Precise Flow-Insensitive May-Alias Analysis is NP-Hard

SUSAN HORWITZ
University of Wisconsin-Madison

Determining aliases is one of the fundamental static analysis problems, in part because the precision with which this problem is solved can affect the precision of other analyses such as live variables, available expressions, and constant propagation. Previous work has investigated the complexity of flow-sensitive alias analysis. In this paper we show that precise flow-*insensitive* may-alias analysis is NP-hard given arbitrary levels of pointers and arbitrary pointer dereferencing.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*; *Optimization*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Computability Theory*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*Decision problems*

General Terms: Languages, Theory

Additional Key Words and Phrases: Alias analysis, dataflow analysis, pointer analysis, static analysis

1. INTRODUCTION

The goal of "may alias" analysis is to determine which pairs of expressions might refer to the same memory location. It has been shown that given multiple levels of pointers, precise flow-sensitive may-alias analysis is NP-hard, even when restricted to the intraprocedural case with no dynamic memory allocation [Landi and Ryder 1991]. Further, it has been shown that given dynamic memory allocation, the problem becomes undecidable [Landi 1992] [Ramalingam 1994]. In this paper, we show that even precise flow-*insensitive* may-alias analysis is NP-hard given arbitrary levels of pointers and arbitrary pointer dereferencing. The proof involves a reduction of a known NP-hard problem, the *Hamiltonian path* problem [Garey and Johnson 1979], to the problem of precise, flow-insensitive, intraprocedural may-alias analysis.

This work was supported in part by the National Science Foundation under grant CCR-8958530, and by the Defense Advanced Research Projects Agency under ARPA Order No. 8856 (monitored by the Office of Naval Research under contract N00014-92-J-1937).

Author's address: Computer Sciences Department; Univ. of Wisconsin; 1210 West Dayton Street; Madison, WI 53706; USA. Electronic mail: horwitz@cs.wisc.edu

2. PROBLEM DEFINITION

The first question we must address is what we mean by *precise flow-insensitive may-alias analysis*. To do this, we first define precise flow-*sensitive* may-alias analysis.

In flow-sensitive may-alias analysis, a problem instance includes a control-flow graph representation of a program. The pair of expressions $< e_1, e_2 >$ is in the precise flow-sensitive solution to the may-alias problem at node n of the control-flow graph iff there exists a path in the control-flow graph from the start node to node n such that executing the sequence of operations along that path causes e_1 and e_2 to refer to the same memory location. (Note that this definition implies the usual assumption that all paths in the control-flow graph are executable.)

Flow-*insensitive* analysis only considers the set of nodes in the control-flow graph, without taking into account their actual ordering. The precise solution to flow-insensitive may-alias analysis must account for *all* possible orderings of the control-flow graph's nodes. The pair of expressions $< e_1, e_2 >$ is in the precise flow-insensitive solution to the may-alias problem for control-flow graph G iff it is possible to construct a control-flow graph G' using the same set of nodes, such that $< e_1, e_2 >$ is in the solution to the precise flow-sensitive may-alias problem at some node n of G.

Example. Figure 1 shows a program's control-flow graph annotated with precise, flow-sensitive may-alias information. The precise flow-insensitive solution for this program includes all of the expression pairs shown in the diagram, as well as the pair $<**a, d>$, because those expressions would refer to the same memory location if the order of the last two statements of the program were reversed.

Figure 2 illustrates the difference between *precise* flow-insensitive analysis and imprecise (but polynomial) methods such as those of [Andersen 1994] and [Steensgaard 1996]. Figure 2 includes seven assignment statements, the corresponding precise may-alias information, and the extra alias pairs that would be included using the methods of [Andersen 1994] and of [Steensgaard 1996].

To show that precise flow-insensitive may-alias analysis is NP-hard, we will consider a simple language that includes only assignment statements of the form "ID = *expression*", where expressions are defined by the following grammar:

expression	→	& ID
	—	*derefExpression*
derefExpression	→	ID
	—	* *derefExpression*

It is important to note that arbitrary levels of pointers and arbitrary pointer dereferencing are allowed; that is, there is no limit to the length of pointer "chains" in memory (a can point to b can point to c can point to ...), and there is no limit to the number of stars that can appear in a "dereference" expression (*e.g.*, $x = *****p$). [1]

