On the Complexity of Flow-Sensitive Dataflow Analyses *

Robert Muth Saumya Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721, U.S.A.
{muth, debray}@cs.arizona.edu

Technical Report 99-12

Abstract

This paper attempts to address the question of why certain dataflow analysis problems can be solved efficiently, but not others. We focus on flow-sensitive analyses, and give a simple and general result that shows that analyses that require the use of *relational attributes* for precision must be PSPACE-hard in general. We then show that if the language constructs are slightly strengthened to allow a computation to maintain a very limited summary of what happens along an execution path, inter-procedural analyses become EXPTIME-hard. We discuss applications of our results to a variety of analyses discussed in the literature. Our work elucidates the reasons behind the complexity results given by a number of authors, improves on a number of such complexity results, and exposes conceptual commonalities underlying such results that are not readily apparent otherwise.

*This work was supported in part by the National Science Foundation under grants CDA-9500991, CCR-9711166, and ASC-9720738.

1 Introduction

Program analysis involves keeping track of properties of variables at different program points. In general, the properties of different variables may depend on each other. When tracking such properties, we may choose to keep track of dependencies between the properties of different variables (leading to analysis information of the form "[$x = a$ and $y = b$]; or [$x = c$ and $y = d$]"), or we may choose to ignore such dependencies (leading to information of the form "[$x = a$ or $x = c$]; and [$y = b$ or $y = d$]"). Jones and Muchnick refer to the former kind of analyses as *relational attributes* analyses, and the latter kind as *independent attributes* analyses [6]. The tradeoff between these methods is that independent attributes analyses are usually more efficient but less precise than relational attributes analyses.

When addressing a program analysis problem, it is useful to consider the computational complexity of obtaining a precise (upto symbolic execution) solution to the problem.[1] If a precise solution can be obtained "efficiently," i.e., in polynomial time, it makes sense to try and find an algorithm that obtains such a solution. If, on the other hand, the existence of efficient algorithms to compute precise solutions is unlikely, it makes sense to sacrifice precision for efficiency. Questions about the computational complexity of various program analyses have been addressed by a number of authors (see Section 5). The current state of knowledge resulting from these works is, by and large, a set of isolated facts about the complexities of various analyses. What is missing are insights into the underlying reasons for these results. For example, Landi's results on the complexity of pointer-induced alias analysis [8, 11] tell us that single-level pointers are, in some sense, easy to handle, but multi-level pointers are not: however, they don't explain exactly *why* multi-level pointers are hard to deal with. The situation is further muddled by the results of Pande *et al.*, who show that the precise construction of inter-procedural def-use chains becomes difficult in the presence of single-level pointers [15]. In other words, single-level pointers complicate some analyses but not others, but we don't have any insights into why such pointers are benign in some situations but problematic in others. Moreover, these results are typically obtained using reductions from problems with known complexity: different problem choices by different authors, and differences in the details of the reductions for different analysis problems, often make it difficult to see whether there are any underlying conceptual commonalities between different such complexity arguments.

The main contribution of this paper is to elucidate the fundamental reasons why certain program analyses can be carried out efficiently (i.e., in polynomial time), while others are difficult. We give a simple and general result that is applicable to a wide variety of intra- and inter-procedural flow-sensitive analyses. This is able to explain, for example, why single-level pointers can be handled efficiently in the context of pointer-induced alias analysis [8, 11] but not for def-use chains [15]. With very little conceptual and notational effort, a number of complexity results given in the literature [8, 11, 12, 13, 15] fall out directly as corollaries of this result. Moreover, for several of these analyses, we are able to improve significantly on the known complexity results reported in the literature [12, 13, 15]. For example, we show that the following analyses are EXPTIME-complete: inter-procedural pointer alias analysis in the presence of two-level pointers (Corollary 4.5; previous best result: PSPACE-hard [8]), inter-procedural reaching definitions in the presence of single-level pointers (Corollary 4.6; previous best result: NP-hard [15]), and inter-procedural liveness analysis and available expressions in the presence of reference parameters (Corollary 4.8; previous best result: NP-hard [13]). In the process, our work exposes conceptual commonalities underlying a variety of program analyses.

2 Preliminaries

From the perspective of program analysis, we may be interested in two different kinds of information about program variables. We may want to know something about a particular variable at a particular

[1] The determination of whether some (nontrivial) property will actually hold at a particular program point at runtime is, of course, undecidable. A standard assumption in the dataflow analysis literature is that all "realizable" paths in a program—by which we mean all paths subject to the constraint that procedure calls are matched up correctly with returns—are executable, or, equivalently, that either branch of any conditional can always be executed. This assumption, which Barth referred to as precision "upto symbolic execution" [2], usually suffices to sidestep the problem of undecidability, and "precision" of program analyses is typically defined with respect to this assumption.

program point, e.g., in the context of constant propagation [1]; or we may want to know something about the relationships among some set of variables, e.g., whether or not two variables can be guaranteed to have different values at a particular program point (useful for reasoning about pointers). We refer to the problem of determining the former kind of information as the *single value problem*, and that of determining the latter kind of information as the *simultaneous value problem*. For the purposes of this paper, we focus on rather restricted classes of such problems, under the assumption, standard in dataflow analysis, that all paths in the program being analyzed are executable:

Definition 2.1 Suppose we are given a program P and an initial assignment E_{init} of values for the variables of P. Let $x, x_1, \ldots, x_n$ be variables in P, $c, c_1, \ldots, c_n$ be values, and let p be a program point in P.

A *single value problem* for P is a problem of the form: *"is there an execution path from the entry node of P to p, with initial variable assignment E_{init}, such that "$x = c$" holds when control reaches p?"*

A *simultaneous value problem* for P is a problem of the form: *"is there an execution path from the entry node of P to p, with initial variable assignment E_{init}, such that "$x_1 = c_1 \wedge x_2 = c_2 \wedge \cdots \wedge x_k = c_k$" holds when control reaches p?"* ∎

In particular, simultaneous value problems where all of the constants $c_1, \ldots, c_k$ are either 0 or 1 are referred to as *binary simultaneous value problems*.

It seems intuitively obvious that solving a simultaneous value problem will require a relational attributes analysis; we will show, however, that while an independent attributes analysis is often adequate for a single value problem, there are some situations where it is necessary to resort to relational attributes analyses even for single value problems.

3 Intra-procedural and Non-recursive Inter-procedural Analyses

3.1 Intra-procedural Analysis

In this section we consider a simple language BASE where variables are all integer-valued, and a program consists of a single procedure containing (labelled) statements that can be assignments, conditionals, or unconditional jumps.[2] Since our primary interest is in dataflow analyses, we make the standard assumption that all paths in the program are executable, i.e., that either branch of a conditional may be executed at runtime, and omit the actual expression being tested in a conditional. To keep the discussion simple and focused, we restrict our attention to expressions that are variables or constants (assuming that an analysis is able to do arithmetic adds an independent source of complexity that can obscure the essence of our results):

$$
\begin{array}{lcl}
\textit{Prog} & ::= & \textit{Stmt} \\
\textit{Stmt} & ::= & \textit{Var} \texttt{ = } \textit{Expr}\texttt{;} \\
 & | & \textbf{if}\texttt{ (-) } \textit{Stmt}_1 \;\ldots\; \textbf{else if}\texttt{ (-) } \textit{Stmt}_i \;\ldots\; \textbf{else } \textit{Stmt}_n \\
 & | & \textit{Label}\texttt{:}\ \textit{Stmt} \\
 & | & \textbf{goto}\ \textit{Label}\texttt{;} \\
 & | & \{\textit{Stmt}_1\texttt{;}\ \ldots\texttt{;}\ \textit{Stmt}_n\texttt{;}\} \\
\textit{Expr} & ::= & \textit{Const} \;\;|\;\; \textit{Var} \\
\textit{Const} & ::= & \texttt{0} \;\;|\;\; \texttt{1}
\end{array}
$$

The simplest analyses are those where there is no need to keep track of relationships between variables:

Theorem 3.1 *The single value problem for programs in* BASE *can be solved in polynomial time, provided that primitive operations of the analysis can be carried out in polynomial time.*

[2] We choose this syntax for simplicity: with a small amount of code duplication, it is straightforward to express our programs in a subset of C consisting of assignments, conditionals, and **while** loops together with **break** and **continue** statements.

Proof: A straightforward independent attribute analysis suffices in this case. Jones and Muchnick ([6], Section 12.2) show that this can be carried out in time quadratic in the size of the program, provided that primitive operations of the analysis, e.g., checking whether two abstract domain elements are equal (which is necessary to determine when a fixpoint has been reached), can be carried out in $O(1)$ time. The requirement of constant-time operations can be relaxed to allow polynomial-time primitive operations and still preserve an overall polynomial time complexity. ∎

We next consider the complexity of simultaneous value problems for BASE. In this context, we mention the following result: this is not the central result of this paper, but is of some historical interest because its proof, given below. is essentially isomorphic to similar NP-hardness results for acyclic programs given by a number of authors [6, 8, 11, 12, 13, 15]. Applications of this theorem include (intra-procedural) type inference problems where the type of a variable depends on the types of other variables (see, e.g., [6, 16]). Theorem 3.5 and Corollary 4.3 give stronger results for more general classes of programs.

