\wedge([m, \langle \ast q, \ast q \rangle]) \) must be \textit{false} at iteration i-1. By the induction hypothesis, there is a path \( \rho n_1 n_2 \ldots m \) on which \( \langle \ast q, \ast q \rangle \) does not hold. Thus \( \langle \ast q, \ast q \rangle \) cannot hold on the path \( \rho n_1 n_2 \ldots mn \).

\( (b) \) if \( a = \ast p \ [a \neq b] \) 
\[ \text{holds}^\wedge([n, \langle \ast p, b \rangle]) = \land_{<m,n> \in E} \left( \text{holds}^\wedge([m, \langle \ast q, b \rangle]) \right) \] 
Thus for some \( <m,n> \in E \), \( \text{holds}^\wedge([m, \langle \ast q, b \rangle]) \) must be \textit{false} at iteration i-1. By the induction hypothesis, there is a path \( \rho n_1 n_2 \ldots m \) on which \( \langle \ast q, b \rangle \) does not hold. Thus \( \langle \ast q, b \rangle \) cannot hold on the path \( \rho n_1 n_2 \ldots mn \).

\( (c) \) if \( a \neq \ast p \) and \( b \neq \ast p \) 
\[ \text{holds}^\wedge([n, \langle a, b \rangle]) = \land_{<m,n> \in E} \left( \text{holds}^\wedge([m, \langle a, b \rangle]) \right) \] 
Thus for some \( <m,n> \in E \), \( \text{holds}^\wedge([m, \langle a, b \rangle]) \) must be \textit{false} at iteration i-1. By the induction hypothesis, there is a path \( \rho n_1 n_2 \ldots m \) on which \( \langle a, b \rangle \) does not hold. Thus \( \langle a, b \rangle \) cannot hold on the path \( \rho n_1 n_2 \ldots mn \).

5. \( n \) is the pointer assignment “\( p = &v \)” \( [(a \neq b \text{ or } a \text{ is a dereferenced variable}) \text{ and} \ n \neq \rho] \) 
\( (a) \) if \( a = b = \ast p \) 
\[ \text{holds}^\wedge([n, \langle \ast p, \ast p \rangle]) = \text{true} \] by definition and cannot possibly be \textit{false}. 

(b) if \( a = *p \ [a \neq b] \)

\[
holds^\wedge([n, \langle *p, b \rangle]) = \land_{\ll m,n \gg \in \mathcal{E}} (holds^\wedge([m, \langle v, b \rangle]))\]

Thus for some \( \ll m,n \gg \in \mathcal{E} \), \( holds^\wedge([m, \langle v, b \rangle]) \) must be \textit{false} at iteration i-1. By the induction hypothesis, there is a path \( \rho n_1 n_2 \ldots m \) on which \( \langle v, b \rangle \) does not hold. Thus \( \langle *p, b \rangle \) cannot hold on the path \( \rho n_1 n_2 \ldots mn \).

(c) if \( a \neq *p \) and \( b \neq *p \)

\[
holds^\wedge([n, \langle a, b \rangle]) = \land_{\ll m,n \gg \in \mathcal{E}} (holds^\wedge([m, \langle a, b \rangle]))\]

Thus for some \( \ll m,n \gg \in \mathcal{E} \), \( holds^\wedge([m, \langle a, b \rangle]) \) must be \textit{false} at iteration i-1. By the induction hypothesis, there is a path \( \rho n_1 n_2 \ldots m \) on which \( \langle a, b \rangle \) does not hold. Thus \( \langle a, b \rangle \) cannot hold on the path \( \rho n_1 n_2 \ldots mn \).

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 \( " p = malloc() " \)

\[
holds^\wedge([n, \langle *p, *p \rangle]) = \text{true} \text{ by definition and can never be false.}
\]

- \( n \) 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 \( b \neq *p \)

\[
holds^\wedge([n, \langle a, b \rangle]) = \land_{\ll m,n \gg \in \mathcal{E}} (holds^\wedge([m, \langle a, b \rangle]))\]

Thus for some \( \ll m,n \gg \in \mathcal{E} \), \( holds^\wedge([m, \langle a, b \rangle]) \) must be \textit{false} at iteration i-1. By the induction hypothesis, there is a path \( \rho n_1 n_2 \ldots m \) on which \( \langle a, b \rangle \) does not hold. Thus \( \langle a, b \rangle \) cannot hold on the path \( \rho n_1 n_2 \ldots mn \).

\( \Box \)

\textbf{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

$\text{back-bind}_{\text{call}_P}(\text{alias-pair})$ specifies which alias holding on $\rho n_1...n_{i-1}[\text{call}_P]$ forces $\text{alias-pair}$ to hold on $\rho n_1...n_{i-1}[\text{call}_P][\text{entry}_P]$.

Let $\text{back-bind}_n$, $n$ a call node in the ICFG, be defined as:

$\text{back-bind}_n(\emptyset) = \emptyset$
\[
\begin{align*}
\langle a, b \rangle & \quad \text{if } a \text{ and } b \text{ are both visible in the procedure called by } n \\
\langle a, *a_j \rangle & \quad \text{if } a \text{ is visible in the procedure called by } n, \text{ and } b \text{ is the dereferenced formal } *f_j \text{ with corresponding actual } a_j \\
\langle *a_j, *a_k \rangle & \quad \text{if } a \text{ is the dereferenced formal } *f_j \text{ with corresponding actual } a_j, \text{ and } b \text{ is the dereferenced formal } *f_k \text{ with corresponding actual } a_k \\
\mathit{false} & \quad \text{otherwise (either } a \text{ or } b \text{ is a local [non-parameter] object name)}
\end{align*}
\]

\[
\text{back-binding}_{\mathit{call}_p}(\langle a, \text{non-visible} \rangle, o) \text{ specifies the alias holding on any path } \rho...[\text{call}_p] \text{ that guarantees } a \text{ will be aliased to the non-visible object } o \text{ on } \rho...[\text{call}_p][\text{entry}_p].
\]

\[
\begin{align*}
\langle a, o \rangle & \quad \text{if } a \text{ is visible in the called procedure} \\
\langle *a_j, o \rangle & \quad \text{if } a \text{ is the dereferenced formal } *f_j \text{ with corresponding actual } a_j \\
\mathit{false} & \quad \text{otherwise (} a \text{ is a local [non-parameter] object name)}
\end{align*}
\]

C.1.2 Proof of correctness

Let \text{called}(n), \text{n} a call node, be the entry node of the procedure called by \text{n}.

\textbf{Lemma C.1.1} For all call nodes, n:
- \( \text{back-bind}_n(\emptyset) = \emptyset \)
  
  On all paths \( \rho n_1 n_2 \ldots n[\text{called}(n)] \), \( \emptyset \) is considered to 'hold'.

- For all possible alias pairs, \( \langle a, b \rangle \) (\( a \neq \text{non_visible} \) and \( b \neq \text{non_visible} \)):
  
  - If \( \text{back-bind}_n(\langle a, b \rangle) = \text{false} \) then
    
    On all paths \( \rho n_1 n_2 \ldots n[\text{called}(n)] \), \( \langle a, b \rangle \) does not hold.
  
  - If \( \text{back-bind}_n(\langle a, b \rangle) = \langle c, d \rangle \) then
    
    On all paths \( \rho n_1 n_2 \ldots n[\text{called}(n)] \), \( \langle a, b \rangle \) holds iff \( \langle c, d \rangle \) holds on \( \rho n_1 n_2 \ldots n \).

- For all possible alias pairs \( \langle a, \text{non_visible} \rangle \) (\( a \neq \text{non_visible} \)), and all possible \( o \) (\( o \) is an object name not visible in the called procedure):
  
  - If \( \text{back-bind}_n(\langle a, \text{non_visible} \rangle, o) = \text{false} \) then
    
    On all paths \( \rho n_1 n_2 \ldots n[\text{called}(n)] \), \( a \) can never be aliased to the non-visible object name \( o \).
  
  - If \( \text{back-bind}_n(\langle a, \text{non_visible} \rangle, o) = \langle c, o \rangle \) then
    
    On all paths \( \rho n_1 n_2 \ldots n[\text{called}(n)] \), \( a \) is aliased to the non-visible object name \( o \) iff \( \langle c, o \rangle \) holds on \( \rho n_1 n_2 \ldots n \).

Proof:

For all call nodes, \( n \):

1. \( \text{back-bind}_n(\emptyset) = \emptyset \)
   
   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, \( \langle a, b \rangle \) (\( a \neq \text{non_visible} \) and \( b \neq \text{non_visible} \)):
   
   (a) \( \text{back-bind}_n(\langle a, b \rangle) = \text{false} \)
    
    This only occurs when either \( a \) or \( b \) are local (non-parameter) object names. In this case, on all paths \( \rho n_1 n_2 \ldots n[\text{called}(n)] \), \( \langle a, b \rangle \) does not hold.

   (b) \( \text{back-bind}_n(\langle a, b \rangle) = \langle c, d \rangle \)
i.  

\[ a \text{ and } b \text{ are both visible in the called procedure} \]

By definition \( \langle c, d \rangle = \langle a, b \rangle \).

Clearly on all paths \( \rho_{n_2...n}[\text{called}(n)] \), \( \langle a, b \rangle \) holds iff \( \langle a, b \rangle \) holds on \( \rho_{n_2...n} \).

ii.  

\[ a \text{ is visible in the called procedure, and } b \text{ is the dereferenced formal } \star f_j \text{ with actual } a_j \]

By definition \( \langle c, d \rangle = \langle a, \star a_j \rangle \).

Clearly on all paths \( \rho_{n_2...n}[\text{called}(n)] \), \( \langle a, \star f_j \rangle \) holds iff \( \langle a, \star a_j \rangle \) holds on \( \rho_{n_2...n} \).

iii.  

\[ a \text{ is the dereferenced formal } \star f_j \text{ with actual } a_j, \text{ and } b \text{ is the dereferenced formal } \star f_k \text{ with actual } a_k \]

By definition \( \langle c, d \rangle = \langle \star a_j, \star a_k \rangle \).

Clearly on all paths \( \rho_{n_2...n}[\text{called}(n)] \), \( \langle \star f_j, \star f_k \rangle \) holds iff \( \langle \star a_j, \star a_k \rangle \) holds on \( \rho_{n_2...n} \).

3. For all possible alias pairs \( \langle a, \text{non_visible} \rangle \) (\( a \neq \text{non_visible} \)), and all possible \( o \) (\( o \) is an object name not visible in the called procedure):

(a) \( \text{back-bind}_n(\langle a, \text{non_visible} \rangle, o) = \text{false} \)

This only occurs when \( a \) is a local (non-parameter) object name.

Thus on all paths \( \rho_{n_2...n}[\text{called}(n)] \), \( a \) can never be aliased to anything.

(b) \( \text{back-bind}_n(\langle a, \text{non_visible} \rangle, o) = \langle c, \text{non_visible} \rangle \)

i.  

if \( a \) is visible in the called procedure

By definition \( \langle c, \text{non_visible} \rangle = \langle a, \text{non_visible} \rangle \).

On all paths \( \rho_{n_2...n}[\text{called}(n)] \), \( a \) is aliased to the non-visible object name \( o \) iff \( \langle a, o \rangle \) holds on \( \rho_{n_2...n} \).

ii.  

if \( a \) is the dereferenced formal \( \star f_j \) with actual \( a_j \)

By definition \( \langle c, \text{non_visible} \rangle = \langle \star a_j, \text{non_visible} \rangle \).

On all paths \( \rho_{n_2...n}[\text{called}(n)] \), \( \star f_j \) is aliased to the non-visible object name \( o \) iff \( \langle \star a_j, o \rangle \) holds on \( \rho_{n_2...n} \).
C.1.3 Computable in constant time

Lemma C.1.2 Given a call node $n$ and an alias $(a, b)$, $\text{back-bind}_n((a, b))$ can be computed in constant time.

Given a call node $n$, an alias $(c, \text{non-visible})$, and object name $o$ which is not visible in the called procedure, $\text{back-bind}_n((c, \text{non-visible}), o)$ can be computed in constant time.

The naive encoding of the definitions of $\text{back-bind}_n((a, b))$ and $\text{back-bind}_n((c, \text{non-visible}), o)$ yields constant time algorithms.

C.2 Bind

C.2.1 Definition

$\text{bind}_{\text{call}_P}(\text{alias-set})$ specifies for all paths $\rho n_1...n_i[\text{call}_P][\text{entry}_P]$ which aliases hold assuming all aliases in $\text{alias-set}$ hold on $\rho n_1...n_i[\text{call}_P]$.

Let $O' = \{ *p \mid p$ is a pointer in the program $\} \cup \{ v \mid v$ is a non-pointer variable in the program $\}$.

$O'$ is the set of all object names in the program which may have aliases.

Let $\text{POSSIBLE-ALIASES}' = (O' \times O')$.

$\text{POSSIBLE-ALIASES}'$ is the set of all possible aliases.

Let $\text{bind}_n(A), n$ a call node in the ICFG and $A \subseteq \text{POSSIBLE-ALIASES}'$, be defined as: $\text{bind}_n(A) = \text{bind}_n'(\emptyset) \cup \bigcup_{(a, b) \in A} (\text{bind}_n((a, b)))$
\[
bind_n(\emptyset) = \left\{ \begin{array}{l}
\langle *f_i, *f_j \rangle \quad \text{if } f_i \text{ and } f_j \text{ are pointer formals with actuals} \\
\langle a_i \text{ and } a_j \text{, respectively, and } a_i = a_j \text{ and neither } a_i \text{ nor } a_j \text{ is a pointer variable} \rangle
\end{array} \right\} \cup
\left\{ \begin{array}{l}
\langle f_i \rangle \quad \text{if } f_i \text{ is a pointer formal with actual } a_i, \\
\langle *f_i, *a_i \rangle \quad \text{and } a_i \text{ is visible in the called procedure and } a_i \text{ is not a pointer variable}
\end{array} \right\}
\]

\[
bind_n(\langle a, b \rangle) = \left\{ \begin{array}{l}
\langle a, b \rangle \quad \text{if } a \text{ and } b \text{ are visible in the called procedure} \\
\langle a, *f_i \rangle \quad \text{if } a \text{ is visible in the called procedure, } f_i \text{ is a pointer formal with actual } a_i, \text{ and } *a_i = b
\end{array} \right\} \cup
\left\{ \begin{array}{l}
\langle *f_i, *f_j \rangle \quad \text{if } f_i \text{ and } f_j \text{ are pointer formals with corresponding actuals } a_i \text{ and } a_j, \text{ and } *a_i = a \text{ and } *a_j = b
\end{array} \right\}
\]

C.2.2 Proof of correctness

Let \texttt{called}(\texttt{call.node}) be the entry node of the procedure called by \texttt{call.node}.

**Lemma C.2.1** For a call node \( n \) and a set of aliases \( \text{alias-set} \), \( bind_n(\text{alias-set}) \) is the set of aliases that hold on \( \rho n_1 \ldots n_{i-1} n[\text{called}(n)] \) given that \( \text{alias-set} \) holds on \( \rho n_1 \ldots n_{i-1} n \).

Lemma 4.2.1 states that we can consider the effect of each alias in \( \text{alias-set} \) 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 \([\text{called}(n)]\) even if no aliases hold at \(n\) are aliases between dereferenced pointer formals that are passed the same (non-pointer) actual and aliases between dereferenced pointer formals and their dereferenced (non-pointer) actuals (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 \([\text{called}(n)]\) 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 \(bind'_n(\langle a, b \rangle)\).

\(\square\)

\section*{C.2.3 Computable in constant time}

\textbf{Lemma C.2.2} Given a call node \(n\) and an alias set alias-set, \(bind_n(\text{alias-set})\) can be computed in time polynomial in \(O(|\text{alias-set}| \times |(\text{number of formals in the called procedure})^2|)\).

The naive encoding of the definition of \(bind_n(\text{alias-set})\) satisfies the above time constraints.

\(\square\)

\footnote{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 $O = \{*p \mid p \text{ is a pointer in the program}\} \cup \\
\{v \mid v \text{ is a non-pointer variable in the program}\} \cup \\
\{\text{non-visible}\}.$

$O$ is the set of all object names in the program which may have aliases.

"non_visible" represents object names which are not visible.

Let $P_{OSSIBLE-ALIASES} = (O \times O) - \{(\text{non-visible}, \text{non-visible})\}$

and $\text{ASSUMED } = P_{OSSIBLE-ALIASES} \cup \{\emptyset\}$

Given an ICFG for a program:

Let $\text{exit}(n), n$ a return node in the ICFG, be the exit node corresponding to $n$.
Let $\text{call}(n), n$ a return node in the ICFG, be the call node corresponding to $n$.
Let $\text{entry}(n), n$ an ICFG node, be the entry node of the procedure containing $n$.

Let $\text{back-bind}_\text{call}, \text{call}$ a call node in the ICFG, be defined as in Appendix C.1.
$\text{back-bind}_\text{call}(\text{alias-pair})$ specifies which alias holding at the call site forces
$\text{alias-pair}$ to hold at the entry of the called procedure.

\begin{align*}
n_1n_2...n_i \cdot m_1m_2...m_j &= n_1n_2...n_im_1m_2...m_j \\
n_1n_2...n_i \circ m_1m_2...m_j &= n_1n_2...n_im_2...m_j \ [n_i \text{ must equal } m_1]
\end{align*}

C.3.2 Building the Pointer Alias Graph (PAG)

Given $\text{ICFG} = (N, E, \rho)$, construct $\text{PAG} = (N', E', \rho')$ and $\text{holds} : N' \rightarrow \{\text{true, false}\}$.

$\text{holds}([\text{node}, \mathcal{AA}, \mathcal{PA}]) = \text{true}$ iff
∃ a realizable path \( P = \text{entry}(\text{node})|n_1n_2...n_i[\text{node}] \) in the ICFG, where
\( n_1n_2...n_i \) contains the same number of calls as returns, such that for every
realizable path \( P' = ρm_1...m_i[\text{entry}(\text{node})] \), \( \mathcal{A}\mathcal{A} \) holding\(^{77} \) on \( P' \) means \( \mathcal{P}\mathcal{A} \)
must hold on \( P' \cdot P \) (i.e., \( ρm_1...m_i[\text{entry}(\text{node})]|n_1n_2...n_i[\text{node}] \)).

