New Results on the Computability and Complexity of Points – to – Analysis

Venkatesan T. Chakaravarthy
Computer Sciences Department
University of Wisconsin–Madison
1210, West Dayton Street, Madison, WI 53706, USA.
venkat@cs.wisc.edu

ABSTRACT

Given a program and two variables p and q, the goal of points-to analysis is to check if p can point to q in some execution of the program. This well-studied problem plays a crucial role in compiler optimization. The problem is known to be undecidable when dynamic memory is allowed. But the result is known only when variables are allowed to be structures. We extend the result to show that, the problem remains undecidable, even when only scalar variables are allowed. Our second result deals with a version of points-to analysis called flow-insensitive analysis, where one ignores the control flow of the program and assumes that the statements can be executed in any order. The problem is known to be NP-Hard, even when dynamic memory is not allowed and variables are scalar. We show that when the variables are further restricted to have well-defined data types, the problem is in P. The corresponding flow-sensitive version, even with further restrictions, is known to be PSPACE-Complete. Thus, our result gives some theoretical evidence that flow-insensitive analysis is easier than flow-sensitive analysis. Moreover, while most variations of the points-to analysis are known to be computationally hard, our result gives a rare instance of a non-trivial points-to problem solvable in polynomial time.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers,optimization*

General Terms

Languages, Theory

Keywords

Pointer analysis, flow-sensitive, flow-insensitive, complexity, undecidability

POPL'03, January 15–17, 2003, New Orleans, Louisiana, USA.

1. INTRODUCTION

Modern compilers use static analysis to optimize and speed-up programs. The analysis gets complicated for languages that support pointers (such as C). In order to analyze a program that involves pointers, it is necessary to know what each variable may point to at a given program statement. For example, consider the following segment of a program.

1. x = 3;
2. *p = 5;
3. y = 4*x;

If we know that p can never point to x at statement (2), we can optimize the program by changing statement (3) to be "y=12". This may lead to further optimization possibilities. Thus points-to analysis is useful in static analysis and optimization of programs. Two types of points-to analysis are prevalent: *flow-sensitive* and *flow-insensitive*. This paper deals with the theoretical aspects of both versions of the problem.

In the flow-sensitive points-to analysis problem, we are given the control flow graph of a program, a pair of variables p and q and a location in the program. The goal is to check if there is some path in the control flow graph such that executing the statements along the path makes p point to q at the given program location. We will discuss the semantics of the programming language in Section 3. Briefly, the program is allowed to use the usual assignment statements, branching statements and loops. As is traditionally done, we make the conservative assumption that all paths in the program are executable. Thus, we ignore the conditions in the branching statements.

The points-to problem has been well studied, in terms of both theory and practice. The theoretical results have dealt with the complexity of the problem, whereas approximation algorithms have been proposed in practice.

Throughout the following discussion, we assume that the input program is made of a single procedure (with no procedure calls). Even with such a restriction, various versions of the problem are known to be undecidable or computationally hard. Thus the assumption, in fact, gives us stronger hardness results.

Our first result deals with classifying various special cases of the flow-sensitive analysis. The special cases are obtained by placing restrictions on the input program. Firstly, we can either allow or disallow the program to use dynamic memory. Secondly, we can either restrict the program to use only scalar variables (e.g. int, int*, int** etc.) or allow non-scalar variables (i.e. arrays and structures) as well. Landi [7]

and Ramalingam [10] proved the following result.

THEOREM 1. *([7, 10]) Single procedural flow-sensitive points-to analysis with dynamic memory and non-scalar variables is undecidable.*

Our first result improves the above undecidability result to

THEOREM 2. *Single procedural flow-sensitive points-to analysis with dynamic memory is undecidable, even when only scalar variables are allowed.*

The new result gains some interest when we interpret it with the known results for the other case where dynamic memory is not allowed. In this case, the problem is decidable, as the configuration space of pointers becomes finite. But, Landi [6] and Muth and Debray [9] proved a very strong PSPACE-Hardness result for this case. To state their result, let us restrict the problem further. First, we assume that the variables are scalars. Next, we assume that the variables have well-defined data types and that the programs conform to typing rules. For example, a pointer of type $int**$ can only point to a variable of type $int*$. Furthermore, we restrict that the number of levels of dereferences in the types is just 2. For example, the variables can only be of type int, $int*$ or $int**$. Now we are ready to state the PSPACE-Hardness result:

THEOREM 3. *([6],[9])* [1] *Single procedure flow-sensitive points-to analysis is PSPACE-Complete, even when all variables are scalars with well defined types and only two levels of dereferencing in the types are allowed.*

The only possible further restrictions are either to allow only one level of dereferencing or to disallow the use of pointers! As the second case does not allow use of any pointers, we ignore it! The first case is known to be solvable in polynomial time [6].

Theorem 3 combined with Theorem 1 gives us a very good understanding of the theory of flow-sensitive analysis. The only case left open is to allow dynamic memory, but restrict variables to be scalars. Theorem 2 shows that this case is also undecidable. We can now summarize the results as Figure 1.

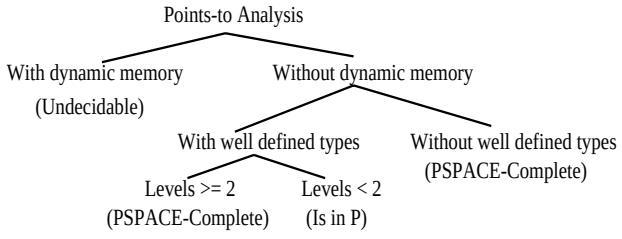

Figure 1: Flow sensitive points-to analysis - a classification

Our second result deals with *flow-insensitive* analysis. In this type of analysis, we ignore the control flow of the program and assume that the statements can be executed in any order (with possible repetition). Thus flow-insensitive analysis is a special case of flow-sensitive analysis, where the input control flow graph is a complete graph.

As flow-sensitive analysis is hard, there have been attempts to do flow-insensitive analysis. Many algorithms for approximate flow-insensitive analysis have been proposed [1, 2, 3, 11, 12, 13]. These algorithms compute a *safe* solution to the problem: given the input program and a pair of pointers (p, q), if there is an execution that makes p point to q, the algorithm would definitely accept the pair. But, the algorithm may accept the pair (p, q), even though there may not be any execution that makes p point to q. Thus, the algorithm compute an approximate solution (with one sided error). In this paper, we focus on the complexity of computing the exact solution to the problem.

One important result in this context is the following.

THEOREM 4. *(Horwitz [5]) Even when dynamic memory is not allowed and variables are scalar, flow-insensitive analysis is NP-Hard, provided arbitrary number of dereferences is allowed in pointer expressions* [2]

Except for this result, unfortunately, not much is known about the computability and complexity of the problem. Several questions remain open: Is the above problem in NP? Is the problem decidable, if we allow dynamic memory? Can the problem be solved in polynomial time, if we bound the number of dereferences allowed in pointer expression? A positive answer to the last question will have practical implications, because in real world programs, a pointer expression like "$***\ldots*p$" uses only a limited number of dereferences. Thus, unlike the flow sensitive case, the complexity of flow-insensitive case is less understood. In this scenario, our second result aims to shed some light on the complexity of flow-insensitive analysis. We show that the problem stated in Theorem 4 can be solved in polynomial time, if we add the restriction that the variables have well-defined data types.

