Acta Informatica 38, 587–598 (2002)

On the Non-Approximability of Points-to Analysis

Venkatesan T. Chakaravarthy, Susan Horwitz

Department of Computer Sciences,
University of Wisconsin, Madison,
Madison, WI 53706. e-mail: {venkat,horwitz}@cs.wisc.edu

Received: 1 November 2001 / 4 February 2002

Abstract. Determining points-to sets is an important static-analysis problem. Most of the classic static analyses (used e.g., by compilers or in programming environments) rely on knowing which variables might be used or defined by each expression in a program. In the presence of pointers, the use/def set of an expression like $*p = *q$ can only be determined given (safe) points-to sets for p and q.

Previous work has shown that both precise flow-sensitive and precise flow-insensitive pointer analysis is NP-Hard, even when restricted to single-procedure programs with no dynamic memory allocation. In this paper, we show that it is not even possible to compute good approximations to the precise solutions (i.e., to compute points-to sets whose sizes are within a constant factor of the sizes of the precise points-to sets) unless P=NP.

1 Introduction

Most of the classic static analyses (e.g., constant propagation, live variable analysis, reaching definitions) rely on knowing which variables might be used or defined by each expression in a program. In the presence of pointers, the use/def set of an expression like $*x = *y$ can only be determined given (safe) information about what x and y might point to.

Previous work has shown that precise flow-sensitive pointer analysis is undecidable, given recursive data structures and dynamic memory allocation [Lan92b,Ram94]. Even without dynamic memory allocation, given at least four levels of pointer dereferencing (i.e., allowing pointer chains to be of length ≥ 4), the problem has been shown to be PSPACE-Complete [Lan92a], and to be NP-Hard given at least two levels of pointer dereferencing [LR91]. Precise flow-insensitive pointer analysis has been shown to be NP-Hard, even when restricted to single-procedure programs with no dynamic memory allocation [Hor97].

Given these results, it is very unlikely that there are precise, polynomial-time algorithms for either flow-sensitive or flow-insensitive pointer analysis. Therefore, a natural question to ask is whether it is possible to define a polynomial-time algorithm that produces a good *approximation* to the precise solution; i.e., a solution that is both safe (is a superset of the precise solution) and reasonably accurate (e.g., whose size is always within a constant factor of the size of the precise solution). Such approximation algorithms have been defined for many of the classic NP-Complete problems, including the vertex-cover problem, the knapsack problem, and the Euclidean travelling salesman problem [Pap94].

Unfortunately, in this paper, we show that no such approximation algorithms exist, either for flow-sensitive or for flow-insensitive pointer analysis, unless P=NP. In the flow-insensitive case, we show that if there were such an algorithm we could solve the NP-Complete Hamiltonian Path problem in polynomial time. In the flow-sensitive case, we prove a stronger result. We relax our notion of accuracy and require that the approximate solution be bounded by a polynomial in the size of the precise solution (instead of by a constant) and show that even such weaker approximate algorithms are infeasible unless P=NP. We prove this by showing that if there were such an algorithm we could solve the NP-Complete 3-SAT problem in polynomial time.

2 Background and Terminology

2.1 The Programming Language

We restrict our attention to programs that consist of a single procedure (with no procedure calls), have no dynamic memory allocation, and have only scalar variables (i.e., no arrays or structures). We use C notation for the "address-of" operator and pointer dereferencing:

x	\\ variable x
$\&x$	\\ the address of x
$*x$	\\ the location pointed-to by x

and we assume that there is no restriction on the number of dereferences in an l-value or r-value expression; i.e., we allow assignment statements of the form ***···*p = ***···*q, with an arbitray number of stars on either side.

2.2 Alias Analysis vs Points-to Analysis

Two versions of the pointer-analysis problem have been discussed in the literature: alias analysis and points-to analysis. Although the two problems are similar, they are not equivalent (see [LBS$^+$98] Section 2.4 for a discussion of some of the similarities and differences).

The complexity results mentioned in the introduction were all for *alias* analysis; however, the proofs can be easily modified to apply to points-to analysis. The proofs given in this paper are for *points-to* analysis. It is not clear whether they can be modified to apply to alias analysis; that is an interesting open problem.