In order to simply if the algorithm description and proof we expand the definition of
holds. For all \( \text{node} \in \text{ICFG} \), all \( \mathcal{A}\mathcal{A} \in \text{ASSUMED} \), \( \text{holds}([\text{node}, \mathcal{A}\mathcal{A}], \text{false}] = \text{false} \).

\[
\mathcal{N}' = \left\{ \text{node} \in \mathcal{N}, \mathcal{A}\mathcal{A} \in \text{ASSUMED}, \right. \\
\text{and } \mathcal{P}\mathcal{A} \in \text{POSSIBLE-ALIASES} \} \cup \{ρ'\}
\]

Specification of \( \mathcal{E}' \) and holds.
For all \( \text{node} \in \mathcal{N} \), for all \( \langle a, b \rangle \in \text{POSSIBLE-ALIASES} \), for all \( \mathcal{A}\mathcal{A} \in \text{ASSUMED} \):

1. \( a = b \) and \( a \) is not a dereferenced pointer

   (Reflexive aliases for variables always hold)

   Add \( \ll ρ', \langle \text{node}, \mathcal{A}\mathcal{A}, \langle a, b \rangle \rangle \gg \) to \( \mathcal{E}' \)

   \( \text{holds}([\text{node}, \mathcal{A}\mathcal{A}, \langle a, b \rangle]) = \text{true} \)

2. \( a \neq b \) or \( a \) is a dereferenced pointer

   (a) \( \text{node} \) is an entry node

      \( \left( \begin{array}{c}
      \text{An alias conditionally holds at an entry node iff we assume it does} \\
      \text{or it is a reflexive alias for a variable.}
      \end{array} \right) \)

      Add \( \ll ρ', \langle \text{node}, \mathcal{A}\mathcal{A}, \langle a, b \rangle \rangle \gg \) to \( \mathcal{E}' \)

      \( \text{holds}([\text{node}, \mathcal{A}\mathcal{A}, \langle a, b \rangle]) = \begin{cases} 
      \text{true} & \text{if } \mathcal{A}\mathcal{A} = \langle a, b \rangle \\
      \text{false} & \text{otherwise}
      \end{cases} \)

   (b) \( \text{node} \) 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 \( \ll m, \text{node} \gg \in \mathcal{E} \), add \( \ll \ll m, \mathcal{A}\mathcal{A}, \langle a, b \rangle \gg, \langle \text{node}, \mathcal{A}\mathcal{A}, \langle a, b \rangle \rangle \gg \) to \( \mathcal{E}' \)

      \( \text{holds}([\text{node}, \mathcal{A}\mathcal{A}, \langle a, b \rangle]) = \bigvee \ll m, \text{node} \gg \in \mathcal{E} \left( \text{holds}([m, \mathcal{A}\mathcal{A}, \langle a, b \rangle]) \right) \)

\(^{77}\)We consider the assumed alias \( \theta \) 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 \( \ll m, \text{node} \gg \in \mathcal{E} \), add \( \ll \ll m, \text{AA} \gg, \langle a, b \rangle, \ll \text{node}, \text{AA} \gg, \langle a, b \rangle \gg \gg \rightarrow \mathcal{E}' \)

\[
\text{holds}(\ll \text{node}, \text{AA} \gg, \langle a, b \rangle) = \bigvee_{\ll m, \text{node} \gg \in \mathcal{E}} \left( \text{holds}(\ll m, \text{AA} \gg, \langle a, b \rangle) \right)
\]

(d) node is a return node

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

\( \left( \right. \text{The procedure call cannot affect this alias. The alias is} \text{ holds after the call} \left. \right) \text{ iff it held before the call.} \)

Add \( \ll \ll \text{call(node)} \gg, \langle a, b \rangle, \ll \text{node}, \text{AA} \gg, \langle a, b \rangle \gg \gg \rightarrow \mathcal{E}' \)

\[
\text{holds}(\ll \text{node}, \text{AA} \gg, \langle a, b \rangle) = \text{holds}(\ll \text{call(node)}, \text{AA} \gg, \langle a, b \rangle)
\]

ii. a is visible in the called procedure, but b is not

\( \left( \langle a, b \rangle \text{ holds on a path to node iff} \langle a, \text{non_visible} \rangle \right) \left( \text{holds at exit(node) and \text{“non_visible” is b} \right) \)

For every \( \langle c, \text{non_visible} \rangle \in \text{ASSUMED} \):

if \( \text{back-binding}'(\ll \langle c, \text{non_visible} \rangle, \ll a, \ll b \gg) \neq \text{false} \) then add

\[
\ll \ll \text{exit(node)} \gg, \langle c, \text{non_visible} \rangle, \ll a, \text{non_visible} \gg, \ll \text{node}, \text{AA} \gg, \langle a, b \rangle \gg \gg \rightarrow \mathcal{E}' \)

\[
\text{holds}(\ll \text{node}, \text{AA} \gg, \langle a, b \rangle) = \bigvee_{\langle c, \text{non_visible} \rangle \in \text{ASSUMED}} \left( \text{holds}(\ll \text{exit(node)}, \langle c, \text{non_visible} \rangle, \langle a, \text{non_visible} \rangle) \right) \land \left( \text{holds}(\ll \text{call(node)}, \text{AA} \gg, \langle c, \text{non_visible} \rangle) \text{, back-binding}'(\ll \langle c, \text{non_visible} \rangle, b) \gg \rightarrow \mathcal{E}' \right)
\]

iii. a and b are visible in the called procedure

\( \left( \langle a, b \rangle \text{ holds on a path to node iff} \langle a, b \rangle \text{ holds at exit(node) } \right) \)

Add \( \ll \ll \text{exit(node)} \gg, \langle a, b \rangle, \ll \text{node}, \text{AA} \gg, \langle a, b \rangle \gg \gg \rightarrow \mathcal{E}' \)

For every \( \langle c, d \rangle \in \text{ASSUMED} \ (c \text{ and } d \neq \text{“non_visible”}) \):

if \( \text{back-binding}'(\ll \langle c, d \rangle) \neq \text{false} \) then add

\[
\ll \ll \text{exit(node)} \gg, \langle c, d \rangle, \ll \text{node}, \text{AA} \gg, \langle a, b \rangle \gg \gg \rightarrow \mathcal{E}' \)

\[
\text{holds}(\ll \text{node}, \text{AA} \gg, \langle a, b \rangle) = \text{holds}(\ll \text{exit(node)}, \langle c, d \rangle) \gg \land \left( \text{holds}(\ll \text{call(node)}, \text{AA} \gg, \langle c, d \rangle) \gg \rightarrow \mathcal{E}' \right)
\]
\[ \forall_{c,d} \in \text{ASSUMED} \left( \text{holds}([\text{exit}(\text{node}), \langle c, d \rangle), \langle a, b \rangle]) \land \text{holds}([\text{call}(\text{node}, \text{AA}), \text{back-bind} \_\text{call} \_\text{of}(c, d)]]) \right) \]

(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 \( \text{held immediately before node} \))

For all \( \ll m, \text{node} \gg \in \mathcal{E} \), add \( \ll \ll [m, \text{AA}], \langle a, b \rangle \], \([\text{node}, \text{AA}], \langle a, b \rangle \] \gg \gg \in \mathcal{E}' \)

\[ \text{holds}([\text{node}, \text{AA}], \langle a, b \rangle]) = \bigvee_{\ll m, \text{node} \gg \in \mathcal{E}} \left( \text{holds}([m, \text{AA}], \langle a, b \rangle) \right) \]

ii. node is the pointer assignment “\( p = q \)”

A. if \( a = b = *p \)

\[ (\langle *p, *p \rangle \) holds on a path to node iff \( \langle *q, *q \rangle \) \]

held immediately before node

For all \( \ll m, \text{node} \gg \in \mathcal{E} \) add \( \langle m, \text{AA} \rangle, \langle *q, *q \rangle \], \([\text{node}, \text{AA}], \langle *p, *p \rangle \] \gg \gg \in \mathcal{E}' \)

\[ \text{holds}([\text{node}, \text{AA}], \langle *p, *p \rangle]) = \bigvee_{\ll m, \text{node} \gg \in \mathcal{E}} \left( \text{holds}([m, \text{AA}], \langle *q, *q \rangle) \right) \]

B. if \( a = *p \) [\( a \neq b \)]

\[ (\langle *p, b \rangle \) holds on a path to node iff \( \langle *q, b \rangle \) \]

held immediately before node

For all \( \ll m, \text{node} \gg \in \mathcal{E} \) add \( \langle m, \text{AA} \rangle, \langle *q, b \rangle \], \([\text{node}, \text{AA}], \langle *p, b \rangle \] \gg \gg \in \mathcal{E}' \)

\[ \text{holds}([\text{node}, \text{AA}], \langle *p, b \rangle]) = \bigvee_{\ll m, \text{node} \gg \in \mathcal{E}} \left( \text{holds}([m, \text{AA}], \langle *q, b \rangle) \right) \]

C. if (\( a \neq *p \)) and (\( b \neq *p \))

\( (\langle a, b \rangle \) holds on a path to node iff \( a, b \) held immediately before node) \]

For all \( \ll m, \text{node} \gg \in \mathcal{E} \) add \( \ll [m, \text{AA}], \langle a, b \rangle \], \([\text{node}, \text{AA}], \langle a, b \rangle \] \gg \gg \in \mathcal{E}' \)

\[ \text{holds}([\text{node}, \text{AA}], \langle a, b \rangle]) = \bigvee_{\ll m, \text{node} \gg \in \mathcal{E}} \left( \text{holds}([m, \text{AA}], \langle a, b \rangle) \right) \]

iii. node is the pointer assignment “\( p = &v \)”

A. if \( a = b = *p \)

\( (\langle *p, *p \rangle \) holds on all paths to node) \]

Add \( \ll \rho', [\text{node}, \text{AA}], \langle *p, *p \rangle \) \gg \gg \in \mathcal{E}' \)

\[ \text{holds}([\text{node}, \text{AA}], \langle *p, *p \rangle]) = \text{true} \]
B. if \( a = \ast p \ [a \neq b] \)
\[
\begin{align*}
\langle \ast p, b \rangle & \text{ holds on a path to node iff } \langle v, b \rangle \\
& \text{ held immediately before node}
\end{align*}
\]
For all \( \ll m, \text{node} \rr \in \mathcal{E} \) add \( \ll [(m, AA), \langle v, b \rangle], [(\text{node}, AA), \langle \ast p, b \rangle] \rr \) to \( \mathcal{E}' \)
\[
\text{holds}([(\text{node}, AA), \langle \ast p, b \rangle]) = \bigvee_{\ll m, \text{node} \rr \in \mathcal{E}} \left( \text{holds}([(m, AA), \langle v, b \rangle]) \right)
\]

C. if \( (a \neq \ast p) \) and \( (b \neq \ast p) \)
\[
\langle a, b \rangle \text{ holds on a path to node iff } \langle a, b \rangle \text{ held immediately before node}
\]
For all \( \ll m, \text{node} \rr \in \mathcal{E} \) add \( \ll [(m, AA), \langle a, b \rangle], [(\text{node}, AA), \langle a, b \rangle] \rr \) to \( \mathcal{E}' \)
\[
\text{holds}([(\text{node}, AA), \langle a, b \rangle]) = \bigvee_{\ll m, \text{node} \rr \in \mathcal{E}} \left( \text{holds}([(m, AA), \langle a, b \rangle]) \right)
\]
iv. node is the pointer assignment \( "p = \text{malloc()}" \) or \( "p = \text{NULL}" \)

A. if \( a = b = \ast p \)
\[
\text{Add } \ll \rho', [(\text{node}, AA), \langle \ast p, \ast p \rangle] \rr \text{ to } \mathcal{E}'
\]
\[
\text{holds}([(\text{node}, AA), \langle \ast p, \ast p \rangle]) = \begin{cases} 
\text{true} & \text{node is } \"p = \text{malloc()}" \\
\text{false} & \text{node is } \"p = \text{NULL}" 
\end{cases}
\]
B. if \( a = \ast p \ [a \neq b] \)
\[
\ast p \text{ can never be aliased to anything on any path to node}
\]
Add \( \ll \rho', [(\text{node}, AA), \langle \ast p, b \rangle] \rr \) to \( \mathcal{E}' \)
\[
\text{holds}([(\text{node}, AA), \langle \ast p, b \rangle]) = \text{false}
\]
C. if \( (a \neq \ast p) \) and \( (b \neq \ast p) \)
\[
\langle a, b \rangle \text{ holds on a path to node iff } \langle a, b \rangle \text{ held immediately before node}
\]
For all \( \ll m, \text{node} \rr \in \mathcal{E} \) add \( \ll [(m, AA), \langle a, b \rangle], [(\text{node}, AA), \langle a, b \rangle] \rr \) to \( \mathcal{E}' \)
\[
\text{holds}([(\text{node}, AA), \langle a, b \rangle]) = \bigvee_{\ll m, \text{node} \rr \in \mathcal{E}} \left( \text{holds}([(m, AA), \langle a, b \rangle]) \right)
\]

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

Let \( O' = \{ \ast p \mid p \text{ is a pointer in the program} \} \cup \{ v \mid v \text{ is a non-pointer variable in the program} \} \).
\( O' \) is the set of all object names in the program which may have aliases.

Let \( \text{POSSIBLE-ALIASES}' = (O' \times O') \).
"POSSIBLE-ALIASES'" is the set of all possible aliases.

Define the lattice [Hec77] \( \mathcal{L}' = \) \( \text{powerset}(\text{POSSIBLE-ALIASES'}) \supseteq (\sqsubseteq, \cup, \text{POSSIBLE-ALIASES'} (\bot, \emptyset (\top)) \)

Let \( \text{bind}_n(AA), n \) a call node in the ICFG and \( AA \in \text{POSSIBLE-ALIASES'}, \)
be defined as in Appendix C.2.

Let \( \text{entry}(n), n \) a node in the ICFG, be the entry node of the procedure containing \( n. \)

For every node \( n \) in the ICFG define \( \text{may-alias}(n) \) as:

1. \( \text{may-alias}(\rho) = \emptyset \)

2. if \( n \) is an entry node then \( \text{may-alias}(n) = \bigcup_{m, n \in \varepsilon} (\text{bind}_m(\text{may-alias}(m))) \)

3. otherwise, \( \text{may-alias}(n) = \)

\[
\begin{align*}
&\{ \langle a, b \rangle \mid \text{neither } a \text{ nor } b \text{ is non-visible and} \\
&[\text{holds}([n, \emptyset, \langle a, b \rangle]) = \text{true}] \lor \\
&[\exists AA \in \text{may-alias(entry}(n))] \text{ holds}([n, AA, \langle a, b \rangle]) = \text{true} \}
\end{align*}
\]

Algorithm:

1. For all \( n \) in the ICFG, all \( \langle a, b \rangle \in \text{POSSIBLE-ALIASES}, \)
all \( AA \in \text{ASSUMED}, \) initialize \( \text{holds}([n, AA, \langle a, b \rangle]) \) to \( \text{false}. \)

2. Calculate the fixed point of \( \text{holds}. \)

3. For all \( n \) in the ICFG, initialize \( \text{may-alias}(n) \) to \( \emptyset. \)

4. Calculate the fixed point of \( \text{may-alias}. \)
### 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 \( P = n_1n_2\ldots n_i \) (where \( n_1 \) is the entry node for the procedure containing \( n_i \) and the number of calls on the path \( n_1n_2\ldots n_{i-1} \) equals the number of returns.),

- and for all possible alias pairs \( \langle a, b \rangle \);

If

all alias pairs in the set \( A = \{A_1, A_2, \ldots, A_m\} \) 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 \( A \). In the first case, the alias that holds must be in \( A \) 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 \( n_1 \ldots n_{i-1} \) 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 \( n_1 \ldots n_{i-1} \) 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 \( n_1 \ldots n_j \) 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 \( n_1 \ldots n_j \) 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 \( n_1 \ldots n_{j+1} \) (an entry node) assuming \( \mathcal{A} \) at \( n_1 \). For \( \langle a, b \rangle \) to hold on \( P \) assuming \( \mathcal{A} \) at \( n_1 \), \( \langle a, \text{non_visible} \rangle \) (where \text{non_visible} is \( b \)) must hold on \( n_{j+1} \ldots n_{i-1} \) assuming \( \mathcal{A}' \) at \( n_{j+1} \). \( n_{j+1} \ldots n_{i-1} \) is shorter than \( P \) and meets the requirements of the Lemma thus by induction hypothesis, there is a \( A'_{k'} \in \mathcal{A}' \) such that assuming only \( A'_{k'} \) at \( n_{j+1} \) is sufficient. By Lemma C.1.1 (p. 160), \text{back_bind}'(A'_{k'}, b) \) must hold on \( n_1 \ldots n_j \) 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 \( n_1 \ldots n_j \) 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 \( n_1 \ldots n_{j+1} \) (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 non_visible is involved.
$n_{j+1} \ldots n_{i-1}$ assuming $A'$ at $n_{j+1}$. $n_{j+1} \ldots n_{i-1}$ 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 $A'_{k'} \in A'$ 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 $n_1 \ldots n_j$ assuming $A$ at $n_1$ for $\langle a, b \rangle$ to hold on $P$ under the same assumptions. The Lemma follows by induction because $n_1 \ldots n_j$ 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 $A$ at $n_1$ only if $\langle a, b \rangle$ holds on $n_1 \ldots n_{i-1}$ assuming $A$ at $n_1$. By induction hypothesis either no assumption or only $A_k \in A$ forces $\langle a, b \rangle$ to hold on $n_1 \ldots n_{i-1}$ 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 $A$ at $n_1$ only if $\langle *q, *q \rangle$ holds on $n_1 \ldots n_{i-1}$ assuming $A$ at $n_1$. By induction hypothesis either no assumption or only $A_k \in A$ forces $\langle *q, *q \rangle$ to hold on $n_1 \ldots n_{i-1}$ and thus the same is true for $P$.

ii. $a = *p \ [a \neq b]$

$\langle *p, b \rangle$ holds on $P$ assuming $A$ at $n_1$ only if $\langle *q, b \rangle$ holds on $n_1 \ldots n_{i-1}$ assuming $A$ at $n_1$. By induction hypothesis either no assumption or only $A_k \in A$ forces $\langle *q, b \rangle$ to hold on $n_1 \ldots n_{i-1}$ and thus the same is true for $P$.

iii. otherwise
\( \langle a, b \rangle \) holds on \( P \) assuming \( A \) at \( n_1 \) only if \( \langle a, b \rangle \) holds on \( n_1 \ldots n_i \) assuming \( A \) at \( n_1 \). By induction hypothesis either no assumption or only \( A_k \in A \) forces \( \langle a, b \rangle \) to hold on \( n_1 \ldots n_i \) 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 \( A \) at \( n_1 \) only if \( \langle v, b \rangle \) holds on \( n_1 \ldots n_i \) assuming \( A \) at \( n_1 \). By induction hypothesis either no assumption or only \( A_k \in A \) forces \( \langle v, b \rangle \) to hold on \( n_1 \ldots n_i \) and thus the same is true for \( P \).

iii. otherwise

\( \langle a, b \rangle \) holds on \( P \) assuming \( A \) at \( n_1 \) only if \( \langle a, b \rangle \) holds on \( n_1 \ldots n_i \) assuming \( A \) at \( n_1 \). By induction hypothesis either no assumption or only \( A_k \in A \) forces \( \langle a, b \rangle \) to hold on \( n_1 \ldots n_i \) 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 \( A \) at \( n_1 \) only if \( \langle a, b \rangle \) holds on \( n_1 \ldots n_i \) assuming \( A \) at \( n_1 \). By induction hypothesis either no assumption or only
\( A_k \in \mathcal{A} \) forces \( \langle a, b \rangle \) to hold on \( n_1 \ldots n_{i-1} \) and thus the same is true for \( P \).