THEOREM 5. *When dynamic memory is not allowed and the variables are scalars with well defined types, the flow insensitive analysis can be solved in polynomial time, even when arbitrary number of dereferences are allowed in an expression.*

The above theorem is interesting for a few reasons. First of all, from our discussion so far, we can see that there are only negative results on points-to analysis problems. The theorem shows that there is a somewhat-natural, non-trivial points-to problem solvable in polynomial time. Secondly, though flow-insensitive analysis (being a special case) seems to be easier than flow-sensitive analysis, there is little theoretical evidence for this thesis. The new result, when compared with Theorem 3, gives us a version of the problem where flow-insensitive analysis is easier than flow-sensitive analysis. When the variables have well-defined types, flow-insensitive analysis can be solved in polynomial time (for arbitrary levels of dereferences in types), whereas flow-sensitive analysis is PSPACE-Complete (even for just

[1]The above result was proved for four levels in [6] and was improved to two levels in [9].

[2]The issue of number of dereferences in an expression did not arise in the context of flow-sensitive analysis, because there, using temporary variables, we can break an expression into a sequence of statements that use one level of dereferencing. But in flow insensitive analysis, breaking statements may not preserve the solution.

two levels). Finally, ideas used in the proof of the theorem may be useful in solving the open questions.

The rest of the paper is organized as follows. In Section 2, we prove Theorem 5 that deals with flow-insensitive analysis. In Section 3, we prove Theorem 2 that deals with flow-sensitive analysis. We state a few open problems and conclusions in Section 4.

2. FLOW-INSENSITIVE ANALYSIS WITH WELL DEFINED TYPES

In this section, we give a polynomial time algorithm for the flow-insensitive analysis with well-defined types and thereby prove Theorem 5. We first consider the problem without well-defined types and briefly discuss the difficulties in solving it (Section 2.1). There, we also develop some notation used throughout the paper. Then, we discuss how these difficulties can be handled when the input program has well-defined types and present a high level overview of the algorithm (Section 2.2). We then solve two subproblems in Sections 2.3 and 2.4 that are used in our algorithm. Finally, we present our algorithm in Section 2.5. Throughout the discussion, we use notations from the C language.

2.1 Flow-Insensitive Analysis Without Types

In this problem, we are given a set of *pointers* (or *variables*) $\{p_1,p_2,\ldots,p_n\}$, a set of statements S and a pair of pointers p and q. A statement can be of two types: (i) $***\ldots*p_i = \&p_j$ or (ii) $***\ldots*p_i = ***\ldots*p_j$, for some variables p_i and p_j. Given the set of pointers, the set of statements and two pointers p and q, the goal is to check if there is a finite sequence $s = s_1, s_2, \ldots, s_r$ of the input statements from S, such that at the end of executing these statements (in order) p points to q. We then say that the sequence s makes p point to q. In such a sequence, a single statement can be used any number of times. We assume the usual semantics of using pointers. In the beginning of any execution all the variables point to a special NULL value. We do not consider sequences that dereference the NULL value any time during their execution. That is, by definition, such a sequence cannot make p point to q.

As stated in Theorem 4, the above problem is known to be NP-Hard. Here, we discuss the problem and some known approximation algorithms. Our aim is to highlight the difficulties involved in solving the problem. Later, we shall see how these difficulties can be overcome if add the typing restrictions to the problem. We first develop some notation.

Notation: We denote the expression $***\ldots*x$, with i stars, as $*^ix$. If a sequence of input statements from S makes x point to y then we say that the sequence *realizes* the pair $\langle x, y\rangle$. If some such sequence exists, we say that the pair $\langle x, y\rangle$ is *realizable*. Observe that, the goal of the problem is to check if the input pair $\langle p, q\rangle$ is realizable. We say that a set $\{\langle x_1, y_1\rangle, \langle x_2, y_2\rangle, \ldots, \langle x_k, y_k\rangle\}$ is realizable, if there is a sequence of input statements from S that makes each x_i point to y_i (simultaneously). As a special case, if a set of pointers $\{\langle x_1, x_2\rangle, \langle x_2, x_3\rangle, \langle x_3, x_4\rangle \ldots, \langle x_{k-1}, x_k\rangle\}$ is realizable, we say that the *chain*

$$x_1 \rightarrow x_2 \rightarrow \cdots \rightarrow x_k$$

is realizable.

Though we are mainly interested in the realizability of the input pair, $\langle p, q\rangle$, it is useful to consider the set of all realizable pairs. For the given set of input statements S, we denote its solution as

$$\lambda(S) = \{\langle x, y\rangle | \text{ the pair } (x, y) \text{ is realizable.}\}$$

It is often convenient to view the solution as a directed graph: the pointers form the vertices of the graph and $\lambda(S)$ serves as the edge set. In other words, (x, y) will be an edge in the graph iff the pair $\langle x, y\rangle$ is realizable. We call the graph the *realizability graph* of the set of input statements S.

Example 1 Consider a set of pointers $P = \{a, b, c, d\}$ and the set of statements

$$S = \{a = \&c, c = \&b, b = \&a, b = a, *b = c, d = *a\}$$

It is easy to see that $\langle a, c\rangle$, $\langle c, b\rangle$ and $\langle b, a\rangle$ are realizable. The sequence $a = \&c, b = a$ realizes the pair $\langle b, c\rangle$. The sequence $c = \&b, b = \&a, *b = c$ realizes the pair $\langle a, b\rangle$. The following sequence would realize the pair $\langle d, a\rangle$:

$$c = \&b, b = \&a, *b = c, b = \&a, d = *a$$

One can construct sequences that realize the pairs $\langle d, b\rangle$ and $\langle b, b\rangle$. One can argue that no other pair is realizable. In particular, note that the pair $\langle d, c\rangle$ is not realizable. So, the solution is

$$\lambda(S) = \{\langle a, c\rangle, \langle c, b\rangle, \langle b, a\rangle, \langle a, b\rangle, \langle b, c\rangle, \langle b, b\rangle, \langle d, a\rangle, \langle d, b\rangle\}$$

The realizability graph is shown in Figure 2 □

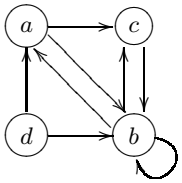

Figure 2: Realizability graph of Example 1

As the problem is NP-Hard, approximation algorithms have been proposed by Andersen [1], Steensgaard [12], and Shapiro and Horwitz [11], to name a few. These algorithms compute a *safe* solution T to a given set of input statements S. We say that T is a safe solution if T is a superset of the exact solution $\lambda(S)$. For example, the set of all pairs of pointers is a safe solution. The accuracy of a solution is measured in terms of its size: smaller the better. The above algorithms consider a restricted version of the problem where only one dereferencing is allowed in any input statement. That is, the only statements allowed are of the form $x = \&y$, $x = y$, $*x = y$ and $x = *y$. The idea is that the given set of statements S can be transformed into a new set of statements S'. We "break" a more general statement $***\ldots*x = y$ in S into a sequence of statements that use at most one dereferencing by using temporary variables. In flow-sensitive analysis, one can do the above transformation without loss of accuracy. But in flow-insensitive analysis, it may lead to some loss of accuracy, because extra pairs of pointers may be realizable in the transformed program. However, the transformation will not compromise safety, because $\lambda(S) \subseteq \lambda(S')$. The three algorithms provide tradeoff between accuracy and running time. We briefly discuss Andersen's algorithm.