2.3 Flow-sensitive Analysis

The goal of flow-sensitive points-to analysis is to determine, for each point in a program (for each node in the control-flow graph representation of the program), the set of points-to facts that might hold at that point. A points-to fact is a pair of variables (x, y), meaning that x might point to y. The fact (x, y) is in the precise solution to the points-to-analysis problem at program-point p iff there is some execution path (some path in the program's control-flow graph starting with the enter node and ending with node p) such that executing the sequence of statements along that path up to but not including node p, sets the value of x to be the address of y.

For the purposes of this paper, we consider a restricted version of the problem: namely, to determine the set of points-to facts that hold at the end of the program (at the exit node of the program's control-flow graph). Clearly, if the restricted version is not approximable, than neither is the more general version.

2.4 Flow-insensitive Analysis

One way to think about flow-insensitive points-to analysis is that it is equivalent to flow-sensitive analysis applied to a *complete* control-flow graph in which each node represents a single statement or predicate (rather than a basic block); i.e., the actual edges in a program's control-flow graph are augmented to include all possible edges (including self loops) before performing flow-sensitive analysis. This is equivalent to saying that statements can execute in an arbitrary order (including arbitrary repetitions). An advantage of this model for flow-insensitive analysis is that it is clear what is meant by a precise solution: a points-to fact $(x,\ y)$ is in the precise solution iff there is some path in the (complete) control-flow graph such that executing the sequence of statements along that path sets the value of x to be the address of y. Since the graph is complete, the same points-to facts will hold at every node (every path from the enter node up to but not including p is also a path from the enter node up to but not including every other node q).

It is interesting to note that for many NP-Hard graph problems, restricting the graphs to be complete makes the solution either trivial or at least easier than for an arbitrary graph. For example, the problem of determining whether a given directed graph has a Hamiltonian path is NP-Hard. But when the input graph is restricted to be complete, the problem becomes trivial (there is always a Hamiltonian path). However, in the case of points-to analysis it has been shown that finding the precise solution even for such a restricted version is NP-Hard [Hor97].

2.5 Example

Figure 1 shows a program's control-flow graph and gives the precise solutions to the flow-sensitive and flow-insensitive points-to-analysis problems for this program. Note that the flow-insensitive solution includes (w, b), which is not in the flow-sensitive solution. This is because the *complete* version of the control-flow graph includes the path:

$$enter \rightarrow x = \&a \rightarrow a = \&b \rightarrow w = {}^{*}x$$

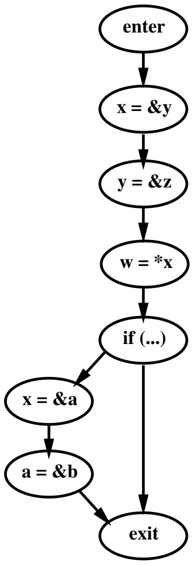
but that is not a path in the original control-flow graph.

2.6 Notation and Definitions

Given a program P, we use $\lambda(P)$ to denote the precise solution to the points-to-analysis problem for P (we use $\lambda(P)$ for both the flow-sensitive and flow-insensitive solutions – it should be clear from the context which is intended).

Next, we define two measures of the precision of an approximate algorithm. Consider an algorithm A for the points-to-analysis problem. Let $A(P)$ denote the points-to set computed by the algorithm for program P.

A is said to be a *k-approximate algorithm*, for some constant k, if



Precise flow–sensitive solution (the set of points–to facts that hold at the end of the program): { (x,y) (x,a) (y,z) (w,z) (a,b) }

Precise flow–insensitive solution: { (x,y) (x,a) (y,z) (w,z) (w,b) (a,b) }

Fig. 1 Example control-flow graph with precise flow-sensitive and flow-insensitive points-to sets.

1. A is a polynomial-time algorithm.
2. For all programs P, $\lambda(P) \subseteq A(P)$.
3. For all programs P, $|A(P)| \leq k|\lambda(P)|$.

A is said to be a *poly-k-approximate algorithm*, for some constant k, if

1. A is a polynomial-time algorithm.
2. For all programs P, $\lambda(P) \subseteq A(P)$.
3. For all programs P, $|A(P)| \leq |\lambda(P)|^k$.

In section 3 we show that there are no *poly*-k-approximate algorithms (for any constant k) for the flow-sensitive points-to analysis, unless P=NP. In section 4 we show that there are no k-approximate algorithms (for any constant k) for flow-insensitive points-to analysis, unless P=NP.