\( \square \)

### C.3.5 Proof of Theorem 4.2.3

Let \( \text{CFG} = (\mathcal{N}, \mathcal{E}, \rho) \).

Let \( \text{PAG} = (\mathcal{N}', \mathcal{E}', \rho') \) and \( \text{holds} : \mathcal{N}' \rightarrow \{ \text{true}, \text{false} \} \) be as specified in Appendix C.3.2.

\[ \text{Lemma C.3.1} \quad \text{The Pointer Alias Graph (PAG) is polynomial in the size of the CFG.} \]

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 \( | \text{variables} |^2 \).

Since a program can use at most \( O(|\mathcal{N}|) \) variables, the number of possible aliases is \( O(|\mathcal{N}|^2) \).

\[ |\mathcal{N}'| = |\mathcal{N}| \ast |\text{ASSUMED}| \ast |\text{POSSIBLE-ALIASES}| + 1 \]

\[ |\mathcal{E}'| \leq |\mathcal{E}| \ast |\text{ASSUMED}|^2 \ast |\text{POSSIBLE-ALIASES}| + |\mathcal{N}'| \]

\( \square \)

\[ \text{Lemma C.3.2} \quad \text{The PAG can be built from the CFG in polynomial time.} \]

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 \( O(\max(\mathcal{E}, |\text{ASSUMED}|)) \) work.

\( \square \)

\[ \text{Lemma C.3.3} \quad \text{Fixed point calculation on the PAG is polynomial time.} \]
Follows from Lemma C.3.1 (p. 175) and the observation that for any node \( n' \in N' \), 
\( \text{holds}(n') \) can change its value at most once.
\( \square \)

**Lemma C.3.4** \( \text{holds}([\text{node}, \mathcal{AA}], \mathcal{P}A) = \text{true} \) iff

\[ \exists \text{ a realizable path } P = [\text{entry(node)}]n_1n_2...n_i[\text{node}] \text{ in the ICFG, where} \]
\[ n_1n_2...n_i \text{ contains the same number of calls as returns, such that for every} \]
\[ \text{realizable path } P' = pm_1...m_i[\text{entry(node)}], \mathcal{AA} \text{ holding on } P' \text{ means } \mathcal{P}A \]
\[ \text{must hold on } P' \bullet P \text{ (i.e., } pm_1...m_i[\text{entry(node)}]n_1n_2...n_i[\text{node}]). \]

**Proof of Lemma C.3.4 (only if)** By induction on number of iterations of the fixed point algorithm:

Assume \( \text{holds}([\text{node}, \mathcal{AA}], \langle a, b \rangle) \) became true in the \( i \)th iteration.

**basis (} i = 0): after initialization.**

For all \( n \in N' \), \( \text{holds}(n) \) 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) \text{ node} is an entry node

By definition \( \text{holds}([\text{node}, \mathcal{AA}], \langle a, b \rangle) \) is true iff \( \mathcal{A}\mathcal{A} = \langle a, b \rangle \). In this case the “only if” direction of the Lemma is clearly satisfied.

\(^{13}\)We consider the assumed alias \( \emptyset \) to hold on all paths
(b) *node* is an exit node

\[ \text{holds}([[\text{node}, \text{AA}], \langle a, b \rangle]) = \bigvee_{\langle m, \text{node} \rangle \in \mathcal{E}} \left( \text{holds}([[m, \text{AA}], \langle a, b \rangle]) \right) \]

Thus for some *m* such that \( \langle m, \text{node} \rangle \in \mathcal{E} \), \( \text{holds}([[m, \text{AA}], \langle a, b \rangle]) \) was set true at an earlier iteration. By induction hypothesis, there is a path \( P(\text{entry}(m) ... m) \) that satisfies the Lemma for \( \text{holds}([[m, \text{AA}], \langle a, b \rangle]) \). Since \( \text{entry}(m) = \text{entry}(\text{node}) \), \( P \circ [\text{node}] \) must satisfy the Lemma for \( \text{holds}([[\text{node}, \text{AA}], \langle a, b \rangle]) \).

(c) *node* is a 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

\[ \text{holds}([[\text{node}, \text{AA}], \langle a, b \rangle]) = \text{holds}([[\text{call}(\text{node}), \text{AA}], \langle a, b \rangle]) \]

\( \text{holds}([[\text{call}(\text{node}), \text{AA}], \langle a, b \rangle]) \) was set true at an earlier iteration. By induction hypothesis, there is a path \( P(\text{entry}(\text{call}(\text{node})) ... [\text{call}(\text{node})]) \) that satisfies the Lemma for \( \text{holds}([[\text{call}(\text{node}), \text{AA}], \langle a, b \rangle]) \). Let \( P^* \) be any realizable path from \( \text{entry}(\text{exit}(\text{node})) \) to \( \text{exit}(\text{node}) \). Since \( \text{entry}(\text{call}(\text{node})) = \text{entry}(\text{node}) \), \( P \circ P^* \circ [\text{node}] \) must satisfy the Lemma for \( \text{holds}([[\text{node}, \text{AA}], \langle a, b \rangle]) \).

ii. a is visible in the called procedure, but b is not

\[ \text{holds}([[\text{node}, \text{AA}], \langle a, b \rangle]) = \bigvee_{\langle c, \text{non-visible} \rangle \in \text{ASSUMED}} \left( \text{holds}([[\text{exit}(\text{node}), \langle c, \text{non-visible} \rangle], \langle a, \text{non-visible} \rangle]) \wedge \text{holds}([[\text{call}(\text{node}), \text{AA}], \text{back-bind}_{\text{call}(\text{node})}(\langle c, \text{non-visible} \rangle, b)]) \right) \]

Since \( \text{holds}([[\text{node}, \text{AA}], \langle a, b \rangle]) \) is true, for some \( \langle c, \text{non-visible} \rangle \) (1)

\[ \text{holds}([[\text{exit}(\text{node}), \langle c, \text{non-visible} \rangle], \langle a, \text{non-visible} \rangle]) \] and (2)

\[ \text{holds}([[\text{call}(\text{node}), \text{AA}], \text{back-bind}_{\text{call}(\text{node})}(\langle c, \text{non-visible} \rangle, b)]) \] must have been made true at a earlier iterations. By induction, (1) implies that there is a path \( P_1 ([\text{entry}(\text{exit}(\text{node}))] ... [\text{exit}(\text{node})]) \) such that when \( \langle c, \text{non-visible} \rangle \) holds on a path \( P'_1 \) to \( \text{entry}(\text{exit}(\text{node})) \), \( \langle a, \text{non-visible} \rangle \) holds on \( P'_1 \cdot P_1 \). Also by induction, (2) implies there is a path \( P_2 \)
([entry(node)] ... [call(node)]) such that when \( \mathcal{A} \mathcal{A} \) holds on a path \( P_2' \) to \( \text{entry(node)} \), \( \text{back-bind}_{\text{call(node)}}(\langle c, \text{non_visible}\rangle, b) \) holds on \( P_2' \cdot P_2 \).

This along with Lemma C.1.1 (p. 160) imply that \( P_2 \circ P_1 \circ [\text{node}] \) is a path that satisfies the Lemma for \( \text{holds}([\text{node}, \mathcal{A} \mathcal{A}], \langle a, b \rangle) \).

iii. \( a \) and \( b \) are both visible in the called procedure

\[
\text{holds}([\langle \text{node}, \mathcal{A} \mathcal{A} \rangle, \langle a, b \rangle]) = \text{holds}([\langle \text{exit(node), \emptyset} \rangle, \langle a, b \rangle]) \lor \\
\bigvee_{c,d} \text{ASSUMED} \left( \text{holds}([\langle \text{exit(node), \langle c, d \rangle} \rangle, \langle a, b \rangle]) \land \\
\text{holds}([\langle \text{call(node), \mathcal{A} \mathcal{A} \rangle, \text{back-bind}_{\text{call(node)}}(\langle c, d \rangle)]) \right)
\]

Since \( \text{holds}([\langle \text{node}, \mathcal{A} \mathcal{A} \rangle, \langle a, b \rangle]) \) is true either \( \text{holds}([\langle \text{exit(node), \emptyset} \rangle, \langle a, b \rangle]) \) was set true in a previous iteration or for some \( \langle c, d \rangle \) (1)

\( \text{holds}([\langle \text{exit(node), \langle c, d \rangle} \rangle, \langle a, b \rangle]) \) and (2)

\( \text{holds}([\langle \text{call(node), \mathcal{A} \mathcal{A} \rangle, \text{back-bind}_{\text{call(node)}}(\langle c, d \rangle)]) \) must have been made true at some earlier iterations. In the former case, by induction there is a path \( P_0 \) ([entry(exit(node))] ... [exit(node)]) such that for any that path \( P_0' \) to [entry(exit(node))], \( \langle a, b \rangle \) holds on \( P_0' \cdot P_0 \). 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 \( n_1...n_i \), \( P^* \circ P_0 \circ [\text{node}] \) satisfies the Lemma for \( \text{holds}([\langle \text{node}, \mathcal{A} \mathcal{A} \rangle, \langle a, b \rangle]) \). 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 \( \text{entry(exit(node))} ), \( \langle a, b \rangle \) holds on \( P_1' \cdot P_1 \). Also by induction, (2) implies there is a path \( P_2 \) ([entry(node)] ... [call(node)]) such that when \( \mathcal{A} \mathcal{A} \) holds on a path \( P_2 \) to \( \text{entry(node)} \), \( \text{back-bind}_{\text{call(node)}}(\langle c, d \rangle) \) holds on \( P_2' \cdot P_2 \). This along with Lemma C.1.1 (p. 160) imply that \( P_2 \circ P_1 \circ [\text{node}] \) is a path that satisfies the Lemma for \( \text{holds}([\langle \text{node}, \mathcal{A} \mathcal{A} \rangle, \langle a, b \rangle]) \).

(e) otherwise (\text{node} is a statement node)

i. \text{node} is not an assignment to a pointer

\[
\text{holds}([\langle \text{node}, \mathcal{A} \mathcal{A} \rangle, \langle a, b \rangle]) = \bigvee_{m, \text{node}} \in \mathcal{E} \left( \text{holds}([\langle m, \mathcal{A} \mathcal{A} \rangle, \langle a, b \rangle]) \right)
\]

Thus for some \( m \) such that \( \ll m, \text{node} \gg \in \mathcal{E} \), \( \text{holds}([\langle m, \mathcal{A} \mathcal{A} \rangle, \langle a, b \rangle]) \) was

\[8^8 \text{entry(node)} = \text{entry(call(node))}.\]
set true at an earlier iteration. By induction hypothesis, there is a path \( P \) (\( entry(m) \ldots m \)) that satisfies the Lemma for \( holds([(m, AA), (a, b)]) \). Since \( entry(m) = entry(node) \), \( P \circ [node] \) must satisfy the Lemma for \( holds([(node, AA), (a, b)]) \).

ii. \( node \) is the pointer assignment “\( p = q \)”

A. if \( a = b = *p \)

\[ holds([(node, AA), (*p, *p)]) = \bigvee_{m, node} E (holds([(m, AA), (*q, *q)])) \]

Thus for some \( m \) such that \( \ll m, node \gg \in E \), \( holds([(m, AA), (*q, *q)]) \) was set true at an earlier iteration. By induction hypothesis, there is a path \( P \) (\( entry(m) \ldots m \)) that satisfies the Lemma for \( holds([(m, AA), (*q, *q)]) \). Since \( entry(m) = entry(node) \), \( P \circ [node] \) must satisfy the Lemma for \( holds([(node, AA), (*p, *p)]) \).

B. if \( a = *p \ [a \neq b] \)

\[ holds([(node, AA), (*p, *b)]) = \bigvee_{m, node} E (holds([(m, AA), (*q, *b)]) \]

Thus for some \( m \) such that \( \ll m, node \gg \in E \), \( holds([(m, AA), (*q, *b)]) \) was set true at an earlier iteration. By induction hypothesis, there is a path \( P \) (\( entry(m) \ldots m \)) that satisfies the Lemma for \( holds([(m, AA), (*q, *b)]) \). Since \( entry(m) = entry(node) \), \( P \circ [node] \) must satisfy the Lemma for \( holds([(node, AA), (*p, *b)]) \).

C. if \( (a \neq *p) \) and \( (b \neq *p) \)

\[ holds([(node, AA), (a, b)]) = \bigvee_{m, node} E (holds([(m, AA), (a, b)]) \]

Thus for some \( m \) such that \( \ll m, node \gg \in E \), \( holds([(m, AA), (a, b)]) \) was set true at an earlier iteration. By induction hypothesis, there is a path \( P \) (\( entry(m) \ldots m \)) that satisfies the Lemma for \( holds([(m, AA), (a, b)]) \). Since \( entry(m) = entry(node) \), \( P \circ [node] \) must satisfy the Lemma for \( holds([(node, AA), (a, b)]) \).

iii. \( node \) is the pointer assignment “\( p = &v \)”

A. if \( a = b = *p \)

\( (*p, *p) \) holds on all paths to \( node \). Thus any path \( P' \) (\( [entry(node)] \)
\( n_1 \ldots n_i \ [node] \)) that is realizable and has an equal number of calls and
returns on \( n_1 \ldots n_i \), satisfies the Lemma for 
\[ \text{holds}([\text{node, AA}, (sp, sp)]). \]

B. if \( a = sp \ [a \neq b] \)
\[ \text{holds}([\text{node, AA}, (sp, b)]) = \bigvee_{m, \text{node} \in E} \text{holds}([m, \text{AA}, (v, b)]) \]
Thus for some \( m \) such that \( \preceq m, \text{node} \in E \), \( \text{holds}([m, \text{AA}, (v, b)]) \)
was set true at an earlier iteration. By induction hypothesis, there
is a path \( P (\text{entry}(m) \ldots m) \) that satisfies the Lemma for
\[ \text{holds}([m, \text{AA}, (v, b)]). \] Since \( \text{entry}(m) = \text{entry}(\text{node}) \), \( P \circ [\text{node} \]
must satisfy the Lemma for \( \text{holds}([\text{node, AA}, (sp, b)]). \)

C. if \((a \neq sp) \) and \((b \neq sp) \)
\[ \text{holds}([\text{node, AA}, (a, b)]) = \bigvee_{m, \text{node} \in E} \text{holds}([m, \text{AA}, (a, b)]) \]
Thus for some \( m \) such that \( \preceq m, \text{node} \in E \), \( \text{holds}([m, \text{AA}, (a, b)]) \)
was set true at an earlier iteration. By induction hypothesis, there
is a path \( P (\text{entry}(m) \ldots m) \) that satisfies the Lemma for
\[ \text{holds}([m, \text{AA}, (a, b)]). \] Since \( \text{entry}(m) = \text{entry}(\text{node}) \), \( P \circ [\text{node} \]
must satisfy the Lemma for \( \text{holds}([\text{node, AA}, (a, b)]). \)

iv. \( \text{node} \) is the pointer assignment “\( p = \text{malloc()} \)” or “\( p = \text{NULL} \)”

A. if \( a = b = sp \)
By definition, \( \text{node} \) must be “\( p = \text{malloc()} \)”.
In which case, \((sp, sp)\) holds on all paths to \( \text{node} \).
Thus any path \( P' ([\text{entry}(\text{node}) \ n_1 \ldots n_i \ 
[\text{node}]) \) that is realizable and has an equal number of calls and returns
on \( n_1 \ldots n_i \), satisfies the Lemma for \( \text{holds}([\text{node, AA}, (sp, sp)]). \)

B. if \( a = sp \ [a \neq b] \)
By definition, \( \text{holds}([\text{node, AA}, (sp, b)]) \) can never be true.

C. if \((a \neq sp) \) and \((b \neq sp) \)
\[ \text{holds}([\text{node, AA}, (a, b)]) = \bigvee_{m, \text{node} \in E} \text{holds}([m, \text{AA}, (a, b)]) \]
Thus for some \( m \) such that \( \preceq m, \text{node} \in E \), \( \text{holds}([m, \text{AA}, (a, b)]) \)
was set true at an earlier iteration. By induction hypothesis, there
is a path \( P (\text{entry}(m) \ldots m) \) that satisfies the Lemma for
\[ \text{holds}([m, \text{AA}, (a, b)]). \] Since \( \text{entry}(m) = \text{entry}(\text{node}) \), \( P \circ [\text{node} \]
must satisfy the Lemma for \( \text{holds}([\text{node, AA}, (a, b)]). \)
Proof of Lemma C.3.4 (if) by induction on length of path $P$:

**basis:** $|P| = 1$

$P = entry\_node$. 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($[[\text{entry}\_\text{node}, \mathcal{AA}], \mathcal{PA}]] = true$ 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($[[\text{entry}\_\text{node}, \mathcal{AA}], \mathcal{PA}]]$ to be true.

**induction hypothesis:**

Assume the “if” direction of Lemma C.3.4 holds for paths of length $< |P|$.

**induction step:**

Let $P = [entry\_node]\mid n_1 ... n_i [node]$ 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 $P' = pm_1 ... m_i [entry\_node]$ 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

   $|P|$ must be 1 and the basis applies.

   (b) $node$ is an exit node

   Since $\langle a, b \rangle$ holds on all paths $P' \bullet [entry\_node]\mid n_1 ... n_i [node]$ it must also hold on $P' \bullet [entry\_node]\mid n_1 ... n_i$. Thus by the induction hypothesis,

   holds($[[n_i, \mathcal{AA}], \langle a, b \rangle]$) is true. By definition,

   holds($[[node, \mathcal{AA}], \langle a, b \rangle]$) $= \bigvee_{\ll m, \text{node} \gg \in \mathcal{E}} \left( \text{holds}([[m, \mathcal{AA}], \langle a, b \rangle]) \right)$.

   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) \textit{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, \textit{node}. Since \(\langle a, b \rangle\) holds on all paths \(P' \bullet [\text{entry(node)}]n_1\ldots n_i[\text{node}]\) it also holds on \(P' \bullet [\text{entry(node)}]n_1\ldots n_j\).