Given the set of input statements S, we start with an empty solution $T = \phi$. We first consider direct assignment statements of the form $x = \&y$ and add the pair $\langle x, y\rangle$ to T. Then, we executte the following procedure iteratively, and

add more pairs to T. If $x = y$ is an input statement in S and $\langle y, z\rangle \in T$, we add the pair $\langle x, z\rangle$ to T. If $x = *y$ is in S and $\langle y, w\rangle$ and $\langle w, z\rangle$ are in T, we add $\langle x, z\rangle$ to T. Similarly, if $*x = y$ is in S, and $\langle x, w\rangle$ and $\langle y, z\rangle$ are in T, we add $\langle w, z\rangle$ to T. We stop when no more pairs can be added to T.

It is easy to see that Andersen's algorithm will output a safe solution. However, it may not be the exact solution. The issue is that there may be pairs $\langle y, w\rangle$ and $\langle w, z\rangle$ that are realizable individually, but not simultaneously (i.e. $\langle y, w\rangle$ and $\langle w, z\rangle$ are realizable, but $\{\langle y, w\rangle, \langle w, z\rangle\}$ is not realizable). Suppose such pairs exist and $x = *y$ is an input statement. Then, the algorithm would include $\langle x, z\rangle$ in its solution T, even though $\langle x, z\rangle$ may not be realizable. In the above situation, informally, we say that the realizable pairs $\langle y, w\rangle$ and $\langle w, z\rangle$ are "dependent" on each other. For instance, consider the set of statements S presented in Example 1. We see that $\langle a, b\rangle$ and $\langle b, c\rangle$ are realizable individually. But they are not realizable simultaneously. So the pairs are dependent. Consider running Andersen's algorithm on Example 1. As pairs $\langle a, b\rangle$ and $\langle b, c\rangle$ are realizable, and $d = *a$ is an input statement, the algorithm would include $\langle d, c\rangle$ in its solution. But $\langle d, c\rangle$ is not actually realizable (i.e. $\langle d, c\rangle \notin \lambda(S)$).

The above discussion shows that in order to compute the exact solution, we have to handle the dependencies among the realizable pairs more carefully (at least, when one tries to extend Andersen's algorithm to compute exact solutions). It remains open whether one can do that efficiently. Recall that, the general problem where arbitrary number of dereferencing is allowed in an input statement is NP-Hard (Theorem 4). But, when the number of dereferences allowed is bounded (say, only one dereferencing per statement is allowed), no such hardness result is known. Hence, one may be able to compute the exact solution in polynomial time. It is a challenging open problem. We believe that some mechanism to handle the dependencies correctly and efficiently may lead to a polynomial time algorithm.

In this paper, we show that when the pointers are required to have well-defined types, we can deal with the dependencies correctly and efficiently (in polynomial time). Moreover, we can do that even when the input statements use arbitrary number of dereferences. We do not "break" such statements and hence obtain exact solutions. The rest of the section presents a polynomial time algorithm for the flow-insensitive analysis with types.

2.2 Flow-Insensitive Analysis With Types

This problem is defined by restricting the general flow-insensitive analysis (discussed in Section 2.1) to programs with well-defined types. A more precise definition follows.

We are given a set of *pointers* (or *variables*) $\{p_1, p_2, \ldots, p_n\}$, a set of statements S and a pair of pointers p and q. A statement can be of two types: (i) $***\ldots*p_i = \&p_j$ or (ii) $***\ldots*p_i = ***\ldots*p_j$, for some variables p_i and p_j. All variables are scalars with well-defined types (e.g. int, int*, int** etc.). A variable can point to only variables of compatible type. For example, a variable of type $int**$ can only point to a variable of type $int*$. We identify the type of a variable by the number of dereferences used. For example, variables of type int and $int**$ are said to have types 0 and 2 respectively. The type information is part of the input, given by a function Type(). The statements satisfy the typing rule mentioned above. For example, if $p_i = \&p_j$ is a statement, then Type(p_i) should be Type(p_j)+1. Similarly, if $***p_i = *p_j$ is a statement, then Type(p_i) should be Type(p_j)+2. Given the set of pointers, the set of statements and two pointers p and q, the goal is to check if there is a finite sequence $s = s_1, s_2, \ldots, s_r$ of the input statements, such that at the end of executing these statements p points to q. We then say that the sequence s makes p point to q. In such a sequence, a single statement can be used any number of times.

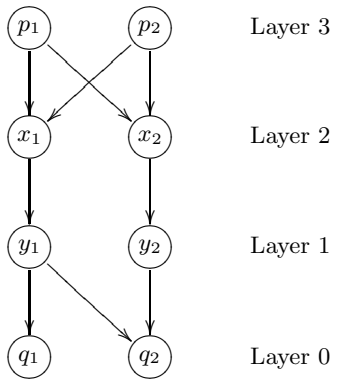

Figure 3: Realizability graph of Example 2

Example 2 Input variables are $\{p_1, p_2, x_1, x_2, y_1, y_2, q_1, q_2\}$ and their types are:

$$Type(p_1) = Type(p_2) = 3$$
$$Type(x_1) = Type(x_2) = 2$$
$$Type(y_1) = Type(y_2) = 1$$
$$Type(q_1) = Type(q_2) = 0.$$

The set of input statements is:

$$\{p_1 = \&x_1, \quad p_2 = \&x_2,$$
$$x_1 = \&y_1, \quad x_2 = \&y_2,$$
$$y_1 = \&q_1, \quad y_2 = \&q_2,$$
$$p_1 = p_2, \quad p_2 = p_1,$$
$$**p_1 = **p_2\}$$

The realizability graph is shown in Figure 3.

In the rest of the section, we make some preliminary observations and present an overview of our polynomial time algorithm for flow-insensitive analysis with types. The algorithm will be presented in detail in Section 2.5.

As the variables have well-defined types, the realizability graph will be acyclic. Moreover, the variables will be arranged into "layers". For instance, variables of type int will be in layer 0, those of type $int*$ will be in layer 1, and so on. In general, a variable of type i will be in layer i. Furthermore, a pointer of type i can only point to a pointer of type $i-1$. Hence, any edge in the realizability graph can only be from layer i to $i-1$ (for some i). Thus, the realizability graph will always be layered:

DEFINITION 1. *(Layered DAG) A directed graph $G = (V, E)$ over n vertices is said to be layered if there is a layering function $l : V \mapsto [0..(n-1)]$ such that for all edges $(u, v) \in E$, $l(v) = l(u) - 1$. We say that a vertex u is in layer $l(u)$ and an edge (x, y) is in layer $l(x)$. Note that layered graphs are acyclic.*

We next observe that pointers in a layer r are not useful in realizing any edge in a layer $l > r$. In other words, if x is a pointer in layer l, and $\langle x, y\rangle$ is a realizable pair, then there is an execution sequence that realizes this pair without using any pointer in a layer $r < l$. As a consequence, if $\langle x, y\rangle$ and $\langle u, v\rangle$ are realizable (individually) and $layer(x) \neq layer(u)$, then $\{\langle x, y\rangle, \langle u, v\rangle\}$ is realizable [3]. Thus, edges in different layers can never be dependent. (Later, we will argue this claim in more detail). However, the claim does not preclude the possibility of edges in the same layer being dependent. For instance, in Example 2, $\langle p_1, x_2\rangle$ and $\langle p_2, x_1\rangle$ are realizable individually, but $\{\langle p_1, x_2\rangle, \langle p_2, x_1\rangle\}$ is not realizable. Thus, even though type restrictions do not eliminate dependencies totally, they do make it easier to handle the dependencies. As dependent pair of edges in the same layer will be important in our algorithm, we give them a special name: we call a pair of edges (x, y) and (u, v) in the same layer a *forbidden pair* if $\langle x, y\rangle$ and $\langle u, v\rangle$ are realizable individually, but they are not realizable simultaneously.