3 Non-Approximability of Flow-sensitive Points-to Analysis

In this section, we show that if there is a *poly*-k-approximate algorithm for flow-sensitive points-to analysis, then we can solve the

3-SAT problem[1] in polynomial time. 3-SAT is known to be NP-Complete [GJ79]; hence, this would mean that P=NP. This proof is an extension of the proof used in [LR91] to show that precise flow-sensitive alias analysis is NP-hard.

Given a *poly-k*-approximate algorithm A for the points-to-analysis problem, our algorithm for 3-SAT works as follows. Let the input 3-SAT formula ϕ be defined over N variables and C clauses. Let the variables be $\{x_1, x_2, \ldots x_N\}$. Let ϕ be $(l_{1,1} \vee l_{1,2} \vee l_{1,3}) \wedge (l_{2,1} \vee l_{2,2} \vee l_{2,3}) \ldots (l_{C,1} \vee l_{C,2} \vee l_{C,3})$. Here, $l_{i,j}$ stands for the j^{th} literal of the i^{th} clause. It could be some variable x_r or its complement $\overline{x_r}$. Given ϕ, we first construct a program P as follows: Irrespective of ϕ, we always include the variables True, False and Yes in P. For each variable x_i, we include two variables, X_i and $\overline{X_i}$ in P. Finally, we include R variables, $Z_1, Z_2 \ldots Z_R$, where R= $(2+4N)((2+4N)^k - 1)$. The reason for choosing this value for R will be clear when we prove the correctness of our 3-SAT algorithm below. The program P is as follows:

program P {
    int Yes, $Z_1, Z_2, \ldots Z_R$;
    int *True, *False;
    int $**X_1$, $**\overline{X_1}$, $**X_2$, $**\overline{X_2}$, ... $**X_N$, $**\overline{X_N}$;

    /**** Part I: Initialization. ****/
    True = &Yes;
    False = &Yes;
    if (...) False = &Z_1;
    if (...) False = &Z_2;
    ......
    if (...) False = &Z_R;

    /***** Part II: Choosing a truth assignment. ****/
    /***** One if-then-else for each variable X_i. ****/
    if (...) { X_1 = &True; $\overline{X_1}$ = &False; }
    else   { X_1 = &False; $\overline{X_1}$ = &True; }
    ...
    if (...) { X_N = &True; $\overline{X_N}$ = &False; }
    else   { X_N = &False; $\overline{X_N}$ = &True; }

[1] Recall that the 3-SAT problem is defined as follows: Given a Boolean formula in conjunctive normal form, where each clause contains three literals (and each literal is either x or $\overline{x}$, for some variable x), is there some assignment of truth values to variables that makes the formula true?

```
    /**** Part III : Validating the truth assignment ****/
    /**** One switch statement for each clause; ****/
    /**** two cases plus a default to prevent fall-through. ****/
    switch(...){
        case ... : *l_{1,1} = &Yes; break;
        case ... : *l_{1,2} = &Yes; break;
        default ... : *l_{1,3} = &Yes; break;
    }
    . . .
    switch(...){
        case ... : *l_{C,1} = &Yes; break;
        case ... : *l_{C,2} = &Yes; break;
        default ... : *l_{C,3} = &Yes; break;
    }
}
```

In the above program, $l_{i,j}$ represents the variable corresponding to the literal. For example, if there is a clause $(x_1 \vee \overline{x_2} \vee x_3)$, then the corresponding switch statement would be:

```
switch(...){
    case ... : *X_1 = &Yes; break;
    case ... : *\overline{X_2} = &Yes; break;
    default ... : *X_3 = &Yes; break;
}
```

In the above program, part II corresponds to choosing a truth assignment. Note that there are 2^N possible execution paths in part II, and each path corresponds to choosing a particular truth assignment t. Part III checks whether the truth assignment satisfies the formula ϕ. Here, each switch statement corresponds to validating a particular clause of ϕ. Note that at the end of part I, False either points to Yes, or it points to some Z_i. When a switch statement in part III is executed, False can remain pointing to Z_i iff t satisfies the corresponding clause (i.e., iff the assignment in the switch changes variable True to point to Yes rather than changing False to point to Yes). If t fails to satisfy a clause, whichever case statement is executed in the corresponding switch statement, False will be changed to point to Yes (since all three of the literals in the clause will be pointing to False). Thus, if False points to Z_i at the end of part I, False can remain pointing to Z_i at the end of the program iff the chosen truth assignment t satisfies ϕ.