Thus by the induction hypothesis, \(\text{holds}([[n_j, \mathcal{AA}], \langle a, b \rangle])\) is true. By definition, \(\text{holds}([[\text{node, AA}], \langle a, b \rangle]) = \text{holds}([[\text{call(node), AA}], \langle a, b \rangle])\) where \(\text{call(node)} = n_j\). Thus \(\text{holds}([[\text{node, AA}], \langle a, b \rangle])\) 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 \textit{node}, \(a\) must have been aliased to \textit{non_visible} on the same path up to (and including) the exit of the called procedure and \textit{non_visible} is \(b\). Let \(n_j\) be the call node that corresponds to the return, \textit{node}. Given that \(\mathcal{AA}\) holds on \(P'\) there is a set, \(\mathcal{A}\) of aliases that hold on \(P' \bullet [\text{entry(node)}]n_1\ldots n_jn_{j+1}\). Now \(n_{j+1}\) is the entry node of the called procedure and we know that \(\langle a, \text{non_visible} \rangle\) holds on \(P' \bullet [\text{entry(node)}]n_1\ldots n_jn_{j+1}\ldots n_i\) thus by Lemma 4.2.1 (p. 34) there is a unique \(A_k \in \mathcal{A}^{\text{true}}\) just assuming \(A_k\) at \(n_{j+1}\) would have been sufficient.

Since \(n_{j+1}\ldots n_i\) is a shorter path than \(P\) that meets the necessary restrictions, by the induction hypothesis \(\text{holds}([[n_i, A_i], \langle a, \text{non_visible} \rangle])\) is true. Now since \(A_k\) holds on \(P' \bullet [\text{entry(node)}]n_1\ldots n_jn_{j+1}\), by Lemma C.1.1 (p. 160) \(\text{back-bind}_y(A_k, b)\) holds on \(P' \bullet [\text{entry(node)}]n_1 \ldots n_j\). By induction \(\text{holds}([[n_j, \mathcal{AA}], \text{back-bind}_y(A_k, b)])\) is true. By definition,

\[
\text{holds}([[\text{node, AA}], \langle a, b \rangle]) = \bigvee_{\langle c, \text{non_visible} \rangle \in \text{ASSUMED}} \left( \text{holds}([[\text{exit(node)}, \langle c, \text{non_visible} \rangle], \langle a, \text{non_visible} \rangle]) \wedge \text{holds}([[\text{call(node), AA}, \text{back-bind}_y(c, \text{call(node)}, \langle c, \text{non_visible} \rangle, b)]) \right)
\]

\(n_j = \text{call(node)}\) and \(n_i = \text{exit(node)}\). Thus \(\text{holds}([[\text{node, AA}], \langle a, b \rangle])\) is true.

iii. \(a\) and \(b\) are visible in the called procedure

\footnote{\(\emptyset\) \(\emptyset\) is not possible since \textit{non_visible} is involved and \(A_k\) must be \(\langle c, \text{non_visible} \rangle\) for some \(c\).}
By definition,
\[
\text{holds}([\text{node} . \text{AA}, \langle a, b \rangle]) = \text{holds}([\text{exit} (\text{node}), \emptyset, \langle a, b \rangle]) \lor \\
\bigvee_{< c, d > \in \text{ASSUMED}} \left( \text{holds}([\text{exit} (\text{node}), \langle c, d \rangle], \langle a, b \rangle) \land \text{holds}([\text{call} (\text{node}), \text{AA}], \text{back-bind}_{\text{exit} (\text{node})} (\langle c, d \rangle)) \right)
\]

For \( a \) to be aliased to \( b \) on a path to \( \text{node} \), \( a \) 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, \( \text{node} \).

Given that \( \text{AA} \) holds on \( P' \) there is a set, \( \text{A} \) of aliases that hold on \( P' \bullet [\text{entry} (\text{node})]n_1 \ldots n_jn_{j+1} \). Now \( n_{j+1} \) is the entry node of the called procedure and we know that \( \langle a, b \rangle \) holds on \( P' \bullet [\text{entry} (\text{node})]n_1 \ldots n_jn_{j+1} \ldots n_i \) thus by Lemma 4.2.1 (p. 34) either no assumptions are necessary at \( n_{j+1} \) or there is a unique \( A_k \in \text{A} \) such that just assuming \( A_k \) at \( n_{j+1} \) is sufficient. In the first case since \( n_{j+1} \ldots n_i \) is a shorter path than \( P \) that meets the necessary restrictions, by the induction hypothesis \text{holds}([\langle n, \emptyset \rangle, \langle a, b \rangle]) \) is true and thus \text{holds}([\langle \text{node} . \text{AA}, \emptyset \rangle, \langle a, b \rangle]) \) must also be true. In the latter case, by the induction hypothesis (again on \( n_{j+1} \ldots n_i \)) \text{holds}([\langle n, A_k \rangle, \langle a, b \rangle]) \) is true. Now since \( A_k \) holds on \( P' \bullet [\text{entry} (\text{node})]n_1 \ldots n_jn_{j+1} \), by Lemma C.1.1 (p. 160) \text{back-bind}_{\text{exit} (\text{node})} (A_k) \) holds on \( P' \bullet [\text{entry} (\text{node})]n_1 \ldots n_j \). By induction \text{holds}([\langle n_j . \text{AA}, \text{back-bind}_{\text{exit} (\text{node})} (A_k) \rangle]) \) is true.

Since \( n_j = \text{call} (\text{node}) \) and \( n_i = \text{exit} (\text{node}) \), \text{holds}([\langle \text{node} . \text{AA}, \emptyset \rangle, \langle a, b \rangle]) \) is true by definition.

(e) otherwise (\text{node} is a statement node)

i. \text{node} is not an assignment to a pointer

Since \( \langle a, b \rangle \) holds on all paths \( P' \bullet [\text{entry} (\text{node})]n_1 \ldots n_i \) it must also hold on \( P' \bullet [\text{entry} (\text{node})]n_1 \ldots n_i \). Thus by the induction hypothesis,
\[
\text{holds}([\langle n_i, \text{AA}, \emptyset \rangle, \langle a, b \rangle]) \) is true. By definition,
\[
\text{holds}([\langle \text{node} . \text{AA}, \emptyset \rangle, \langle a, b \rangle]) = \bigvee_{m . \text{node} \triangleright} \epsilon E \left( \text{holds}([\langle m . \text{AA}, \emptyset \rangle, \langle a, b \rangle]) \right).
\]

Thus \text{holds}([\langle \text{node} . \text{AA}, \emptyset \rangle, \langle a, b \rangle]) \) is true.

ii. \text{node} is the pointer assignment “\( p = q \)”

A. if \( a = b = *p \)
Since \( \ast p, \ast p \) holds on all paths \( P' \cdot [entry(node)] n_1 \ldots n_i [node] \),
\( \ast q, \ast q \) must hold on \( P' \cdot [entry(node)] n_1 \ldots n_i \). Thus by the induction hypothesis, \( holds([n_i, AA], \ast q, \ast q) \) is true. By definition,
\( holds([n_i, AA], \ast q, \ast q) \) is true. By definition,
\( \ast p, \ast p \)

**B. if \( a = \ast p \ [a \neq b \)**

Since \( \ast p, b \) holds on all paths \( P' \cdot [entry(node)] n_1 \ldots n_i [node] \),
\( \ast q, b \) must hold on \( P' \cdot [entry(node)] n_1 \ldots n_i \). Thus by the induction hypothesis, \( holds([n_i, AA], \ast q, b) \) is true. By definition,
\( holds([n_i, AA], \ast q, b) \) is true. By definition,
\( \ast p, b \)

**C. if \( (a \neq \ast p) \) and \( (b \neq \ast p) \)**

Since \( \langle a, b \rangle \) holds on all paths \( P' \cdot [entry(node)] n_1 \ldots n_i [node] \) it must also hold on \( P' \cdot [entry(node)] n_1 \ldots n_i \). Thus by the induction hypothesis, \( holds([n_i, AA], \langle a, b \rangle) \) is true. By definition,
\( holds([n_i, AA], \langle a, b \rangle) \) is true. By definition,
\( \langle a, b \rangle \)

**iii. node is the pointer assignment \( \text{“} p = &v \text{”} \)**

**A. if \( a = b = \ast p \)**

By definition \( holds([node, AA], \ast p, \ast p) \) is true.

**B. if \( a = \ast p \ [a \neq b \)**

Since \( \ast p, b \) holds on all paths \( P' \cdot [entry(node)] n_1 \ldots n_i [node] \),
\( \ast v, b \) must hold on \( P' \cdot [entry(node)] n_1 \ldots n_i \). Thus by the induction hypothesis, \( holds([n_i, AA], \ast v, b) \) is true. By definition,
\( holds([n_i, AA], \ast v, b) \) is true. By definition,
\( \ast v, b \)

**C. if \( (a \neq \ast p) \) and \( (b \neq \ast p) \)**

Since \( \langle a, b \rangle \) holds on all paths \( P' \cdot [entry(node)] n_1 \ldots n_i [node] \) it must also hold on \( P' \cdot [entry(node)] n_1 \ldots n_i \). Thus by the induction hypothesis, \( holds([n_i, AA], \langle a, b \rangle) \) is true. By definition,
\[ \text{holds}([\text{node}, \mathcal{AA}], \langle a, b \rangle) = \bigvee_{\langle m, \text{node}\rangle} \in \mathcal{E} \left( \text{holds}([m, \mathcal{AA}], \langle a, b \rangle) \right). \]

Thus \( \text{holds}([\text{node}, \mathcal{AA}], \langle a, b \rangle) \) is true.

iv. node is the pointer assignment “\( p = \text{malloc()} \)” or “\( p = \text{NULL} \)”

A. if \( a = b = \star p \)

\( \text{node} \) must be “\( p = \text{malloc()} \)” in which case,

\[ \text{holds}([\text{node}, \mathcal{AA}], \langle \star p, \star p \rangle) = \text{true}. \]

B. if \( a = \star p \ [a \neq b] \)

No such alias can hold on a path to node.

C. if \( (a \neq \star p) \) and \( (b \neq \star p) \)

Since \( \langle a, b \rangle \) holds on all paths \( P' \cdot [\text{entry(node)}]n_1 \ldots n_i[\text{node}] \) it must also hold on \( P' \cdot [\text{entry(node)}]n_1 \ldots n_i \). Thus by the induction hypothesis, \( \text{holds}([n_i, \mathcal{AA}], \langle a, b \rangle) \) is true. By definition,

\[ \text{holds}([\text{node}, \mathcal{AA}], \langle a, b \rangle) = \bigvee_{\langle m, \text{node}\rangle} \in \mathcal{E} \left( \text{holds}([m, \mathcal{AA}], \langle a, b \rangle) \right). \]

Thus \( \text{holds}([\text{node}, \mathcal{AA}], \langle a, b \rangle) \) is true.

\[ \square \]

**Lemma C.3.5** The fixed point of may-alias can be calculated in time polynomial in the size of the \( \text{ICFG} = (\mathcal{N}, \mathcal{E}, p) \).

Each variable in the program contributes one object name which may be involved in aliases; for each pointer \( p \), \( \star p \) 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 \(| \text{variables} |^2 \). Since a program can use at most \( O(|\mathcal{N}|) \) variables, the number of possible aliases is \( O(|\mathcal{N}|^2) \).

During the fixed point calculation for any \( \text{node} \), may-alias\((\text{node}) \) can never decrease in size. Since the maximum size of all may-alias\((\text{node}) \) is \( O(|\mathcal{N}|^2) \), any may-alias\((\text{node}) \) can change its value at most that many times. Thus we have \(|\mathcal{N}| \) nodes that can change their value at most \( O(|\mathcal{N}|^2) \) times. Therefore the fixed point
calculation for may-alias takes a polynomial amount of time \(O(|N|^3))\).
\[\square\]

---

**Lemma C.3.6**  For all realizable paths \(P = n_1n_2\ldots n_i\) (where \(n_1 = \text{entry}(n_i)\) and the number of calls on \(n_2\ldots n_{i-1}\) equals the number of returns),

If

all alias pairs in the set \(A = \{A_1, \ldots, A_m\}\) holding at \(n_1\) and that the execution of \(P\) forces all aliases in the set \(A'\) to hold at \(n_i\)

then

for any alias \(B \notin A\), assuming \(A \cup \{B\}\) at \(n_1\) and that the execution of \(P\)

forces all aliases in the set \(A''\) to hold at \(n_i\), \(A'\) must be a subset of \(A''\).

---

Additional alias assumptions on a path can only create additional aliases, they cannot destroy existing ones.
\[\square\]

---

**Lemma C.3.7**  For all \(n \in N\) and all \(\langle a, b \rangle \in O'\), \(\langle a, b \rangle \in \text{may-alias}(n)\) 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 \(n\), \(\text{may-alias}(n)\) 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 \textit{may-alias}(n) in iteration \( i \).

1. \( n \) is an entry node

By definition, \( \textit{may-alias}(n) = \bigcup_{\langle m,n \rangle \in \mathcal{E}} (\text{bind}_m(\textit{may-alias}(m))). \)

For every \( \langle m,n \rangle \in \mathcal{E} \), let \( \mathcal{A}^{i-1}_{\langle m,n \rangle} \) be the set of \([m, \langle c,d \rangle] \) such that \( \langle c,d \rangle \in \textit{may-alias}(m) \) at iteration \( i - 1 \). By the induction hypothesis, \( \mathcal{A}^{i-1}_{\langle m,n \rangle} \subseteq \text{precise solution for Interprocedural May Alias.} \) Since \( \langle a,b \rangle \) was added to \( \textit{may-alias}(n) \) in the \( i^{th} \) iteration, \( \langle a,b \rangle \in \bigcup_{\langle m,n \rangle \in \mathcal{E}} (\text{bind}_m(\{\langle c,d \rangle | [m, \langle c,d \rangle] \in \mathcal{A}^{i-1}_{\langle m,n \rangle}\})) \)

which, by Lemmas C.2.1 (p. 163) and C.3.6 (p. 186), is a subset of \( \bigcup_{\langle m,n \rangle \in \mathcal{E}} (\text{bind}_m(\textit{may-alias}(m))) \) which equals, by definition, \( \textit{may-alias}(n) \). Therefore, \([n, \langle a,b \rangle]\) must be in the precise solution for Interprocedural May Alias.

2. otherwise

By definition, \( \textit{may-alias}(n) = \)

\[
\begin{cases}
\langle a, b \rangle & \text{neither } a \text{ nor } b \text{ is non-visible and } \\
\text{[holds}([n, \emptyset], \langle a,b \rangle)] = \text{true}] \lor \\
\text{[} \exists \mathcal{A} \mathcal{A} \in \textit{may-alias(entry}(n)) \text{ holds}([n, \mathcal{A} \mathcal{A}, \langle a,b \rangle]) = \text{true} \]
\end{cases}
\]

Thus for \( \langle a,b \rangle \in \textit{may-alias}(n) \) either \( \text{holds}([n, \emptyset], \langle a,b \rangle) \) is true in which case the Lemma follows from Lemma C.3.4 (p. 176) or there is some \( \mathcal{A} \mathcal{A} \in \textit{may-alias(entry}(n)) \) at iteration \( i - 1 \) for which \( \text{holds}([n, \mathcal{A} \mathcal{A}, \langle a,b \rangle]) \) is true.

Since \( \mathcal{A} \mathcal{A} \in \textit{may-alias(entry}(n)) \) at iteration \( i - 1 \), by the induction hypothesis \( \textit{entry}(n), \mathcal{A} \mathcal{A} \) 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.

\textbf{basis:} path is simply \( \rho \)

No aliases hold before execution of the program, thus the Lemma holds vacuously.

\textbf{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, $may-alias(n) = \bigcup_{\langle i, j \rangle \in \mathcal{E}} \left( bind_{m_i}(may-alias(m)) \right)$.

   Let $\mathcal{A}'$ be the set of all aliases that hold on $\rho n_1...n_k'$. By induction, $\mathcal{A}' \subseteq may-alias(n_{k'})$. 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 $bind_{n_k}(may-alias(n_k))$ and thus by definition in $may-alias(m_0)$.

2. otherwise

   By definition, $may-alias(n) = \{ \langle a, b \rangle \mid holds([\langle a, b \rangle]) = \text{true} \} \cup \{ \exists \forall A \in may-alias(entry(n)) holds([n, A, \langle a, b \rangle]) = \text{true} \}$

   Let $\mathcal{A}'$ be the set of 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 $\mathcal{A}A$ in $\mathcal{A}'$ such that assuming only $\mathcal{A}A$ 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 $\langle a, b \rangle \in may-alias(m_k)$ by definition of $may-alias$. In the second case, $holds([m_k, AA, \langle a, b \rangle])$ is true by Lemma C.3.4 (p. 176), and since $\mathcal{A}A$ holds on $\rho n_1...n_k[entry(m_k)]$, by the induction hypothesis $\mathcal{A}A \in may-alias(entry(m_k))$. Thus $\langle a, b \rangle$ has to be in $may-alias(m_k)$ by definition of $may-alias.$

$\square$

**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.

\[\square\]

**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 \( \mathcal{O} = \{sp \mid p \text{ is a pointer in the program}\} \cup \{v \mid v \text{ is a non-pointer variable in the program}\} \cup \{\text{non-visible}\} \).

\( \mathcal{O} \) is the set of all object names in the program which may have aliases.

"non-visible" represents object names which are not visible.

Let \( \text{POSSIBLE-ALIASES} = (\mathcal{O} \times \mathcal{O}) - \{(\text{non-visible, non-visible})\} \).

\( \text{POSSIBLE-ALIASES} \) is the set of all possible aliases.

Define the lattice[Hec77] \( \mathcal{L} = (\mathcal{S}, \sqsubseteq, \cap, \bot, \top) \) as

- \( \mathcal{S} = \text{powerset(POSSIBLE-ALIASES)} \cup false \)
- \( a \sqsubseteq b \text{ iff } (a = false) \text{ or } (a \supseteq b)^{82} \)
- \( a \cap b = \begin{cases} 
false & \text{if } (a = false) \text{ or } (b = false) \\
\mathcal{a} \cup \mathcal{b} & \text{otherwise}
\end{cases} \)
- \( \top = \emptyset \)

---

\(^{82}\)NOTE: If \( b \) is \( false \), \( a \sqsubseteq b \) iff \( a \) is \( false \) also.
\[ \bot = \text{false} \]

Given an ICFG for a program:

Let \( \text{exit}(n) \), \( n \) a return node in the ICFG, be the exit node corresponding to \( n \).

Let \( \text{call}(n) \), \( n \) a return node in the ICFG, be the call node corresponding to \( n \).

Let \( \text{entry}(n) \), \( n \) a node in the ICFG, be the entry node of the procedure containing \( n \).

Let \( \text{back-bind}_{\text{call}} \), call a call node in the ICFG, be defined as in Appendix C.1.

\( \text{back-bind}_{\text{call}}(\text{alias-pair}) \) specifies which alias holding at the call site forces \( \text{alias-pair} \) to hold at the entry of the called procedure.

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

Let \( \text{must-holds}(n, \text{false}) = \text{false} \) for all nodes \( n \) in the ICFG.