Next we extend the above claim as follows. If individual links of a chain are realizable, then the chain is realizable. That is, if each of $\langle p_1, p_2\rangle$, $\langle p_2, p_3\rangle$,$\langle p_3, p_4\rangle$, $\ldots$,$\langle p_{k-1}, p_k\rangle$ is realizable, then the chain

$$p_1 \rightarrow p_2 \rightarrow \cdots \rightarrow p_k$$

is realizable.

With these observations, the idea is to compute the realizability graph layer by layer, starting with the highest layer. Suppose, by induction, we have computed all layers from the highest down to layer $l+1$ and we want to compute edges in layer l. We first consider statements of the form $*^d p = \&z$ (with $d \geq 0$). For each x in layer l, if there is a path of length d from p to x (in the graph constructed so far) then we add the edge (x, y).

Handling copying statements of the form $*^{d_1} p = *^{d_2} q$ is more tricky. Let x and y be pointers in layer l. It is tempting to conclude that, if there is a path of length d_1 from p to x and a path of length d_2 from q to y, then values of y can be copied to x. This is not true in general. We must also make sure that the two paths can be realized simultaneously. That involves checking for two necessary conditions. First, the paths should be vertex disjoint. Second, any two edges in the same layer used by the paths must be realizable simultaneously. So, it is not enough if we compute only the edges in each layer. In each layer, we also need to know which pair of edges are realizable simultaneously and which are forbidden. Now we are faced with two new subproblems. First, we should be able to check for existence of vertex disjoint paths that do not use forbidden pairs. Second, we should also be able to compute forbidden pair of edges in each layer. We formalize the first problem and solve it in Section 2.3. As for the second problem, the set of forbidden pairs in a layer is determined solely by how copy propagation works in that layer. We formalize the problem as *concurrent copy propagation* and solve it in Section 2.4.

2.3 Disjoint Paths With Forbidden Pairs

In this problem, we are given a directed graph G, a pair of source vertices s_1 and s_2, a pair of target vertices t_1 and t_2, and a set of pairs of edges $F \subseteq E \times E$. We call the pairs in F as forbidden pairs. The goal of the problem is to check if there are two paths p_1 and p_2 such that i) p_1 is a path from s_1 to t_1 and p_2 is a path from s_2 to t_2; ii) p_1 and p_2 are vertex disjoint (i.e., they do not share any vertex); iii) for any forbidden pair $(e_1, e_2) \in F$, if p_1 uses e_1 then p_2 does not use e_2.

The well-known disjoint paths problem is a special case of the above problem where there are no forbidden pairs (i.e. $F = \phi$). The problem is known to be NP-Complete [4]. Here we consider a special case of the problem where the input graph is layered and for any forbidden pair $\langle e_1, e_2\rangle$, the edges e_1 and e_2 are in the same layer of the graph. We show that this restriction can be solved in polynomial time.

THEOREM 6. *The problem of disjoint paths with forbidden pairs can be solved in polynomial time, if the input graph is layered and each forbidden pair of edges appear in the the same layer.*

PROOF. We reduce the given problem to a question of reachability in directed graphs. Let the input instance consist of a layered graph $G = (V, E)$ with a layering function l, source vertices s_1 and s_2, target vertices t_1 and t_2, and a set of forbidden pairs F. Without loss of generality, we assume that the two source vertices are in the same layer. (Suppose s_2 is in a layer higher than s_1. We can add new vertices $u_1, u_2, \ldots, u_d$ to G, where $d = l(s_2) - l(s_1)$, and add $d-1$ edges $(u_1, u_2), (u_2, u_3), \ldots, (u_{d-1}, u_d)$, creating a chain. Then we add the edge (u_d, s_1) and make u_1 as a source vertex, instead of s_1.) Similarly, assume that the two target vertices are in the same layer. Now we construct a new graph $G' = (V', E')$. $V' \subseteq V \times V$ consists of pairs of distinct vertices in the same layer of G. Two vertices in G' are connected by an edge if the corresponding components of these vertices are connected by a pair of non-forbidden edges in G. Formally let,

$$\begin{aligned} V' &= \{\langle u, v\rangle | u, v \in V, u \neq v, \text{ and } l(u) = l(v)\} \\ E' &= \{(\langle u_1, v_1\rangle, \langle u_2, v_2\rangle) | \langle (u_1, u_2), (v_1, v_2)\rangle \in E \times E - F\} \end{aligned}$$

It is clear that G' is also layered.

Now we show that the given instance has a solution iff there is a path from $\langle s_1, s_2\rangle$ to $\langle t_1, t_2\rangle$ in G'. Suppose the given instance has a solution via the disjoint paths $p_1 = u_1, u_2, \ldots, u_k$ and $p_2 = v_1, v_2, \ldots, v_k$, where $u_1 = s_1, u_k = t_1, v_1 = s_2$ and $v_k = t_2$. We made sure that $l(s_1) = l(s_2)$ and $l(t_1) = l(t_2)$. So for any i, $l(u_i) = l(v_i)$. As the paths are vertex disjoint, $u_i \neq v_i$. Hence, $\langle u_i, v_i\rangle \in V'$, for all $1 \leq i \leq k$. Moreover, for $1 \leq i < k$, the pair of edges (u_i, u_{i+1}) and (v_i, v_{i+1}) is not forbidden. Hence, $(\langle u_i, v_i\rangle, \langle u_{i+1}, v_{i+1}\rangle) \in E'$. So $p = \langle u_1, v_1\rangle, \ldots, \langle u_k, v_k\rangle$ is a path from $\langle s_1, s_2\rangle$ to $\langle t_1, t_2\rangle$ in G'.

For the other direction, assume that there is a path $p = \langle u_1, v_1\rangle, \langle u_2, v_2\rangle, \ldots, \langle u_k, v_k\rangle$ from $\langle s_1, s_2\rangle$ to $\langle t_1, t_2\rangle$ in G'. Consider the paths $p_1 = u_1, u_2, \ldots, u_k$ and $p_2 = v_1, v_2, \ldots, v_k$. It is clear that p_1 and p_2 are indeed legitimate paths in G, from s_1 to t_1 and s_2 to t_2, respectively. We first show that they are vertex disjoint. By our construction and as G is layered, for $i \neq j$, $u_i \neq v_j$. Moreover, as $\langle u_i, v_i\rangle \in V'$, $u_i \neq v_i$. Hence the paths are vertex disjoint. To show that no forbidden pair is used in p_1 and p_2, note that a forbidden pair of edges will be in the same layer (by problem definition). The presence of the edge $(\langle u_i, v_i\rangle, \langle u_{i+1}, v_{i+1}\rangle)$ implies that the pair of edges (u_i, u_{i+1}) and (v_i, v_{i+1}) are not forbidden. Hence p_1,p_2 is a valid solution to our problem.

[3]This claim fails if the pointers do not have well-defined types, or if the goal is to do flow-sensitive analysis.