Since at the end of part I, False can be pointing to *any* of the Z_i, it can still be pointing to any of them at the end of the program iff

ϕ is satisfiable. Therefore, if ϕ is satisfiable, then at the end of the program, False might point to Yes, or it might point to any of the Z_i, and so the precise solution to the flow-sensitive points-to-analysis problem is as follows:

$$\begin{aligned}\lambda(P) = \{&(\text{True}, \text{Yes}), (\text{False}, \text{Yes})\} \bigcup \\ &\{(X_i, \text{True}), (X_i, \text{False}), (\overline{X_i}, \text{True}), (\overline{X_i}, \text{False}), 1 \le i \le N\} \bigcup \\ &\{(\text{False}, Z_i), 1 \le i \le R\}\end{aligned}$$

and $|\lambda(P)|$=2+4N+R.

If ϕ is *not* satisfiable, then at the end of the program, False can only point to Yes; i.e., in this case:

$$\begin{aligned}\lambda(P) = \{&(\text{True}, \text{Yes}), (\text{False}, \text{Yes})\} \bigcup \\ &\{(X_i, \text{True}), (X_i, \text{False}), (\overline{X_i}, \text{True}), (\overline{X_i}, \text{False}), 1 \le i \le N\}\end{aligned}$$

and $|\lambda(P)|$=2+4N.

Now we present our algorithm for solving 3-SAT, given an approximation algorithm A:

Algorithm for 3-SAT

1. Given a 3-SAT formula ϕ, use the above construction to get the program P.
2. Run the *poly-k*-approximate algorithm A on P to get $A(P)$.
3. Define $D = \{(\text{False}, Z_i), 1 \le i \le R\}$. If $D \subseteq A(P)$ then report that ϕ is satisfiable; otherwise, report that ϕ is non-satisfiable.

To prove the correctness of the algorithm, all we need to prove is that ϕ is satisfiable iff $D \subseteq A(P)$. Consider the two cases. Let ϕ be satisfiable. Then, as we have already observed, $D \subseteq \lambda(P)$. And by the definition of *poly-k*-approximation, $\lambda(P) \subseteq A(P)$. Hence, $D \subseteq A(P)$.

Consider the case where ϕ is not satisfiable. We claim that $D \not\subseteq A(P)$. By contradiction let $D \subseteq A(P)$. In this case, as observed already, $|\lambda(P)| = 2{+}4N$ and $D \bigcap \lambda(P)$ is empty. By definition, $\lambda(P) \subseteq A(P)$. Thus, since by assumption $D \subseteq A(P)$, and $|D| = R$, $|A(P)| \ge 2 + 4N + R = (2 + 4N) + ((2 + 4N)^k - 1)(2 + 4N) = (2{+}4N)^{k+1}$. So, $|A(P)| \ge |\lambda(P)|^{k+1}$. This contradicts the assumption that A is a *poly-k*-approximate algorithm, because $|\lambda(P)| = (2{+}4N)$.

Thus, we have proved that if there is a *poly-k*-approximate algorithm for precise, flow-sensitive points-to analysis, we can solve 3-SAT in polynomial time, which can only be true if P=NP.

4 Non-Approximability of Flow-Insensitive Points-to Analysis

In this section, we show that if there is a k-approximate algorithm for flow-insensitive points-to analysis, then we can solve the NP-Complete Hamiltonian Path problem[2] [GJ79], for graphs with at least $k+2$ nodes, in polynomial time. This proof is an extension of the proof used in [Hor97] to show that precise flow-insensitive alias analysis is NP-hard.

Let the input problem instance be a directed graph $G = (V, E)$. Let $V = \{n_1, n_2, \ldots n_N\}$ (so $|V| = N$), such that $N \geq (k + 2)$. The problem is to determine whether there exists a Hamiltonian path from n_1 to n_N. Without loss of generality, assume that the in-degree of n_1 and out-degree of n_N are zero. Given this problem instance, we first construct a program P as follows. (Note that since we are interested in flow-insensitive analysis we only need to produce a set of statements – the flow of control of the program is irrelevant.)