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

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

1. \( n \) is an entry node
   \[
   \text{must-holds}(n, \langle a, b \rangle) = \{ \langle a, b \rangle \}
   \]

2. \( n \) is an exit node
   \[
   \text{must-holds}(n, \langle a, b \rangle) = \bigcap_{m.n. \in \varepsilon} \text{must-holds}(m, \langle a, b \rangle)
   \]

3. \( n \) is a call node
   \[
   \text{must-holds}(n, \langle a, b \rangle) = \bigcap_{m.n. \in \varepsilon} \text{must-holds}(m, \langle a, b \rangle)
   \]

4. \( n \) is a return node
   
   \begin{itemize}
   \item \( a \) and \( b \) are both visible in the called procedure
   \end{itemize}
must-holds\( (n, \langle a, b \rangle) = \begin{cases} \text{false} & \text{if } \text{must-holds} (\text{exit}(n), \langle a, b \rangle) = \text{false} \\ \emptyset & \text{if } \text{must-holds} (\text{exit}(n), \langle a, b \rangle) = \emptyset \\ R_1 & \text{otherwise} \end{cases} \)

\( R_1 : \quad \bigcap_{A \in \text{must-holds} (\text{exit}(n), \langle a, b \rangle)} \left( \text{must-holds} (\text{call}(n), \text{back-bind}_{\text{call}(n)}(A)) \right) \)

(b) \( a \) is visible in the called procedure, but \( b \) is not

\[
\begin{align*}
\text{must-holds}(n, \langle a, b \rangle) &= \begin{cases} 
\text{false} & \text{if } \text{must-holds}(\text{exit}(n), \langle a, \text{non-visible} \rangle) = \text{false} \\
\langle \text{not possible} \rangle & \text{if } \text{must-holds}(\text{exit}(n), \langle a, \text{non-visible} \rangle) = \emptyset \\
R_2 & \text{otherwise} 
\end{cases} \\
R_2 : \quad \bigcap_{(x, \text{non-visible}) \in \text{must-holds}(\text{exit}(n), \langle a, \text{non-visible} \rangle)} (\text{must-holds}(\text{call}(n), \text{back-bind}_{\text{call}(n)}((x, \text{non-visible}), b))) 
\end{align*}
\]

(c) \( a \) and \( b \) are not visible in the called procedure

\[
\text{must-holds}(n, \langle a, b \rangle) = \text{must-holds}(\text{call}(n), \langle a, b \rangle) 
\]

5. \( n \) is statement node

(a) \( n \) is not an assignment to a pointer

\[
\text{must-holds}(n, \langle a, b \rangle) = \bigcap_{m,n \in \mathcal{E}} (\text{must-holds}(m, \langle a, b \rangle)) 
\]

(b) \( n \) is the assignment “\( p = q \)” \( (p, q \) pointers)

i. if \( a = b = \ast p \)

\[
\text{must-holds}(n, \langle \ast p, \ast p \rangle) = \bigcap_{m,n \in \mathcal{E}} (\text{must-holds}(m, \langle \ast q, \ast q \rangle)) 
\]

ii. if \( a = \ast p \ [a \neq b] \)

\[
\text{must-holds}(n, \langle \ast p, b \rangle) = \bigcap_{m,n \in \mathcal{E}} (\text{must-holds}(m, \langle \ast q, b \rangle)) 
\]

iii. if \( (a \neq \ast p) \) and \( (b \neq \ast p) \)

\[
\text{must-holds}(n, \langle a, b \rangle) = \bigcap_{m,n \in \mathcal{E}} (\text{must-holds}(m, \langle a, b \rangle)) 
\]

(c) \( n \) is the assignment “\( p = \& v \)” \( (p \ a \) pointer)

i. if \( a = b = \ast p \)

\[
\text{must-holds}(n, \langle \ast p, \ast p \rangle) = \emptyset 
\]
\textbf{Algorithm for Computing Precise Must Alias Sets in the Presence of Single Level Pointers}

Let $O' = \{*p \mid p \text{ is a pointer in the program}\} \cup \{v \mid v \text{ is a non-pointer variable in the program}\}$. 

$O'$ is the set of all object names in the program which may have aliases.

Let $POSSIBLE\text{-ALIASES}' = (O' \times O')$. 

$POSSIBLE\text{-ALIASES}'$ is the set of all possible aliases.

Define the lattice $\mathcal{L}' =$

\[(\text{powerset}(POSSIBLE\text{-ALIASES'}), \subseteq, \cap(\subseteq), \emptyset (\perp), POSSIBLE\text{-ALIASES'} (\top)\)

Let $bind_n(A)$, $n$ a call node in the ICFG and $A \in POSSIBLE\text{-ALIASES'}$, be defined as in Appendix C.2.

Let $entry(n)$, $n$ a node in the ICFG, be the entry node of the procedure containing $n$. 

\begin{itemize}
  \item[ii.] if $a = *p \ [a \neq b]$
    \[\text{must-holds}(n, \langle *p, b \rangle) = \prod_{m,n \in E} \left( \text{must-holds}(m, \langle v, b \rangle) \right) \]
  \item[iii.] if $(a \neq *p)$ and $(b \neq *p)$
    \[\text{must-holds}(n, \langle a, b \rangle) = \prod_{m,n \in E} \left( \text{must-holds}(m, \langle a, b \rangle) \right) \]
  \end{itemize}

(d) $n$ is the assignment “$p = malloc()$” or “$p = NULL$” ($p$ a pointer)

\begin{itemize}
  \item[i.] if $a = b = *p$
    \[\text{must-holds}(n, \langle *p, *p \rangle) = \begin{cases} 
    \emptyset & \text{if } n \text{ is } "p = malloc()" \\
    \text{false} & \text{if } n \text{ is } "p = NULL" 
  \end{cases} \]
  \item[ii.] if $a = *p \ [a \neq b]$
    \[\text{must-holds}(n, \langle *p, b \rangle) = \text{false} \]
  \item[iii.] if $(a \neq *p)$ and $(b \neq *p)$
    \[\text{must-holds}(n, \langle a, b \rangle) = \prod_{m,n \in E} \left( \text{must-holds}(m, \langle a, b \rangle) \right) \]
\end{itemize}
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 $must-alias(n)$ as:

1. $must-alias(\rho) = \emptyset$

2. $n$ is an entry node

   $must-alias(n) = \bigcap_{\langle m, n \rangle \in \mathcal{E}} \left\{ \langle a, b \rangle \mid \langle a, b \rangle \in bind_m(must-alias(m)) \text{ and } \text{neither } a \text{ nor } b \text{ contain } \text{non_visible} \right\}$

3. otherwise

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

Algorithm

1. For all $n$ in the ICFG, all $\langle a, b \rangle \in POSSIBLE-ALIASES$ initialize $must-hold(n, \langle a, b \rangle)$ to $\emptyset$.

2. Calculate the fixed point of $must-hold$.

3. For all $n$ in the ICFG, initialize $must-alias(n)$ to $POSSIBLE-ALIASES'$.

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 $COND^{alias-pair}_{node}$ 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 $COND_{\text{node}}^{\text{alias-pair}}$ values

Lemma C.4.1 For all $n_i$ in the ICFG, for all $\langle a, b \rangle$, if

for all paths $\rho n_1 n_2 \ldots n_i$

$\langle a, b \rangle$ holds on $\rho n_1 n_2 \ldots n_i$ if $\langle c, d \rangle$ holds on $\rho n_1 n_2 \ldots n_{i-1}$

and $n_i$ is not an entry node

then

$COND_{n_i}^{\langle a, b \rangle} = \bigcap_{\langle m, n \rangle \in \mathcal{E}} \left( COND_m^{\langle c, d \rangle} \right)$

Proof:

Consider any $n$ and $\langle a, b \rangle$ such that for all paths $\rho n_1 n_2 \ldots n_i$ ($n_i = n$), $\langle a, b \rangle$ holds on $\rho n_1 n_2 \ldots n_i$ if $\langle c, d \rangle$ holds on $\rho n_1 n_2 \ldots n_{i-1}$ 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 $[entry(m)]n_1 n_2 \ldots m$ 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 $[entry(m)]n_1 \ldots mn$ regardless of which aliases hold at $[entry(m)] = [entry(n)]$. Thus $COND_n^{\langle a, b \rangle}$ is false.

- Otherwise $(\forall \langle m, n \rangle \in \mathcal{E}) COND_m^{\langle c, d \rangle} \neq \text{false}$

$COND_n^{\langle a, b \rangle} \subseteq \bigcup_{\langle m, n \rangle \in \mathcal{E}} \left( COND_m^{\langle c, d \rangle} \right)$ because, by definition of $COND$, assuming $\bigcup_{\langle m, n \rangle \in \mathcal{E}} \left( COND_m^{\langle c, d \rangle} \right)$ holds at $entry(n)$ means that on all paths $[entry(n)]n_1 n_2 \ldots m$, where $\langle m, n \rangle \in \mathcal{E}$, $\langle c, d \rangle$ must hold and thus $\langle a, b \rangle$ must hold on $[entry(n)]n_1 n_2 \ldots mn$. Also $COND_n^{\langle a, b \rangle} \supseteq \bigcup_{\langle m, n \rangle \in \mathcal{E}} \left( COND_m^{\langle c, d \rangle} \right)$. Otherwise there is a $m$ such that $\langle m, n \rangle \in \mathcal{E}$ and $COND_n^{\langle a, b \rangle} \nsubseteq COND_m^{\langle c, d \rangle}$. 
Thus there exists a path \([\text{entry}(m)]n_1n_2\ldots m\) which when assuming \(COND_n^{(a,b)}\) holds at \(\text{entry}(m)\), \(\langle c, d \rangle\) does not hold. Under those assumptions \(\langle a, b \rangle\) does not hold on \([\text{entry}(m)]n_1n_2\ldots m\) (\(\text{entry}(m) = \text{entry}(n)\)). Contradiction.

\[\square\]

---

**Lemma C.4.2** \(COND\) obeys the following relationships:

\(COND_n^{false} = false\) for all nodes \(n\) in the \(ICFG\).

\(COND_n^{(o,o)} = \emptyset\) for all nodes \(n\) in the \(ICFG\), all non-dereferenced pointer object names \(o \in \mathcal{O}\).

Consider \(COND_n^{(a,b)}\), \(n\) a node of the \(ICFG=(N,E,p)\) and \(\langle a, b \rangle \in \mathcal{A}\):

1. \(n\) is an entry node
   \[COND_n^{(a,b)} = \{\langle a, b \rangle\}\]

2. \(n\) is an exit node
   \[COND_n^{(a,b)} = \bigcap_{\langle m,n \rangle \in \mathcal{E}} (COND_m^{(a,b)})\]

3. \(n\) is a call node
   \[COND_n^{(a,b)} = \bigcap_{\langle m,n \rangle \in \mathcal{E}} (COND_m^{(a,b)})\]

4. \(n\) is a return node
   
   (a) \(a\) and \(b\) are both visible in the called procedure
   
   \[COND_n^{(a,b)} = \begin{cases} 
   false & \text{if } COND_{\text{exit}(n)}^{(a,b)} = false \\
   \emptyset & \text{if } COND_{\text{exit}(n)}^{(a,b)} = \emptyset \\
   \mathcal{R}_1 & \text{otherwise} 
   \end{cases}\]

   \[\mathcal{R}_1 : \bigcap_{A \in COND_{\text{exit}(n)}^{(a,b)}} (COND_{\text{call}(n)}^{\text{back-bind}_{\text{call}(n)}}(A))\]

   (b) \(a\) is visible in the called procedure, but \(b\) is not
\[ COND_{n}^{(a,b)} = \begin{cases} 
false & \text{if } COND_{\text{exit}(n)}^{(a,\text{non-visible})} = false \\
\text{(not possible)} & \text{if } COND_{\text{exit}(n)}^{(a,\text{non-visible})} = \emptyset \\
R_{2} & \text{otherwise} 
\end{cases} \]

\[ R_{2} : \]

\[ \bigcap_{x, \text{non-visible} \in COND_{\text{exit}(n)}^{(a,\text{non-visible})}} \left( COND_{\text{call}(n)}^{\text{back-bin}(x)}(x, \text{non-visible}, b) \right) \]

(c) a and b are not visible in the called procedure

\[ COND_{n}^{(a,b)} = COND_{\text{call}(n)}^{(a,b)} \]

5. n is statement node

(a) n is not an assignment to a pointer

\[ COND_{n}^{(a,b)} = \bigcap_{\ll m,n \gg \in \mathcal{E}} \left( COND_{m}^{(a,b)} \right) \]

(b) n is the assignment \(p = q\) (p, q pointers)

i. if \(a = b = *p\)

\[ COND_{n}^{(*p, *p)} = \bigcap_{\ll m,n \gg \in \mathcal{E}} \left( COND_{m}^{(*p, *q)} \right) \]

ii. if \(a = *p \ [a \neq b]\)

\[ COND_{n}^{(*p, b)} = \bigcap_{\ll m,n \gg \in \mathcal{E}} \left( COND_{m}^{(*q, b)} \right) \]

iii. if \(a \neq *p\) and \(b \neq *p\)

\[ COND_{n}^{(a,b)} = \bigcap_{\ll m,n \gg \in \mathcal{E}} \left( COND_{m}^{(a,b)} \right) \]

(c) n is the assignment \(p = \&v\) (p a pointer)

i. if \(a = b = *p\)

\[ COND_{n}^{(*p, *p)} = \emptyset \]

ii. if \(a = *p \ [a \neq b]\)

\[ COND_{n}^{(*p, b)} = \bigcap_{\ll m,n \gg \in \mathcal{E}} \left( COND_{m}^{(v, b)} \right) \]

iii. if \(a \neq *p\) and \(b \neq *p\)

\[ COND_{n}^{(a,b)} = \bigcap_{\ll m,n \gg \in \mathcal{E}} \left( COND_{m}^{(a,b)} \right) \]

(d) n is the assignment \(p = malloc()\)” or \(p = NULL\)” (p a pointer)
i. if $a = b = *p$  
\[
COND_{n}^{(a,b)} = \begin{cases} 
\emptyset & \text{if } n \text{ is } "p = malloc()" \\
false & \text{if } n \text{ is } "p = NULL" 
\end{cases}
\]

ii. if $a = *p \ [a \neq b]$  
\[COND_{n}^{(a,p,b)} = false\]

iii. if $a \neq *p$ and $b \neq *p$  
\[COND_{n}^{(a,b)} = \prod_{\langle m,n \rangle \in \mathcal{E}} (COND_{m}^{(a,b)})\]

Proof:

\[COND_{n}^{false} = false\] for all nodes $n$ in the CFG.

This is just a notational convenience.

\[COND_{n}^{(o,o)} = \emptyset\] for all nodes $n$ in the CFG, all non-dereferenced pointer object names $o \in \mathcal{O}$.

Reflexive aliases always hold between variables.

Consider $COND_{n}^{(a,b)}$, $n$ a node of the CFG=$\langle N,\mathcal{E},\rho \rangle$ and $\langle a, b \rangle \in A$ ($a \neq b$ or $a$ is a dereferenced pointer):

1. $n$ is an entry node  
   \[COND_{n}^{(a,b)} = \{\langle a, b \rangle\}\]. 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), $\prod_{\langle m,n \rangle \in \mathcal{E}} (COND_{m}^{(a,b)})$.

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), $\prod_{\langle m,n \rangle \in \mathcal{E}} (COND_{m}^{(a,b)})$.

4. $n$ is a return node
(a) \(a\) and \(b\) are both visible in the called procedure

- \(COND_{\text{call}}^{(a,b)} = \text{false}\) because \(COND_{\text{exit}(n)}^{(a,b)} = \text{false}\)

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.

- \(COND_{\text{call}}^{(a,b)} = \emptyset\) because \(COND_{\text{exit}(n)}^{(a,b)} = \emptyset\)

\(COND_{\text{exit}(n)}^{(a,b)} = \emptyset\) means that whenever the procedure containing \(\text{exit}(n)\) is called, \(a\) is aliased at \(b\) on exit and thus \(a\) and \(b\) must also be aliased whenever \(n\) is executed.

\(- COND_{\text{call}}^{(a,b)} = \bigcap_{A \in COND_{\text{exit}(n)}^{(a,b)}} \left( COND_{\text{call}(n)}^{\text{back-bind}_{\text{call}}(A)} \right) \)

\(\langle a, b \rangle\) holds on a path \(\rho \ldots [\text{exit}(n)]n\) iff it holds on \(\rho \ldots [\text{exit}(n)]\). By definition of \(COND\), \(\langle a, b \rangle\) must hold at \(\text{exit}(n)\) iff \(COND_{\text{exit}(n)}^{(a,b)}\) holds at \(\text{entry}(\text{exit}(n))\). By Lemma C.1.1 (p. 160), \(COND_{\text{call}(n)}^{(a,b)}\) holds at \(\text{entry}(\text{exit}(n))\) iff all aliases \(\{\text{back-bind}_{\text{call}}(A) \mid A \in COND_{\text{exit}(n)}^{(a,b)}\}\) must hold at \(\text{call}(n)\). This is true iff all aliases in

\(- \bigcap_{A \in COND_{\text{exit}(n)}^{(a,b)}} \left( COND_{\text{call}(n)}^{\text{back-bind}_{\text{call}}(A)} \right) \) hold at \(\text{entry}(n)\).

(b) \(a\) is visible in the called procedure, but \(b\) is not

- \(COND_{\text{call}}^{(a,b)} = \text{false}\) because \(COND_{\text{exit}(n)}^{(a,\text{non-visible})} = \text{false}\)

If \(\langle a, \text{non-visible} \rangle\) can under no circumstances hold on exit of a procedure then \(\langle a, \text{nv} \rangle\), \(\text{nv}\) a non-visible object name, can never hold on return from that procedure.

\(- COND_{\text{call}}^{(a,b)} = \bigcap_{\langle x, \text{non-visible} \rangle \in COND_{\text{exit}(n)}^{(a,\text{non-visible})}} \left( COND_{\text{call}(n)}^{\text{back-bind}_{\text{call}}(\langle x, \text{non-visible} \rangle, b)} \right) \)

\(\langle a, b \rangle\) holds on a path \(\rho \ldots [\text{exit}(n)]n\) iff \(\langle a, \text{non-visible} \rangle\) holds on \(\rho \ldots [\text{exit}(n)]\) where “\(\text{non-visible}\)” is the non-visible object name “\(b\)”. By definition of \(COND\), \(\langle a, \text{non-visible} \rangle\) must hold at \(\text{exit}(n)\) iff

\(COND_{\text{exit}(n)}^{(a,\text{non-visible})}\) holds at \(\text{entry}(\text{exit}(n))\). By Lemma C.1.1 (p. 160),

\(COND_{\text{exit}(n)}^{(a,\text{non-visible})}\) holds at \(\text{entry}(\text{exit}(n))\) iff all aliases

\(\{\text{back-bind}_{\text{call}}(A, b) \mid A \in COND_{\text{exit}(n)}^{(a,\text{non-visible})}\}\) must hold at \(\text{call}(n)\).