[1] The need for arbitrary levels of pointer dereferencing may not be immediately obvious. A simplifying assumption that is sometimes made when designing alias-analysis algorithms is that all statements are of the form: ID = ID or ID = *ID or *ID = ID. A statement that contains multiple levels of dereferencing can be replaced by a sequence of statements of the above form, by introduc-

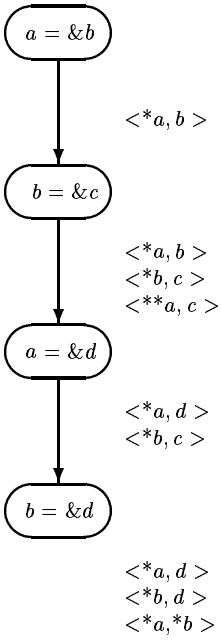

Fig. 1. Example illustrating flow-sensitive may-alias analysis.

Statements	*Precise Flow-Insensitive May-Alias Pairs*	*Extra Pairs Included by Andersen's Algorithm*	*Additional Extra Pairs Included by Steensgaard's Algorithm*
a = &b;	< *a, b >	< *b, z >	< *b, y >
a = &x;	< *a, x >	< *x, d >	< *x, c >
b = &c;	< *b, c >		< *c, z >
c = &d;	< *b, d >		< *y, d >
x = &y;	< *x, y >		<**b, z >
y = &z;	< *x, z >		<**x, d >
*a = **a;	< *c, d >		
	< *y, z >		
	< **a, c >		
	< **a, y >		
	< **b, d >		
	< **x, z >		
	<***a, d >		
	<***a, z >		

Fig. 2. Example illustrating precise and imprecise flow-insensitive may-alias analysis.

3. PRECISE FLOW-INSENSITIVE MAY-ALIAS ANALYSIS IS NP-HARD

THEOREM. *Given a program in the language defined above, it is NP-hard to determine whether a given pair of expressions is in the precise solution to the flow-insensitive may-alias analysis problem for that program.*

PROOF. The proof involves showing that, given an instance of the Hamiltonian path problem, it is possible to produce (in polynomial time) an instance of the may-alias problem (including a pair of expressions $< e_1, e_2 >$) such that e_1 and e_2 are in the solution to the may-alias problem iff there is a solution to the Hamiltonian path problem.

The Hamiltonian path problem is as follows: Given a directed graph G with designated *start* and *end* nodes, does there exist a path from *start* to *end* that visits every node exactly once?

To transform an instance of the Hamiltonian path problem (in which the graph G has N nodes, n_1 is the *start* node, and n_N is the *end* node) to an instance of the may-alias problem:

(1) For every edge $n_j \rightarrow n_k$ in G, add the statement $n_j = \&n_k$ to the program.
(2) Add the statement $n_N = \&END$ to the program, where END is a new variable.
(3) Add the statement $x = ****...*n_1$ to the program, where x is a new variable, and the number of stars is N.

CLAIM. *There is a Hamiltonian path in G from n_1 to n_N iff the pair of expressions <*x, END> is in the solution to the may-alias problem.*

Proof Part I. (Hamiltonian path $\Rightarrow$ $*x$ may-alias END): Suppose there is a Hamiltonian path in G of the form:

$$n_1 \rightarrow n_2 \rightarrow ... \rightarrow n_{N-1} \rightarrow n_N.$$

Then the statements:

$$n_1 = \&n_2;\ n_2 = \&n_3;\ ...;\ n_{N-1} = \&n_N;\ n_N = \&END;\ x = ****...*n_1;$$

are all in the program, and it is clear that using exactly that ordering of those statements causes $*x$ to be aliased to END.

Proof Part II. ($*x$ may-alias END $\Rightarrow$ Hamiltonian path): Now suppose that the pair of expressions $<*x, END>$ is in the solution to the may-alias problem; *i.e.*, there is some sequence of statements that causes the value of x to be the address of END. Consider the shortest such sequence. We observe that:

—The sequence must end with the statement $x = ****...*n_1$, since that is the only statement that assigns to x, and x's value is unaffected by any statements that follow that one.

---

ing new temporaries. Such a transformation is valid for flow-*sensitive* alias analysis (the precise flow-sensitive solution is the same for the original and the transformed programs); however, this is not the case for flow-insensitive analysis (the precise flow-insensitive solution to the transformed program may include more alias pairs—*i.e.*, may be less precise—than the precise flow-insensitive solution to the original program). This happens because the nodes of the control-flow graph determine the granularity of the flow-insensitivity. Separating a statement like $x = **p$; into two statements: $tmp_1 = *p$; $x = *tmp_1$; permits the analysis to "assume" that other statements might intervene between these two.