The variables in program P are determined as follows:

1. For each node $n_i \in V$, include a variable n_i in P. We call these node-variables.
2. Include variables $x, x_1, x_2, \ldots x_N$.
3. Include variables $\text{End}, \text{End}_1, \text{End}_2, \ldots \text{End}_T$, where $\text{T} = k(N+1)^2$.

(T plays a role similar to the variable R used in the previous section; the proof of correctness of our Hamiltonian-Path algorithm will clarify the choice of value for T.)

The statements in P are:

1. **Edge Statements:** For each edge $(n_i, n_j) \in E$, include the statement $n_i = \&n_j$.
2. **Validating statements:** Include the statements $n_N = \&\text{End}$ and $x = ^{****\cdots*}n_1$, where the number of stars is N.
3. **Approximation Statements:** For $1 \leq j \leq T$, include $\text{End} = \&\text{End}_j$. For $1 \leq i \leq N$, include $x_i = {}^*x$.

The construction given in the previous section generates programs where variables have well-defined types and the program would type check; however, the programs generated in our present construction do not enjoy this property. It is clear that allowing structures (with pointer fields) would permit essentially the same construction to produce programs that do type check. It is an interesting open problem

[2] Recall that the Hamiltonian Path problem is defined as follows: Given a directed graph with designated *start* and *end* nodes, does there exist a path from *start* to *end* that visits every node exactly once?

whether a similar non-approximability result holds for programs with only scalar variables with well-defined types.

Our algorithm for solving the Hamiltonian Path problem is similar to the algorithm for solving the 3-SAT problem that we presented earlier:

Algorithm for Hamiltonian Path:

1. Construct the program P (the set of statements) using the construction given above.
2. Run the approximate algorithm A and obtain $A(P)$.
3. Define $D = \{(x_i, \mathrm{End}_j),\ 1 \leq i \leq N,\ 1 \leq j \leq T\}$. If $D \subseteq A(P)$ report that there is a Hamiltonian path from n_1 to n_N and none otherwise.

The construction used here is the same as the one used in [Hor97], except that the approximation statements are new. Therefore, as in [Hor97], we first argue that x can be made to point to End iff there is a Hamiltonian path from n_1 to n_N.

First, suppose there is an Hamiltonian path in G of the form:

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

Then the statements:

$$n_1 = \&n_2; n_2 = \&n_3; \ldots ; n_{N-1} = \&n_N; n_N = \&\mathrm{End}; x = ^{****}\ldots{}^{*}n_1$$

are all in the program, and it is clear that using exactly that ordering of those statements causes x to point to End.

Next, suppose there is a sequence of statements at the end of which x points to End. The only statement that affects x is $x = ^{****}\ldots{}^{*}n_1$. So we can assume that this is the last statement used in the sequence. There are N dereferences made in this statement. Thus, before executing this statement, there must be a chain of pointers of length $N+1$ that starts with n_1 and ends with End. The only variable that can point to End is n_N. Thus the above chain has a sub-chain of length N, starting with n_1 and ending with n_N. From the construction, it is clear that any node-variable n_i, except n_N, can only point to some other node-variable n_j. Thus, the above sub-chain can consist only of node-variables. As there are exactly N node-variables, each one of them must occur exactly once in this chain. Consider any link in this chain $n_i \rightarrow n_j$. The only statement that can create this link is $n_i = \&n_j$. This means that the edge (n_i, n_j) is in the graph. So there is a path in the graph from n_1 to n_N that visits each node exactly once. In other words, there is a Hamiltonian path from n_1 to n_N in the graph.

We have argued that x can be made to point to End iff there is a Hamiltonian path from n_1 to n_N. By the nature of the construction, it is clear that for any x_i and End_j, $(x_i, \mathrm{End}_j) \in \lambda(P)$ iff $(x, \mathrm{End}) \in \lambda(P)$. Therefore, $D \subseteq \lambda(P)$ if there is a Hamiltonian path from n_1 to n_N and $D \cap \lambda(P)$ is empty if there is no such Hamiltonian path.

Now we must show that $D \subseteq A(P)$ iff there is a Hamiltonian path from n_1 to n_N. Consider the case where there is a Hamiltonian path. Then, $D \subseteq \lambda(P) \subseteq A(P)$. The first inclusion is by the previous observation and the second one is by the definition of approximate algorithm.