This is true iff all aliases in
\[ \bigwedge_{A \in \text{COND}^{\text{in} \circ \cdots \circ \text{in}(n)}} \left( \text{COND}^{\text{backward bind (A,b)}}_{\text{call}(n)} \right) \text{ hold at entry}(n). \]

(c) \(a\) and \(b\) are not visible in the called procedure

For all paths \(\rho_{n_1 \ldots n_i} (n_i = n), \langle a, b \rangle\) holds at \(n_i\) iff \(\langle a, b \rangle\) holds at \(\text{call}(n_i)\).

By an argument similar to the proof of Lemma C.4.1 (p. 194) (replace \(n_{i-1}\) with \(\text{call}(n_i)\)), \(\text{COND}^{(a,b)}_{\text{call}(n)} = \text{COND}^{(a,b)}_{\text{call}(n_i)}\).

5. \(n\) is statement node

(a) \(n\) is not an assignment to a pointer

For all paths \(\rho_{n_1 n_2 \ldots n_i}, \langle a, b \rangle\) holds on \(\rho_{n_1 n_2 \ldots n_i}\) iff \(\langle a, b \rangle\) holds on \(\rho_{n_1 \ldots n_{i-1}}\).

By Lemma C.4.1 (p. 194), \(\text{COND}^{(a,b)}_{\text{call}(n)} = \bigwedge_{\langle m,n \rangle} \epsilon \left( \text{COND}^{(a,b)}_{\text{call}(n)} \right)\).

(b) \(n\) is the assignment \(\langle p = q \rangle\) (\(p, q\) pointers)

i. if \(a = b = *p\)

For all paths \(\rho_{n_1 n_2 \ldots n_i}, \langle *p, *p \rangle\) holds on \(\rho_{n_1 n_2 \ldots n_i}\) iff \(\langle *q, *q \rangle\) holds on \(\rho_{n_1 \ldots n_{i-1}}\).

By Lemma C.4.1 (p. 194), \(\text{COND}^{(*p,*p)}_{\text{call}(n)} = \bigwedge_{\langle m,n \rangle} \epsilon \left( \text{COND}^{(*p,*q)}_{\text{call}(n)} \right)\).

ii. if \(a = *p [a \neq b]\)

For all paths \(\rho_{n_1 n_2 \ldots n_i}, \langle *p, b \rangle\) holds on \(\rho_{n_1 n_2 \ldots n_i}\) iff \(\langle *q, b \rangle\) holds on \(\rho_{n_1 \ldots n_{i-1}}\).

By Lemma C.4.1 (p. 194), \(\text{COND}^{(*p,b)}_{\text{call}(n)} = \bigwedge_{\langle m,n \rangle} \epsilon \left( \text{COND}^{(*q,b)}_{\text{call}(n)} \right)\).

iii. if \(a \neq *p\) and \(b \neq *p\)

For all paths \(\rho_{n_1 n_2 \ldots n_i}, \langle a, b \rangle\) holds on \(\rho_{n_1 n_2 \ldots n_i}\) iff \(\langle a, b \rangle\) holds on \(\rho_{n_1 \ldots n_{i-1}}\).

By Lemma C.4.1 (p. 194), \(\text{COND}^{(a,b)}_{\text{call}(n)} = \bigwedge_{\langle m,n \rangle} \epsilon \left( \text{COND}^{(a,b)}_{\text{call}(n)} \right)\).

(c) \(n\) is the assignment \(\langle p = & v \rangle\) (\(p\) a pointer)

i. if \(a = b = *p\)

On all paths \(\rho_{n_1 \ldots n}\) \(\langle *p, *p \rangle\) has to hold. Thus \(\text{COND}^{(*p,*p)}_{\text{call}(n)} = \emptyset\).

ii. if \(a = *p [a \neq b]\)

For all paths \(\rho_{n_1 n_2 \ldots n_i}, \langle *p, b \rangle\) holds on \(\rho_{n_1 n_2 \ldots n_i}\) iff \(\langle v, b \rangle\) holds on \(\rho_{n_1 \ldots n_{i-1}}\).
By Lemma C.4.1 (p. 194), \( COND_n^{(*p, b)} = \bigcap_{\ll m, n \gg \in \varepsilon} (COND_m^{(v,b)}) \).

iii. if \( a \neq *p \) and \( b \neq *p \)

For all paths \( pn_1n_2...n_i, \langle a, b \rangle \) holds on \( pn_1n_2...n_i \) iff \( \langle a, b \rangle \) holds on \( pn_1...n_{i-1} \).

By Lemma C.4.1 (p. 194), \( COND_n^{(a,b)} = \bigcap_{\ll m, n \gg \in \varepsilon} (COND_m^{(a,b)}) \).

(d) \( n \) is the assignment “\( p = malloc() \)” or “\( p = NULL \)” (\( p \) a pointer)

i. if \( a = b = *p \)

\[ *n \text{ is “} p = malloc() \text{”} \]

On all paths \( pn_1...n \langle *p, *p \rangle \) has to hold. Thus \( COND_n^{(*p, *p)} = \emptyset \).

\[ *n \text{ is “} p = NULL \text{”} \]

On all paths \( pn_1...n \langle *p, *p \rangle \) does not hold. Thus \( COND_n^{(*p, *p)} = \text{false} \).

ii. if \( a = *p \ [a \neq b] \)

On all paths \( pn_1...n \langle *p, b \rangle \) does not hold. Thus \( COND_n^{(*p, b)} = false \).

iii. if \( a \neq *p \) and \( b \neq *p \)

For all paths \( pn_1n_2...n_i, \langle a, b \rangle \) holds on \( pn_1n_2...n_i \) iff \( \langle a, b \rangle \) holds on \( pn_1...n_{i-1} \).

By Lemma C.4.1 (p. 194), \( COND_n^{(a,b)} = \bigcap_{\ll m, n \gg \in \varepsilon} (COND_m^{(a,b)}) \).

\[ \square \]

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 \( \text{must-holds}(n, \text{alias-pair}) = COND_n^{\text{alias-pair}} \).

We show \( \text{must-holds}(n, \text{alias-pair}) \subseteq COND_n^{\text{alias-pair}} \) by Lemma C.4.4 (p. 201) and Lemma C.4.5 (p. 201) and \( \text{must-holds}(n, \text{alias-pair}) \supseteq COND_n^{\text{alias-pair}} \) by Lemma
Lemma C.4.4 For every \( n \) and \( \langle a, b \rangle \in \text{POSSIBLE-ALIASES} \), let \( A_n^{(a,b)} \) in \( L \) be such that \( A_n^{(a,b)} \) 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 \( A_n^{(a,b)} \) so that on all paths \( P = \text{entry}(n_i)n_2n_3...n_i(n_i = n) \), there exists an \( A' \in L \) such that \( A_n^{(a,b)} \subseteq A' \). If \( A' = \text{false} \) then \( \langle a, b \rangle \) can never hold after the execution of \( P \), otherwise, when all aliases in the set \( A' \) hold at entry(\( n_i \)), the execution of \( P \) forces \( \langle a, b \rangle \) to hold at \( n_i \). Claim: \( \text{COND}_n^{(a,b)} \supseteq A_n^{(a,b)} \).

Assume that for some \( n \) and \( \langle a, b \rangle \) the Lemma was false. Thus \( \text{COND}_n^{(a,b)} \supseteq A_n^{(a,b)} \), but by definition \( \text{COND}_n^{(a,b)} \) 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\(^{83}\).

\( \square \)

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 must-holds(\( n, \langle a, b \rangle \)) = \( A \). For all paths \( P = \text{entry}(n_i)n_2n_3...n_i(n_i = n) \), there exists an \( A' \in L \) such that if \( A' = \text{false} \) then \( \langle a, b \rangle \) can never hold after the execution of \( P \), otherwise all aliases in the set \( A' \) hold at entry(\( n_i \)) and the execution of \( P \) forces \( \langle a, b \rangle \) to hold at \( n_i \) and \( A \subseteq A' \).

Proof by induction on \( i \), the path length.

**basis**: \( i = 1 \)

\(^{83}\)Remember, \( a \subseteq b \) iff \( \langle a = \text{false} \rangle \) or \( \langle a \supseteq b \rangle \)
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
  \[ \text{must-holds}(n, \langle a, a \rangle) = \emptyset. \] Lemma must hold because reflexive aliases always hold ($A' = \emptyset$).

- Otherwise
  \[ \text{must-holds}(n, \langle a, b \rangle) = \{ \langle a, b \rangle \}. \] Lemma must hold ($A' = \{ \langle a, b \rangle \}$).

**Induction hypothesis:** Lemma C.4.5 holds for $j < i$.

**Induction step:**

Let $\mathcal{P} = \text{entry}(n_i)n_2...n_i$.

Let $\mathcal{P}_{i-1} = \text{entry}(n_i)n_2...n_{i-1}$.

Consider $\text{must-holds}(n_i, \langle a, b \rangle)$

if $a = b$ and $a$ is not a dereferenced pointer then

\[ \text{must-holds}(n, \langle a, a \rangle) = \emptyset. \] 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

   \[ \text{must-holds}(n_i, \langle a, b \rangle) = \bigcap_{m \in n_i} (\text{must-holds}(m, \langle a, b \rangle)). \]

**Argument C.4.1** By the induction hypothesis, for $\mathcal{P}_{i-1}$ there is an $A'_{i-1} \supseteq \text{must-holds}(n_{i-1}, \langle a, b \rangle)$. $\text{must-holds}(n_{i-1}, \langle a, b \rangle) \supseteq \text{must-holds}(n_i, \langle a, b \rangle)$ by definition of $\text{must-holds}$. $A'_{i-1}$ must also be valid for $A'$ on $\mathcal{P}_i$. Thus $A' = A'_{i-1} \supseteq \text{must-holds}(n_{i-1}, \langle a, b \rangle) \supseteq \text{must-holds}(n_i, \langle a, b \rangle)$.

3. $n_i$ is a call node

   \[ \text{must-holds}(n_i, \langle a, b \rangle) = \bigcap_{m \in n_i} (\text{must-holds}(m, \langle a, b \rangle)). \]

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

- \( \text{must-holds}(n_i, \langle a, b \rangle) = \text{false} \) because \( \text{must-holds}(n_{i-1}, \langle a, b \rangle) = \text{false} \).

  Lemma must hold because \text{false} is less than everything.

- \( \text{must-holds}(n_i, \langle a, b \rangle) = \emptyset \) because \( \text{must-holds}(n_{i-1}, \langle a, b \rangle) = \emptyset \).

  By induction, whenever the procedure containing \( \text{exit}(n) \) 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.

(b) \( a \) is visible in the called procedure, but \( b \) is not

- \( \text{must-holds}(n_i, \langle a, b \rangle) = \text{false} \) because

  \( \text{must-holds}(n_{i-1}, \langle a, \text{non-visible} \rangle) = \text{false} \).

  Lemma must hold because \text{false} is less than everything.

- \( (1) \text{must-holds}(n_i, \langle a, b \rangle) = \)

  \[ \prod_{B \in \text{must-holds}({\text{exit}(n_i), \langle a, \text{non-visible} \rangle})} \left( \text{must-holds}(\text{call}(n_i), \text{back-bind}_{\text{call}(n_i)}(B, b)) \right) \]

  (2) By the induction hypothesis there is an

  \( A'_{i-1} \supseteq \text{must-holds}(n_{i-1}, \langle a, \text{non-visible} \rangle) \).

  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

  \( A'_{j} \supseteq \text{must-holds}(n_j, \langle c, d \rangle) \).

  Consider \( A' = \prod_{X \in A'_{i-1}} \left( A'_{j} \text{back-bind}_{\text{call}(n_i)}(X, X) \right) \).

  Certainly, \( A' \supseteq \text{must-holds}(n_i, \langle a, b \rangle) \) because of (1), (2), and (3) and Lemma C.1.1 (p. 160).

  Also \( A' \) holding at \( \text{entry}(n_i) \) and executing \( \mathcal{P}_i \) must force \( \langle a, b \rangle \) to hold at \( n_i \) because of (2) and (3).
(3) By the induction hypothesis for all \( \langle c, d \rangle \) there is an
\[ A^c_{j} \supseteq \text{must-holds}(n_j, \langle c, d \rangle). \]
Consider \( A' = \bigcap_{B \in A'_{i-1}} A^\text{back-bind}_{j}(\text{call}(n_i), B, b) \).

Certainly, \( A' \supseteq \text{must-holds}(n_i, \langle a, b \rangle) \) because of (1), (2), and (3) and
Lemma C.1.1 (p. 160).

Also \( A' \) holding at \( \text{entry}(n_i) \) 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


5. \( n_i \) is a statement node

(a) \( n_i \) is not an assignment to a pointer
\[ \text{must-holds}(n_i, \langle a, b \rangle) = \bigcap_{\langle m, n_i \rangle \in \mathcal{E}} \text{must-holds}(m, \langle a, b \rangle). \]


(b) \( n_i \) is the assignment \( \langle p = q \rangle \) (p,q pointers)

i. if \( a = b = \ast p \)
\[ \text{must-holds}(n_i, \langle \ast p, \ast p \rangle) = \bigcap_{\langle m, n_i \rangle \in \mathcal{E}} \text{must-holds}(m, \langle \ast q, \ast q \rangle). \]

By the induction hypothesis, for \( \mathcal{P}_{i-1} \) there is an \( A'_{i-1} \supseteq \text{must-holds}(n_{i-1}, \langle \ast q, \ast q \rangle). \text{must-holds}(n_{i-1}, \langle \ast q, \ast q \rangle) \supseteq \text{must-holds}(n_i, \langle \ast p, \ast p \rangle) \) by definition of \( \text{must-holds} \). \( A'_{i-1} \) must also be valid for \( A' \) on \( \mathcal{P}_i \) for \( \text{must-holds}(n_i, \langle \ast p, \ast p \rangle) \) because \( A' = A'_{i-1} \supseteq \text{must-holds}(n_{i-1}, \langle \ast q, \ast q \rangle) \supseteq \text{must-holds}(n_i, \langle \ast p, \ast p \rangle). \)

ii. if \( a = \ast p \) \( [a \neq b] \)
\[ \text{must-holds}(n_i, \langle \ast p, b \rangle) = \bigcap_{\langle m, n_i \rangle \in \mathcal{E}} \text{must-holds}(m, \langle \ast q, b \rangle). \]

By the induction hypothesis, for \( \mathcal{P}_{i-1} \) there is an \( A'_{i-1} \supseteq \text{must-holds}(n_{i-1}, \langle \ast q, b \rangle). \text{must-holds}(n_{i-1}, \langle \ast q, b \rangle) \supseteq \text{must-holds}(n_i, \langle \ast p, b \rangle) \) by definition of \( \text{must-holds} \). \( A'_{i-1} \) must also be valid for \( A' \) on \( \mathcal{P}_i \) for \( \text{must-holds}(n_i, \langle \ast p, b \rangle) \) because \( A' = A'_{i-1} \supseteq \text{must-holds}(n_{i-1}, \langle \ast q, b \rangle) \supseteq \text{must-holds}(n_i, \langle \ast p, b \rangle). \)
iii. if \( a \neq *p \) and \( b \neq *p \)

\[
\text{must-holds}(n_i, \langle a, b \rangle) = \bigcap_{\langle m, n_i \rangle \in \mathcal{E}} \text{must-holds}(m, \langle a, b \rangle)
\]


(c) \( n_i \) is the assignment “\( p = &v \)” (\( p \) a pointer)

i. if \( a = b = *p \)

\[
\text{must-holds}(n_i, \langle *p, *p \rangle) = \emptyset, \text{but } \langle *p, *p \rangle \text{ holds on all paths to } n_i \text{ (even with no assumptions at entry}(n_i)\text{)}, \text{thus the Lemma holds } (A' = \emptyset).
\]

ii. if \( a = *p \) [\( a \neq b \)]

\[
\text{must-holds}(n_i, \langle *p, b \rangle) = \bigcap_{\langle m, n_i \rangle \in \mathcal{E}} \text{must-holds}(m, \langle v, b \rangle)
\]

By the induction hypothesis, for \( P_{i-1} \) there is an \( A'_{i-1} \supseteq \text{must-holds}(n_{i-1}, \langle v, b \rangle) \). \( \text{must-holds}(n_i, \langle *p, b \rangle) \supseteq \text{must-holds}(n_{i-1}, \langle v, b \rangle) \) by definition of \( \text{must-holds} \). \( A'_{i-1} \) must also be valid for \( A' \) on \( P_i \) for \( \text{must-holds}(n_i, \langle *p, b \rangle) \) because \( A' = A'_{i-1} \supseteq \text{must-holds}(n_{i-1}, \langle v, b \rangle) \supseteq \text{must-holds}(n_i, \langle *p, b \rangle) \).

iii. if \( a \neq *p \) and \( b \neq *p \)

\[
\text{must-holds}(n_i, \langle a, b \rangle) = \bigcap_{\langle m, n_i \rangle \in \mathcal{E}} \text{must-holds}(m, \langle a, b \rangle)
\]


(d) \( n_i \) is the assignment “\( p = malloc() \)” or “\( p = NULL \)” (\( p \) a pointer)

i. if \( a = b = *p \)

• \( n_i \) is “\( p = malloc() \)”

\[
\text{must-holds}(n_i, \langle *p, *p \rangle) = \emptyset, \text{but } \langle *p, *p \rangle \text{ holds on all paths to } n_i \text{ (even with no assumptions at entry}(n_i)\text{), thus the Lemma holds } (A' = \emptyset).
\]

• \( n_i \) is “\( p = NULL \)”

\[
\text{must-holds}(n_i, \langle *p, b \rangle) = false.
\]

Lemma must hold because \( false \) is less than everything.

ii. if \( a = *p \) [\( a \neq b \)]

\[
\text{must-holds}(n_i, \langle *p, b \rangle) = false.
\]

Lemma must hold because \( false \) is less than everything.
iii. if $a \neq *p$ and $b \neq *p$

\[
\text{must-holds}(n_i, \langle a, b \rangle) = \bigcap_{m, n_i \in \mathcal{E}} (\text{must-holds}(m, \langle a, b \rangle))
\]


□

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 $n \in \text{ICFG}$ and $\langle a, b \rangle \in \text{POSSIBLE-ALIASES}$,

\[
\text{must-holds}(n, \langle a, b \rangle) \supseteq \text{COND}_n^{(a,b)}.
\]

Proof by induction on $j$, number of iterations of the maximum fixed point algorithm.