Theorem 3.2 *The (binary) simultaneous value problem for acyclic programs in* BASE *is NP-complete.*

Proof: The proof of NP-hardness is by reduction from the 3-SAT problem, which is the problem of determining, given a set of clauses φ each containing three literals, whether φ is satisfiable. This problem is known to be NP-complete [5]. Given a formula $\varphi \equiv (u_{11} \vee \cdots \vee u_{13}) \wedge \cdots \wedge (u_{m1} \vee \cdots \vee u_{m3})$ over a set of variables $\{x_1, \ldots, x_n\}$, where each of the literals u_{ij} is either a variable or its negation, we generate a program P_φ, with variables {`x_1t`, ..., `x_nt`, `x_1f`, ..., `x_nf`, `c1`, ..., `cm`}, of the following form:

```
    if ( - ) { x_1t = 0; x_1f = 1; } else { x_1t = 1; x_1f = 0; }
    if ( - ) { x_2t = 0; x_2f = 1; } else { x_2t = 1; x_2f = 0; }
    ...
    if ( - ) { x_nt = 0; x_nf = 1; } else { x_nt = 1; x_nf = 0; }

    if ( - ) c1 = w11;
    else if ( - ) c1 = w12;
    else c1 = w13;
    ...
    if ( - ) cm = wm1;
    else if ( - ) cm = wm2;
    else cm = wm3;
L:
```

Here, w_{ij} are defined as follows: if the literal u_{ij} is a variable x_k for some k, then w_{ij} = `x_kt`; if the literal u_{ij} is a negated variable $\overline{x_k}$ for some k, then w_{ij} = `x_kf`. Intuitively, `x_it` = 1 in P_φ represents an assignment of a truth value *true* to x_i in φ, while `x_if` = 1 represents a truth value of *false*. Each path through the first group of conditionals represents a truth assignment for the variables of φ. The second group of conditionals represents the evaluation of the clauses: the i^{th} clause evaluates to true if and only if there is a path through the i^{th} conditional in the second group that assigns 1 to the variable `ci`. The simultaneous value problem we pose at the program point labelled `L` is

`c1 = 1` $\wedge$ `...` $\wedge$ `cm = 1`.

This is true if and only if there is a path through all of the statements in P_φ that assigns 1 to each of the `ci`, i.e., if and only if there is a truth assignment to the variables of φ that causes each of its clauses to evaluate to *true*.

To see that the simultaneous value problem is in NP, given any acyclic program in BASE we simply guess a path through the program and check whether the assignments along this path make the problem true. ∎

The main result of this section is for simultaneous value problems for all programs in BASE. We show that this class of problems is PSPACE-complete: the idea is that given an arbitary polynomial-space-bounded Turing machine, we can construct a simultaneous value problem over a program in BASE that can be used to determine whether or not the Turing machine accepts its input. Suppose we are given a single tape deterministic polynomial-space-bounded Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where Σ is the input alphabet; $\Gamma = \{0, 1, \ldots, ns\}$ is the tape alphabet, with 0 being the blank symbol; $\delta \in Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathsf{L}, \mathsf{R}\}$ is the transition function; $q_0 \in Q$ is the initial state; and $F = \{q_1\}$ is the set of final states, such that M halts on all inputs x after using at most $|x|^k$ cells of the tape. For simplicity we assume that M erases its tape before halting and that the tape is cyclic, i.e., after the last cell the tape "wraps around" to the first cell: these are not serious restrictions, and it is not difficult to see how a Turing machine that does not satisfy these assumptions can be transformed into one that does. The use of a cyclic tape allows us to simulate the movement of the tape head to the left (respectively, right) by rotating the tape to the right (respectively, left), so that the tape cell being scanned by the head is always cell 0: this simplifies the simulation of the Turing machine, since we don't have to keep track of the position of the tape head. We construct a program $P_{M,x}$ that emulates M on an input x. This program contains three sets of (boolean) variables:

1. $\mathtt{Q}_0, \ldots, \mathtt{Q}_{nq}$, where $nq = |Q| - 1$: These variables represent the current state of M: intuitively, $\mathtt{Q}_i = 1$ denotes that M is in state i.
2. $\mathtt{T}_{0,0}, \ldots, \mathtt{T}_{nt,ns}$, where $nt = |x|^k - 1, ns = |\Gamma| - 1$: These variables represent the contents of M's tape: intuitively, $\mathtt{T}_{i,j} = 1$ denotes that cell i of M's tape contains symbol j.
3. $\mathtt{X}_0, \ldots, \mathtt{X}_{ns}$: these variables are temporaries for copying the tape contents while we "rotate" the tape.

A configuration where M is in state q_k, the tape contents are $s_0 s_1 \ldots s_{nt}$, and where M's tape head is scanning the m^{th} tape square, is described by the following variable settings:

$$\mathtt{Q}_i = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise} \end{cases} ; \quad \mathtt{X}_i = 0, \text{for all } i; \quad \mathtt{T}_{i,j} = \begin{cases} 1 & \text{if } s_{(i-m) \bmod (nt+1)} = j \\ 0 & \text{otherwise} \end{cases}$$

The code corresponding to M's move when it is state q_i and scanning a cell containing a symbol s_j, i.e., $\delta(q_i, s_j)$, is represented by $MOV_{i,j}$, and is defined as follows:

$\delta(q_i, s_j) = (q_k, s_m, \mathsf{L})$	$\delta(q_i, s_j) = (q_k, s_m, \mathsf{R})$
$\mathtt{Q}_i$ = $\mathtt{Q}_k$;	$\mathtt{Q}_i$ = $\mathtt{Q}_k$;
$\mathtt{Q}_k$ = 1;	$\mathtt{Q}_k$ = 1;
$\mathtt{T}_{0,j}$ = $\mathtt{T}_{0,m}$;	$\mathtt{T}_{0,j}$ = $\mathtt{T}_{0,m}$;
$\mathtt{T}_{0,m}$ = 1;	$\mathtt{T}_{0,m}$ = 1;
`goto copy_left;`	`goto copy_right;`

The first two lines of this code update the state variable, the next two lines update the contents of the tape cell being scanned, and the last line corresponds to the rotation of the tape, simulating the movement of the tape head.

The program $P_{M,x}$ that emulates M on input x is shown in Figure 1. After initializing the $\mathtt{T}_{i,j}$ variables appropriately for the input x, the program goes into a loop, repeatedly guessing the current state and the symbol under the tape head, then updating the state and tape cell, and finally rotating the tape appropriately in order to simulate the movement of the tape head. A wrong guess leads to a state where multiple $\mathtt{Q}_i$ variables, or multiple $\mathtt{T}_{i,j}$ variables, are set to 1. Once such an "illegal" state is entered, the structure of the program ensures that the number of variables set to 1 does not decrease, which means that subsequent states remain illegal. This allows us to use a simultaneous value problem to identify legal states in $P_{M,x}$, i.e., those that correspond to valid configurations of M, and thence to determine whether M accepts its input. For notational convenience, we introduce the following abbreviations:

```
/* Program P_{M,x} to emulate a given polynomial space-bounded Turing Machine M
   on input x */
/* int Q_0, ..., Q_{nq};
   int T_{0,0}, ..., T_{nt,ns};
   int X_0, ..., X_{ns}; */
{
   T_{0,0} = ···; ...; T_{nt,ns} = ···;  /* initialize T_{i,j} based on input string x */
   Q_0 = 1; Q_1 = 0; ...Q_{nq} = 0;   /* initial state */

   Start:                        /* emulation loop */
      X_0 = 0; ...;  X_{ns} = 0;    /* clear temps */

   Dispatch:            /* transitions based on current state and tape symbol */
      if ( - )
      { /* Q_0 == 1? */
         if ( - ) { /* T_{0,0} == 1? */ MOV_{0,0}; }
         ...
         else if ( - ) { /* T_{0,i} == 1? */ MOV_{0,i}; }
         ...
         else if ( - ) { /* T_{0,ns} == 1? */ MOV_{0,ns}; }
      }
      else if ( - ) goto Done; /* Q_1 == 1? : q_1 = final state */
      else if ( - )
      { /* Q_2 == 1? */
         ...
      }
      ...
      else if ( - )
      { /* Q_{nq} == 1? */
         if ( - ) { /* T_{0,0} == 1? */ MOV_{nq,0}; }
         ...
         else if ( - ) { /* T_{0,i} == 1? */ MOV_{nq,i}; }
         ...
         else if ( - ) { /* T_{0,ns} == 1? */ MOV_{nq,ns}; }
      }
   /* copy tape left or right */
   copy_right:
      X_0 = T_{0,0}; ...; X_{ns} = T_{0,ns};
      T_{0,0} = T_{1,0}; ...; T_{0,ns} = T_{1,ns};
      ...
      T_{nt,0} = X_0; ...; T_{nt,ns} = X_{ns};
      goto Start;
   copy_left:
      X_0 = T_{nt,0}; ...; X_{ns} = T_{nt,ns};
      T_{nt,0} = T_{nt-1,0}; ...; T_{nt,ns} = T_{nt-1,ns};
      ...
      T_{0,0} = X_0; ...; T_{0,ns} = X_{ns};
   goto Start;
   Done:
      X_0 = 0; ...X_{ns} = 0;
   End:
}
```

Figure 1: The program $P_{M,x}$ to emulate Turing machine M on input x

$$
\begin{array}{lcl}
\textit{UnambiguousFinalState} & \equiv & (\mathtt{Q}_0 = 0 \wedge \mathtt{Q}_1 = 1 \wedge \mathtt{Q}_2 = 0 \wedge \cdots \mathtt{Q}_{nq} = 0) \\
\textit{TempsClear} & \equiv & (\mathtt{X}_0 = 0 \wedge \cdots \mathtt{X}_{ns} = 0) \\
\textit{TapeClear} & \equiv & ((\mathtt{T}_{0,0} = 1 \wedge \cdots \wedge \mathtt{T}_{nt,0} = 1) \wedge \\
 & & \;(\mathtt{T}_{0,1} = 0 \wedge \cdots \wedge \mathtt{T}_{nt,1} = 0) \wedge \cdots \wedge \\
 & & \;(\mathtt{T}_{0,ns} = 0 \wedge \cdots \wedge \mathtt{T}_{nt,ns} = 0)).
\end{array}
$$

Intuitively, *UnambiguousFinalState* is true if and only if the only state variable that is 1 is $\mathtt{Q}_1$, corresponding to the final state of M; *TempsClear* is true if and only if the variables $\mathtt{X}_i$ are all 0; and *TapeClear* is true if and only if the contents of the variables $\mathtt{T}_{i,j}$ correspond to all the tape cells of M containing a blank.