—The statement $x = ****...*n_1$ can cause *x and END to be aliases only if some variable holds the address of END when that statement is executed. Since the only statement that assigns the address of END to another variable is the statement $n_N = \&END$, that statement must occur somewhere in the sequence. Furthermore, since (by construction) no statement copies the value of one variable into another variable, variable n_N must hold the address of END when $x = ****...*n_1$ is executed, and thus there can be no other assignments to n_N in the sequence.

The program's state just before the final statement, $x = ****...*n_1$, is executed, can be represented by a directed graph G', whose nodes are labeled with the names of the variables $(n_1, n_2, ..., n_N, x, END)$, and whose edges represent the values of the variables (*i.e.*, there is an edge from n_j to n_k iff the value of n_j is the address of n_k). Since in a program state a variable has only one value, each node has at most one outgoing edge.

It is clear that the statement $x = ****...*n_1$ sets the value of x to the address of END iff there is a path in G' of length $N + 1$ from n_1 to END. We have argued above that only variable n_N contains the address of END. Therefore, the path in G' from n_1 to END must be of the form: $n_1 \rightarrow ... \rightarrow n_N \rightarrow END$; *i.e.*, it must consist of a path of length N from n_1 to n_N, followed by the edge $n_N \rightarrow END$.

Furthermore, the path from n_1 to n_N must be acyclic: n_N contains the address of END, which by construction never occurs on the left-hand side of an assignment, so node n_N cannot be involved in a cycle; all other nodes in the path have exactly one outgoing edge, so they cannot be involved in cycles, either (or the END node would never be reached). Therefore, this path involves exactly N distinct nodes, each of which corresponds to one node in graph G (the graph for the Hamiltonian path problem). The presence of edge $n_j \rightarrow n_k$ in G' means that there is a corresponding edge $n_j \rightarrow n_k$ in G. Thus, given the path in G' from n_1 to n_N, there must be a corresponding path in G, and so G contains a Hamiltonian path.

□

4. CONCLUSIONS AND FUTURE WORK

We have shown that even for a very simple language, precise flow-insensitive may-alias analysis is NP-hard, given arbitrary levels of pointers and arbitrary pointer dereferencing. Some interesting open questions remain:

—It was shown in [Landi and Ryder 1991] that precise flow-sensitive may-alias analysis is NP-hard given just *two* levels of pointer indirection. We believe that this result does *not* hold for flow-insensitive may-alias analysis. In fact, we believe that (at least in the intraprocedural case) precise flow-insensitive may-alias analysis becomes polynomial even with arbitrary levels of pointer indirection (maintaining the restriction that there is no dynamic memory allocation) if the number of levels of dereferencing (*i.e.*, the maximum number of stars in an expression) is restricted to some fixed k. We believe that the time will be polynomial in the size of the program, but exponential in k. However, this remains to be proved.

—It was shown in [Landi 1992], [Ramalingam 1994] that flow-sensitive may-alias analysis is undecidable in the presence of dynamic memory allocation. Does a

similar result hold for flow-insensitive analysis?

—We have shown that flow-insensitive may-alias analysis is NP-hard, but we have not been able to determine whether it is in NP.

—How important is our NP-hardness result in practice? How much information is lost by safe (but imprecise) polynomial-time flow-insensitive may-alias algorithms such as those defined in [Andersen 1994], [M.Burke et al. 1994], [Steensgaard 1996]?

REFERENCES

ANDERSEN, L. O. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU, University of Copenhagen. (DIKU report 94/19).

GAREY, M. AND JOHNSON, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company.

LANDI, W. 1992. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems 1,* 4, 323–337.

LANDI, W. AND RYDER, B. 1991. Pointer induced aliasing: A problem classification. In *ACM Symposium on Principles of Programming Languages.* 93–103.

M.BURKE, CARINI, P., CHOI, J., AND HIND, M. 1994. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Languages and Compilers for Parallel Computing: Proceedings of the 7th International Workshop*, K. Pingali, U. Banerjee, D. Galernter, A. Nicolau, and D. Padua, Eds. Lecture Notes in Computer Science, vol. 892. Springer-Verlag, Ithaca, NY, 234–250.

RAMALINGAM, G. 1994. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems 16,* 5, 1467–1471.

STEENSGAARD, B. 1996. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages.* 32–41.

Received February 1996; revised June 1996; accepted August 1996