Let $\text{must-holds}^j(n, \langle a, b \rangle)$ be the value of must-holds$(n, \langle a, b \rangle)$ on the $j^{th}$ iteration of the fixed point algorithm.

basis: $j = 0$

\[
\text{must-holds}^0(n, \langle a, b \rangle) \quad \text{for all } n \in \text{ICFG} \text{ and all } \langle a, b \rangle \in \mathcal{A} \text{ is initialized to } \emptyset = \top. \text{ Lemma holds.}
\]

induction hypothesis: Lemma C.4.6 holds for $k < j$.

induction step: Say the value of $\text{must-holds}^j(n, \langle a, b \rangle)$ changed at iteration $j$.

if $a = b$ and $a$ is not a dereferenced pointer then

\[
\text{must-holds}^j(n, \langle a, a \rangle) = \emptyset. \text{ Lemma must hold because } \emptyset = \top.
\]

otherwise

1. $n$ is an entry node

\[
\text{must-holds}^j(n, \langle a, b \rangle) = \{\langle a, b \rangle\}. \text{ By Lemma C.4.2 (p. 197) } \text{COND}_n^{(a,b)} = \{\langle a, b \rangle\}
\]

and thus the Lemma holds.
2. $n$ is an exit node

\[ \text{must-holds}^i(n, \langle a, b \rangle) = \bigcap_{\langle m, n \rangle \in \mathcal{E}} \text{must-holds}^{i-1}(m, \langle a, b \rangle). \]

**Argument C.4.2** By the induction hypothesis for all $\langle m, n \rangle \in \mathcal{E}$, \n\[ \text{must-holds}^{i-1}(m, \langle a, b \rangle) \sqsupseteq \text{COND}_{m}^{(a,b)}. \]

By Lemma C.4.2 (p. 197), \n\[ \text{COND}_{n}^{(a,b)} = \bigcap_{\langle m, n \rangle \in \mathcal{E}} \left( \text{COND}_{m}^{(a,b)} \right) \] and by definition, \n\[ \text{must-holds}^i(n, \langle a, b \rangle) = \bigcap_{\langle m, n \rangle \in \mathcal{E}} \left( \text{must-holds}^{i-1}(m, \langle a, b \rangle) \right) \]
\[ \sqsupseteq \bigcap_{\langle m, n \rangle \in \mathcal{E}} \left( \text{COND}_{m}^{(a,b)} \right) = \text{COND}_{n}^{(a,b)}. \] Thus Lemma C.4.6 must hold.

3. $n$ is a call node

\[ \text{must-holds}^i(n, \langle a, b \rangle) = \bigcap_{\langle m, n \rangle \in \mathcal{E}} \text{must-holds}^{i-1}(m, \langle a, b \rangle). \]

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

- \[ \text{must-holds}^i(n_i, \langle a, b \rangle) = \text{false} \]

because \[ \text{must-holds}^{i-1}(\text{exit}(n), \langle a, b \rangle) = \text{false}. \]

By the induction hypothesis, \n\[ \text{false} = \text{must-holds}^{i-1}(\text{exit}(n), \langle a, b \rangle) \sqsupseteq \text{COND}_{\text{exit}(n)}^{(a,b)}. \]

Thus \[ \text{COND}_{\text{exit}(n)}^{(a,b)} = \text{false}, \] which means by Lemma C.4.2 (p. 197), \n\[ \text{COND}_{n}^{(a,b)} \] must be \text{false}.

- \[ \text{must-holds}^i(n_i, \langle a, b \rangle) = \emptyset \]

because \[ \text{must-holds}^{i-1}(\text{exit}(n), \langle a, b \rangle) = \emptyset. \]

Lemma must hold because $\emptyset = \top$.

- \[ \text{must-holds}^i(n, \langle a, b \rangle) = \bigcap_{A \in \text{COND}_{\text{exit}(n)}^{(a,b)}} \left( \text{must-holds}^{i-1}(\text{call}(n), \text{back-bind}_{\text{call}(n)}(A)) \right) \]

\[ \sqsupseteq [\text{by induction}] \bigcap_{A \in \text{COND}_{\text{exit}(n)}^{(a,b)}} \left( \text{must-holds}^{i-1}(\text{call}(n), \text{back-bind}_{\text{call}(n)}(A)) \right) \]

\[ \sqsupseteq [\text{by induction}] \bigcap_{A \in \text{COND}_{\text{exit}(n)}^{(a,b)}} \left( \text{COND}_{\text{call}(n)}^{\text{back-bind}_{\text{call}(n)}(A)} \right) \]

\[ = [\text{by Lemma C.4.2 (p. 197)}] \text{COND}_{n}^{(a,b)}. \]

Thus lemma must hold.

(b) $a$ is visible in the called procedure, but $b$ is not
• \textit{must-holds}^j(n_i, \langle a, b \rangle) = false

because \textit{must-holds}^{j-1}(\langle \text{exit}(n), \langle a, \text{non-visible} \rangle \rangle) = false.

By the induction hypothesis, false = \textit{must-holds}^{j-1}(\langle \text{exit}(n), \langle a, \text{non-visible} \rangle \rangle) \supseteq \text{COND}^{\langle a, \text{non-visible} \rangle}_{\text{exit}(n)}.

Thus \text{COND}^{\langle a, \text{non-visible} \rangle}_{\text{exit}(n)} = false which means (by Lemma C.4.2 (p. 197)) \text{COND}^{(a,b)}_n must be false.

• \textit{must-holds}^j(n, \langle a, b \rangle) = [by definition]

\[
\bigcap_{B \in \text{must-holds}^{j-1}(\langle \text{exit}(n), \langle a, \text{non-visible} \rangle \rangle)} (\text{must-holds}^j[\langle \text{call}(n), \text{back-bind}_{\text{call}(n)}(B, b) \rangle])
\]

\[
\supseteq [\text{by induction}] \bigcap_{B \in \text{COND}^{\langle a, \text{non-visible} \rangle}_{\text{call}(n)}} (\text{must-holds}^{j-1}[\langle \text{call}(n), \text{back-bind}_{\text{call}(n)}(B, b) \rangle])
\]

\[
\supseteq [\text{by induction}] \bigcap_{B \in \text{COND}^{\langle a, \text{non-visible} \rangle}_{\text{call}(n)}} (\text{COND}^{\text{back} \cdot \text{bind}_{\text{call}(n)}}(B, b))
\]

= [by Lemma C.4.2 (p. 197)] \text{COND}^{(a,b)}_n

Thus Lemma must hold.

(c) \(a\) and \(b\) are not visible in the called procedure

\textit{must-holds}^j(n, \langle a, b \rangle) = [by definition]

\[
\text{must-holds}^{j-1}[\langle \text{call}(n), \langle a, b \rangle \rangle] \supseteq [\text{by induction}]
\]

\text{COND}^{\langle a,b \rangle}_{\text{call}(n)} = [by Lemma C.4.2 (p. 197)] \text{COND}^{\langle a,b \rangle}_n

Thus Lemma holds.

5. \(n\) is a statement node

(a) \(n\) is not an assignment to a pointer

\textit{must-holds}^j(n, \langle a, b \rangle) = \bigcap_{\langle m, n \rangle \in \mathcal{E}} (\text{must-holds}^{j-1}(m, \langle a, b \rangle)).

Lemma holds by Argument C.4.2 (p. 201).

(b) \(n\) is the assignment “\(p = q\)” (\(p,q\) pointers)

i. if \(a = b = *p\)

\textit{must-holds}^j(n, \langle *p, *p \rangle) = \bigcap_{\langle m, n \rangle \in \mathcal{E}} (\text{must-hold}^{j-1}(m, \langle *q, *q \rangle))

By the induction hypothesis for all \(\langle m,n \rangle \in \mathcal{E},\)
must-holds^{i-1}(m, (\langle q, q \rangle)) \supseteq COND^i_{m}(\langle q, q \rangle).

By Lemma C.4.2 (p. 197),

\[ COND^i_{m}(\langle q, q \rangle) = \bigcap_{\ll m, n \gg \in \mathcal{E}} \left( COND^i_{m}(\langle q, q \rangle) \right). \]

Thus Lemma must hold.

ii. if \( a \neq *p \ [a \neq b] \)

must-holds^{i}(n, (\langle p, b \rangle)) = \bigcap_{\ll m, n \gg \in \mathcal{E}} \left( must-hold^{i-1}(m, (\langle q, b \rangle)) \right)

By the induction hypothesis for all \( \ll m, n \gg \in \mathcal{E} \),

must-holds^{i-1}(m, (\langle q, b \rangle)) \supseteq COND^i_{m}(\langle q, b \rangle).

By Lemma C.4.2 (p. 197),

\[ COND^i_{m}(\langle q, b \rangle) = \bigcap_{\ll m, n \gg \in \mathcal{E}} \left( COND^i_{m}(\langle q, b \rangle) \right). \]

Thus Lemma must hold.

iii. if \( a \neq *p \) and \( b \neq *p \)

must-holds^{i}(n, (\langle a, b \rangle)) = \bigcap_{\ll m, n \gg \in \mathcal{E}} \left( must-hold^{i-1}(m, (\langle a, b \rangle)) \right)

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-holds^{i}(n, (\langle p, *p \rangle)) = \emptyset

By Lemma C.4.2 (p. 197), \( COND^i_{n}(\langle p, *p \rangle) = \emptyset \)

Thus Lemma must hold.

ii. if \( a = *p \ [a \neq b] \)

must-holds^{i}(n, (\langle p, b \rangle)) = \bigcap_{\ll m, n \gg \in \mathcal{E}} \left( must-hold^{i-1}(m, (\langle v, b \rangle)) \right)

By the induction hypothesis for all \( \ll m, n \gg \in \mathcal{E} \),

must-holds^{i-1}(m, (\langle v, b \rangle)) \supseteq COND^i_{m}(\langle v, b \rangle).

By Lemma C.4.2 (p. 197), \( COND^i_{m}(\langle v, b \rangle) = \bigcap_{\ll m, n \gg \in \mathcal{E}} \left( COND^i_{m}(\langle v, b \rangle) \right). \)

Thus Lemma must hold.

iii. if \( a \neq *p \) and \( b \neq *p \)

must-holds^{i}(n, (\langle a, b \rangle)) = \bigcap_{\ll m, n \gg \in \mathcal{E}} \left( must-hold^{i-1}(m, (\langle a, b \rangle)) \right)

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 \( "p = malloc()" \)

\[
\text{must-holds}^i(n, (\ast p, \ast p)) = \emptyset
\]

By Lemma C.4.2 (p. 197), \( COND_{n}^{\ast p, \ast p} = \emptyset \)

Thus Lemma must hold.

- \( n \) is \( "p = NULL" \)

\[
\text{must-holds}^i(n, (\ast p, \ast p)) = false
\]

By Lemma C.4.2 (p. 197), \( COND_{n}^{\ast p, \ast p} = false \)

Thus Lemma must hold.

ii. if \( a = \ast p \ [a \neq b] \)

\[
\text{must-holds}^i(n, (\ast p, b)) = false
\]

By Lemma C.4.2 (p. 197), \( COND_{n}^{\ast p, b} = false \)

Thus Lemma must hold.

iii. if \( a \neq \ast p \) and \( b \neq \ast p \)

\[
\text{must-holds}^i(n, \langle a, b \rangle) = \bigcap_{m, n \in \varepsilon} \text{must-holds}^{i-1}(m, \langle a, b \rangle)
\]

Lemma holds by Argument C.4.2 (p. 207).

\[\square\]

**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 \( p \), \( \ast p \) 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 \(| variables |^2 \).

Since a program can use at most \( O(|\mathcal{N}|) \) variables, the number of possible aliases is \( O(|\mathcal{N}|^2) \).

For any \( n \in ICFG \) and any \( \langle a, b \rangle \in POSSIBLE-ALIASES \) and all iterations \( i \), \( \text{must-holds}(n, \langle a, b \rangle) \) at iteration \( i \) \( \supseteq \text{must-holds}(n, \langle a, b \rangle) \) at iteration \( i+1 \). Thus \( \text{must-holds}(n, \langle a, b \rangle) \) can change its value at most \( O(|\mathcal{N}|^2) \) times. This means there are at most \( O(|\mathcal{N}|^5) \) iterations in the fixed point algorithm and since each iteration can take
at most $O(|\mathcal{E}| + |\mathcal{N}|^2)$ time, the fixed point can be calculated in time polynomial in the size of the ICFG.

\[
\square
\]

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 $= (\mathcal{N}, \mathcal{E}, \rho)$.

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 $|\text{variables}|^2$. Since a program can use at most $O(|\mathcal{N}|)$ variables, the number of possible aliases is $O(|\mathcal{N}|^2)$.

During the fixed point calculation for any node, must-alias($\text{node}$) can never increase in size. Since the maximum size of all must-alias($\text{node}$) is $O(|\mathcal{N}|^2)$, any must-alias($\text{node}$) can change its value at most that many times. Thus we have $|\mathcal{N}|$ nodes that can change their value at most $O(|\mathcal{N}|^2)$ times. Therefore the fixed point calculation for must-alias takes a polynomial amount of time ($O(|\mathcal{N}|^2)$).

\[
\square
\]

Lemma C.4.9 \langle a, b \rangle \text{ not in must-alias}(n) implies } \exists \text{ a realizable paths to } n \text{ in the ICFG on which } \langle a, b \rangle \text{ does not hold.} 

Proof by induction on $j$, the iteration of the fixed point algorithm.

Let $\text{must-alias}^j(n)$ 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 CFG, must-alias$(n)$ is initialized to

\textit{POSSIBLE-ALIASES}'.

\textbf{induction hypothesis}: Lemma C.4.9 holds for $k < j$.

\textbf{induction step}:

Say the value of must-alias$^j(n)$ changed at iteration $j$.

1. $n$ is $\rho$

   must-alias$(\rho) = \emptyset$. No aliases hold before the program is executed.

2. $n$ is an entry node

   must-alias$^j(n) = \bigcap_{\ll m, n \gg \in \varepsilon} \langle \text{bind}_m(\text{must-alias}^{j-1}(m)) \rangle$.

   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 CFG, either

   \begin{itemize}
   \item back-bind$_m(\langle a, b \rangle)$ is $false$
     
     In this case, by Lemma C.1.1 (p. 160), on all paths $pn_1n_2...n_mn \langle a, b \rangle$ does not hold.

   \item 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 $P \circ n$.
   \end{itemize}

3. $n$ is not an entry node

   must-alias$(n) = \{ \langle a, b \rangle \mid$ must-hold$(n, \langle a, b \rangle) \subseteq$ must-alias(entry$\langle n \rangle) \}$

   Let $\langle a, b \rangle$ be any arbitrary alias pair in (must-alias$^{j-1}(n) - $ must-alias$^j(n)$).

   Either

   \begin{itemize}
   \item 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.

   \item There is a $\langle c, d \rangle$ in must-hold$(n, \langle a, b \rangle)$, but not in must-alias(entry$\langle n \rangle)$. 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.

\[ \square \]

**Lemma C.4.10** \( \exists \) a realizable path, \( \mathcal{P}_i = \rho n_1 \ldots n_i \), in the ICFG on which \( \langle a, b \rangle \) does not hold implies \( \langle a, b \rangle \) not in \( \text{must-alias}(n_i) \).

Proof by induction on \( i \), the path length

**basis:** \( i = 0 \)

\[ \mathcal{P}_i = \rho. \] No aliases hold and \( \text{must-alias}(\rho) = \emptyset \).

**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 \( \rho \)

   See the basis.

2. \( n_i \) is an entry node

   \[
   \text{must-alias}(n_i) = \bigcap_{\langle m, n_i \rangle \in \mathcal{E}} \langle \text{bind}_m(\text{must-alias}(m)) \rangle.
   \]

   - \( \text{back-bind}_{n_i-1}(\langle a, b \rangle) = \text{false} \)

     By Lemma C.1.1 (p. 160) and Lemma C.2.1 (p. 163), Lemma C.4.10 must hold.

   - otherwise

     \( \text{back-bind}_{n_i-1}(\langle a, b \rangle) \) 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, \( \text{back-bind}_{n_i-1}(\langle a, b \rangle) \) is not an element of \( \text{must-alias}(n_{i-1}) \) and, by Lemma C.2.1 (p. 163), \( \langle a, b \rangle \) is not an element of \( \text{must-alias}(n_i) \) because \( \langle n_{i-1}, n_i \rangle \in \mathcal{E} \).

3. \( n_i \) is not an entry node

   \[
   \text{must-alias}(n_i) = \{ \langle a, b \rangle \mid \text{must-hold}(n_i, \langle a, b \rangle) \subseteq \text{must-alias}(\text{entry}(n_i)) \}\]
• $\text{must-hold}(n_i, \langle a, b \rangle) = \text{false}$

By Lemma C.4.3 (p. 200), Lemma C.4.10 must hold.

• otherwise

By Lemma C.4.3 (p. 200), there is a $\langle c, d \rangle \in \text{must-hold}(n_i, \langle a, b \rangle)$ that does not hold on $P_{\text{entry}(n_i)}$. By induction, $\langle c, d \rangle$ is not in $\text{must-alias}(\text{entry}(n_i))$. Thus $\text{must-hold}(n_i, \langle a, b \rangle) \not\subseteq \text{must-alias}(\text{entry}(n_i))$. Thus $\langle a, b \rangle \not\in \text{must-hold}(n_i, \langle a, b \rangle)$.

\[
\square
\]

**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.

\[
\square
\]

C.5 Replacing reference formals with pointers

**Lemma 4.3.1** For all realizable paths $pn_1n_2...n_i$ 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 "\(p = \ldots\)" where \(x = \ast p\) 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 \( \rho \). 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 \( P_i = \rho n_1 n_2 \ldots n_i - 1 n_i \) in the original program.

Let \( P_{i-1} = \rho n_1 n_2 \ldots n_{i-1} \).

Let \( P_i^f \) 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 \( P_{i-1}^f \) in the transformed program. This requires that \( \langle *f_j, *f_k \rangle \equiv \langle *x, *y \rangle \) hold on \( P_i^f \).
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 “$p = \ldots$” 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 in-
duction hypothesis $\langle *a_j, *f \rangle$ holds on $P^t_{i-1}$ in the trans-
formed 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 trans-
formed 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 \(P_i^t\) in the transformed program. Thus \(\langle *f_j, y \rangle \equiv \langle *x, y \rangle\) holds on \(P_i^t\) 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 \(P_i^t\) 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 \&\(a_j\) 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 \(P_i^t\) 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 \&\(a_j\) 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 \(P_i^t\) in the transformed program. Thus \(\langle *f_j, y \rangle \equiv \langle *x, y \rangle\) holds on \(P_i^t\) 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 $\langle \ast x, \ast f \rangle$ holds on $P^{t}_{i-1}$ in the transformed program. Thus $\langle \ast x, \ast f_k \rangle \equiv \langle \ast x, y \rangle$ holds on $P^{t}_{i}$ since $f_k$ is passed $f$ in the transformed program.

• $a_k$ is not a $\&$ reference formal

$\langle x, \ast a_k \rangle$ holds on $P_{i-1}$ in the original program, thus by induction hypothesis $\langle \ast x, \ast a_k \rangle$ holds on $P^{t}_{i-1}$ in the transformed program. Thus $\langle \ast x, \ast f_k \rangle \equiv \langle \ast x, y \rangle$ 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, $\langle \ast x, y \rangle$ 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 $\langle \ast x, y \rangle$ 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 $x = \ast p$ because $x$ is a reference formal.

i. $n_i$ is the pointer assignment "$p = q$" in the original program

A. if $y = \ast p$

$\langle x, \ast q \rangle$ holds on $P_{i-1}$ in the original program, thus by induction hypothesis, $\langle \ast x, \ast q \rangle$ holds on $P^{t}_{i-1}$ in the transformed program and thus $\langle \ast x, \ast p \rangle \equiv \langle \ast x, y \rangle$ holds on $P^{t}_{i}$.