Lemma 3.3 *A given polynomial-space-bounded Turing machine M accepts its input x if and only if AcceptingConfig may hold at the end of the program, where*

$$\textit{AcceptingConfig} \equiv \textit{UnambiguousFinalState} \wedge \textit{TempsClear} \wedge \textit{TapeClear}.$$

Proof: (sketch) Let a configuration α of M *correspond to* a state $\widehat{\alpha}$ of $P_{M,x}$, written $\alpha \sim \widehat{\alpha}$, if and only if the following holds: in α, M is in state q_k, scanning tape cell m, with tape contents $s_0 s_1 \ldots s_{nt}$; and in $\widehat{\alpha}$, $P_{M,x}$ has the following values for its variables, with control at the point labelled `Dispatch`:

$$\mathtt{Q}_i = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise} \end{cases} ; \quad \mathtt{X}_i = 0, \text{for all } i; \quad \mathtt{T}_{i,j} = \begin{cases} 1 & \text{if } s_{(i-m) \bmod (nt+1)} = j \\ 0 & \text{otherwise} \end{cases}$$

We use the following notation: if M can go from configuration α to configuration β via a sequence of transitions, we write $\alpha \vdash_M^* \beta$; if there is a path in the program $P_{M,x}$ that transforms a state u to a state v, with control being at the point labelled `Dispatch` in each case, we write $u \vdash_P^* v$.

We first show that if, given configurations α and β for M and states $\widehat{\alpha}$ and $\widehat{\beta}$ for $P_{M,x}$ such that $\alpha \sim \widehat{\alpha}$ and $\beta \sim \widehat{\beta}$, if $\alpha \vdash_M^* \beta$ then $\widehat{\alpha} \vdash_P^* \widehat{\beta}$. Pictorially:

$$
\begin{tikzcd}
\alpha \arrow[r, "\vdash_M^*"] \arrow[d, dash, "\sim"'] & \beta \arrow[d, dash, "\sim"] \\
\widehat{\alpha} \arrow[r, "\vdash_P^*"'] & \widehat{\beta}
\end{tikzcd}
$$

The proof is by induction on the length n of the transition sequence of M. The base case, for $n = 0$, is trivial. For the inductive case, suppose that the claim holds for transition sequences of length n, and consider configurations α, β and γ of M and states $\widehat{\alpha}$ and $\widehat{\gamma}$ of $P_{M,x}$, with $\alpha \sim \widehat{\alpha}$ and $\gamma \sim \widehat{\gamma}$, such that $\alpha \vdash_M^n \gamma \vdash_M \beta$. From the induction hypothesis, we have $\widehat{\alpha} \vdash_P^* \widehat{\gamma}$. Suppose that in the transition $\gamma \vdash_M \beta$ M goes from state q_a, scanning tape symbol c, to state q_b. In $P_{M,x}$, consider state resulting from $\widehat{\gamma}$ by taking the path from the point labelled `Dispatch` to that referred to as $MOV_{a,c}$. An examination of the definition of the code corresponding to $MOV_{i,j}$ shows that the resulting state $\widehat{\beta}$ of $P_{M,x}$ corresponds to the configuration β of M after the $n+1^{st}$ transition. The claim follows.

Since, from the definition of $P_{M,x}$, the initial configuration of M corresponds to the state of $P_{M,x}$ when control first reaches `Dispatch`, it follows from this that if M accepts its input and halts—i.e., reaches a configuration with state q_1 and its tape erased (recall that q_1 is the final state of M, and we assumed that M would erase its tape prior to halting)—then there is a path in $P_{M,x}$ that leads to a corresponding state, which is described by *AcceptingConfig*. This means that *AcceptingConfig* holds at the point `End`. Conversely, if there is a path through $P_{M,x}$ such that *AcceptingConfig* holds at its end at the point labelled `End`, then we can use the sequence of $MOV_{i,j}$ code executed along this path to reconstruct a sequence of moves of M leading to acceptance. This establishes that M accepts its input if and only if there is a path in $P_{M,x}$, consisting of "good" guesses, at the end of which *AcceptingConfig* holds at the point `End`.

Next, consider any path in $P_{M,x}$ that does not correspond to a valid computation of M. This must come from a "bad guess" in $P_{M,x}$ of either the state (variables $\mathtt{Q}_i$) or the tape symbol (variables $\mathtt{T}_{j,k}$), resulting in the execution of a code fragment $MOV_{i,k}$. It can be seen, from the definition of $MOV_{i,k}$, that the variable setting that results when control next returns to the point `Dispatch` has more than one the variables $\mathtt{Q}_i$ set to 1, or more than one of the variables $\mathtt{T}_{i,j}$ set to 1. Such a variable setting is called illegal because it does not represent any valid configuration. Furthermore, once we obtain an illegal variable setting we cannot turn this back into a legal one because each of the $MOV_{i,j}$ code segments preserves or increases the number variables set to 1. This means that *AcceptingConfig* will not hold at the end of such a path in $P_{M,x}$.

Together, it follows from these that *AcceptingConfig* will hold at the point labelled `End` if and only if M accepts x. ∎

Lemma 3.4 *Given a polynomial-space-bounded Turing machine M and input x, the program $P_{M,x}$ illustrated in Figure 1 can be generated in space $O(\log(|M| + |x|))$.*

Proof: Suppose we are given a Turing machine M that, on any input of length n, is $p(n)$-space-bounded for some polynomial $p(n)$. The code for the corresponding program $P_{M,x}$ can be divided into three distinct, and independent, components: the initialization code; the code for the emulation loop, consisting of the code to clear the variables $\mathtt{X}_i$ followed by the code for the transitions of M; and the code for "rotating" the tape, labelled `copy_right` and `copy_left`, and the "cleanup" computation at the label `Done`. The space requirements for each of these components is as follows:

- The initialization step consists of $|\Gamma| \times p(|x|)$ assignments, where each assignment statement is of fixed size. To generate this code we need a counter of size $\log(|\Gamma| \times p(|x|)) = \log|\Gamma| + \log p(|x|)$ bits. Since $|\Gamma| = O(|M|)$ and $\log p(n) = O(\log n)$ for any polynomial $p(n)$, this component requires $O(\log|M| + \log|x|)$ space.
- For the emulation loop, clearing the temporary variables requires $\log|\Gamma| = O(1)$ bits. The outer **if** statement in the emulation loop consists of $|Q|$ cases, where each case (with the exception of that for $\mathtt{Q}_1 = 1$) consists of an inner **if** statement with $O(|\Gamma|)$ cases, each of which consists of a fixed amount of code. Thus the space requirement for generating this is $\log(|Q| \times |\Gamma|) = \log|Q| + \log|\Gamma| = O(\log|M|)$. Thus, the total space required for this component is $O(\log|M|)$.
- Each of the `copy_right` and `copy_left` portions of the program consists of $|\Gamma| + |\Gamma| \times p(|x|) = O(|\Gamma| \times p(|x|))$ assignments, where each assignment statement is of fixed size. The cleanup code at the label `Done` consists of $|\Gamma| \times p(|x|)$ assignments, where each assignment statement is of fixed size. To generate these assignments we need a counter of size $\log(|\Gamma| \times p(|x|)) = \log|\Gamma| + \log p(|x|)$ bits. Since $|\Gamma| = O(|M|)$ and $\log p(n) = O(\log n)$ for any polynomial $p(n)$, this component requires $O(\log|M| + \log|x|)$ space.

The total space required is therefore $O(\log|x| + \log|M|)$. Since $\log|x| \leq \log(|M| + |x|)$ and $\log|M| \leq \log(|M| + |x|)$, we have $O(\log|x| + \log|M|) = O(\log(|M| + |x|))$. The lemma follows. ∎

Theorem 3.5 *The (binary) simultaneous value problem for programs in* BASE *is PSPACE-complete.*

Proof: (sketch) PSPACE-hardness follows directly from Lemmas 3.3 and 3.4.