Thus we have reduced the given instance to a question of reachability in G'. Size of G' is polynomial in size of G and reachability can be solved in polynomial time (say, via depth first search). Our reduction algorithm runs in polynomial time. Hence disjoint paths with forbidden pairs on layered graphs can be solved in polynomial time. □

2.4 Concurrent Copy Propagation Problem

k-Concurrent Copy Propagation Problem (k-CCP): Let k be a positive integer. In the k-CCP problem, we are given a set of *variables* V, a set of *constants* (or values) C and a set of *statements* S. We represent variables by capital letters and constants by small letters. A statement can be of two types: (i) $X := a$, for some $X \in V$ and $a \in C$ or (ii) $X := Y$, for some $X, Y \in V$. Executing the former statement assigns the constant a to X, whereas executing the latter statement copies the current value of Y to X. Consider a set of *goals* $G = \{\langle X_1, a_1\rangle, \langle X_2, a_2\rangle, \ldots, \langle X_k, a_k\rangle\}$, where X_i are distinct. We say that G can be *realized* if there is some finite sequence $s_1, s_2, \ldots, s_r$, with each $s_i \in S$, such that at the end of executing this sequence, for each $1 \leq i \leq k$, the value X_i is a_i. The solution λ of k-CCP is the set of all such realizable sets of k goals:

$$\lambda = \{s | s \text{ is a realizable set of size } k\}$$

If k can be arbitrary (and is part of the input) the decision version of the problem is known to be PSPACE-Complete [9]. But when k is a constant, k-CCP can be solved in polynomial time. For our purposes, we only need the cases where k is either 1 or 2. When $k = 1$, it is easy to see that the problem can be reduced to a question of reachability in directed graphs and hence can be solved in polynomial time.

THEOREM 7. *The 1-CCP problem can be solved in polynomial time.*

THEOREM 8. *The 2-CCP problem can be solved in polynomial time.*

PROOF. Given a set of variables V, constants C and statements S, we compute the solution λ using the idea of transitive closure. We start with

$$\lambda = \{\{\langle X, x\rangle, \langle Y, y\rangle\} | X \neq Y \text{ and } X := x \text{ and } Y := y \text{ are statements in } S\}$$

Now we iterate and add more elements to λ. For each element $\{\langle X, x\rangle, \langle Y, y\rangle\}$ in λ,

1. For each statement $Z := z$ in S, with $Z \neq X$, add $\{\langle X, x\rangle, \langle Z, z\rangle\}$ to λ.
2. For each statement $Z := X$, with $Z \neq X$, add $\{\langle X, x\rangle, \langle Z, x\rangle\}$ to λ.
3. For each statement $Z := X$, with $Z \neq Y$, add $\{\langle Z, x\rangle, \langle Y, y\rangle\}$ to λ.

A variable cannot hold two constants simultaneously. The inequality checks in the above procedure are to ensure this property. Now we apply the above procedure iteratively and stop when we cannot add any more elements to λ. The boundary condition in which X is the only variable that gets a constant assigned to it directly has to be handled separately. The details are omitted for brevity.

λ can have at most $|V|^2 \times |C|^2$ elements. Thus the above algorithm would need no more than that many iterations. Hence it runs in polynomial time. One can prove the correctness of the algorithm using induction. □

2.5 Main Algorithm

Given a set of pointers $\{p_1, p_2, \ldots, p_n\}$ (with types) and a set of statements S, we describe a polynomial time algorithm to compute the realizability graph. Let the number of types used by the pointers be L. Recall that the graph would be layered with $L + 1$ layers numbered $0, 1, \ldots, L$. We will find the edges of the graph layer by layer, from L down to 1. For each layer i, we will also compute the set F_i of all forbidden pairs of edges. By induction, assume that we have computed the edges and set of forbidden pairs for layers L down to $l + 1$ correctly and we want to compute the same information for layer l.

The computation of edges and forbidden pairs in a given layer l of G consists of four phases:

1. Direct Assignments: Analyze each statement of the form $*^d p = \&z$ and find the variables q, in level l, that can be made to point to z by this statement.
2. Copying Statements: Analyze each statement of the form $*^{d_1} p_1 = *^{d_2} p_2$ and find pairs of variables q_1 and q_2, in level l, such that values of q_2 can be copied to q_1 using this statement.
3. Edge Computation: Compute edges in layer l in this phase.
4. Forbidden Pair Computation: Compute the forbidden pairs of edges in layer l.

Direct Assignments: We consider each (direct assignment) statement of the form $*^d p = \&z$. As variables in layer l can only point to those in layer $l - 1$, we ignore the statement if $layer(z) \neq l - 1$. So, assume that z is in layer $l - 1$. Then, we claim that if there exists a path of length d from p to q (in the graph constructed so far) then $\langle q, z\rangle$ is realizable (where, length of a path is the number of edges used). Suppose there is such a path $r_d, r_{d-1}, \ldots, r_0$, where $r_d = p$, $r_0 = q$. We have to exhibit a sequence that realizes the pair $\langle q, z\rangle$. The edges (r_d, r_{d-1}), (r_{d-1}, r_{d-2}), $\ldots$, (r_1, r_0), are all in some layer higher than l. Thus, by our induction hypothesis, each of these edges is realizable. So there exist execution sequences, E_1, E_2,... E_d that realize the pairs $\langle r_1, r_0\rangle$, $\langle r_2, r_1\rangle$,..., $\langle r_d, r_{d-1}\rangle$, respectively. Because of the type restrictions, variables in a layer are not useful in realizing any edge in a higher layer. Hence, without loss of generality, we can assume that, for each $1 \leq i \leq d$, the sequence E_i does not use the variables r_{i-1},r_{i-2},...,r_0. So, executing the sequence E_i will not change the value of r_{i-1},r_{i-2},...,r_0. Now consider the sequence obtained by concatenating E_1,E_2,...,E_d (in that order [4]). Executing this sequence would realize the chain,

$$(p =) r_d \rightarrow r_{d-1} \rightarrow \cdots \rightarrow r_0 (= q)$$

By appending the statement $*^d p = \&z$ to the above sequence we can realize the pair $\langle q, z\rangle$. Thus, we have proved the claim that if there is a path of length d from p to q (in the graph constructed so far), then $\langle q, z\rangle$ is realizable. So, if there is a path of length d exists, we add the edge (q, z).

Copying Statements: Next we consider each (copying) statement of the form $*^{d_1} p_1 = *^{d_2} p_2$. We first check if

[4]Here is where we use the fact that we are doing flow insensitive analysis. If there is a input control flow graph that regulates the ordering among statements, we cannot execute the statements in the order we want.

$layer(p) = l + d_1$ or $layer(p_2) = l + d_2$. If not, the statement is not relevant at layer l and we ignore it. Then, we consider each pair of pointers q_1 and q_2 in layer l. We want to check if the above statement can copy values of q_2 to q_1. It is tempting to say that this is the case if there is a path from p_1 to q_1 and a path from p_2 to q_2. But these paths should satisfy two necessary conditions.

Condition 1: The paths should be vertex disjoint. Suppose the paths are not vertex disjoint. Let the paths be $p_1, \ldots, x, x_1, \ldots, q_1$ and $p_2, \ldots, x, x_2, \ldots, q_2$. At any point during an execution, x can point to either x_1 or x_2, but not both. Hence, though these two chain of pointers can be realized individually, they cannot be realized concurrently. So we need to make sure that the two paths are vertex disjoint.

Condition 2: As forbidden pair of edges cannot be realized simultaneously, we need to make sure that the paths do not use them. That is, for every pair of forbidden pair of edges e_1 and e_2, if one of the paths uses e_1, the other path should not use e_2.