Suppose there is no Hamiltonian path from n_1 to n_N. We claim that $D \not\subseteq A(P)$. By contradiction, let $D \subseteq A(P)$. We will now show that our choice of values of T causes $A(P)/\lambda(P) > k$, which will contradict the assumption that A is a k-approximate algorithm. We first characterize $\lambda(P)$ when there is no Hamiltonian path. Then, $\lambda(P)$ *may* include only the following pairs:

1. (n_i, n_j), iff $(n_i, n_j) \in E$.
2. (n_N, End).
3. $(\mathrm{End}, \mathrm{End}_j)$, for all End_j.
4. (x, n_i), for any n_i.
5. (x_i, n_j) for any x_i and n_j.
6. (x_i, End) for any x_i.

Using the above observation, we set an upper bound on $|\lambda(P)|$:

$$|\lambda(P)| \leq |E| + 1 + T + N + N^2 + N$$

Now $|E|$, the number of edges in G is $\leq N^2$. Recall $\mathrm{T} = k(N+1)^2$, where k is the approximation constant. Substituting $|E| = N^2$, we get:

$$|\lambda(P)| \leq N^2 + 1 + T + N^2 + N + N \ = \ T + N^2 + (N+1)^2$$

which is less than:

$$T + 2(N+1)^2$$

Substituting $T = k(N+1)^2$, we get:

$$|\lambda(P)| \leq (k+2)(N+1)^2$$

By assumption, $D \subseteq A(P)$, and by definition $\lambda(P) \subseteq A(P)$, so $A(P) \supseteq D \bigcup \lambda(P)$. As $D \cap \lambda(P)$ is empty,

$$|A(P)| \geq |D| + |\lambda(P)| \ = \ NT + |\lambda(P)| \ = \ Nk(N+1)^2 + |\lambda(P)|$$

Dividing both sides by $|\lambda(P)|$ we get:

$$\frac{|A(P)|}{|\lambda(P)|} \geq \frac{Nk(N+1)^2}{|\lambda(P)|} + \frac{|\lambda(P)|}{|\lambda(P)|}$$

Since (as shown above) $|\lambda(P)| \leq (k+2)(N+1)^2$, we can substitute to obtain:

$$\frac{|A(P)|}{|\lambda(P)|} \geq \frac{Nk(N+1)^2}{(k+2)(N+1)^2} + 1$$

and simplify to:

$$\frac{|A(P)|}{|\lambda(P)|} \geq \frac{kN}{k+2} + 1$$

Since we have assumed that $N \geq (k+2)$, this gives the required contradiction that the approximation factor is greater than k.

Thus, we have proved that if there is a k-approximate algorithm for precise, flow-insensitive points-to-analysis, we can solve the Hamiltonian Path problem in polynomial time, which can only be true if P=NP.

5 Conclusions

Although there have been previous results on the complexity of various versions of the pointer-analysis problem, and previous results on approximation algorithms for other kinds of problems, to our knowledge this is the first attempt to study the approximability of pointer analysis. Unfortunately, our results are negative: we have shown that there can be no k-approximate algorithms for flow-insensitive points-to analysis, unless P=NP. In the case of flow-sensitive analysis, we have proved a stronger result: that there cannot even be *poly-k*-approximate algorithms, unless P=NP. Our proofs involve showing that if such algorithms exist, known NP-Hard problems can be solved in polynomial time.

An interesting open problem is whether there are *poly-k*-approximate algorithms for the flow-insensitive case. In Section 4, the variables used in the programs do not have well-defined types. It would be of interest to generalize the non-approximability results even when the variables are required to have well-defined types.

References

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[Hor97] S. Horwitz. Precise flow-insensitive alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1), January 1997.

[Lan92a] W. Landi. *Interprocedural Aliasing in the Presence of Pointers*. PhD thesis, Rutgers University, January 1992.

[Lan92b] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[LBS+98] W. Landi, B.Ryder, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural side effect analysis with pointer aliasing. Technical Report DCS-TR-336, Department of Computer Science, Rutgers University, May 1998.

[LR91] W. Landi and B.G. Ryder. Pointer induced aliasing: A problem classification. In *ACM Symposium on Principles of Programming Languages*, pages 93–103, January 1991.

[Pap94] C. Papadimitriou. *Computational Complexity.* Addison Wesley Publishing Company, 1994.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.