To show that the simultaneous value problem is in PSPACE, we show that a given such a problem for a program P, we can construct a nondeterministic multi-tape polynomial-space-bounded Turing machine

M_P to solve the problem. Given a program P, the input to M_P consists of the control flow graph G_P of P, an initial assignment E_{init} of values for the variables of P, a target program point n_t, and a target environment E_t for the variables of P: $E_t = \{x_0 \mapsto c_0, x_1 \mapsto c_1, \ldots, x_n \mapsto c_n\}$ specifies the simultaneous value problem $x_0 = c_0 \wedge x_1 = c_1 \wedge \ldots \wedge x_n = c_n$. We want M to halt iff there is a path from the initial node of G_P to n_t that transforms E_{init} to the target environment E_t. M_P copies G_P and E_t to two work tapes and maintains another work tape T_{env} that contains a list of (*variable*, *value*) pairs, one for each program variable. T_{env} is initialized from the initial assignment E_{init}. M_P then starts simulating the execution of P by traversing G_P At each vertex of the control flow graph, it simulates the effects of assignments and updates T_{env} appropriately. At branch nodes M_P nondeterministically chooses a successor to continue processing. Whenever M_P reaches the target node n_t it checks whether the variable values on T_{env} match the desired environment E_t, and halts if this is the case. It is clear that if there is an execution path in P such that, starting from the initial variable assignment E_{init}, execution can reach the point n_t with the desired values E_t for the variables, then M can guess this path and will eventually halt and accept its input. Conversely, if M_P halts and accepts, there must have been such a path.

The space needs for M_P are bounded by the space required to store the G_P and E_{init} and the space required for the tape T_{env}. The space required for G_P and E_{init} is $O(n)$, where n is the size of the input program. Under the assumption that the we have a fixed number of constants to deal with (i.e., that the analysis is being carried out over a fixed finite domain), we need $O(1)$ bits for the value of a variable at any program point; there can be at most $O(n)$ variables in P, so the space requirements for T_{env} are $O(n)$. It follows that M is polynomial-space-bounded. ∎

In the context of program analysis, this is representative of the simplest kind of simultaneous value problem, where we have two distinct properties (here represented by "equal to 0" and "equal to 1") of a language with a minimally interesting set of control constructs. The (hardness) result therefore extends directly to more complex analysis problems. Unlike the PSPACE-hardness result given by Jones and Muchnick for relational attributes analyses [6], our result does not require interpreted conditionals. In other words, our result complies with the standard assumption of dataflow analysis, namely, that all paths in a program are executable. As such, it is applicable to a wider variety of dataflow analyses.

3.2 Inter-procedural Analysis of Non-recursive Programs

Suppose we extend the language BASE with procedures where parameters are passed by value: let the resulting language be BASE+PROC. For non-recursive programs in this language, the complexity of simultaneous value problems does not change:

Theorem 3.6 *Inter-procedural simultaneous values problems for non-recursive programs in* BASE+PROC *is PSPACE-complete.*

Proof: PSPACE-hardness follows from Theorem 3.5. To see that the problem remains in PSPACE, consider a non-recursive program containing k procedures. The runtime call stack of this program can have depth at most k. We use a nondeterministic Turing machine similar to that used to show membership in PSPACE in the proof of Theorem 3.5, except that it uses a tape that is k times longer than before. This tape is used as a stack: at a procedure call, it "pushes" a frame by copying the values of the arguments after the "current frame" at the end of the tape; and on a return from a procedure, it "pops" the current frame by erasing the appropriate tape cells and moves to the next frame. The space requirement of this machine is still polynomial in the length of the input, whence it follows that the analysis is in PSPACE. ∎

3.3 Applications to the Complexity of Dataflow Analyses

This section discusses applications of the results of the previous section to various program analyses discussed in the literature.

3.3.1 Intra-procedural Pointer Alias Analysis

We first add single-level pointers to the BASE language, yielding the language BASE+1PTR. This language contains two classes of variables: *base variables*, which range over integers, and *pointers to base variables*, which range over addresses (which are assumed to be disjoint from the set of integers). The new operations in this language, compared to BASE, are: taking the address of a (base) variable `v`, denoted by `&v`, and dereferencing a pointer `p`, denoted by `*p`.

It is not hard to see that the simultaneous value problem in this case is still in PSPACE, since we can construct a polynomial-space-bounded Turing machine to solve this problem in a manner similar to that in the proof of Theorem 3.5. By contrast to the language BASE, where the single value problem is in P, the complexity of the single value problem for BASE+1PTR depends on whether we are concerned with base variables or pointers. For a single-value problem for a base variable, an independent attribute analysis is not sufficient. This is illustrated by the following program fragment:

```
a = 0;
if ( - ) { p = &a; x = 0; }  else { p = &b; x = 1; }
*p = x;
```

Suppose we are interested in the single-value problem of whether `a = 1` may hold immediately after the assignment `*p = x`. An independent attributes analysis would infer that immediately after the conditional, `p` can point to either `a` or `b`, and therefore that after the assignment '`*p = x`' the value of `a` may or may not be 1. A relational attributes analysis, on the other hand, would be able to infer that the value of `a` cannot be 1 after the indirect assignment. In other words, for a precise analysis we need relational attributes, i.e., the ability to solve simultaneous value problems.

Theorem 3.7 *The single-value problem for pointer variables in* BASE+1PTR *can be solved in polynomial time. The single-value problem for base variables in* BASE+1PTR *is PSPACE-complete.*

Proof: For a single-value problem for a pointer variable, the analysis need concern itself only with assignments to pointer variables, and a straightforward independent attributes analysis is sufficient. Reasoning as for Theorem 3.1 shows that this is solvable in polynomial time.

To prove PSPACE-hardness, we show how a binary simultaneous value problem in BASE can be reduced to a single-value problem for base variables in BASE+1PTR . Given a program P in BASE the idea is to generate a program P' as follows (here, $X_1, X_2, \ldots$ denote variables in P while $\mathtt{X}_1, \mathtt{X}_2, \ldots$ denote variables in P'). The program P' contains two variables, `Zero` and `One`, that are initialized to the constants 0 and 1 respectively. For each variable X in P we have two variables `X` and $\overline{\mathtt{X}}$ in P'. Assignments in P are translated into P' as follows:

- An assignment '$X = 0$' in P is translated to a pair of assignments '`X = &Zero;` $\overline{X}$ `= &One`' in P'; an assignment '$X = 1$' is translated to '`X = &One;` $\overline{X}$ `= &Zero`.'
- An assignment '$X = Y$ in P is translated to a pair of assignments '`X = Y;` $\overline{\mathtt{X}}$ `=` $\overline{\mathtt{Y}}$.'

The intuition is that `X` tells us what the value of the original variable X is, while $\overline{\mathtt{X}}$ tells us what it is not. Other constructs, such as conditionals and control transfers, remain unchanged in the translation.

Suppose we are given a binary simultaneous value problem in of the form $X_1 = c_1 \wedge \cdots \wedge X_n = c_n$ at a point p in the original program P, where $c_i \in \{0, 1\}$. Consider the conjunct $X_1 = c_1$: if $c_1 \equiv 0$ then, in the generated program program, we want to test whether $\mathtt{X}_1$ points to `Zero`. If $c_1 \equiv 1$, we want to test whether $\mathtt{X}_1$ points to `One`; or equivalently, whether $\mathtt{X}_1$ does not point to `Zero` (since the variables `One` and `Zero` are the only base variables in the program, and hence the only things that $\mathtt{X}_1$ could point to); or equivalently, whether $\overline{X}_1$ points to `Zero`. Let p' be the program point in P' that corresponds to the point p in P, and let $u \rightsquigarrow v$ denote that u points to v. We want to determine whether there is an execution path in P' upto p' such that $x_1 \rightsquigarrow \texttt{Zero} \wedge \cdots \wedge x_n \rightsquigarrow \texttt{Zero}$, where x_i is $\texttt{X}_i$ if $c_i \equiv 0$, and $\overline{\texttt{X}}_i$ if $c_i \equiv 1$. We do this by inserting the following code fragment at the point p' (where x_i is either $\texttt{X}_i$ or $\overline{\texttt{X}}_i$, depending on whether c_i is 0 or not, as just described).

```
    if ( - ) {
       *x1 = 0; ...; *xn = 0;
L:     goto End;          /* go to end of program and halt */
    }
```

If, for some execution path leading to p' in the program P', $x_i \rightsquigarrow \texttt{Zero}$ for each x_i, then all of the assignments $*x_i$ = 0 will write to the variable Zero. This means that the initial assignment of 1 to the variable One will not be overwritten (since there are no other assignments to either Zero or One, or any indirect assignments through any of the variables $\texttt{X}_i$ or $\overline{\texttt{X}}_i$, elsewhere in the program), so One will have the value 1 at the point labelled L in the code fragment above. On the other hand, if for every execution path leading to p' we have $x_j \not\rightsquigarrow \texttt{Zero}$ for some j, it must be the case that $x_j \rightsquigarrow \texttt{One}$, which means that the assignment $*x_j$ = 0 will overwrite the initial assignment to One. Thus, by answering the single-value problem of whether or not One has the value 1 at the point L, we can solve the original binary simultaneous value problem for the program P. The result follows from Theorem 3.7. ∎

As an example application of this, the following result is immediate:

Corollary 3.8 *Precise intra-procedural constant propagation in* BASE+1PTR *is PSPACE-complete.* ∎

Next, we consider multi-level pointers. The simplest case involving multi-level pointers is when we have two-level pointers, i.e., pointers to pointers. In this case we have three classes of variables: base variables; pointers to base variables, or 1-pointers; and pointers to 1-pointers (i.e., pointers to pointers to base variables), or 2-pointers. We call this language BASE+2PTR.