We illustrate the need for the second condition, using Figure 3 and the associated example. That graph has vertex disjoint paths of length two from p_1 to y_2 and p_2 to y_1. But, the pair of edges (p_1, x_2) and (p_2, x_1) cannot be realized simultaneously. So the edges (p_1, x_2) and (p_2, x_1) form a forbidden pair. Hence, even though the two chains $p_1 \rightarrow x_2 \rightarrow y_2$ and $p_2 \rightarrow x_1 \rightarrow y_1$ can be realized individually, they cannot be realized simultaneously. If we do not check for Condition 2, while processing the input statement $**p_1 = **p_2$, we will add the edge (y_2, q_1). But the edge (y_2, q_1) cannot be realized by any sequence.

Based on above discussion, our algorithm to process the statement $*^{d_1}p_1 = *^{d_2}p_2$ is as follows. We consider each pair of pointers q_1 and q_2 in level l. We claim that values of q_2 can be copied to q_1 if there is a path from p_1 to q_1 and a path from p_2 to q_2 such that the paths are vertex disjoint and they respect the forbidden pair of edges in layers L through $l + 1$. Suppose such paths exist. Let the paths be

$$(p_1 =)x_{d_1} \rightarrow x_{d_2-1} \cdots \rightarrow x_0 (= q_1)$$

and

$$(p_2 =)y_{d_2} \rightarrow y_{d_2-1} \rightarrow \cdots \rightarrow y_0 (= q_2).$$

Assume that $d_1 \leq d_2$ (the other case is similar). By our induction hypothesis, there exist execution sequences E_1, E_2,...,E_{d_2} such that, for $d_2 \leq i < d_1$, E_i realizes the pair $\langle y_i, y_{i-1} \rangle$. Moreover, by our induction hypothesis, for each $d_1 \leq i \leq 1$, E_i realizes the pairs $\langle x_i, x_{i-1} \rangle$ and $\langle y_i, y_{i-1} \rangle$ simultaneously, because they are not forbidden. Then the sequence obtained by concatenating E_1,E_2,...,E_{d_2} (in that order) would realize the two chains of pointers. We can copy values of q_2 to q_1 by appending the statement $*^{d_1}p_1 = *^{d_2}p_2$ to the above sequence.

Edge Computation: From the Direct Assignments phase, we have identified the set of pairs $\langle q, z \rangle$ such that q is layer l, z is in layer $l - 1$ and q can be made to point to z. From the Copying Statements phase, we have pairs $\langle q_1, q_2 \rangle$ with both q_1 and q_2 in layer l such that values from q_2 can be copied to q_1. We use the above information to compute the edges in layer l by solving a copy propagation problem. The set of pointers in layer l correspond to the variables for the copy propagation problem and the set of pointers in layer $l - 1$ correspond to the constants. More precisely, construct a set V_{CCP} by adding a variable X_q for each pointer q in layer l. Then construct a set C_{CCP} by adding a constant c_z for each pointer z in layer $l-1$. Define a set of statements S_{CCP} for the copy propagation problem as follows. For each pair $\langle q, z \rangle$ identified in Direct Assignments phase, add a statement $X_q := c_z$ to S_{CCP} and for each pair $\langle q_1, q_2 \rangle$ identified in Copying Statements phase, add a statement $X_{q_1} := X_{q_2}$ to S_{CCP}. Then using the algorithm claimed in Theorem 7 we solve the 1-CCP problem with V_{CCP} as the set of variables, C_{CCP} as the set of constants and S_{CCP} as the set of statement. For each pair $\langle X_q, c_z \rangle$ in the solution to the 1-CCP problem, we add the edge (q, z) to the graph.

Forbidden Pair Computation: If (q_1, z_1) and (q_2, z_2) are edges in layer l, whether they can be realized simultaneously is determined solely by the way copy propagation works in layer l. So using the algorithm claimed in Theorem 8 we solve the 2-CCP problem with V_{CCP} as the set of variables, C_{CCP} as the set of constants and S_{CCP} as the set of statements. For any $\langle \langle X_{q_1}, c_{z_1} \rangle, \langle X_{q_2}, c_{z_2} \rangle \rangle$ in the solution to the 2-CCP problem, we can realize the pairs $\langle q_1, z_1 \rangle$ and $\langle q_2, z_2 \rangle$ simultaneously. Every other pair of realizable edges in layer l is declared forbidden.

The algorithm is presented in Figure 4. It is easy to see that the algorithm runs in polynomial time. Our discussion shows that if the algorithm includes an edge (u, v) in its output graph, then pair $\langle u, v \rangle$ is indeed realizable. One can prove the claim formally, by using an induction on the layer number of (u, v). On the other hand, suppose a $\langle u, v \rangle$ is realizable. Then, we can prove that (u, v) will be an edge in the output graph by using induction on the length of the shortest sequence realizing the pair $\langle u, v \rangle$. Though straightforward, these proofs are somewhat lengthy. For want of space, we omit them.

3. UNDECIDABILITY OF FLOW-SENSITIVE ANALYSIS WITH SCALAR VARIABLES

In this section we prove Theorem 2.

Problem Definition: We are given a set of pointers, a program (or say, its control flow graph) and two pointers p and q. Three types of assignment statements are allowed in the program: (1) $***\ldots*x = \&y$, (2) $***\ldots*x = ***\ldots*y$ and (3) $***\ldots*x =$ NEW . The third statement creates a new unnamed variable and makes $***\ldots*x$ point to it. Two types of control flow statements are allowed: $if(..)then...else...$ and $while(..)$.... Now the goal is to check if there is some path from the start node to the exit node in the control flow graph, such that, at the end of executing the statements along the path, p points to q.

As it is traditionally done, we make the conservative assumption that every path in the program is executable. This means that, no matter through which path we arrived at an $if(..)then...else...$ statement, we can proceed through either the if part or the $else$ part. In other words, we ignore the conditional expression. A similar assumption is made about the WHILE(..) statement. One can see that relaxing this assumption makes the problem even harder.

A variation of the problem where variables could be structures is known to be undecidable. Landi proved this theorem by giving a reduction from the halting problem [7]. Ramalingam gave a simpler proof using a reduction from the Post's correspondence problem [10]. Here we prove the undecidability making use of only scalar variables.

Input: A set of n pointers with L types and a set of statements S.
Output: Realizability graph G of S.

For l from L to 1 do
1. Let $V_{CCP} = \{X_q | q$ is a pointer in layer $l\}$.
2. Let $C_{CCP} = \{c_z | z$ is a pointer in layer $l-1\}$.
3. Let $S_{CCP} = \{\}$.
4. For each statement $*^d p = \&z$, such that, $layer(z) = l-1$ and $layer(p) = l+d$
   For each each pointer q in layer l
   If there is a path from p to q then add the statement ``$X_q := c_z$'' to S_{CCP}.
5. Let $F = \bigcup_{i=l+1}^{L} F_i$
6. For each statement $*^{d_1} p_1 = *^{d_2} p_2$ such that $layer(p_1) = l + d_1$ and $layer(p_2) = l + d_2$
   For each pair of pointers q_1 and q_2 in layer l
   Use algorithm in Theorem 6 to check if there are vertex disjoint paths
   from p_1 to q_1 and p_2 to q_2 that respect the set of forbidden pairs F.
   If such paths exist add the statement ``$X_{q_1} := X_{q_2}$'' to S_{CCP}.
7. Use algorithm for 1-CCP (Theorem 7) with V_{CCP} as variables, C_{CCP} as constants and S_{CCP} as statements and compute the solution λ_{1-CCP}.
8. For each pair $\langle X_q, c_z \rangle$ in λ_{1-CCP}, add the edge (q,z) to G.
9. Use algorithm for 2-CCP (Theorem 8) with V_{CCP} as variables, C_{CCP} as constants and S_{CCP} as statements and compute the solution λ_{2-CCP}.
10. Let $F_l = \{\}$.
11. For each $\langle X_{q_1}, c_{z_1} \rangle$ and $\langle X_{q_2}, c_{z_2} \rangle$ in λ_{1-CCP}, with $q_1 \neq q_2$,
    If $\{\langle X_{q_1}, c_{z_1} \rangle, \langle X_{q_2}, c_{z_2} \rangle\}$ is not in λ_{2-CCP}, add $\{(q_1, z_2), (q_2, z_2)\}$ to F_l.