B. otherwise

$\langle x, y \rangle$ holds on $P_{i-1}$ in the original program, thus by induction hypothesis, $\langle \ast x, y \rangle$ holds on $P^{t}_{i-1}$ in the transformed
program and thus holds on $P_i^d$.

ii. $n_i$ is the pointer assignment "$p = \&v$" in the original program

A. if $y = \ast p$

$\langle x, v \rangle$ holds on $P_{i-1}$ in the original program, thus by induction hypothesis, $\langle \ast x, v \rangle$ holds on $P_{i-1}^d$ in the transformed program and thus $\langle \ast x, \ast p \rangle \equiv \langle \ast x, y \rangle$ holds on $P_i^d$.

B. otherwise

$\langle x, y \rangle$ holds on $P_{i-1}$ in the original program, thus by induction hypothesis, $\langle \ast x, y \rangle$ holds on $P_{i-1}^d$ in the transformed program and thus holds on $P_i^d$.

iii. $n_i$ is the pointer assignment "$p = malloc()$" or "$p = NULL$" in the original program

$y$ can not be $\ast p$ since $\ast p$ 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, $\langle \ast x, y \rangle$ holds on $P_{i-1}^d$ in the transformed program and thus holds on $P_i^d$.

3. $x$ and $y$ are not reference formals

(a) $n_i$ is an entry node

i. $x = \ast f_j$ and $y = \ast f_k$, $f_j$ 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 $\langle \ast f_1, \ast f_2 \rangle$ holds on $P_{i-1}^d$ in the transformed program. Thus $\langle \ast f_j, \ast f_k \rangle \equiv \langle x, y \rangle$ holds on $P_i^d$ 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 \( P^d_{i-1} \) in the transformed program. Thus \( \langle *f_j, *f_k \rangle \equiv \langle x, y \rangle \) holds on \( P^d_i \) 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 \( P^d_{i-1} \) in the transformed program. Thus \( \langle *f_j, *f_k \rangle \equiv \langle x, y \rangle \) holds on \( P^d_i \) 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 \( P^d_{i-1} \) in the transformed program. Thus \( \langle *f_j, y \rangle \equiv \langle x, y \rangle \) holds on \( P^d_i \) 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 \( P^d_{i-1} \) in the transformed program. Thus \( \langle *f_j, y \rangle \equiv \langle x, y \rangle \) holds on \( P^d_i \) since \( f_j \) is passed \( a_j \) in the transformed program.

iii. neither \( x \) nor \( y \) are dereferenced pointer forms 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 \( P^d_{i-1} \) in the transformed program. This implies that it also holds on \( P^d_i \).

(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 \( x = y = \ast p \)

By definition, \( \langle \ast p, \ast p \rangle \equiv \langle x, y \rangle \) 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 \( x = \ast p \ [x \neq y] \)

Not possible, since \( \ast p \) 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 \( \rho \). 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 \( P_i^t = \rho n_1 n_2 \ldots n_{i-1} n_i \) in the transformed program.

Let \( P_{i-1}^t = \rho n_1 n_2 \ldots n_{i-1} \).

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. \( \langle a, b \rangle \equiv \langle \ast x, \ast y \rangle \), \( x \) 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 $P_{i-1}^t$, thus by induction hypothesis $\langle a_j, a_k \rangle$ holds on $P_{i-1}$. This requires that $\langle f_j, f_k \rangle 
abla \langle x, y \rangle$ 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 $P_{i-1}^t$, thus by induction hypothesis $\langle a_j, a_k \rangle$ holds on $P_{i-1}$. This requires that $\langle f_j, f_k \rangle 
abla \langle x, y \rangle$ hold on $P_i$.

C. $a_j$ and $a_k$ are not reference formal

$\langle *a_j, *a_k \rangle$ must hold on $P_{i-1}^t$, 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 formal). 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 $P_{i-1}^t$. 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 $P_{i-1}^t$. 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 $P_{i-1}^t$, 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 $P_{i-1}^t$, 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 \( P_{i-1}^d \),
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 \( P_{i-1}^d \). Thus by induction hy-
 pothesis \( \langle a_j, f \rangle \) 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 \( P_{i-1}^d \). Thus by induction hy-
 pothesis \( \langle a_j, *a_k \rangle \) 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 \( P_{i-1}^d \). 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. \( z \) 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 \( P_{i-1}^t \). Thus by induction hypothesis \( \langle f_1, f_2 \rangle \) holds on \( P_{i-1}^t \). 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 \( P_{i-1}^t \). Thus by induction hypothesis \( \langle f_1, *a_k \rangle \) holds on \( P_{i-1}^t \). 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 \( P_{i-1}^t \). Thus by induction hypothesis \( \langle *a_j, *a_k \rangle \) holds on \( P_{i-1}^t \). 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 \( P_{i-1}^t \). Thus by induction hypothesis \( \langle f, y \rangle \) holds on \( P_{i-1}^t \). 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 $P_{i-1}$. 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 $P_{i-1}$, 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 $P_{i-1}$, 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 $P_{i-1}$, 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 $P_{i-1}$, 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 $P_{i-1}$, 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 $P_{i-1}$, 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 $P_{i-1}$, 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, $(*p, *p) \equiv \langle x, y \rangle$ holds on $P_i$ 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 $P_{i-1}'$, 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

\texttt{approximate-when-both-non_visible} (call-node, \langle n v_1, n v_2 \rangle, \\
\texttt{non_visible-name1-is, non_visible-name2-is, name1-AA, name2-AA}) substitutes \\
\texttt{non_visible-name1-is} for \texttt{non_visible} in \texttt{nv}_1, substitutes \texttt{non_visible-name2-is} for \\
\texttt{non_visible} in \texttt{nv}_2, and uses the assumed aliases \texttt{name1-AA} and \texttt{name2-AA} to \\
establish a safe assumed alias condition at procedure entry.

\begin{verbatim}
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,< non_visible-name1-is,non_visible-name2-is>) name2-AA
            to worklist
        }
    }
}
\end{verbatim}
{  
    /* 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 \emptyset if both are \emptyset because requiring both \emptyset and 
       A,\emptyset is the same as requiring just A,\emptyset). */ 
    if name1-AA is \emptyset or name2-AA contains non_visible 
        AA = name2-AA†  
    else AA = name1-AA  
    for each \langle x, y \rangle 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],\langle x, y \rangle]) is false  
            {  
                set may-hold([[return,AA],\langle x, y \rangle]) to true  
                add (return,AA,\langle x, y \rangle) to worklist  
            }  
    }  
}  

† Both name1-AA and name2-AA are necessary aliasing conditions at the entry of 
the procedure for z 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

\[
\text{alias\_at\_call\_implies}(\text{call, assumed\_alias}, \langle a, b \rangle) \\
\text{let return be the return node that corresponds to call} \\
\text{let exit be the exit node of the called procedure} \\
\text{let entry be the entry node of the called procedure} \\
\text{if both } a \text{ and } b \text{ are not visible in the called procedure /* Rule 1 */} \\
\quad \text{if } \text{may\_hold}([[\text{return, assumed\_alias}], \langle a, b \rangle]) \text{ is false} \\
\quad \text{set } \text{may\_hold}([[\text{return, assumed\_alias}], \langle a, b \rangle]) \text{ to true} \\
\quad \text{add } (\text{return, assumed\_alias}, \langle a, b \rangle) \text{ to worklist} \\
\text{/* Below an expanded version of the code found on page 97 with} \\
\text{the effects on entry added. */} \\
\text{for each } A_A \text{ and } \langle A_A, n_v \rangle \text{ in } \text{bind\_call}(\langle a, b \rangle) \\
\quad \text{if } \text{may\_hold}([[\text{entry, } A_A], A_A]) \text{ is false /* effects on entry */} \\
\quad \text{set } \text{may\_hold}([[\text{entry, } A_A], A_A]) \text{ to true} \\
\quad \text{add } (\text{entry, } A_A, A_A) \text{ to worklist} \\
\text{for every } \langle x, y \rangle \text{ or } \langle x, y \rangle, \text{other\_condition} \text{ in } \text{TUA}(\text{exit, } A_A) \\
\quad \text{if neither } x \text{ nor } y \text{ contain non\_visible /* Rule 2 */} \\
\quad \quad \text{if } \text{may\_hold}([[\text{return, assumed\_alias}], \langle x, y \rangle]) \text{ is false} \\
\quad \quad \quad \text{set } \text{may\_hold}([[\text{return, assumed\_alias}], \langle x, y \rangle]) \text{ to true} \\
\quad \quad \quad \text{add } (\text{return, assumed\_alias}, \langle x, y \rangle) \text{ to the worklist.} \\
\quad \text{else}
\]
{ if both \( z \) and \( y \) contain non-visible /* Rule 3- special case */
for each \((\mathcal{AA},nv)\) in \( \text{BET}(call,\langle\text{non-visible},\text{other.condition}\rangle) \)
approximate-when-both-non-visible\( (call,\langle x,y \rangle,nv,nv_2,\)
\text{assumed.alias},\mathcal{AA})
else (assume \( z \) contains non-visible): /* Rule 3 */
\text{let } x' = nv; \text{apply\_trans}(\text{non-visible},x,x')
if \text{may\_hold}(\langle\text{return, assumed.alias},(x',y)\rangle) \text{ is false}
\{ set \text{may\_hold}(\langle\text{return, assumed.alias},(x',y)\rangle) \text{ to true}
\text{add } (\text{return, assumed.alias},(x',y)) \text{ to the worklist.}
\}
\}
} /* end for each element of \( \text{TUA} \) */
} /* end for each element of \( \text{bind} \) */
D.3 Alias_at_exit_implies

alias_at_exit_implies(exit, AA, \langle x, y \rangle)
{ if both \( x \) and \( y \) contain non_visible

\{ let AA = \langle a_1, a_2 \rangle

  for each call node (call) that invokes the procedure containing exit

  for each (assumed_1, nv_1) in BBT(call, \langle \text{non_visible}, a_1 \rangle)

  for each (assumed_2, nv_2) in BBT(call, \langle \text{non_visible}, a_2 \rangle)

      approximate-when-both-non_visible(call, \langle x, y \rangle, nv_1, nv_2, assumed_1, assumed_2)

\} 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 \( \begin{cases} 
    \text{the object names } x \text{ and } y \text{ at call} \\
    \text{are visible in the called procedure}
    \end{cases} \)

    /* e.g., two aliased locals in a recursive procedure */

    if may-hold([\langle \text{return, assumed} \rangle, \langle x, y \rangle]) is false

    \{ set may-hold([\langle \text{return, assumed} \rangle, \langle x, y \rangle]) to true

      add (return, assumed, \langle x, y \rangle) 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)

            }
        }
    }
}
}
if \( z = p \) /* Case 3 */

\{
\text{for each} \ (a, b) \ \text{in} \ \text{alias.consequences}(\langle p, y \rangle) \ \text{or}
\}

\begin{align*}
\text{provided} \ & \text{!is.prefix}(y, q) \ \text{and} \ \text{!is.prefix}(p, q)) \\
\text{alias.consequences}(\langle \#y, \#q \rangle)
\end{align*}

\{
\text{if} \ \text{may-hold}([[\text{succ.assumed.alias}, (a, b)]) \ \text{is false}
\}

\{
\text{set} \ \text{may-hold}([[\text{succ.assumed.alias}, (a, b)]) \ \text{to true}
\}

\text{add} \ (\text{succ.assumed.alias}, (a, b)) \ \text{to the worklist}
\}
\}\}

\text{for each} \ (A, A, (a, b)) \ \text{such that}

\text{may-hold}([[\text{node.aa}, (a, b)]) \ \text{and}

\text{is.prefix.with.deref}(q, a) \ \text{and} \ \text{!is.prefix}(p, b)

\text{if} \ y \ \text{contains non.visible or} \ \text{!is.prefix}(y, b)

\{
\text{let} \ y' = y
\}

\text{apply.trans}(q, a, y')

\text{safely.make.alias}(\text{succ}, y', \text{assumed.alias}, b, A, A)
\}
\}

/* node is an assignment to a pointer */

} /* end for each successor */
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.

\[
\begin{aligned}
\langle p, q \rangle & \quad \downarrow \quad \langle *z, *y \rangle \\
\text{t: } & \quad p = x \\
\langle *q, *y \rangle & ?
\end{aligned}
\]

If both \( \langle p, q \rangle \) and \( \langle *z, *y \rangle \) 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.

\[
\begin{aligned}
\langle p, q \rangle & \quad \downarrow \quad \langle *q, *z \rangle \\
\text{t: } & \quad p = x \\
\langle *q, *z \rangle & ?
\end{aligned}
\]

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 \(p, z\)) also holds, then \(\langle \ast q, \ast z \rangle\) does not necessarily hold on any path to \(t\). In either case, for safety, we assume \(\langle \ast q, \ast z \rangle\) holds on some path to \(t\).

4.

\[
\begin{array}{c}
(p, \ast u) \\
(p, \ast v)
\end{array}
\]

\[
t: \begin{array}{c}
p.n = v \rightarrow n \rightarrow n; \\
\langle \ast (u \rightarrow n), \ast (v \rightarrow n \rightarrow n) \rangle
\end{array}
\]

Normally, \(\langle \ast (u \rightarrow n), \ast (v \rightarrow n \rightarrow n) \rangle\) should hold on a path to \(t\) because assigning \(v \rightarrow n \rightarrow n\) to \(p.n\) is also assigning \(v \rightarrow n \rightarrow n\) to \(u \rightarrow n\).\(^{85}\) However, \(\langle \ast (u \rightarrow n), \ast (v \rightarrow n \rightarrow n) \rangle\) need not hold. If, for example, \(\langle p, \ast u \rangle\) and \(\langle p, \ast v \rangle\) hold on the same path then \(\langle \ast (u \rightarrow n), \ast (v \rightarrow n \rightarrow n) \rangle\) 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 \(\text{may\-hold}([\text{node}, \mathcal{A}A], \mathcal{P}A)\) is true then either \(\mathcal{P}A\) is a type 2, 3, or 4 approximation, derived (directly or indirectly) from a type 2, 3 or 4 approximation, or \((\text{node}, \mathcal{P}A) \in \text{limit}_k(\text{precise solution})\).\(^{86}\)

---

Proof of Lemma E.1.1 by induction on \(i\), the iteration number of the algorithm.

**basis**: \(i = 0\)

- Aliases introduced by assignment \((\text{node})\) (Chapter 5.2.1)

  When \(p\) is not a prefix of \(q\) (let \(\text{node} \) 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 \(\langle p, \ast u \rangle\) holds

\(^{86}\) Somehow we must account for aliases involving non-visible object names. Assume that \texttt{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 \( \text{true} \) is set \( \text{true} \) 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\([\langle entry, \langle a, b \rangle \rangle, \langle a, b \rangle \]) is set \( \text{true} \) 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):

\[
\begin{array}{c}
\text{call}_{p(a_1,a_2)} \\
\text{store}_1: \\
\text{entry}_{p(f_1,f_2)} \\
\text{store}_2:
\end{array}
\]

\[
\begin{array}{c}
\text{1}\text{st way: } \langle **a_1,**a_2 \rangle \text{ at call } \Rightarrow \langle **a_1,*a_2 \rangle \text{ at entry} \\
\text{2}\text{nd way: } \langle **a_1,*a_2 \rangle \text{ at call } \Rightarrow \langle *f_2,**a_1 \rangle \text{ at entry} \\
\text{3}\text{rd way: } \langle **a_1,*a_2 \rangle \text{ at call } \Rightarrow \langle **f_1,*f_2 \rangle \text{ at entry}
\end{array}
\]

- 1\text{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\text{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.

- 3rd 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, AA], \(\langle x, y \rangle\)), where AA is in bind_{call}(possible.alias), is either a type 2, 3, or 4 approximation or (exit, \(\langle x, y \rangle\)) is in limith(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 AA at entry for some path the encoding is valid and the lemma follows. If \(\langle x, y \rangle\) depends on 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, 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 limith(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 limith(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 AA, PAA]),
where $\text{asserted_alias}$ in $\text{bind}_{\text{call}}(\mathcal{P.A})$, is either a type 2, 3, or 4 approximation or $(\text{call}, \mathcal{P.A})$ is in $\text{limit}_k$(precise solution). In the first case, by definition may-hold($([\text{return, asserted_alias}], (x, y))$) 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 $\mathcal{P.A}$ depends only on $\mathcal{A}$ at call for some path the encoding is valid and the lemma follows. If $\mathcal{P.A}$ depends on $\mathcal{A.A}$ plus other assumptions on some path then $\mathcal{P.A}$ 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 $\mathcal{P.A}$ is an approximation at call, by induction (may-hold($([\text{call, A.A}], \mathcal{P.A})$) must have been made true at an earlier iteration) $\mathcal{P.A}$ must be a type 2, 3, or 4 approximation and thus $\mathcal{P.A}$ is derived from a type 2, 3, or 4 approximation.

3. Any other alias implies(node, asserted_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 $\text{limit}_k$(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 $\text{limit}_k$(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 \textit{succ} given that it holds on a path to \textit{node} is if either \( y_1 \) or \( z_1 \) is redefined by \textit{succ} ("\( p = q \)"") on every path to \textit{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 \textit{succ} only if \( \langle p, y_1 \rangle \) or \( \langle p, z_1 \rangle \) holds on every path up to (but not including) \textit{succ} on which \( \langle *y_1, *z_1 \rangle \) does. That is the definition of the 3\textsuperscript{rd} type of approximation and thus the lemma holds.

- What are the effects of an alias of \( *q \)?

This case is applicable when \textsc{is\_prefix\_with\_deref}(\( q, y \)).

Consider \( y \equiv *q \). If \( z \) is something like \( p\rightarrow\textit{next} \) (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 \textit{succ} whenever \( \langle *q, z \rangle \) holds to \textit{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 \textit{succ}. However, when \( \langle p, z_1 \rangle \) holds on a path to \textit{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\textsuperscript{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 \textit{possible\_alias} is in \textit{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 \textit{may\_holds} that is set true is set true because of \textit{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., \textit{possible\_alias} is in \textit{limit}\(_k\) (precise solution)).
For case 3.i (Chapter 5.2.5), \( \langle *z, *q \rangle \) holds on a path to \( \text{succ} \) if \( \langle y, z \rangle \) holds to \( \text{node} \) 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 4th 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 \( \text{succ} \) by the Assumption 5.1.1 (NULL).

This leaves only case 3.iii (Chapter 5.2.5), but that case specifically handles the 2nd source of approximation and the lemma must follow.

\[ \square \]
References


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.


1992    Ph.D. in Computer Science.