The role of 2-pointers with respect to 1-pointers in the language BASE+2PTR is exactly analogous to that of pointers to base variables in the language BASE+1PTR. In particular, to determine the possible aliases of 1-pointers, we need to determine the values that can be assigned to them through 2-pointers. By direct analogy with Theorem 3.7, therefore, we have the following result:

Theorem 3.9 *The single-value problem for 2-pointers in* BASE+2PTR *is solvable in polynomial time. The single-value problem for 1-pointers in* BASE+2PTR *is PSPACE-complete.* ∎

Landi's dissertation shows that intra-procedural pointer alias analysis is PSPACE-complete if at least four levels of indirection are permitted [8]; his proof can be adapted to require only two levels of indirection [10]. Landi's conclusion is that the difficulty with pointer alias analysis is caused by multiple levels of indirection. This is obviously a valid conclusion, but does not get to the heart of the matter: *what is the fundamental difference between single-level and multi-level pointers that causes the analysis of multi-level pointers to become so difficult?* The answer, as we have shown above, is that alias analysis in the presence of at most one level of indirection can be carried out using an independent attributes analysis, while the presence of even two levels of indirection requires a relational attributes analysis.

A similar line of reasoning can be used to derive a recent result by Chatterjee *et al.* [4], namely, that intra-procedural concrete type inference for Java programs with single-level types and exceptions without subtyping, and without dynamic dispatch, is PSPACE-hard.

3.3.2 Intra-procedural Reaching Definitions with Single-Level Pointers

Consider the problem of computing intra-procedural reaching definitions in the language BASE+1PTR, i.e., in the presence of single-level pointers. The following example illustrates that an independent attributes analysis is not enough for a precise solution to this problem, and that a relational attributes analysis is necessary:

```
      int a, b, *p, *q;
      ...
   D: a = 0;
      if ( - ) { p = &a; q = &b; } else { q = &a; p = &b; }
      *p = 1;
      *q = 1;
   L:
```

We want to know whether the definition labelled `D` can reach the program point labelled `L`. An independent attributes analysis would infer that `p` can point to either `a` or `b` after the conditional, and therefore that the assignment `*p = 1` might not kill the definition `D`. A similar reasoning would apply to `q` and the indirect assignment `*q = 1`. Such an analysis would therefore conclude that definition `D` could reach `L`. A relational attributes analysis, by contrast, would determine that one of `p` or `q` would point to `a`, so that one of the assignments `*p = 1` or `*q = 1` would definitely kill the definition `D`—i.e., definition `D` does not reach `L`. Thus, the independent attributes analysis is not precise, and a relational attributes analysis is necessary. The following theorem discusses the complexity of precise analyses; its proof uses a reduction very similar to that for Theorem 3.7.

Theorem 3.10 *The determination of precise solutions for the following intra-procedural analysis problems for base variables in programs in* BASE+1PTR *is PSPACE-complete: (a) reaching definitions; (b) live variables; and (c) available expressions.*

Proof: The proof of PSPACE-hardness uses a translation from the simultaneous value problem in BASE that is identical to that used in the proof of Theorem 3.7. Let p' be the program point in P' that corresponds to the point p in P, and let $u \rightsquigarrow v$ denote that u points to v. We inser the following code fragment at the point p' (where x_i is either $\mathtt{X}_i$ or $\overline{\mathtt{X}}_i$, depending on whether c_i is 0 or not, as in the proof of Theorem 3.7).

```
      if ( - ) {
         *x_1 = 0; ...; *x_n = 0;
   L:    goto End;          /* go to end of program and halt */
      }
```

If, for some execution path leading to p' in the program P', $x_i \rightsquigarrow$ `Zero` for each x_i, then all of the assignments $*x_i$ `= 0` will write to the variable `Zero`, which means that the initial assignment of 1 to the variable `One` will reach the point labelled `L` in the code fragment above (since there are no other assignments to either `Zero` or `One`, or any indirect assignments through any of the variables $\mathtt{X}_i$ or $\overline{\mathtt{X}}_i$, elsewhere in the program). On the other hand, if for every execution path leading to p' we have $x_j \not\rightsquigarrow$ `Zero` for some j, it must be the case that $x_j \rightsquigarrow$ `One`, which means that the assignment $*x_j$ `= 0` will kill the initial assignment to `One`. Thus, by answering the question of whether the initial assignment to the variable `One` can reach the point labelled `L` in the program P', we can solve the original binary simultaneous value problem for the program P. The result follows from Theorem 3.7.

Similar arguments can be used to establish PSPACE-completeness for liveness analysis and available expressions. ∎

Theorem 3.10 improves on a result due to Pande, Landi and Ryder, who show that the problem of computing inter-procedural def-use chains in the presence of single-level pointers is NP-hard [15].

4 Inter-procedural Analysis of Recursive Programs

To study the complexity of inter-procedural analyses in the presence of recursion, we add a very limited enhancement to the control flow constructs of the language BASE+PROC (i.e., the base language together with procedures). Each program now has a distinguished global variable `NoErr` whose value is initially 1. We add a statement Error-if-Zero($-$) that behaves as follows: when Error-if-Zero(x) is executed, `NoErr` is set to 0 if x has the value 0, otherwise it is not modified. In a general programming context, such a construct could be used to determine, for example, whether system calls such as `malloc()` have executed without errors during execution; in the context of this paper we use it in a much more limited way, though with a very similar overall goal, namely, to determine whether anything "goes wrong" in an execution path. We refer to the language obtained by adding this facility to BASE+PROC as BASE+PROC+ERR.

We show that the single-value problem for arbitrary programs in BASE+PROC+ERR is complete for deterministic exponential time. Our proof relies on a result of Chandra *et al.* [3], who show that APSPACE = EXPTIME, where APSPACE is the class of languages accepted by polynomial-space-bounded alternating Turing machines, and EXPTIME = $\cup_{c \geq 0}$DTIME$[2^{n^c}]$.

Definition 4.1 An (single-tape) alternating Turing machine M is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, \tau)$, where Q is a finite set of states; Σ is the input alphabet; Γ is the tape alphabet; $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\mathsf{L}, \mathsf{R}\})$ is the transition function; $q_0 \in Q$ is the initial state; and $\tau : Q \to \{\mathsf{accept}, \mathsf{reject}, \forall, \exists\}$ is a labelling function on states.[3] ■

To simplify the discussion that follows, we additionally assume that a state q that is existential (i.e., $\tau(q) = \exists$) or universal (i.e., $\tau(q) = \forall$) has exactly two successor states for any given tape symbol; it is not hard to see how any ATM can be transformed to satisfy this restriction: if a state q has a single successor for some tape symbol we add a second successor that is either an accepting state if q is universal, or a rejecting state if q is existential; if q has more than 2 successors for some tape symbol, we use a "binary tree of transitions" instead. As before, we assume that the tape "wraps around," so that the cell being scanned is always cell 0. Thus, a configuration of an ATM is of the form qx where q is a state and x the tape contents.

The notion of acceptance for alternating Turing machines is a generalization of that for ordinary non-deterministic Turing machines: the main difference is that each successor of a universal state is required to lead to acceptance. To define this more formally, we use the notion of *computation trees* due to Ladner *et al.* [7]. A computation tree for an ATM M is a finite, nonempty labelled tree with the following properties: each node of the tree is labelled with a configuration of M; if p is an internal node of a tree with label qu and q is an existential state, then p has exactly one child labelled $q'u'$ such that $qu \vdash q'u'$; and if p is an internal node of a tree with label qu and q is a universal state with successors q' and q'', such that $qu \vdash q'u'$ and $qu \vdash q''u''$, then p has two children labelled $q'u'$ and $q''u''$. An *accepting computation tree* is one where all the leaf nodes are accepting configurations, i.e., of the form qu where q is an accepting state. An ATM M with start state q_0 accepts an input x if it has an accepting computation tree whose root is labelled q_0x.

Let M be a $p(n)$-space-bounded ATM with tape alphabet Γ, where $p(n)$ is some polynomial, and let x be an input for M. Let $nt = p(|x|) - 1$ and $ns = |\Gamma| - 1$. The program $P_{M,x}$ in BASE+PROC+ERR that simulates the behavior of M on input x behaves as sketched below. There is a function $\mathtt{f}_q\mathtt{()}$ for each state q of M. Each such function has a tuple of parameters $\mathtt{T}_{0,0}$, ..., $\mathtt{T}_{nt,ns}$ that represents the contents of M's tape in a way that is conceptually similar to the construction described in Section 3.1, the main difference being that these variable are now locals rather than globals. State transitions in M are simulated by function calls in $P_{M,x}$: moves to the successors of an existential state are simulated using an **if-then** construct, while moves to the successors of a universal state are simulated by a sequence of function calls.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \tau)$ be a $p(n)$-space-bounded ATM, where $p(n)$ is some polynomial, and let x be an input for M. Let $nt = p(|x|) - 1$ and $ns = |\Gamma| - 1$. We generate a program $P_{M,x}$ in BASE+PROC+ERR

[3] There is a more general formulation of alternating Turing machines where states can also be labelled as "negating" states, which are labelled by $\neg$. However, this adds nothing to their power (Theorem 2.5 of Chandra *et al.* [3]), so for simplicity we restrict ourselves to alternating Turing machines without negating states.

as discussed below. The code necessary to simulate M's actions when it makes a transition from state q_i to state q_k upon scanning a tape cell containing symbol s_j is represented by $\mathsf{TRANSITION}(q_i, s_j, q_k)$ and is defined as follows:

$\delta(q_i, s_j) = (q_k, s_m, \mathsf{L})$	$\delta(q_i, s_j) = (q_k, s_m, \mathsf{R})$	*Explanation*
$\mathtt{T}_{0,0} = \mathtt{X}_{0,0}$;	$\mathtt{T}_{0,0} = \mathtt{X}_{0,0}$;	restore tape
...	...	
$\mathtt{T}_{nt,ns} = \mathtt{X}_{nt,ns}$;	$\mathtt{T}_{nt,ns} = \mathtt{X}_{nt,ns}$;	restore tape
Error-if-Zero($\mathtt{T}_{0,j}$) ;	Error-if-Zero($\mathtt{T}_{0,j}$) ;	verify scanned symbol
$\mathtt{T}_{0,j} = 0$;	$\mathtt{T}_{0,j} = 0$;	update tape
$\mathtt{T}_{0,m} = 1$;	$\mathtt{T}_{0,m} = 1$;	update tape
COPY_LEFT;	*COPY_RIGHT*;	rotate tape
$\mathtt{f}_{q_k}(\mathtt{T}_{0,0}, \ldots, \mathtt{T}_{nt,ns})$;	$\mathtt{f}_{q_k}(\mathtt{T}_{0,0}, \ldots, \mathtt{T}_{nt,ns})$;	move to state q_k

Here, *COPY_LEFT* and *COPY_RIGHT* correspond to the code fragments labelled `copy_left` and `copy_right` respectively in Figure 1, their purpose being to rotate the tape appropriately to simulate the movement of the tape head.

Corresponding to each state $q \in Q$ there is a function $\mathtt{f}_q$ in $P_{M,x}$ that is defined as follows:

1. q_i is an accepting state. The function $\mathtt{f}_{q_i}$ is defined as

   $\mathtt{f}_{q_i}$($\mathtt{T}_{0,0}$, ..., $\mathtt{T}_{nt,ns}$) { /* do nothing */ }

2. q_i is a rejecting state. The function $\mathtt{f}_{q_i}$ is defined as

   $\mathtt{f}_{q_i}$($\mathtt{T}_{0,0}$, ..., $\mathtt{T}_{nt,ns}$) { Error-if-Zero(0) ; }

3. q_i is a universal state. Let the successors of q_i on tape symbol s_j be $q_{j'}$ and $q_{j''}$ (recall our assumption that q_i has exactly two successors on any given tape symbol). The function $\mathtt{f}_{q_i}$ is defined as

```
f_{q_i}( T_{0,0} , ..., T_{nt,ns} )
{
   local X_{0,0} = T_{0,0}, ..., X_{nt,ns} = T_{nt,ns};

   if ( - ) { TRANSITION(q_i, s_j, q'_j); TRANSITION(q_i, s_j, q''_j); } /* moves on s_j */
   ...
   else { TRANSITION(q_i, s_k, q'_k); TRANSITION(q_i, s_k, q''_k); } /* moves on s_k */
}
```

4. q_i is an existential state. Let the successors of q_i on tape symbol s_j be $q_{j'}$ and $q_{j''}$. The function $\mathtt{f}_{q_i}$ is defined as

```
f_{q_i}( T_{0,0} , ..., T_{nt,ns} )
{
   local X_{0,0} = T_{0,0}, ..., X_{nt,ns} = T_{nt,ns};

   if ( - ) {    /* moves on s_j */
      if ( - ) { TRANSITION(q_i, s_j, s'_j) }; else { TRANSITION(q_i, s_j, q''_j) };
   }
   ...
   else if ( - ) {    /* moves on s_k */
      if ( - ) { TRANSITION(q_i, s_k, q'_k) }; else { TRANSITION(q_i, s_k, q''_k) };
   }
}
```

The entry point of the program $P_{M,x}$ is the function `main()`, defined as

```
main()
{
Start:
   local  T_{0,0} , ...,  T_{nt,ns} ;

   INIT_TAPE;      /* initialize  T_{i,j}  based on M's input x */
   f_{q0}(T_{0,0},...,T_{nt,ns});
End:
}
```

The crucial point in the construction is that the Error-if-Zero(−) construct is used to keep track of whether anything "goes wrong" along an execution path: it sets the global variable `NoErr`, which is initialized to 1 when execution starts, to 0 along an execution path if either (i) the execution path does not correspond to a computation of M, because $P_{M,x}$ guesses incorrectly on the tape cell being scanned by M; or (ii) because the path encounters a rejecting state of M. Once `NoErr` has been set to 0 the structure of the program ensures that it cannot be reset to 1. Thus, at the end of the execution path, the value of `NoErr` can be used to determine whether that path corresponds to a valid accepting computation of M.

The dynamic analog of the call (multi-)graph of $P_{M,x}$ is the *valid call tree*, which is a finite tree where each vertex is labelled with a procedure name and a tuple of arguments. A vertex $(f, \bar{u})$ in such a tree has children $(f_1, \bar{u}_1), \ldots, (f_k, \bar{u}_k)$ if there is an execution path in $P_{M,x}$, starting with the call $f(\bar{u})$ with the value of `NoErr` $= 1$, that executes the procedure calls $f_1(\bar{u}_1), \ldots, f_k(\bar{u}_k)$ in f's body and returns with the value of `NoErr` still at 1 (the conditions on the value of `NoErr` ensure that nothing has gone wrong along the corresponding execution path). The following results establish the connection between the behaviors of the alternating Turing machine M and the program $P_{M,x}$. Here, $\overline{\mathtt{T}}_{i,j} \approx u$ denotes that the values of the tuple of variables $(\mathtt{T}_{0,0}, \ldots, \mathtt{T}_{nt,ns})$ in $P_{M,x}$ correctly reflect the tape contents u in M.

Theorem 4.1 *$P_{M,x}$ has a valid call tree with root $(\mathtt{f}_q, \overline{\mathtt{T}}_{i,j})$ if and only if M has an accepting computation tree with root qu, where $\overline{\mathtt{T}}_{i,j} \approx u$.*

Proof: We first show that $P_{M,x}$ has a valid call tree T_P with root $(\mathtt{f}_q, \overline{\mathtt{T}}_{i,j})$ if M has an accepting computation tree T_M with root qu, where $\overline{\mathtt{T}}_{i,j} \approx u$. We proceed by induction on the height of T_M.

The base case is for $n = 0$, which means that q is an accepting state. Suppose that the root of T_M is labelled qu. From the construction of $P_{M,x}$, it follows that the tree consisting of the single node $(\mathtt{f}_q, \overline{\mathtt{T}}_{i,j})$, where $\overline{\mathtt{T}}_{i,j} \approx u$, is a valid call tree.

For the inductive case, assume that $P_{M,x}$ has a valid call tree with root $(\mathtt{f}_{q'}, \bar{v}')$ whenever M has an accepting computation tree with root $q'u'$ and height $\leq k$, where $\bar{v}' \approx u'$, and consider an accepting computation tree T_M of M with height $k+1$. Let the root of T_M be qu, and suppose that $\overline{\mathtt{T}}_{i,j} \approx u$. We have two possibilities:

1. q is an existential state. From the definition of computation trees, T_M's root has a single child $q'u'$, and the subtree T'_M rooted at this child is also an accepting computation tree of M. Since T'_M has height less than $k+1$, it follows from the induction hypothesis that $P_{M,x}$ has a valid call tree T'_P whose root is labelled $(\mathtt{f}_{q'}, \bar{v}')$ such that $\bar{v}' \approx u'$.

   Suppose that the transition from q to q' occurs on tape symbol s_i. From the construction of $P_{M,x}$, the function $\mathtt{f}_q$ contains an execution path through the code defined by TRANSITION(q, s_i, q') that verifies that the tape symbol scanned is s_i, adjusts the variables $\overline{\mathtt{T}}_{i,j}$ as necessary to correspond to the tape contents u', and calls $\mathtt{f}_{q'}$. It follows from this that a tree with root $(\mathtt{f}_q, \overline{\mathtt{T}}_{i,j})$ that has a single subtree T'_P is a valid call tree for $P_{M,x}$.

2. q is a universal state. This means that T_M's root has two children $q'u'$ and $q''u''$, and that the subtrees T'_M and T''_M rooted at each of these children are accepting computation trees for M. Since

each of these subtrees has height less than $k+1$, it follows from the induction hypothesis that $P_{M,x}$ has valid call trees T'_P, with root labelled $(\mathtt{f}_{q'}, \bar{v}')$, and T''_P, with root labelled $(\mathtt{f}_{q''}, \bar{v}'')$, where $\bar{v}' \approx u'$ and $\bar{v}'' \approx u''$.

Suppose that the transitions from q to q' and q'' occur on tape symbol s_i. From the construction of $P_{M,x}$, the function $\mathtt{f}_q$ contains an execution path

`if ( - ) {` TRANSITION(q, s_i, q')`;` TRANSITION(q, s_i, q'')`; }`

that simulates each of these transitions by verifying that the tape symbol scanned is s_i, adjusting the variables $\overline{\mathtt{T}}_{i,j}$ as necessary, and calling the appropriate function in $P_{M,x}$. It follows that a tree with root $(\mathtt{f}_q, \overline{\mathtt{T}}_{i,j})$ that has two subtrees T'_P and T''_P is a valid call tree for $P_{M,x}$.