Figure 4: Algorithm to compute realizability graph

The problem of checking whether a multivariate polynomial has integer roots is known to be undecidable. In this problem, we are given a polynomial $P(x_1, x_2, \ldots, x_n)$ over the variables $x_1, x_2, \ldots, x_n$. A sequence of (positive or negative) integer constants $a_1, a_2, \ldots, a_n$, not all zero, is called an integer root of P if $P(a_1, a_2, \ldots, a_n) = 0$. Given the polynomial, the problem is to check if it has any integer roots. The problem is also known as the Hilbert's tenth problem. Building on the work of Davis, Putnam and Robinson, Matijasevič proved the undecidability of the Hilbert's tenth problem [8].

We prove our undecidability result via a reduction from the above problem. The polynomial $P(x_1, x_2, x_3) = x_1 + x_1x_2 - x_2x_3$ is used as a running example to illustrate our reduction. The output program for this example is given in Appendix A. Here we explain the ideas used. Our output program starts with the following piece of code:

$D = \&Success$;
$Zero = temp =$ NEW ;
WHILE(..) { *temp=NEW ; temp=*temp;}

The first statement makes D point to *Success*. We will make sure that the polynomial has integer roots if and only if there is an execution path in which D remains pointing to *Success* at the exit statement of the program.

This would prove the required undecidability. The next few statements in the above code create a singly linked list with *Zero* as the head. This would simulate the positive integer number line, with the k^{th} node representing integer k.

The next segment of the output program simulates choosing constant values for each variable x_i. The value has a magnitude and a (positive or negative) sign. To choose the magnitude of x_i we use a pointer X_i and traverse the linked list. For our example polynomial, the next segment of code is:

$X_1 = Zero$; WHILE(..) $\{X_1 = *X_1;\}$
$X_2 = Zero$; WHILE(..) $\{X_2 = *X_2;\}$
$X_3 = Zero$; WHILE(..) $\{X_3 = *X_3;\}$

If a path iterates the first loop a_1 times, X_1 would point to the a_1^{th} node in our linked list. This corresponds to assigning $x_1 = a_1$. In general, let $X_1, X_2, \ldots, X_n$ point to nodes $a_1, a_2, \ldots, a_n$ of the linked list. Then, this simulates choosing these values for the variables.

Recall that we want to check if there is a nonzero integer root. Thus, we need to ensure that at least one variable is assigned a nonzero value. The next segment of output our program, ensures that by using a multiway branch with n branches. The code segment for our example would be:

SWITCH(..) {
  CASE: $X_1 = *X_1$;
  CASE: $X_2 = *X_2$;
  CASE: $X_3 = *X_3$;
}

As one of the branches must be executed, not all variables can point to node zero of the linked list.

Next we simulate choosing signs for the variables. As each of the n variables can be positive or negative, the signs can be chosen in 2^n possible ways. We use a multiway branch [5] (a "switch" statement) with 2^n branches to do the simulation. Each branch represents choosing a particular combination of signs for the variables. Consider any one branch with one such fixed combination of signs. Then the sign of any term in the polynomial also gets fixed. The sign of a term is determined by the its sign in the input poly-

[5]The multiway branch can easily be translated into a sequence of $if(..)then...else...$ statements.

nomial and the combination fixed by the branch. In our example, consider the branch that fixes x_1, x_3 to be positive and x_2 to be negative. Then, sign of the term $-x_2x_3$ would be positive. We separate the terms of the polynomial into groups of positive and negative terms and get two polynomials P_1 and P_2. Then, $a_1, a_2, \ldots, a_n$ is an integer root of P iff if $P_1(a_1, a_2, \ldots, a_n) = P_2(a_1, a_2, \ldots, a_n)$. In our example, consider a branch that represents choosing x_1, x_3 to be positive and x_2 to be negative. Now, irrespective of the magnitudes, the terms x_1 and $-x_2x_3$ would be positive, whereas the term x_1x_2 would be negative. So $P_1 = x_1 + x_2x_3$ and $P_2 = x_1x_2$.

Before proceeding further, we define a macro [6] used in the remainder of our program. The macro takes two parameters A and B and checks if they point to the same location:

$\text{ALIAS} - \text{CHECK}(A, B)$:
$temp_1 = *A;$
$temp_2 = *B;$
$*A = \&D;$
$*B = \&dummy;$
$**A = \&Failure;$
$*A = temp_1;$
$*B = temp_2;$

Ignore the first two and the last two statements for the moment. Then, if A and B point to different locations then the variable D would point to $Failure$. On the the other hand, if they point to the same location then D would remain pointing to $Success$ and only a dummy variable will be made to point to $Failure$. The first two and the last two statements ensure that no other variable is affected by this macro.

Recall that we want to check if the polynomials P_1 and P_2 evaluate to the same value. For this purpose, we use two pointers p_1 and p_2. We first set p_1 and p_2 to point to node zero of the linked list. Then we consider the terms of the polynomial one by one. A term would contribute to P_1 if it is positive and to P_2 if it is negative. The sign of the term is determined by two factors: the sign given to the term in the polynomial and which branch of the SWITCH statement we are dealing with. Suppose the term contributes to P_1. In that case we would move p_1 forward on the linked list. If it contributes to P_2, we would move the pointer p_2. In either case, the number of nodes by which the pointer moves is determined by the chosen values $a_1, a_2, \ldots, a_n$. For example, let us take the term x_1x_2. Suppose we are writing code for the branch that represents choosing x_1 and x_3 to be positive and x_2 to be negative. The term x_1x_2 would be negative. So, we move p_2. We need to move it by $a_1 \times a_2$ number of nodes. We use nested loops to achieve this:

$r_1 = Zero;$
WHILE(..) {
    $r_1 = *r_1;$
    $r_2 = Zero;$
    WHILE(..) $\{r_2 = *r_2;\ p2 = *p2;\}$
}

The problem with the above code is that the loops may be executed arbitrary number times. But we want the inner loop to run for exactly a_2 iterations and the outer loop for exactly a_1 times. To ensure this, we use the fact that X_1 and X_2 are pointing to a_1 and a_2 and do alias checks. The new program fragment is:

$r_1 = Zero;$
WHILE(..) {
    $r_1 = *r_1;$
    $r_2 = Zero;$
    WHILE(..) $\{r_2 = *r_2;\ p = *p;\}$
    $\text{ALIAS} - \text{CHECK}(X_2, r_2);$
}
$\text{ALIAS} - \text{CHECK}(X_1, r_1);$

Now either p_2 points to node numbered $a_1 \times a_2$ or D points to $Failure$. Our program will make sure that it never goes back to $Success$.

After evaluating all the terms of the polynomial we finally check whether p_1 and p_2 point to the same location, using the macro $\text{ALIAS} - \text{CHECK}$. Thus D can point to $Success$ at the exit of the program iff the polynomial has integer roots.