The proof in the other direction is very similar, except that the induction is on the height of the valid call trees of $P_{M,x}$. ∎

Corollary 4.2 *M accepts x if and only if there is an execution path p in $P_{M,x}$ from the program point labelled* `Start` *to that labelled* `End` *such* `NoErr` *= 1 at the end of p.*

Proof: We observe that by construction of $P_{M,x}$, the code at the point labelled `Start` sets `NoErr` to 1 and initializes the variables $\mathtt{T}_{i,j}$ according to the input x.

Suppose that M accepts x, i.e., there is an accepting computation tree T_M rooted at q_0x. It follows from Theorem 4.1 that there is a valid call tree T_P for $P_{M,x}$ with root $(\mathtt{f}_{q_0}, \overline{\mathtt{T}}_{i,j})$ where $\overline{\mathtt{T}}_{i,j} \approx x$. This means that there is an execution path in $P_{M,x}$ from `Start` to `End` such that `NoErr` = 1 at `End`.

Suppose that M does not accept x, i.e., there is no accepting computation tree T_M rooted at q_0x. From Theorem 4.1, it follows that there is no valid call tree in $P_{M,x}$ with root $(\mathtt{f}_{q_0}, \overline{\mathtt{T}}_{i,j})$ such that $\overline{\mathtt{T}}_{i,j} \approx x$. It follows that there is no execution path from `Start` to `End` along which the value of `NoErr` remains 1. ∎

It is easy to show, moreover, that $P_{M,x}$ can be generated using $O(\log|M| + \log|x|)$ space. The following result is then immediate:

Corollary 4.3 *The inter-procedural single-value problem for* Base+Proc+Err *is EXPTIME-hard.* ∎

It is interesting and instructive to compare this result with Theorem 3.5. For the intra-procedural case considered in Theorem 3.5, we can use ordinary assignments to program variables to keep track of whether or not an execution path in the program corresponds to a valid accepting computation of the Turing machine being simulated. We don't know whether the same technique works in the case of inter-procedural analysis of recursive programs: specifically, when simulating an alternating Turing machine, the handling of universal states seems problematic. Instead, we use a language mechanism—the Error-if-Zero$(-)$ construct—that allows us to accumulate a highly constrained summary of an execution path into a variable. This allows us to determine, from the value of this variable, whether or not anything went wrong at any point in an execution path. Notice that even though Corollary 4.3 gives a complexity result for single-value problems in Base+Proc+Err, the availability of the Error-if-Zero$(-)$ construct in fact allows us to incrementally accumulate (in a limited way) the values of a number of variables along an execution path. In fact, while the (intra-procedural) single-value problem for Base is solvable in polynomial time (Theorem 3.1), adding the Error-if-Zero$(-)$ construct makes it PSPACE-hard; this can be used to simplify the proof of the 1-pointer case in Theorem 3.9.

4.1 Applications to the Complexity of Inter-procedural Dataflow Analysis

4.1.1 Inter-procedural Pointer Alias Analyses

The following theorem gives the complexity of single-value problems for arbitrary programs in Base+Proc+1ptr. The proof relies on using an indirect assignment through a pointer to set a global variable to 0 if anything "goes wrong" along an execution path, and thereby simulate the Error-if-Zero$(-)$ construct.

Theorem 4.4 *The inter-procedural single-value problem for base variables in* BASE+PROC+1PTR *is EXPTIME-complete.*

Proof: The proof is by reduction from the inter-procedural single-value problem for BASE+PROC+ERR . We show how any program $P_{M,x}$ in BASE+PROC+ERR , generated for an ATM M and input x as discussed in Section 4, can be translated to a program P' in BASE+1PTR (here, $X_1, X_2, \ldots$ denote variables in P while $\mathtt{X}_1, \mathtt{X}_2, \ldots$ denote variables in P'):

1. P' contains global variables `Zero` and `One`, which are initialized to 0 and 1 respectively. Additionally, for each global variable V in P there is a global pointer variable `V` in P'; in particular, the distinguished (base) variable *NoErr* in P corresponds to a global pointer variable `NoErr` in P', which is initialized to the value `&One`.

2. For each n-argument function f in P there is an n-argument function `f` in P'. For each such pair of corresponding functions, for each local variable V in f there is a local pointer variable `V` in `f`.

3. Assignment statements in P are translated as follows: a statement '$X = e$' in P translates to the statement '`X =` e'', where e' is given by

$$e' = \begin{cases} \texttt{\&Zero} & \text{if } e \equiv 0 \\ \texttt{\&One} & \text{if } e \equiv 1 \\ \texttt{Y} & \text{if } e \equiv Y \text{ for some variable } Y \end{cases}$$

   Function calls are translated as follows: a call '$f(e_1, \ldots, e_n)$' translates to '$\mathtt{f}(e'_1, \ldots, e'_n)$', where the e'_i are given by:

$$e'_i = \begin{cases} \texttt{\&Zero} & \text{if } e_i \equiv 0 \\ \texttt{\&One} & \text{if } e_i \equiv 1 \\ \texttt{Y} & \text{if } e_i \equiv Y \text{ for some variable } Y \end{cases}$$

   Conditionals are translated unchanged.

4. A statement Error-if-Zero(X) is translated to '`*NoErr = *X`.'

5. The single-value problem '*NoErr* $= 0$' in P corresponds to the base-variable single-value problem '`One =` 1' in P'.

Each variable V in P is translated to a pointer variable `V` in P'; a value of 0 for V in P corresponds to `V` being a pointer to the base variable `Zero` in P', while a value of 1 for V corresponds `V` being a pointer to the variable `One`.

Consider the program $P_{M,x}$ generated for a given ATM M and input x. In the corresponding program $P'_{M,x}$ in BASE+1PTR , the variable `NoErr` is initially set to point to `One`, which has the value 1. Now consider any execution path p in P. If p does not contain any occurrence of a Error-if-Zero$(-)$ statement, the execution along the corresponding path in P' simply parallels that in P, the only difference being that instead of the values 0 and 1 in P we have `&Zero` and `&One` in P'. If the path p contains a statement Error-if-Zero(X) , then the corresponding statement in P' is '`*NoErr = *X`.' We have the following possibilities:

1. `NoErr` points to `One`, `X` points to `One`, and the value of `One` is 1 (corresponding to the variables *NoErr* and X in P both having the value 1). In this case this assignment to `*NoErr` has no effect on the value of any variable in P'. This parallels the behavior of P.

2. `NoErr` points to `One` and `X` points to `Zero` (corresponding to X having the value 0 in P). In this case the assignment sets the variable `One` to have the value 0. This again parallels the behavior of P.

3. `NoErr` points to `One`, but the value of `One` is 0 (due to an assigment corresponding to the previous case earlier in the execution). In this case, regardless of whether `X` points to `One` or to `Zero`, the value of `*X` is 0, so the assignment '`*NoErr = *X`' does not change the value of any variable in P'. In particular this means that `*NoErr` remains 0. Again, this parallels the behavior of P.

Thus, at the end of the execution of P', the variable `One` has the value 1 if and only if, at the end of the corresponding execution path in P, the value of *NoErr* is 1. The reduction described above establishes that the inter-procedural single-value problem for base variables in BASE+1PTR is EXPTIME-hard.

We next show how a program P in BASE+PROC+1PTR can be simulated by a $p(n)$-space-bounded ATM M_P, where n is the program size. M_P has its tape divided into four regions: *Globals*, *AnticipatedGlobals*, *TempGlobals*, and *Locals*. *Globals* contains the current snapshot of the global variables. *AnticipatedGlobals* shows the Globals as we expect them to be upon return from the current subroutine. *TempGlobals* is an auxiliary region big enough to hold *Globals* and *AnticipatedGlobals*. *Locals* contains the contents of local variables and subroutine arguments; the scope of these variables extends only to the end of the current subroutine (parameter passing and returning of results can be achieved using global variables). These regions are obviously polynomially bounded by the size of P.

M_P works as follows: It interprets the current subroutine `f` in P, updating *Globals* and *Locals* appropriately. When P is nondeterministic because of uninterpreted conditionals so is M_P, which "guesses" one of the branches of the conditional to continue interpreting (using existential states). When `f` returns M_P compares *Globals* with *AnticipatedGlobals* and goes into an accepting state if they are equal and otherwise into a rejecting state.

The key mechanism is how calls to a subroutine `g` are simulated. First M_P copies the *AnticipatedGlobals* into *TempGlobals* M_P then guesses the effect of the subroutine call on *Globals* and writes this guess into *AnticipatedGlobals*. Immediately after this M_P switches into a universal state. One successor of this state starts interpreting subroutine `g`. This computation branch will reach an accepting state only if *AnticipatedGlobals* was guessed correctly. The other successor continues interpreting subroutine f assuming the call to g behaves as expected, i.e., it copies *AnticipatedGlobals* to *Globals* and *TempGlobals* back to *AnticipatedGlobals*.

The subroutine `main()`, where the simulation begins is handled slightly differently. At the beginning of `main()` *Globals* is initialized and upon return from `main` M_P always enters an accepting state.

It is not hard to see that this will faithfully simulate P. If we interested in solving a single or simultaneous value problem—which we assume, without loss of generality, to be posed at the end of `main`—we can make M_P test the condition at the end of `main` and either go into an accepting state if the condition is satisfied or in a rejecting state otherwise. ∎

Corollary 4.5 *The complexity of precise inter-procedural pointer alias analysis in the presence of 2-level pointers is EXPTIME-complete.* ∎

Corollary 4.6 *The determination of precise solutions for the following inter-procedural analysis problems for base variables in* BASE+PROC+1PTR *is EXPTIME-complete:* (*a*) *reaching definitions;* (*b*) *live variables; and* (*c*) *available expressions.* ∎

4.1.2 Inter-procedural Analysis of Procedures with Reference Formals

Consider extending the language BASE along another direction: instead of allowing explicit pointers, as in Section 3.3.1, we allow (non-recursive) functions with reference formal parameters. It does not come as a surprise that an independent attributes analysis is inadequate for solving the single value problem in this case. To see this, consider the following program:

```
var a, b, x: integer;                  proc q(u: ref integer; v: integer)
main()                                 {
{                                         u = v;
   a = 0;                              }
   if (...) { x = 0; q(a,x); }
   else     { x = 1; q(b,x); }
}
```

We want to know whether or not `a = 1` can hold immediately after the conditional in `main()`. We need a relational attributes analysis of `q`'s arguments in order to determine that `q`'s first argument, `u`, cannot be a reference to `a` if its second argument `v` has the value 1. Thus, an independent attributes analysis is inadequate for this single value problem. The following result shows that (non-recursive) procedures with reference parameters can be used to solve arbitrary simultaneous-value problems for BASE.

Theorem 4.7 *The single value problem for* BASE *extended with procedures with reference parameters is PSPACE-complete for non-recursive programs and EXPTIME-complete for arbitrary programs.*

Proof: (sketch) The proof is very similar to that for Theorem 4.4, the primary difference being that instead of explicit pointer variables we use reference parameters. Each procedure in the program takes two additional arguments that are references to the global variables `Zero` and `One`. Instead of explicit assignments of `&Zero` and `&One`, as in the construction in the proof of Theorem 4.4, we use these reference parameters. The remainder of the proof remains essentially unchanged. ■

Corollary 4.8 *Precise inter-procedural liveness analysis and available expressions analysis for* BASE *extended with procedures with reference parameters are both PSPACE-complete for non-recursive programs, and EXPTIME-complete for arbitrary programs.*

Proof: The proof follows the lines of that of Theorem 4.7, modified in a manner analogous to that in Theorem 3.10. ■

This result corrects a minor flaw in Myers' original proof of the difficulty of such analysis problems [13]. Myers considered inter-procedural analyses in the presence of reference parameters, and claimed to show NP-completeness for liveness analysis and co-NP-completeness for available expressions; in fact, he proved only hardness results. Our results establish that membership in NP holds for acyclic non-recursive programs (Theorem 3.2), but stronger results can be given for general programs.

4.1.3 Inter-procedural Control Flow Analysis of Programs with Function Pointers

In this section we consider extending BASE in another direction, by adding C-style function pointers. These differ from general-purpose pointers in that (*i*) the objects pointed at are functions, rather than data; and (*ii*) the object obtained by dereferencing a function pointer cannot be modified by the program. The primary purpose of function pointers, therefore, is to affect control flow. The corresponding analysis problem is therefore a control flow analysis problem. The following result, whose proof follows the lines of those for Theorem 3.9 and Corollary 4.5, improves on an NP-hardness result by Zhang and Ryder [17]:

Theorem 4.9 *Precise control flow analysis in the presence of function pointers is PSPACE-complete for non-recursive programs and EXPTIME-complete for arbitrary programs.* ■

5 Summary and Related Work

The contributions of this paper can be summarized as follows:

1. **New Results** : To the best of our knowledge, the following are are new results: Corollary 3.8, Theorem 3.10(b,c), Corollary 4.5, and Corollary 4.6.

2. **Improvements to Existing Results** : Theorem 3.10 and Corollary 4.6 improve on a result by Pande *et al.* [15]. Corollary 4.5 improves on a result by Landi [8, 11]. Theorem 4.7 and Corollary 4.8 improve on a result by Myers [13].

3. **Explanations of Existing Results** : Theorems 3.7 and 3.9 explain the underlying reasons for Landi's complexity results for pointer alias analysis [8, 11]. Theorems 3.9 and 3.10(a) together explain why single-level pointers are hard to deal with when constructing intra-procedural def-use chains but not when considering intra-procedural pointer analyses. Theorem 4.9 explains the difficulty of inter-procedural control flow analysis in the presence of function pointers.

The distinction between independent attributes analyses and relational attributes analyses was first defined by Jones and Muchnick [6], who also examined the complexity of these approaches to program analysis. They showed that independent attributes analyses over a fixed finite domain has worst case complexity that is polynomial in the size of the program, while relational attributes analysis for programs consisting of assignments, sequencing, and "uninterpreted" conditionals—i.e., where we always assume that either branch of a conditional may be taken, or, equivalently, that all paths in the program are executable—but not containing any loops, is NP-hard [6]. Variations on the basic idea of this proof have been used for NP-hardness results by a number of authors [8, 11, 12, 13, 15], as well as in the proof of Theorem 3.2 in this paper. Jones and Muchnick also show that when loops and "interpreted" conditionals are added, the problem becomes PSPACE-hard. Unfortunately, since most dataflow analyses in practice treat conditionals as uninterpreted, the latter result is not directly applicable to them.

Nielson and Nielson consider, in a very general denotational setting, the number of iterations necessary to compute the least fixpoint of a functional over a finite lattice, under various assumptions about the kinds of functions considered [14]. By contrast, our work focuses on the overall computational complexity for certain kinds of program analyses. While the number of iterations needed to attain a fixpoint is an important factor in determining the amount of work done by an analysis, it is not the only such factor, and hence does not give a complete picture of the complexity of an analysis. To see this, observe that if we restrict our attention to intra-procedural analyses of loop-free programs, the resulting dataflow equations are not recursive, so a single iteration suffices to compute the least fixpoint; nevertheless, relational attributes analyses for such programs are NP-complete (Theorem 3.2).

Many researchers have given complexity results for specific program analysis problems (see, for example, [8, 11, 12, 13, 15]). As discussed earlier, these results do not generally provide insights into the underlying reasons for the efficiency, or lack thereof, of the analyses.

6 Conclusions

This paper attempts to elucidate the fundamental reasons why precise solutions to certain program analyses are computationally difficult to obtain. We give simple and general results that relate the complexity of a problem to whether or not it requires a relational attributes analysis. The applicability of this result is illustrated using a number of analyses discussed in the literature: we are able to derive the complexity results originally given by the authors, and in several cases even stronger complexity results, as direct corollaries to the results presented here, with little conceptual and notational effort.

Acknowledgements

Discussions with William Landi have been very helpful in clarifying complexity questions for pointer alias analysis.

References

[1] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers – Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] J. M. Barth, "A practical interprocedural data flow analysis algorithm", *Communications of the ACM* vol. 21 no. 9, pp. 724–736, 1978.

[3] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer, "Alternation", *J. ACM* vol. 28 no. 1, Jan. 1981, pp. 114–133.

[4] R. Chatterjee, B. G. Ryder, and W. A. Landi, "Complexity of Concrete Type-inference in the Presence of Exceptions", *Proc. European Symposium on Programming*, 1998.

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

[6] N. D. Jones and S. S. Muchnick, "Complexity of flow analysis, inductive assertion synthesis, and a language due to Dijkstra", In S. S Muchnick and N. D Jones, eds., *Program Flow Analysis: Theory and Applications*, chapter 12, pp. 380–393. Prentice-Hall, 1981.

[7] R. E. Ladner, R. J. Lipton, and L. J. Stockmeyer, "Alternating Pushdown Automata", *Proc. 19th IEEE Symposium on Foundations of Computer Science*, Oct. 1978, pp. 92–106.

[8] W. A. Landi, *Interprocedural Aliasing in the Presence of Pointers*, Ph.D. Dissertation, Rutgers University, New Brunswick, NJ, Jan. 1992.

[9] W. A. Landi, "Undecidability of Static Analysis", *ACM Letters on Programming Languages and Systems* vol. 1 no. 2, Dec. 1992, pp. 323–337.

[10] W. Landi, personal communication, June 1998.

[11] W. Landi and B. G. Ryder, "Pointer-induced Aliasing: A Problem Classification", *Proc. 18th ACM Symposium on Principles of Programming Languages*, Jan. 1991, pp. 93–103.

[12] J. R. Larus, *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*, Ph.D. Dissertation, University of California, Berkeley, 1989. Also available as Technical Report UCB/CSD 89/502, Computer Science Division (EECS), University of California, Berkeley, May 1989.

[13] E. W. Myers, "A Precise Inter-Procedural Data Flow Algorithm", *Proc. 8th ACM Symposium on Principles of Programming Languages*, Jan. 1981, pp. 219–230.

[14] H. R. Nielson and F. Nielson, "Bounded Fixed Point Iteration", *Proc. Nineteenth ACM Symposium on Principles of Programming Languages*, Jan. 1992, pp. 71–82.

[15] H. D. Pande, W. A. Landi, and B. G. Ryder, "Interprocedural Def-Use Associations for C Systems with Single Level Pointers", *IEEE Transactions on Software Engineering* vol. 20 no. 5, May 1994, pp. 385–403.

[16] H. D. Pande and B. G. Ryder, "Static Type Determination for C++", *Proc. Sixth USENIX C++ Technical Conference*, April 1994, pp. 85–97.

[17] S. Zhang and B. Ryder, "Complexity of single level function pointer aliasing analysis", Technical Report LCSR-TR-233, Laboratory of Computer Science Research, Rutgers University, October 1994.