4. CONCLUSIONS AND OPEN PROBLEMS

In this paper, we proved two main results. We showed that flow sensitive pointer analysis is undecidable even if the input programs have only scalar variables, provided dynamic memory is allowed. This extends the previously known result that required non-scalar variables as well [7, 10]. Our result presents a complete classification of flow-sensitive analysis as shown in Figure 1.

Unlike flow-sensitive analysis, the complexity of flow insensitive analysis is less understood. When arbitrary levels of dereferencing is allowed in pointer expressions and variables are scalars, the problem in known to be NP-Hard [5]. Except for this result, we know little about the complexity of the problem. In this paper, we showed that the above problem can be solved in polynomial time, with a further restriction that the variables have well-defined types. In contrast, it is known that flow-sensitive analysis is PSPACE-Complete even for two levels of types. Though there has been empirical evidence that flow-insensitive analysis is computationally easier than flow-sensitive analysis, this result gives, to the best of our knowledge, the first theoretical evidence. Moreover, while most of the results pertaining to the points-to analysis problem are undecidability and hardness results, our result is a rare instance of a non-trivial points-to analysis problem solvable in polynomial time.

There are several open problems related to flow-insensitive analysis:

- When dynamic memory is not allowed but arbitrary number of levels of dereferencing is allowed, the problem is NP-Hard [5]. Is it in NP?
- Consider the above problem but with bounded number of dereferences. Is this problem in P?
- When dynamic memory is allowed, is the problem decidable?

5. ACKNOWLEDGMENTS

I thank Jin-Yi Cai, Susan Horwitz, Raghav Kaushik and Rajasekar Krishnamurthy for extensive discussions and helpful comments. I thank Rajasekar Krishnamurthy and the anonymous referees for their suggestions on improving the presentation of the paper. I thank an anonymous referee of an earlier paper for suggesting the problem considered in

[6] Using macros is not an issue, as they can always be expanded.

Theorem 5. This work was supported in part by the National Science Foundation under grants CCR-9634665 and CCR-0208013.

6. REFERENCES

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[2] M. Burke, P. Carini, J.-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Language and Compilers for Parallel Computing, 7th International Workshop*, LNCS 892, pages 234–250. Springer-Verlag, Aug. 1994.

[3] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *ACM Symposium on Principles of Programming Languages*, pages 232–245, Jan. 1993.

[4] S. Fortune, J. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10:111–121, 1980.

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

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

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

[8] Y. Matiyasevič. *Hilbert's 10th Problem*. MIT Press, 1993.

[9] R. Muth and S. Debray. On the complexity of flow-sensitive dataflow analyses. In *ACM Symposium on Principles of Programming Languages*, pages 67–80, 2000.

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

[11] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *ACM Symposium on Principles of Programming Languages*, Jan. 1997.

[12] B. Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages*, pages 32–41, Jan. 1996.

[13] S. Zhang, B. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 81–92, 1996.

APPENDIX

A. APPENDIX A

This Appendix is related to Section 3. Here we take $P(x_1, x_2, x_3) = x_1 + x_1x_2 - x_2x_3$ as an example polynomial and present the complete program output by the reduction given in Section 3. For ease of understanding we use a few macros. The first macro is as follows:

ALIAS − CHECK(A, B) :

$temp_1 = *A$;
$temp_2 = *B$;
$*A = \&D$;
$*B = \&dummy$;
$**A = \&Failure$;
$*A = temp_1$;
$*B = temp_2$;

The macro checks if A and B point to the same node. If they are aliased a dummy variable is made point to $Failure$. If not, it makes D to point to $Failure$. The other pointers A, B, $*A$ and $*B$ are not affected by the macro.

For each term of the polynomial, we define a macro. The macro takes a parameter p. Suppose the the value of the term at the chosen constants is v, Then the macro either moves forward p by v nodes on the linked list or makes D point to $Failure$.

TERM$_1(p)$:
$r_1 = Zero$;
WHILE(..) $\{r_1 = *r_1;\ p = *p;\}$
ALIAS − CHECK(X_1, r_1);

TERM$_2(p)$:
$r_1 = Zero$;
WHILE(..) {
$r_1 = *r_1$;
$r_2 = Zero$;
WHILE(..) $\{r_2 = *r_2;\ p = *p;\}$
ALIAS − CHECK(X_2, r_2);
}
ALIAS − CHECK(X_1, r_1);

TERM$_3(p)$:
$r_1 = Zero$;
WHILE(..) {
$r_1 = *r_1$;
$r_3 = Zero$;
WHILE(..) $\{r_3 = *r_3;\ p = *p;\}$
ALIAS − CHECK(X_3, r_3);
}
ALIAS − CHECK(X_1, r_1);

Now we are ready to present the code:

Variables : $D, Success, Failure$;
Variables : X_1, X_2, X_3;
Variables : p_1, p_2;
Variables : $r_1, r_2, r_3, temp, dummy$;

/* Initialize D */
$D = \&Success$;

/* Setup number line */

$Zero = temp =$ NEW ;
WHILE(..) { *temp=NEW ; temp=*temp;}

/* Choose values */
$X_1 = Zero$; WHILE(..) $\{X_1 = *X_1;\}$
$X_2 = Zero$; WHILE(..) $\{X_2 = *X_2;\}$
$X_3 = Zero$; WHILE(..) $\{X_3 = *X_3;\}$

/* Make sure not all values are zero */

SWITCH(..) {
CASE: $X_1 = *X_1$;
CASE: $X_2 = *X_2$;
CASE: $X_3 = *X_3$;
}
/* Initialize p_1 and p_2 */
$p_1 = Zero$;
$p_2 = Zero$;

/*
Choose signs and evaluate the two polynomials. Each branch chooses a particular combination of signs. In any branch, we consider all the three terms. And move p_1 if the term is positive and move p_2 if the term is negative Whether a term is positive or negative is determined by the sign of term in the input polynomial and the combination of signs represented by the branch.
*/
SWITCH(..) {
CASE: : $\text{TERM}_1(p_1); \text{TERM}_2(p_1); \text{TERM}_3(p_2);$
/*$+X_1, +X_2, +X_3$*/
CASE: : $\text{TERM}_1(p_1); \text{TERM}_2(p_1); \text{TERM}_3(p_1);$
/*$+X_1, +X_2, -X_3$*/
CASE: : $\text{TERM}_1(p_1); \text{TERM}_2(p_2); \text{TERM}_3(p_1);$
/*$+X_1, -X_2, +X_3$*/
CASE: : $\text{TERM}_1(p_1); \text{TERM}_2(p_2); \text{TERM}_3(p_2);$
/*$+X_1, -X_2, -X_3$*/
CASE: : $\text{TERM}_1(p_2); \text{TERM}_2(p_2); \text{TERM}_3(p_2);$
/*$-X_1, +X_2, +X_3$*/
CASE: : $\text{TERM}_1(p_2); \text{TERM}_2(p_2); \text{TERM}_3(p_1);$
/*$-X_1, +X_2, -X_3$*/
CASE: : $\text{TERM}_1(p_2); \text{TERM}_2(p_1); \text{TERM}_3(p_1);$
/*$-X_1, -X_2, +X_3$*/
CASE: : $\text{TERM}_1(p_2); \text{TERM}_2(p_1); \text{TERM}_3(p_2);$
/*$-X_1, -X_2, -X_3$*/
}

/* Finally check if p_1 and p_2 point to same node in the number line */
$\text{ALIAS} - \text{CHECK}(p_1, p_2)$

The polynomial has non-zero integer roots if and only if there is a execution path in the program such that at the last statement D points to $Success$.